Programmation ESP32 avec MicroPython

E/S Numériques, PWM, Bus série et Capteurs

Houssem LAHMER

Plan du cours

L'ESP32 et MicroPython

ESP32

- Microcontrôleur dual-core 240 MHz
- Wi-Fi et Bluetooth intégrés
- 520 KB de RAM
- GPIO, ADC, DAC, PWM, I^2C, SPI, UART
- Faible consommation d'énergie

MicroPython

- Python 3 pour microcontrôleurs
- Interpréteur complet
- Facile à apprendre
- Accès aux fonctionnalités matérielles
- REPL (Read-Eval-Print Loop)

Installation et configuration

Étapes de préparation

1. Installer esptool (pour flasher le firmware)
2. Télécharger le firmware MicroPython pour ESP32
3. Flasher le firmware sur l'ESP32
4. Installer un outil de communication (Thonny, ampy, rshell)

Composants nécessaires

- ESP32 DevKit
- Capteur BME280 (I^2C)
- Écran OLED SSD1306 128x64 (I^2C)
- LED RGB (commune cathode)
- Module carte microSD (SPI)
- Résistances, fils de connexion

E/S numériques : Concepts de base

GPIO (General Purpose Input/Output)

- Broches pouvant être configurées en entrée ou sortie
- Niveaux logiques : 0V (LOW) et 3.3V (HIGH)
- Certaines broches ont des fonctions spéciales
- Résistances de pull-up/pull-down internes disponibles

Précautions importantes

- ESP32 fonctionne à 3.3V (Non compatible 5V directement)
- Courant max : 12mA par GPIO
- Certaines broches réservées (strapping pins)
- Attention aux broches utilisées au démarrage (0, 2, 5, 12, 15)

Pilotage de LEDs — Allumage / Extinction

Contrôle d'une LED simple

```
from machine import Pin
import time

# Configurer la broche 2 en sortie
led = Pin(2, Pin.OUT)

# Allumer la LED
led.value(1)  # ou led.on()

# Attendre 1 seconde
time.sleep(1)

#  teindre  la LED
led.value(0)  # ou led.off()
```

Pilotage de LEDs — Clignotement

Faire clignoter une LED

```
from machine import Pin
import time

# Boucle infinie de clignotement
while True:
    led.value(not led.value())  # Inversion de l'état
    time.sleep(0.5)
```

Lecture de boutons — Configuration

Configuration d'un bouton

```
from machine import Pin

# Configurer la broche 4 en entr e avec pull-up
bouton = Pin(4, Pin.IN, Pin.PULL_UP)

# Lecture simple de l' tat  du bouton
etat = bouton.value()  # 0 si appuy , 1 sinon
```

Lecture de boutons — Anti-rebond

Gestion du rebond (debouncing)

```
import time

def lire_bouton_debounce(pin, delai=20):
    etat_initial = pin.value()
    time.sleep_ms(delai)
    # On v rifie que l' tat  est stable
    return pin.value() == etat_initial and pin.value()

# Exemple d'utilisation
if lire_bouton_debounce(bouton):
    print("Bouton stablement appuy ")
```

Projet : Contrôle LED par bouton

Exemple complet

```
from machine import Pin
import time

led = Pin(2, Pin.OUT)
bouton = Pin(4, Pin.IN, Pin.PULL_UP)
etat_led = False

while True:
    if bouton.value() == 0:   # Bouton appuy
        time.sleep_ms(20)   # Anti-rebond
        if bouton.value() == 0:
            etat_led = not etat_led
            led.value(etat_led)
            # Attendre que le bouton soit rel ch
            while bouton.value() == 0:
                time.sleep_ms(10)
    time.sleep_ms(50)
```

PWM : Modulation de Largeur d'Impulsion

Principe du PWM

- Génère un signal carré avec rapport cyclique variable
- Simule une tension analogique par variation du duty cycle
- Fréquence et rapport cyclique ajustables
- Applications : contrôle de luminosité, vitesse moteur, etc.

25% duty cycle 50% duty cycle 75% duty cycle

Génération de PWM avec ESP32

Configuration PWM de base

```
from machine import Pin, PWM
import time

# Cr er un objet PWM sur la broche 5
pwm = PWM(Pin(5))

# Configurer la fr quence (100Hz)
pwm.freq(100)

# Configurer le duty cycle (0-1023)
# 0 = 0%, 1023 = 100%
pwm.duty(512)   # 50%

# D sactiver le PWM quand termin
# pwm.deinit()
```

Fade in/out LED — Montée progressive

Fade in (augmentation de la luminosité)

```
from machine import Pin, PWM
import time

# Cr er un objet PWM sur la broche 5
led_pwm = PWM(Pin(5))
led_pwm.freq(1000)  # 1 kHz, bonne fr quence pour
    LEDs

# Fade in
for i in range(0, 1024, 10):
    led_pwm.duty(i)
    time.sleep_ms(20)

# Maintenir allum
time.sleep(1)
```

Fade in/out LED — Descente progressive

Fade out (diminution de la luminosité) et arrêt

```
# Fade out
for i in range(1023, -1, -10):
    led_pwm.duty(i)
    time.sleep_ms(20)

# D sactiver le PWM
led_pwm.deinit()
```

Contrôle de LED RGB — Configuration

Initialisation des canaux PWM

```
from machine import Pin, PWM
import time

# Configurer les canaux RGB sur ESP32
red   = PWM(Pin(27))
green = PWM(Pin(26))
blue  = PWM(Pin(25))

# D finir la fr quence PWM
for chan in (red, green, blue):
    chan.freq(1000)  # 1 kHz pour un bon rendu de
        couleurs

# Fonction pour d finir une couleur RGB (0-255 chacun
    )
def set_color(r, g, b):
    # Conversion de   l  chelle   0  255       0  1023
        pour duty()
```

Contrôle de LED RGB — Exemples d'utilisation

Changer la couleur de la LED

```
# Exemples de couleurs
set_color(255, 0, 0)     # Rouge
time.sleep(1)
set_color(0, 255, 0)     # Vert
time.sleep(1)
set_color(0, 0, 255)     # Bleu
time.sleep(1)

# Fondu entre deux couleurs
for i in range(0, 256, 5):
    set_color(i, 0, 255-i)  # Du bleu au rouge
    time.sleep_ms(50)

#  teindre  la LED (noir)
set_color(0, 0, 0)
```

Contrôle de LED RGB — Couleurs de base

Exemples d'utilisation (partie 1)

```
# Rouge pur
set_color(255, 0, 0)
time.sleep(1)

# Vert pur
set_color(0, 255, 0)
time.sleep(1)

# Bleu pur
set_color(0, 0, 255)
time.sleep(1)

# Jaune (Rouge + Vert)
set_color(255, 255, 0)
time.sleep(1)
```

Contrôle de LED RGB — Couleurs étendues

Exemples d'utilisation (partie 2)

```
# Cyan (Vert + Bleu)
set_color(0, 255, 255)
time.sleep(1)

# Magenta (Rouge + Bleu)
set_color(255, 0, 255)
time.sleep(1)

# Blanc
set_color(255, 255, 255)
time.sleep(1)
```

Contrôle de moteur — Configuration PWM

Initialisation du canal PWM

```
from machine import Pin, PWM

# Configurer la broche PWM pour le moteur
moteur_pwm = PWM(Pin(12))
moteur_pwm.freq(500)  # 500 Hz, adapt aux moteurs DC
```

Contrôle de moteur — Fonction de vitesse

Vue d'ensemble des bus série

UART

- 2 fils (TX/RX)
- Asynchrone
- Point à point
- Facile à utiliser
- Vitesse moyenne

I²C

- 2 fils (SDA/SCL)
- Synchrone
- Multi-esclave
- Adressage
- Vitesse moyenne

SPI

- 4 fils (MOSI/MISO/CLK/CS)
- Synchrone
- Multi-esclave
- Sélection par CS
- Haute vitesse

UART : Communication série

Configuration de base UART

```
from machine import UART, Pin
import time

# Configuration UART2 (TX: GPIO17, RX: GPIO16)
uart = UART(2, baudrate=9600, tx=17, rx=16)

# Param tres optionnels
uart.init(baudrate=9600, bits=8, parity=None, stop=1)

# Envoi de donn es
uart.write('Hello UART\r\n')

# R ception de donn es
if uart.any():
    data = uart.read()
    print(data)
```

Lecture GPS — Configuration UART

Configuration du port série pour GPS

```
from machine import UART, Pin

# Configuration UART pour GPS (typiquement 9600 baud)
gps_uart = UART(2, baudrate=9600, tx=17, rx=16)
```

Lecture GPS — Parsing et boucle

Lecture et affichage des trames GPRMC

```
import time

def parse_gps():
    if gps_uart.any():
        line = gps_uart.readline()
        try:
            line = line.decode('utf-8').strip()
            if line.startswith('$GPRMC'):
                print(line)
        except:
            pass

# Boucle principale
while True:
    parse_gps()
    time.sleep_ms(100)
```

I²C : Communication avec capteurs et afficheurs

Configuration I²C de base

```
from machine import Pin, I2C
import time

# Configuration I2C (SCL=Pin 22, SDA=Pin 21)
i2c = I2C(0, scl=Pin(22), sda=Pin(21), freq=100000)

# Scanner les p riph riques I2C
devices = i2c.scan()
print("P riph riques I2C trouv s:", [hex(device)
    for device in devices])

#  crire  des donn es
i2c.writeto(0x3C, b'\x00\x01\x02')

# Lire des donn es
data = i2c.readfrom(0x3C, 3)  # Lire 3 octets
```

Afficheur OLED I²C — Initialisation

Configuration du bus I²C et initialisation de l'écran

```
from machine import Pin, I2C
import ssd1306

# Configuration I2C
i2c = I2C(0, scl=Pin(22), sda=Pin(21), freq=100000)

# Initialiser l'cran OLED (128x64 pixels)
oled = ssd1306.SSD1306_I2C(128, 64, i2c)

# Effacer l'cran
oled.fill(0)
```

Afficheur OLED I²C — Affichage de contenu

Texte et formes graphiques

```
# Afficher du texte
oled.text("ESP32 + OLED", 0, 0)
oled.text("MicroPython", 0, 10)
oled.text("I2C Demo", 0, 20)

# Dessiner un rectangle vide et rempli
oled.rect(0, 30, 128, 20, 1)
oled.fill_rect(2, 32, 124, 16, 0)

# Mettre    jour l'affichage
oled.show()
```

SPI — Configuration et écriture

```
from machine import Pin, SPI
import time

# Configuration SPI
# sck=Pin18, mosi=Pin23, miso=Pin19
spi = SPI(1, baudrate=1000000, polarity=0, phase=0,
          sck=Pin(18), mosi=Pin(23), miso=Pin(19))

# CS (Chip Select)     g rer manuellement
cs = Pin(5, Pin.OUT)
cs.value(1)  # D sactiver le p riph rique

#   crire  des donn es
cs.value(0)  # Activer le p riph rique
spi.write(b'\x01\x02\x03')
cs.value(1)  # D sactiver le p riph rique
```

SPI — Lecture de données

```
# Lire des donn es
cs.value(0)                 # Activer le p riph rique
data = spi.read(3)          # Lire 3 octets
cs.value(1)                 # D sactiver le
    p riph rique

print("Donn es re ues:", data)
```

SD via SPI — Configuration et écriture

Montage et écriture sur la carte SD

```
import os
from machine import Pin, SPI
import sdcard

# Configurer SPI et CS
spi = SPI(1, baudrate=10000000, polarity=0, phase=0,
          sck=Pin(18), mosi=Pin(23), miso=Pin(19))
cs = Pin(5, Pin.OUT)

# Monter la carte SD
sd = sdcard.SDCard(spi, cs)
os.mount(sd, '/sd')

#  crire dans un fichier
with open('/sd/test.txt', 'w') as f:
    f.write('Test de stockage sur carte SD')
```

SD via SPI — Lecture et exploration

Lecture et liste des fichiers

```
# Lire un fichier
with open('/sd/test.txt', 'r') as f:
    contenu = f.read()
    print(contenu)

# Lister les fichiers sur la SD
print(os.listdir('/sd'))
```

Acquisition analogique avec ADC

Convertisseur Analogique-Numérique (ADC)

- ESP32 : 2 ADCs avec 18 canaux au total
- Résolution configurable (9-12 bits)
- Plage de tension : 0-3.3V (attention : max 3.3V !)
- Ports ADC disponibles sur broches GPIO 32-39
- ADC2 non disponible si WiFi activé

Limitations à connaître

- Non-linéarité aux extrêmes de la plage
- Précision : environ 12 bits (mais bruit)
- Impédance d'entrée variable
- Besoin de calibration pour mesures précises

Lecture ADC — Configuration

Initialisation de l'ADC

```
from machine import ADC, Pin

# Cr er un objet ADC sur la broche 34 (ADC1_6)
adc = ADC(Pin(34))

# Configurer l'att nuation (plage de tension)
# 0dB: 0  1V  , 2.5dB: 0  1  .34V, 6dB: 0  2V  , 11dB: 0
    3  .3V
adc.atten(ADC.ATTN_11DB)  # Plage compl te 0  3  .3V

# Configurer la r solution
adc.width(ADC.WIDTH_12BIT)  # 12 bits (0  4095  )
```

Lecture ADC — Lecture et conversion

Lecture de la valeur et estimation de la tension

```
import time

# Lecture simple
valeur = adc.read()
print("Valeur ADC:", valeur)

# Conversion approximative en tension
tension = valeur * 3.3 / 4095
print("Tension estim e:", tension, "V")
```

Configuration ADC pour potentiomètre

```
from machine import ADC, Pin
import time

# Configuration ADC pour potentiom tre
pot = ADC(Pin(34))
pot.atten(ADC.ATTN_11DB)       # Plage 0  3  .3V
pot.width(ADC.WIDTH_12BIT)     # R solution 12 bits (0
    4095  )
```

Potentiomètre LED PWM — Application à la LED

Mapping ADC → PWM et contrôle de luminosité

```
from machine import PWM

# Configuration PWM pour LED
led_pwm = PWM(Pin(5))
led_pwm.freq(1000)                  # 1 kHz pour LED

while True:
    # Lecture du potentiom tre
    val = pot.read()                # 0  4095

    # Conversion pour PWM (0  1023  )
    pwm_val = int(val / 4)    # (4095/4      1023)

    # Appliquer     la LED
    led_pwm.duty(pwm_val)

    print("ADC:", val, "PWM:", pwm_val)
    time.sleep_ms(100)
```

Capteur de température analogique (LM35) – Configuration

Lecture d'un capteur LM35

```
from machine import ADC, Pin
import time

# Configuration ADC (LM35 connect      la broche 35)
adc = ADC(Pin(35))
adc.atten(ADC.ATTN_11DB)
adc.width(ADC.WIDTH_12BIT)

def lire_temperature():
    # Faire plusieurs lectures pour moyenner
    somme = 0
    n_echantillons = 10

    for _ in range(n_echantillons):
        somme += adc.read()
        time.sleep_ms(10)

    moyenne = somme / n_echantillons
```

Capteur de température analogique (LM35) – Affichage

Affichage de la température mesurée

```
while True:
    temp = lire_temperature()
    print("Temp rature:", temp, " C ")
    time.sleep(1)
```

Contrôle de servomoteurs – Configuration de base

Connexion et configuration PWM

```
from machine import Pin, PWM
import time

# Cr er un objet PWM pour le servo
servo = PWM(Pin(13), freq=50)  # 50Hz pour servos
    standards

# Fonction pour d finir l'angle (0-180  )
def set_angle(angle):
    # Conversion angle en duty cycle
    # Typiquement: 0.5ms (0  )    2.5ms (180  ) sur
        p riode de 20ms
    # Calcul: (0.5 + angle * 2 / 180) / 20 * 1023
    min_duty = 26   # 0.5ms/20ms * 1023
    max_duty = 123   # 2.5ms/20ms * 1023

    duty = min_duty + (max_duty - min_duty) * angle /
        180
```

Contrôle de servomoteurs – Test et positionnement

Déplacement du servo à différents angles

```
# Test servo
for angle in range(0, 181, 10):
    set_angle(angle)
    print("Angle:", angle)
    time.sleep(0.2)

# Position milieu
set_angle(90)
```

Pilotage de moteurs pas à pas

Contrôle d'un moteur pas à pas avec ULN2003

```
from machine import Pin
import time

# Configuration des 4 broches pour le ULN2003
in1 = Pin(19, Pin.OUT)
in2 = Pin(18, Pin.OUT)
in3 = Pin(5, Pin.OUT)
in4 = Pin(17, Pin.OUT)

# S quence pour le mode pas complet
sequence = [
    [1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
]

def step(direction):
```

Pilotage de moteurs pas à pas (suite)

Fonction de rotation

```
# Initialisation du compteur
step_counter = 0

# Fonction pour tourner d'un certain nombre de pas
def rotate(steps, direction, delay_ms=2):
    for _ in range(steps):
        step(direction)
        time.sleep_ms(delay_ms)

    # D sactiver toutes les bobines pour   conomiser
       l' nergie
    for pin in [in1, in2, in3, in4]:
        pin.value(0)

# Exemples d'utilisation
# Rotation de 200 pas dans le sens horaire
rotate(200, True)
```

Intégration capteur I²C : BME280

Lecture d'un capteur environnemental BME280

```
from machine import I2C, Pin
import time
import BME280  # Biblioth que pour le capteur

# Configuration I2C
i2c = I2C(0, scl=Pin(22), sda=Pin(21), freq=100000)

# Scanner les p riph riques
devices = i2c.scan()
if 0x76 in devices or 0x77 in devices:
    print("BME280 trouv !")
else:
    print("BME280 non trouv !")

# Initialiser le capteur
bme = BME280.BME280(i2c=i2c)

# Lecture des valeurs
```

MPU6050 : Initialisation

Configuration I2C et initialisation du capteur

```
from machine import I2C, Pin
import time
from mpu6050 import MPU6050  # Biblioth que pour le
    capteur

# Configuration de l'interface I2C
i2c = I2C(0, scl=Pin(22), sda=Pin(21), freq=400000)

# Initialisation du capteur MPU6050
mpu = MPU6050(i2c)
```

MPU6050 : Lecture des données

Lecture des valeurs d'accélération, gyroscope et température

```
while True:
    # Lecture des donn es
    accel = mpu.get_accel_data()
    gyro = mpu.get_gyro_data()
    temp = mpu.get_temp()

    # Affichage des r sultats
    print("Acc l ration: x={:.2f}g, y={:.2f}g, z
        ={:.2f}g".format(
         accel['x'], accel['y'], accel['z']))
    print("Gyroscope: x={:.2f}  /s, y={:.2f}  /s, z
        ={:.2f}  /s".format(
         gyro['x'], gyro['y'], gyro['z']))
    print("Temp rature: {:.1f} C ".format(temp))
    print("-" * 20)

    time.sleep(0.5)
```

Mini-projet 1 : Schéma de câblage

Connexions des composants

- **Bus I²C partagé :**
  - SCL → GPIO22
  - SDA → GPIO21
- **LED RGB :**
  - Rouge → GPIO27 (via résistance 220)
  - Vert → GPIO26 (via résistance 220)
  - Bleu → GPIO25 (via résistance 220)
- **Module SD (SPI) :**
  - MOSI → GPIO23
  - MISO → GPIO19
  - SCK → GPIO18
  - CS → GPIO5

Mini-projet 2 : Contrôleur moteur intelligent

Objectifs

- Piloter un servomoteur avec un potentiomètre
- Utiliser un accéléromètre MPU6050 pour détecter les mouvements
- Contrôler un moteur DC avec PWM basé sur l'accélération
- Afficher l'état sur un terminal série (UART)

Composants nécessaires

- ESP32 DevKit
- Servomoteur standard
- Potentiomètre 10k
- Capteur MPU6050 (accéléromètre/gyroscope)
- Moteur DC + transistor ou driver L293D
- Fils de connexion, résistances

Mini-projet 2 : Exemple de code

Lecture MPU6050 et contrôle moteur

```
from machine import I2C, Pin, ADC, PWM
import time
from mpu6050 import MPU6050

# Configuration I2C pour MPU6050
i2c = I2C(0, scl=Pin(22), sda=Pin(21), freq=400000)
mpu = MPU6050(i2c)

# Configuration PWM pour moteur DC
motor_pwm = PWM(Pin(12))
motor_pwm.freq(500)

# Configuration ADC pour potentiom tre
pot = ADC(Pin(34))
pot.atten(ADC.ATTN_11DB)

# Configuration PWM pour servomoteur
servo = PWM(Pin(13), freq=50)
```

Mini-projet 2 : Exemple de code (suite)

```
def set_servo_angle(angle):
    # Limiter l'angle entre 0-180
    if angle < 0:
        angle = 0
    elif angle > 180:
        angle = 180
    # Convertir en duty cycle
    min_duty = 26   # 0.5ms/20ms * 1023
    max_duty = 123   # 2.5ms/20ms * 1023
    duty = min_duty + (max_duty - min_duty) * angle /
        180
    servo.duty(int(duty))

# Boucle principale
while True:
    try:
        # Lecture du potentiom tre pour servomoteur
        pot_value = pot.read()
        servo_angle = int(pot_value * 180 / 4095)
        set_servo_angle(servo_angle)
```

Bonnes pratiques MicroPython sur ESP32

Optimisation des performances

- Éviter les allocations dynamiques dans les boucles
- Préférer les entiers aux nombres à virgule flottante
- Utiliser des constantes locales plutôt que des attributs
- Minimiser les opérations d'entrée/sortie
- Structurer le code en modules pour la réutilisation

Gestion de la mémoire

- ESP32 dispose de peu de RAM (320KB pour l'application)
- Régulièrement appeler `gc.collect()` pour le garbage collector
- Utiliser `micropython.mem_info()` pour diagnostiquer
- Créer un fichier `boot.py` pour configuration initiale
- Diviser les grosses applications en modules

Gestion de l'énergie

Modes d'économie d'énergie

- **Mode actif :** Consommation 240mA
- **Mode modem-sleep :** WiFi désactivé, 30mA
- **Mode light-sleep :** CPU en pause, 0.8mA
- **Mode deep-sleep :** Presque tout éteint, 10µA

Exemple de mode deep-sleep

```
Excuter quelque chose print("Mesure de capteur...") time.sleep(2)
Mettre en veille profonde pendant 10 secondes print("Mise en
              veille pour 10 secondes...")
              machine.deepsleep(10000)
```

Gestion des erreurs et robustesse

Techniques pour code robuste

```
import machine
import time

# Watchdog pour red marrer en cas de blocage
wdt = machine.WDT(timeout=5000)  # 5 secondes

# D tection de cause de red marrage
reset_cause = machine.reset_cause()
if reset_cause == machine.DEEPSLEEP_RESET:
    print("R veil de deep sleep")
elif reset_cause == machine.WDT_RESET:
    print("Red marrage par watchdog")
elif reset_cause == machine.SOFT_RESET:
    print("Red marrage logiciel")

# Gestion des exceptions
try:
    # Code principal
```

WiFi et connectivité réseau

Configuration WiFi de base

```
import network
import time

# Configuration WiFi en mode station
wlan = network.WLAN(network.STA_IF)
wlan.active(True)

# Connexion au r seau WiFi
ssid = 'MonReseau'
password = 'MonMotDePasse'
wlan.connect(ssid, password)

# Attendre la connexion
max_wait = 10
while max_wait > 0:
    if wlan.isconnected():
        break
    max_wait -= 1
```

Requêtes HTTP et IoT

Envoi de données à un service web

```
import urequests
import json

def envoyer_donnees(temp, hum, pression):
    # Pr parer les donn es
    donnees = {
        "temperature": temp,
        "humidite": hum,
        "pression": pression
    }

    # Convertir en JSON
    json_data = json.dumps(donnees)

    # D finir les en-t tes
    headers = {
        'Content-Type': 'application/json'
    }
```

Récapitulatif des concepts clés

Points essentiels

- ESP32 offre une large gamme d'interfaces (GPIO, ADC, PWM, bus série)
- MicroPython facilite le développement avec une syntaxe simple
- L'ESP32 est idéal pour projets IoT, capteurs et contrôle
- Combinaison puissante pour prototypage rapide

Bonnes pratiques

- Structurer le code en modules réutilisables
- Optimiser la consommation d'énergie selon les besoins
- Gestion des erreurs et watchdog pour fiabilité
- Documenter le code et les connexions matérielles

Ressources pour aller plus loin

Documentation officielle

- Documentation MicroPython : `https://docs.micropython.org/`
- Documentation ESP32 : `https://docs.espressif.com/`
- Référence Python : `https://docs.python.org/fr/3/`

Tutoriels et exemples

- Adafruit Learning System : `https://learn.adafruit.com/`
- Random Nerd Tutorials : `https://randomnerdtutorials.com/`
- MicroPython Forum : `https://forum.micropython.org/`
- GitHub MicroPython : `https://github.com/micropython/micropython`

Bibliothèques utiles

- Pilotes de capteurs : `https://github.com/micropython/micropython-lib`
- Bibliothèques I^2C pour écrans et capteurs
- Utilitaires pour WiFi, serveur web et MQTT

Questions et contact

Merci pour votre attention !

Des questions ?

contact@formateur-esp32.fr

Supports et exemples de code disponibles sur le site du cours