kit de développement MakePython ESP32

tout dans une boîte

Tam Hanna (Slovaquie)

Les microcontrôleurs modernes, tels que l'ESP32, sont si puissants qu'ils peuvent être programmés en MicroPython. Grâce aux puissantes bibliothèques, cela vous permet de réaliser rapidement un projet. Avec le kit de développement MakePython ESP32, qui est d'une part, un manuel et d'autre part, un kit matériel, Dogan Ibrahim, auteur bien connu d'Elektor, nous présente le MicroPython avec des exemples concrets.

Nous n'avons pas vraiment besoin de discuter du fait que MicroPython n'est pas la voie vers une efficacité logicielle maximale. D'un autre côté, il est vrai que les microcontrôleurs modernes tels que l'ESP32 sont plus que capables de suivre le rythme d'un x486 en termes de performances. En particulier pour réaliser de petites séries, il peut donc être raisonnable d'échanger l'effort de programmation contre la vitesse en utilisant des langages de haut niveau.

Le manuel

Commençons par le manuel, qui est disponible en anglais : il est livré avec un design assez fantaisiste, qui fait croire à une (fausse) reliure en spirale.

La partie didactique commence par une brève explication de l'ESP32 dans son ensemble. Si vous êtes un lecteur d'Elektor et que vous avez déjà une expérience approfondie ou au moins basique des microcontrôleurs, vous aurez un aperçu « rapide » des différents périphériques fournis par Espressif dans le contrôleur. Si vous n'avez pas du tout d'expérience, les explications pourraient parfois être trop courtes pour une compréhension solide.

L'installation des EDI est expliquée en détail pour Windows, tandis que Linux est à peine abordé. Ibrahim présente à la fois uPyCraft et Thonny, mais ne travaille ensuite presque exclusivement qu'avec Thonny. S'ensuit un chapitre qui illustre l'exécution de quelques petits extraits Python à l'aide de Thonny. Si vous n'avez aucune connaissance de la syntaxe Python, vous ne pouvez pas vraiment vous lancer à ce stade. En revanche, les explications sont plus que suffisantes pour comprendre les bases de MicroPython et de l'EDI.

La présentation proprement dite des 46 projets contenus dans le livre se fait ensuite dans le style classique de Dogan Ibrahim : dans un premier temps, le professeur présente toujours la tâche à accomplir, puis il présente le code et les explications de la conception. Les amateurs de codage compact en particulier pourraient être quelque peu agacés par le fait que les listings comportent toujours

Figure 1. Le kit. L'emballage en plastique protège de manière fiable les composants contre les dommages.

Figure 2. Cette carte permet un démarrage rapide avec MicroPython.

un en-tête standardisé d'une bonne dizaine de lignes. D'autre part, les exemples, sans être compliqués, illustrent bien les aspects essentiels du MicroPython.

Après l'avoir lu, vous n'aurez certainement plus à craindre la recherche désespérée de la manière de mettre en service certains périphériques. De l'avis de l'auteur, le fait qu'Ibrahim explique comment restaurer une carte « ratée » avec un nouveau firmware MicroPython est tout à fait louable.

Un regard sur la carte

En parlant de la carte à évaluer, le manuel est disponible en combinaison avec le kit de la **figure 1**, qui convient également pour être emporté en vacances grâce à son emballage plastique assez robuste. Mais voyons maintenant le matériel proprement dit. L'expérience de l'auteur en matière d'enseignement a montré que les développeurs qui ne sont pas familiers avec les systèmes embarqués auront une expérience d'apprentissage plus confortable si la plate-forme de développement fournit un écran (idéalement entièrement graphique).

Elektor contourne astucieusement ce problème en fournissant le circuit imprimé rouge présenté sur la **figure 2**. Le pavé noir est un écran OLED populaire à SSD1306 avec une résolution de 128 par 64 pixels.

C'est ici que se cache le plus gros point de critique : les connecteurs de 2,54 mm ne sont pas soudés. C'est dommage, car une personne qui achète le kit dans un magasin et l'emporte avec elle en vacances est bien embêtée si elle n'a pas de fer à souder à portée de main et que le réparateur de téléphones portables du village ne veut pas l'aider.

Pour le reste, la carte n'a aucune raison d'être critiquée. La variante ESP32 utilisée est l'un des plus grands modèles et dispose d'une mémoire suffisante. L'interface MicroUSB permet d'utiliser le câble de n'importe quel ancien téléphone portable. En outre, le kit comprend également un câble (très court) de qualité décente.

Une démo

Bien que le manuel soit principalement orienté vers les besoins des développeurs travaillant sous Windows, le testeur se référera à Ubuntu 20.04 LTS dans les étapes suivantes, ne serait-ce que pour des raisons de commodité. L'EDI disponible à l'adresse [1] est un choix plutôt mauvais, car son exécution dans toutes les versions d'Ubuntu postérieures à 16.04 échoue avec un message d'erreur du type :

```
ImportError: /tmp/_MEIOhQKhz/libz.so.1: version
`ZLIB_1.2.9' not found (required by /usr/lib/x86_64-
linux-gnu/libpng16.so.16)
```

Une meilleure option est Thonny, qui peut être installé automatiquement en entrant :

```
bash <(wget -O - https://thonny.org/installer-for-linux)
```

Et peut ensuite être lancé en entrant la commande :

```
/home/tamhan/apps/thonny/bin/thonny
```

Pour désinstaller, on utilise la commande :`/home/tamhan/apps/thonny/bin/uninstall`. Après le premier démarrage, il ne reste plus qu'à confirmer le choix de la langue.

L'étape suivante consiste à connecter la carte à une station de travail. Heureusement, Elektor a bien pensé à livrer la carte avec un *runtime* MicroPython préconfiguré. Dans ce contexte, il est particulièrement pratique que la carte allume en même temps l'écran, ce qui facilite la vérification des fonctions. À propos, une puce pont USB-UART CP210x de Silicon Labs est utilisée comme convertisseur.

À ce stade, vous pouvez passer sur Thonny (**figure 3**). Veillez à cliquer sur la version de Python dans le menu de sélection qui apparaît en bas à droite et sélectionnez la version *MicroPython (ESP32)*. Thonny recherchera alors automatiquement les cartes ESP32.

La question de savoir si vous devez exécuter Thonny avec les privilèges de superutilisateur est discutable. Le compte utilisateur de l'auteur étant membre du groupe *plugdev*, il a pu exécuter Thonny avec son compte utilisateur normal et interagir avec l'ESP32.

Mon premier programme en MicroPython

Comme prochaine action, passons à la mise en service de l'écran OLED de l'appareil. Pour ce faire, nous ouvrons l'URL : *www.makerfabs.com/makepython-esp32-starter-kit.html*. Télécharger l'archive *MakePython ESP32 Lessons Source Code*. L'utilisation de RAR est un peu ennuyeuse sous Linux. Dans tous les cas, extrayez l'archive à un endroit facilement accessible. Pour les étapes

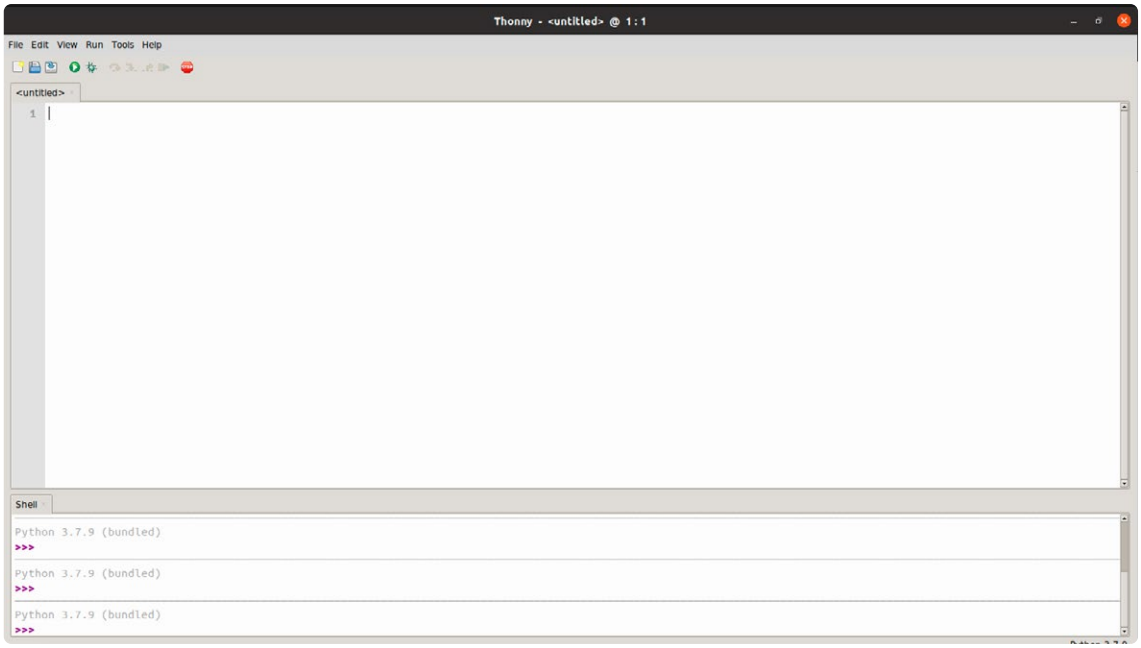

Figure 3. Capture d'écran de Thonny.

suivantes, nous avons particulièrement besoin du fichier *ssd1306.py*, qui fournit le code du pilote OLED.

Dans la première étape, cliquez sur *File -> Open*, puis choisissez l'option *This Computer*. Dans l'étape suivante, on navigue vers le fichier pour le charger dans l'éditeur. Ensuite, nous choisissons *File -> Save as* et enfin l'option *MicroPython Board*.

Si Thonny s'énerve à ce stade parce que le backend est *busy*, vous pouvez cliquer sur l'icône rouge d'arrêt dans la barre d'outils pour ordonner un arrêt. À l'étape suivante, vous enregistrez le fichier sous son nom d'origine (si vous l'avez modifié). Sinon, le fichier est déjà en place.

Comme c'est souvent le cas avec MicroPython, nous devons d'abord installer certaines bibliothèques et définir des constantes contenant des informations supplémentaires sur les spécifications de l'écran :

```
import machine
import ssd1306
import time
WIDTH = const(128)
HEIGHT = const(64)
```

Pour la communication proprement dite avec l'écran connecté via I²C, ce qui n'est pas courant, car on s'attend généralement à ce qu'il s'agisse de SPI, nous utilisons ensuite une instance de la classe I²C du logiciel :

```
pscl = machine.Pin(5, machine.Pin.OUT)
psda = machine.Pin(4, machine.Pin.OUT)
i2c = machine.I2C(scl=pscl, sda=psda)
oled = ssd1306.SSD1306_I2C(WIDTH, HEIGHT, i2c)
```

Enfin, nous devons envoyer du texte à l'écran selon le schéma suivant :

```
while True:
    oled.fill(0)
    oled.text('Hello World!',10,0)
    oled.show()
    time.sleep(1)
```

À ce stade, vous pouvez cliquer sur l'icône de lecture dans l'EDI pour voir le texte apparaître sur l'écran de la carte.

Ça se corse : MQTT

Comme prochaine tâche, nous voulons aller un peu au-delà de la portée du projet de base inclus dans le kit et réaliser une tâche avancée. Cela montre également que le kit est adapté à des tâches plus compliquées. Plus précisément, nous voulons utiliser le protocole MQTT, universellement connu. La tâche devient particulièrement intéressante du fait que nous voulons travailler avec une toute nouvelle version du broker MQTT Mosquitto, disponible. À l'URL [2] vous trouvez une description des problèmes que les *parser* (qui sont configurés de plus en plus strictement pour des raisons de sécurité) peuvent rencontrer en interaction avec les pilotes MicroPython.

En ce qui concerne les pilotes MicroPython, il existe désormais deux implémentations concurrentes de MQTT pour MicroPython. Cependant, nous voulons ici nous appuyer sur la variante la plus simple disponible sur [3]. Télécharger le fichier :

https://github.com/micropython/micropython-lib/blob/master/micropython/umqtt.simple/umqtt/simple.py

Et enregistrez-le sous le nom *simple.py* sur l'ordinateur de traitement.

Dans l'étape suivante, nous devons mettre les émetteurs wifi de la carte en mode station selon le schéma suivant :

```
. . .
wlan=network.WLAN(network.STA_IF)
```

Pour l'établissement effectif de la connexion, nous utilisons ensuite une méthode construite selon le schéma suivant (largement reprise du projet fourni par MakerFabs et implémente un compte à rebours

Listage 1.

```
def connectWiFi():
  i=0
  wlan.active(True)
  wlan.disconnect()
  wlan.connect("ssid","pass")
  while(wlan.ifconfig()[0]=='0.0.0.0'):
    i=i+1
    oled.fill(0)
    oled.text('connecting WiFi',0,16)
    oled.text('Countdown:'+str(20-i)+'s',0,48)
    oled.show()
    time.sleep(1)
    if(i>20):
      break
  oled.fill(0)
  oled.text('Makerfabs',25,0)
  oled.text('MakePython ESP32',0,32)
  if(i<20):
    oled.text('WIFI connected',0,16)
  else:
    oled.text('NOT connected!',0,16)
  oled.show()
  time.sleep(3)
  return True
```

incluant l'établissement de la connexion). Bien sûr, assurez-vous d'adapter les chaînes passées à `wlan.connect` à votre situation de réseau sans fil (voir **listage 1**).

L'établissement de la connexion de test proprement dite ne peut être plus simple :

```
connectWiFi()
while True:
    time.sleep(1)
```

Dans l'étape suivante, nous voulons nous doter d'un serveur Mosquitto. Bien qu'il ne soit pas le broker MQTT le plus rapide, il est open source, gratuit et très strict dans son implémentation MQTT.

Grâce à son utilisation répandue, il existe une image plus ou moins clés en main dans le Docker Hub qui permet le déploiement du serveur sans configurations profondes de l'ordinateur hôte.

Pour une première tentative, il suffit d'entrer la commande suivante :

```
docker run -it -p 1883:1883 -p 9001:9001 -v mosquitto.
conf:/mosquitto/config/mosquitto.conf eclipse-mosquitto
```

Tout d'abord, les paramètres `-p 1883:1883` and `-p 9001:9001` sont importants ici. Après tout, chaque conteneur Docker est doté d'un ensemble complet de ports TCP/IP par le *runtime*. Chacun de ces paramètres connecte ensuite l'un de ces ports à la carte réseau de l'ordinateur hôte. Via le paramètre `-v mosquitto.conf:/mosquitto/config/mosquitto.conf`, nous intégrons également un fichier de configuration.

À ce stade, c'est une bonne idée de faire une première tentative de téléchargement avec une connexion Internet existante. Normalement, Docker se connectera au hub et téléchargera les composants requis, puis quittera avec le message d'erreur suivant :

```
docker: Error response from daemon: source /var/lib/
docker/aufs/mnt/a22b9e557c37d99eb71f17e7bc6d38df6e7677
d09225376d416612adf0977ccd/mosquitto/config/mosquitto.
conf is not directory.
```

La cause de l'erreur est qu'il n'existe pas encore de fichier nommé *mosquitto.conf* sur le poste de travail hôte que nous pourrions mettre à la disposition du conteneur.

Pour résoudre le problème, il suffit de créer un nouveau fichier via la ligne de commande dans le répertoire de travail avec `gedit` :

```
(base) tamhan@tamhan-thinkpad:~$ gedit mosquitto.conf
```

Le passage de la version 1 de Mosquitto à la version 2 s'est accompagné d'un renforcement massif dans le domaine de la configuration de la sécurité. Alors qu'un Mosquitto 1.x (démarré avec les paramètres par défaut) acceptait les requêtes du serveur provenant de n'importe quel client, Mosquitto 2 le refuse. Toutefois, si vous placez les lignes suivantes dans le fichier de configuration, tout fonctionnera à nouveau comme d'habitude (et de manière non sécurisée) :

```
listener 1883
allow_anonymous true
```

L'exécution réelle peut alors être commandée selon le schéma suivant. Notez que le paramètre `-v` nécessite toujours un chemin d'accès complet, c'est pourquoi nous appelons la commande `pwd` et assemblons les résultats par la magie du shell. D'ailleurs, `pwd` signifie *Print Working Directory*, la **figure 4** montre le comportement :

```
(base) tamhan@tamhan-thinkpad:~$ docker run -it -p
1883:1883 -p 9001:9001 -v $(pwd)/mosquitto.conf:/
mosquitto/config/mosquitto.conf eclipse-mosquitto
. . .
```

Si le conteneur Docker confirme le succès de son démarrage en renvoyant `1658673183: mosquitto version 2.0.14 running`, nous sommes prêts à ce stade.

```
tamhan@tamhan-thinkpad: ~/Desktop/Stuff
Support: command not found
(base) tamhan@tamhan-thinkpad:~$ pwd
/home/tamhan
(base) tamhan@tamhan-thinkpad:~$ cd Desktop/Stuff/
(base) tamhan@tamhan-thinkpad:~/Desktop/Stuff$ pwd
/home/tamhan/Desktop/Stuff
(base) tamhan@tamhan-thinkpad:~/Desktop/Stuff$
```

Figure 4. La commande `pwd` renvoie le répertoire de travail actuel du shell.

Utiliser MQTT sur l'ESP32

Tout d'abord, nous avons besoin de quelques constantes pour travailler avec MQTT. Veillez à adapter l'adresse IP transmise dans la chaîne de caractères du serveur à votre situation d'exploitation locale :

```
SERVER = "192.168.178.146"
CLIENT_ID = ubinascii.hexlify(machine.unique_id())
TOPIC = b"led"
state = 0
```

Nous avons également besoin d'une fonction de rappel (*callback*) que le pilote MQTT appellera lorsqu'un message est reçu :

```
def sub_cb(topic, msg):
    global state
    print((topic, msg))
    . . .
```

La configuration réelle de la connexion MQTT est alors relativement simple :

```
connectWiFi()
c = MQTTClient(CLIENT_ID, SERVER, keepalive=30)
c.set_callback(sub_cb)
c.connect()
c.subscribe(TOPIC)
print("Connected to %s, subscribed to %s topic" % (SERVER, TOPIC))

while True:
    c.wait_msg()
```

L'appel de `c.subscribe` garantit que le pilote MQTT informe le serveur des canaux de messages qui l'intéressent actuellement. Il est également important d'appeler périodiquement la méthode `c.wait_msg()` afin de fournir au serveur MQTT du temps CPU pour traiter les informations entrantes ou à envoyer. Dans la ligne de commande ou dans la fenêtre de sortie, vous pouvez alors voir le message `Connected to 192.168.178.146, subscribed to b'led' topic`.

Dans l'étape suivante, nous voulons connecter l'une des LED incluses, ainsi que la résistance incluse, à la broche GPIO 12. L'auteur suppose que le lecteur a des connaissances suffisantes en électronique. L'initialisation du port GPIO ne nécessite alors que du code ESP32 ordinaire :

```
led = Pin(12, Pin.OUT, value=1)
```

Pour traiter les messages entrants, nous devons ensuite étendre la fonction de rappel en combinant la chaîne fournie dans la valeur `topic` avec les constantes fournies pour les différentes commandes :

```
def sub_cb(topic, msg):
    global state
    print((topic, msg))
    if msg == b"on":
        led.value(1)
        state = 1
    elif msg == b"off":
        led.value(0)
        state = 0
    elif msg == b"toggle":
        led.value(state)
```

Pour les tests réels, nous voulons utiliser l'utilitaire de ligne de commande *mosquitto_pub* : c'est un outil disponible sous Linux qui permet d'envoyer des commandes directement à un serveur MQTT. Les commandes suivantes sont ensuite utilisées pour allumer et éteindre la LED :

```
(base) tamhan@tamhan-thinkpad:~$ mosquitto_pub -t led
-m 'on'
(base) tamhan@tamhan-thinkpad:~$ mosquitto_pub -t led
-m 'off'
```

Si l'ESP32 et le serveur sont dans le même réseau, vous pouvez maintenant allumer et éteindre la LED.

Conclusion et perspectives

Le kit de développement MakePython ESP32 est une boîte à outils fascinante et compacte qui permet aux développeurs Python et/ou aux électroniciens de réunir rapidement les deux mondes. En particulier parce que tout est si joliment emballé de manière pratique, c'est un produit que l'auteur est heureux de recommander.

220429-04 — VF : Maxime Valens

Produits

- **Kit de développement MakePython ESP32 (SKU 20137)**
  www.elektor.fr/20137

LIENS

[1] EDI uPyCraft : https://github.com/DFRobot/uPyCraft
[2] umqtt.simple et mosquitto 2.0.12 : https://github.com/micropython/micropython-lib/issues/445
[3] umqtt.simple - client MQTT simple pour MicroPython :
https://github.com/micropython/micropython-lib/tree/master/micropython/umqtt.simple