Champignonnière

1. Présentation

Ce système utilise une carte ESP32 pour mesurer l'environnement, contrôler des actionneurs, afficher les données via une interface web et enregistrer les mesures sur carte SD.

2. Objectif

Mesurer : température, humidité, pression, CO2, TVOC, humidité du sol, pH.
Contrôler : ventilateur et brumisateur.
Afficher : interface web.
Enregistrer : données sur carte SD.

3. Architecture

Fonctionnement en boucle : CAPTEURS → TRAITEMENT → DÉCISION → ACTION → AFFICHAGE → ENREGISTREMENT

4. Câblage

I2C : SDA → GPIO21, SCL → GPIO22
Sol → GPIO34
pH → GPIO35
Ventilateur → GPIO26
Brumisateur → GPIO27
SD : CS→5, SCK→18, MOSI→23, MISO→19

5. Fonctionnement du code

Le programme initialise les capteurs, le WiFi, la carte SD puis exécute une boucle toutes les 5 secondes pour lire les capteurs et agir.

6. Lecture des capteurs

BME280 : température, humidité, pression
SGP30 : CO2 et TVOC
Sol : lecture analogique convertie en %
pH : tension convertie en valeur pH

7. Logique

Ventilateur ON si CO2 >=1000 ppm, OFF si <=900 ppm
Brumisateur ON si humidité <=70%, OFF si >=80%
Avec hystérésis et temporisation pour éviter les oscillations.

8. Interface Web

Accessible via http://192.168.4.1
Affiche les mesures et permet de contrôler les relais.
Actualisation automatique toutes les 10 secondes.

9. Carte SD

Enregistre les données au format CSV compatible Excel.

10. Calibration

Sol : définir valeurs sec/humide
pH : calibrer avec solutions pH4 et pH7

11. Debug

Utiliser le moniteur série pour vérifier les valeurs et le fonctionnement.

12. Problèmes fréquents

SD non détectée, changer de carte SD

capteurs mal câblés,

relais inversé,

alimentation insuffisante.

13. Conclusion

Système autonome permettant surveillance et régulation environnementale avec enregistrement des données.

Le code complet commenté

```
// === ESP32 : BME280 + SGP30 + Sonde sol + 2 Relais (Ventilo +
Brumisateur) + pH + LOG SD ===
// Titre du projet : ce programme pilote une station de mesure complète
avec capteurs,
// actionneurs (relais), affichage web et enregistrement sur carte SD.

// (Commentaires alignés sur le fichier « code complet.xlsx » — Q1 → Q40)
// Indique que les commentaires sont liés à ton tableau de
questions/réponses.

#include <Arduino.h>
// Bibliothèque de base Arduino : indispensable pour setup(), loop(),
pinMode(), digitalWrite(), etc.

#include <WiFi.h>
// Bibliothèque Wi-Fi de l'ESP32 : permet de créer un réseau sans fil ou
de s'y connecter.

#include <WebServer.h>
// Bibliothèque qui permet de créer un serveur web sur l'ESP32.

#include <Adafruit_Sensor.h>
// Bibliothèque générique Adafruit pour uniformiser l'accès aux capteurs.

#include <Adafruit_BME280.h>
// Bibliothèque spécifique au capteur BME280 : température, humidité,
pression.

#include <Adafruit_SGP30.h>
// Bibliothèque spécifique au capteur SGP30 : eCO2 et TVOC.

#include <math.h>
// Bibliothèque mathématique : utilisée ici pour la fonction expf().

#include <esp32-hal-adc.h>
// Bibliothèque liée au convertisseur analogique-numérique (ADC) de
l'ESP32.
```

```
#include <SPI.h>
// Bibliothèque du bus SPI : utilisée pour communiquer avec la carte SD.

#include <SD.h>
// Bibliothèque de gestion de carte SD.

// ====== Réseau (AP) ======
const char* SSID     = "CHAMPY";
// Nom du réseau Wi-Fi créé par l'ESP32 en mode point d'accès.

const char* PASSWORD = "12345678";
// Mot de passe du réseau Wi-Fi.

// ====== Broches capteurs / actionneurs ======
#define SOIL_PIN       34
// Broche analogique utilisée pour la sonde d'humidité du sol.

#define PH_PIN         35
// Broche analogique utilisée pour la sonde pH.

#define RELAY_PIN      26
// Broche qui commande le relais du ventilateur.

#define MIST_RELAY_PIN 27
// Broche qui commande le relais du brumisateur.

// ====== SD (HM-704, VSPI: SCK=18, MOSI=23, MISO=19) ======
#define SD_CS_PIN      5
// Broche CS (Chip Select) du module carte SD.

#define CSV_PATH       "/data.csv"
// Nom et emplacement du fichier CSV sur la carte SD.

// ====== Relais (logique) ======
const bool RELAY_ACTIVE_LOW      = false;
// false = relais activé quand la sortie est HIGH.
// true = relais activé quand la sortie est LOW.

const bool MIST_RELAY_ACTIVE_LOW = false;
// Même principe pour le relais du brumisateur.

#define RELAY_ON_LEVEL        (RELAY_ACTIVE_LOW ? LOW  : HIGH)
// Niveau logique à envoyer pour allumer le ventilateur.

#define RELAY_OFF_LEVEL       (RELAY_ACTIVE_LOW ? HIGH : LOW)
```

```
// Niveau logique à envoyer pour éteindre le ventilateur.

#define MIST_RELAY_ON_LEVEL   (MIST_RELAY_ACTIVE_LOW ? LOW  : HIGH)
// Niveau logique à envoyer pour allumer le brumisateur.

#define MIST_RELAY_OFF_LEVEL  (MIST_RELAY_ACTIVE_LOW ? HIGH : LOW)
// Niveau logique à envoyer pour éteindre le brumisateur.

// ====== Seuils CO2 (ventilo) ======
uint16_t CO2_ON_PPM  = 1000;
// Si le CO2 atteint ou dépasse 1000 ppm, on allume le ventilateur.

uint16_t CO2_OFF_PPM = 900;
// Si le CO2 redescend à 900 ppm ou moins, on éteint le ventilateur.
// Cela crée une hystérésis pour éviter des ON/OFF trop fréquents.

// ====== pH : calibration (affichage) ======
const float ADC_VREF    = 3.30f;
// Tension de référence supposée pour convertir une valeur ADC en volts.

const float PH_VOLT_PH7 = 2.50f;
// Tension mesurée quand la sonde est dans une solution tampon pH 7.

const float PH_VOLT_PH4 = 3.00f;
// Tension mesurée quand la sonde est dans une solution tampon pH 4.

// ====== Humidité air (brumisateur) ======
float HUM_ON_PCT   = 70.0f;
// Si l’humidité de l’air est inférieure ou égale à 70 %, on peut activer
le brumisateur.

float HUM_OFF_PCT  = 80.0f;
// Si l’humidité remonte à 80 % ou plus, on coupe le brumisateur.

const uint32_t MIST_MIN_ON_MS  = 30UL * 1000;
// Durée minimale pendant laquelle le brumisateur doit rester allumé.

const uint32_t MIST_MIN_OFF_MS = 30UL * 1000;
// Durée minimale pendant laquelle le brumisateur doit rester éteint.

// ====== Sonde sol (calibration) ======
#define SOIL_DRY 0
// Valeur brute correspondant à un sol sec.

#define SOIL_WET 2900
```

```
// Valeur brute correspondant à un sol très humide.
// Cette valeur doit être ajustée expérimentalement.

// ====== Cadence ======
const uint32_t INTERVAL_MS = 5000;
// La boucle principale complète s’exécute toutes les 5 secondes.

// ====== Objets ======
Adafruit_BME280 bme;
// Création de l’objet bme pour manipuler le capteur BME280.

Adafruit_SGP30  sgp;
// Création de l’objet sgp pour manipuler le capteur SGP30.

WebServer server(80);
// Création d’un serveur web HTTP sur le port 80.

// ====== États / mesures ======
float    T = NAN, H = NAN, P = NAN;
// Variables pour stocker la température, l’humidité et la pression.
// NAN signifie "Not A Number", utile quand on n’a pas encore de mesure
valide.

uint16_t ECO2 = 0,  TVOC = 0;
// Variables pour stocker le eCO2 et le TVOC.

uint16_t SOIL_RAW = 0;
// Valeur brute lue sur la sonde du sol.

float SOIL_PCT = NAN;
// Pourcentage estimé d’humidité du sol.

uint16_t PH_RAW   = 0;
// Valeur brute lue pour le pH.

float PH_V     = NAN;
// Tension calculée à partir de la valeur brute du capteur pH.

float PH = NAN;
// Valeur finale du pH calculée.

bool     fanOn  = false;
// État logique du ventilateur : false = OFF, true = ON.

bool     mistOn = false;
```

```
// État logique du brumisateur.

uint32_t lastMs = 0;
// Mémorise le dernier instant où la boucle périodique a été exécutée.

uint32_t mistLastChangeMs = 0;
// Mémorise le dernier changement d’état du brumisateur.

bool     sdOK = false;
// Indique si la carte SD a bien été initialisée.

// ====== ADC ======
constexpr adc_attenuation_t ATTN_11DB = (adc_attenuation_t)3;
// Définit l’atténuation ADC à 11 dB.
// Cela permet de mesurer une tension plus élevée sur les entrées
analogiques.

// ====== Helpers ======
static inline uint16_t readAnalogAveraged(uint8_t pin, int samples=8, int
ms_between=5){
// Fonction utilitaire qui lit une entrée analogique plusieurs fois puis
calcule la moyenne.
// pin = broche à lire.
// samples = nombre de lectures.
// ms_between = délai entre deux lectures.

  uint32_t acc=0;
// Variable d’accumulation pour additionner les lectures.

  for(int i=0;i<samples;i++){
// Boucle qui effectue plusieurs lectures.

    acc += analogRead(pin);
// Ajoute la lecture analogique à la somme.

    delay(ms_between);
// Petite pause entre les lectures pour stabiliser la mesure.
  }

  return (uint16_t)(acc/(uint32_t)samples);
// Retourne la moyenne des lectures.
}

static inline float soilRawToPercent(uint16_t raw){
// Convertit la valeur brute de la sonde sol en pourcentage d’humidité.
```

```
  const uint16_t lo = min(SOIL_DRY, SOIL_WET), hi = max(SOIL_DRY,
SOIL_WET);
// Détermine la plus petite et la plus grande valeur de calibration.

  raw = constrain(raw, lo, hi);
// Force la valeur brute à rester dans l'intervalle [lo ; hi].

  float pct = (float)(raw-lo)*100.0f/(float)(hi-lo);
// Transforme la valeur brute en pourcentage entre 0 et 100.

  return (SOIL_WET>SOIL_DRY) ? pct : (100.0f-pct);
// Si la calibration est croissante, retourne pct.
// Sinon inverse le résultat.
}

static inline uint32_t absoluteHumidityForSGP30(float temperatureC, float
relHumidityPct){
// Calcule l'humidité absolue à partir de la température et de l'humidité
relative.
// Le SGP30 utilise cette information pour améliorer la précision de ses
mesures.

  float T=temperatureC, RH=relHumidityPct;
// Copie locale des paramètres pour rendre la formule plus lisible.

  float abs_h = 216.7f * ((RH/100.0f) * 6.112f *
expf((17.62f*T)/(243.12f+T))) / (273.15f+T);
// Formule de type Magnus pour calculer l'humidité absolue en g/m³.

  return (uint32_t)(1000.0f * abs_h);
// Conversion en mg/m³, format attendu par le SGP30.
}

static inline float adcRawToVolt(uint16_t raw){
// Convertit une valeur ADC brute en tension.

  return (raw*ADC_VREF)/4095.0f;
// 4095 = valeur max sur 12 bits.
}

static inline float phFromVoltage(float v){
// Convertit une tension en valeur de pH grâce à une calibration 2 points.

  if(PH_VOLT_PH7==PH_VOLT_PH4) return NAN;
```

```
// Évite une division par zéro si les deux tensions de calibration sont
identiques.

  const float slope=(7.00f-4.00f)/(PH_VOLT_PH7-PH_VOLT_PH4);
// Calcule la pente de la droite reliant les points (Vph7,7) et (Vph4,4).

  const float inter=7.00f - slope*PH_VOLT_PH7;
// Calcule l'ordonnée à l'origine de cette droite.

  return slope*v + inter;
// Applique l'équation de la droite pour obtenir le pH.
}

// ====== Relais ======
void safeInitRelay(int pin, bool activeLow){
// Initialise un relais de manière sécurisée pour éviter qu'il ne s'active
au démarrage.

  if(activeLow){
// Cas d'un relais actif à l'état bas.

    pinMode(pin, INPUT_PULLUP);
// Met temporairement la broche en entrée avec résistance de tirage au +.

    digitalWrite(pin, HIGH);
// Prépare un état OFF logique pour ce type de relais.
  }
  else{
// Cas d'un relais actif à l'état haut.

    pinMode(pin, INPUT_PULLDOWN);
// Met temporairement la broche en entrée avec résistance de tirage au -.

    digitalWrite(pin, LOW);
// Prépare un état OFF logique.
  }

  pinMode(pin, OUTPUT);
// Passe ensuite la broche en sortie.
}

static inline void writeFanPin(bool on){
// Fonction qui écrit physiquement l'état voulu sur la broche du
ventilateur.
```

```
  digitalWrite(RELAY_PIN, on ? RELAY_ON_LEVEL : RELAY_OFF_LEVEL);
// Si on=true, met le niveau logique d’allumage, sinon celui d’arrêt.
}

static inline void writeMistPin(bool on){
// Fonction qui écrit physiquement l’état voulu sur la broche du
brumisateur.

  digitalWrite(MIST_RELAY_PIN, on ? MIST_RELAY_ON_LEVEL :
MIST_RELAY_OFF_LEVEL);
// Même principe que pour le ventilateur.
}

static inline void setFan(bool on){
// Met à jour l’état logiciel du ventilateur puis l’état matériel.

  fanOn=on;
// Enregistre le nouvel état dans la variable.

  writeFanPin(on);
// Applique l’état sur la sortie.
}

static inline void setMist(bool on){
// Met à jour l’état logiciel du brumisateur puis l’état matériel.

  mistOn=on;
// Enregistre le nouvel état.

  writeMistPin(on);
// Applique l’état sur la broche.

  mistLastChangeMs = millis();
// Mémorise l’instant exact du changement d’état.
}

// ====== CSV (Excel FR) ======
#define CSV_SEP ';'
// Le séparateur des colonnes sera le point-virgule, compatible avec Excel
en français.

static inline void csvSep(File &f){
// Fonction pratique pour écrire le séparateur CSV dans le fichier.

  f.print(CSV_SEP);
```

```
// Écrit ';'
}

static inline void csvPrintFloat(File &f, float v, uint8_t digits){
// Écrit un nombre flottant dans le fichier CSV avec un certain nombre de
décimales.

  String s = String((double)v, (int)digits);
// Convertit le float en texte.

  for (size_t i = 0; i < s.length(); ++i)
    if (s[i] == '.') s.setCharAt(i, ',');
// Remplace le point décimal par une virgule pour le format français.

  f.print(s);
// Écrit la chaîne dans le fichier.
}

bool csvExists(){
// Vérifie si le fichier CSV existe déjà sur la carte SD.

  File f = SD.open(CSV_PATH, FILE_READ);
// Tente d’ouvrir le fichier en lecture.

  if(!f) return false;
// Si l’ouverture échoue, le fichier n’existe pas.

  f.close();
// Ferme le fichier si ouverture réussie.

  return true;
// Retourne vrai si le fichier existe.
}

bool writeCsvHeader(){
// Écrit la ligne d’en-tête du fichier CSV.

  File f = SD.open(CSV_PATH, FILE_WRITE);
// Ouvre le fichier en écriture.

  if(!f) return false;
// Si échec, retourne false.

  f.print("uptime_s"); csvSep(f); f.print("T_C");      csvSep(f);
f.print("H_pct");    csvSep(f);
```

```
  f.print("P_hPa");    csvSep(f); f.print("eCO2_ppm"); csvSep(f);
f.print("TVOC_ppb"); csvSep(f);
  f.print("soil_raw"); csvSep(f); f.print("soil_pct"); csvSep(f);
f.print("PH_RAW");   csvSep(f);
  f.print("PH_V");     csvSep(f); f.print("PH");       csvSep(f);
f.print("FAN");      csvSep(f);
  f.println("MIST");
// Écrit le nom de chaque colonne du tableau.

  f.close();
// Ferme le fichier.

  return true;
// Indique que l’écriture s’est bien passée.
}

void appendCsvRow(){
// Ajoute une ligne de mesures dans le fichier CSV.

  File f = SD.open(CSV_PATH, FILE_APPEND);
// Ouvre le fichier en mode ajout.

  if(!f){
    Serial.println("SD: echec ouverture append");
    return;
  }
// Si impossible d’ouvrir, affiche une erreur et sort.

  if(f.size()==0){
    writeCsvHeader();
  }
// Si le fichier est vide, écrit d’abord l’en-tête.

  f.print(millis()/1000);                  csvSep(f);
// Temps de fonctionnement en secondes.

  csvPrintFloat(f, T, 1);                  csvSep(f);
// Température avec 1 décimale.

  csvPrintFloat(f, H, 1);                  csvSep(f);
// Humidité avec 1 décimale.

  csvPrintFloat(f, P, 1);                  csvSep(f);
// Pression avec 1 décimale.
```

```
  f.print(ECO2);                              csvSep(f);
// eCO2 en ppm.

  f.print(TVOC);                              csvSep(f);
// TVOC en ppb.

  f.print(SOIL_RAW);                          csvSep(f);
// Valeur brute de la sonde sol.

  csvPrintFloat(f, SOIL_PCT, 1);              csvSep(f);
// Pourcentage d’humidité du sol.

  f.print(PH_RAW);                            csvSep(f);
// Valeur brute pH.

  csvPrintFloat(f, PH_V, 3);                  csvSep(f);
// Tension de la sonde pH avec 3 décimales.

  csvPrintFloat(f, PH, 2);                    csvSep(f);
// pH estimé avec 2 décimales.

  f.print(fanOn ? 1 : 0);                     csvSep(f);
// État du ventilateur : 1 = ON, 0 = OFF.

  f.println(mistOn ? 1 : 0);
// État du brumisateur : 1 = ON, 0 = OFF.

  f.close();
// Ferme le fichier après écriture.
}

// ====== Web ======
void handleRoot(){
// Fonction appelée lorsqu’on ouvre la page principale "/".

  String html="<!DOCTYPE html><html><head><meta charset='utf-8'>";
// Débute la construction de la page HTML.

  html+="<meta name='viewport' content='width=device-width,initial-
scale=1'>";
// Rend la page adaptée aux téléphones.

  html+="<meta http-equiv='refresh' content='10'><title>Station
ESP32</title></head><body>";
// Recharge automatiquement la page toutes les 10 secondes.
```

```
  html+="<h1>Mesures (5 s)</h1>";
// Titre principal de la page.

  html+="<p><b>Temp&eacute;rature:</b> "+String(T,1)+" &deg;C</p>";
// Affiche la température.

  html+="<p><b>Humidit&eacute; air:</b> "+String(H,1)+" %</p>";
// Affiche l’humidité relative de l’air.

  html+="<p><b>Pression:</b> "+String(P,1)+" hPa</p><hr>";
// Affiche la pression atmosphérique.

  html+="<p><b>eCO2:</b> "+String(ECO2)+" ppm</p>";
// Affiche le eCO2.

  html+="<p><b>TVOC:</b> "+String(TVOC)+" ppb</p>";
// Affiche le TVOC.

  html+="<p>Seuils CO2: ON &ge; "+String(CO2_ON_PPM)+" ppm, OFF &le;
"+String(CO2_OFF_PPM)+" ppm</p><hr>";
// Rappelle les seuils utilisés pour le ventilateur.

  html+="<h2>Humidit&eacute; du sol</h2>";
// Titre de la section sol.

  html+="<p><b>Brut:</b> "+String(SOIL_RAW)+" / 4095</p>";
// Affiche la valeur brute de la sonde sol.

  html+="<p><b>Estimation:</b> "+String(SOIL_PCT,1)+" %</p><hr>";
// Affiche le pourcentage estimé d’humidité du sol.

  html+="<h2>pH</h2>";
// Titre de la section pH.

  html+="<p><b>RAW:</b> "+String(PH_RAW)+" / 4095</p>";
// Affiche la valeur brute pH.

  html+="<p><b>Tension:</b> "+String(PH_V,3)+" V</p>";
// Affiche la tension calculée.

  html+="<p><b>pH estim&eacute;:</b> "+String(PH,2)+"</p><hr>";
// Affiche le pH estimé.

  html+="<h2>Ventilation</h2>";
```

```
// Titre de la section ventilateur.

  html+="<p><b>&Eacute;tat:</b> "+String(fanOn?"ON":"OFF")+"</p>";
// Affiche l'état actuel du ventilateur.

  html+="<p><a href='/fan?set=on'><button>Forcer ON</button></a> ";
// Bouton pour allumer le ventilateur.

  html+="<a href='/fan?set=off'><button>Forcer OFF</button></a></p><hr>";
// Bouton pour éteindre le ventilateur.

  html+="<h2>Brumisateur</h2>";
// Titre de la section brumisateur.

  html+="<p><b>&Eacute;tat:</b> "+String(mistOn?"ON":"OFF")+"</p>";
// Affiche l'état du brumisateur.

  html+="<p>R&egrave;gle: ON si H &le; "+String(HUM_ON_PCT,1)+" %, OFF si
H &ge; "+String(HUM_OFF_PCT,1)+" %</p>";
// Rappelle les seuils d'humidité.

  html+="<p><a href='/mist?set=on'><button>Forcer ON</button></a> ";
// Bouton pour activer le brumisateur.

  html+="<a href='/mist?set=off'><button>Forcer OFF</button></a></p>";
// Bouton pour désactiver le brumisateur.

  server.send(200,"text/html",html);
// Envoie la page HTML au navigateur avec le code HTTP 200 = succès.
}

void handleFan(){
// Fonction appelée quand on visite /fan?set=on ou /fan?set=off

  String set = server.hasArg("set") ? server.arg("set") : "";
// Récupère l'argument "set" de l'URL.

  if(set=="on"){
    setFan(true);
    Serial.println("[WEB] FAN → ON");
  }
// Si l'utilisateur demande "on", le ventilateur est activé.

  if(set=="off"){
    setFan(false);
```

```
    Serial.println("[WEB] FAN → OFF");
  }
// Si l’utilisateur demande "off", le ventilateur est désactivé.

  server.sendHeader("Location","/");
// Prépare une redirection vers la page principale.

  server.send(302,"text/plain","");
// Envoie la redirection HTTP 302.
}

void handleMist(){
// Fonction appelée quand on visite /mist?set=on ou /mist?set=off

  String set = server.hasArg("set") ? server.arg("set") : "";
// Récupère l’argument "set".

  if(set=="on"){
    setMist(true);
    Serial.println("[WEB] MIST → ON");
  }
// Active le brumisateur si demandé.

  if(set=="off"){
    setMist(false);
    Serial.println("[WEB] MIST → OFF");
  }
// Désactive le brumisateur si demandé.

  server.sendHeader("Location","/");
// Redirection vers la page principale.

  server.send(302,"text/plain","");
// Envoie la réponse HTTP de redirection.
}

// ====== Setup ======
void setup(){
// Fonction exécutée une seule fois au démarrage de l’ESP32.

  Serial.begin(115200);
// Démarre la communication série pour afficher les messages de debug.

  delay(200);
// Petite pause pour laisser le port série se stabiliser.
```

```
  // Relais OFF garanti au boot
  safeInitRelay(RELAY_PIN,      RELAY_ACTIVE_LOW);
// Initialise le relais ventilateur proprement.

  safeInitRelay(MIST_RELAY_PIN, MIST_RELAY_ACTIVE_LOW);
// Initialise le relais brumisateur proprement.

  setFan(false);
// Force le ventilateur à l'arrêt au démarrage.

  setMist(false);
// Force le brumisateur à l'arrêt au démarrage.

  // Capteurs
  if(!bme.begin(0x76)){
    Serial.println("BME280 absent");
    while(1){}
  }
// Tente d'initialiser le BME280 à l'adresse I2C 0x76.
// Si échec, affiche une erreur et bloque le programme.

  if(!sgp.begin()){
    Serial.println("SGP30 absent");
    while(1){}
  }
// Tente d'initialiser le SGP30.
// Si échec, le programme s'arrête.

  if(!sgp.IAQinit()){
    Serial.println("SGP30 IAQinit KO");
    while(1){}
  }
// Lance l'initialisation interne du SGP30 pour les mesures qualité d'air.

  // ADC
  analogReadResolution(12);
// Fixe la résolution ADC à 12 bits : valeurs entre 0 et 4095.

  analogSetPinAttenuation(SOIL_PIN, ATTN_11DB);
// Applique une atténuation de 11 dB sur la pin de la sonde sol.

  analogSetPinAttenuation(PH_PIN,   ATTN_11DB);
// Applique aussi cette atténuation à la pin du pH.
```

```
  // Wi Fi + Web
  WiFi.softAP(SSID,PASSWORD);
// Crée un point d’accès Wi-Fi nommé CHAMPY.

  server.on("/",handleRoot);
// Associe la page principale "/" à la fonction handleRoot().

  server.on("/fan",handleFan);
// Associe la route /fan à la fonction handleFan().

  server.on("/mist",handleMist);
// Associe la route /mist à la fonction handleMist().

  server.begin();
// Démarre le serveur web.

  // SD
  pinMode(SD_CS_PIN, OUTPUT);
// Met la broche CS de la carte SD en sortie.

  sdOK = SD.begin(SD_CS_PIN);
// Tente d’initialiser la carte SD.

  if(!sdOK){
    Serial.println("SD: init KO (verifie cablage/alim/CS)");
  }
// Si la carte SD ne démarre pas, affiche une erreur.

  else {
    if(!csvExists()){
      if(writeCsvHeader()) Serial.println("SD: entete CSV cree");
      else Serial.println("SD: echec ecriture entete");
    }
  }
// Si la SD fonctionne et que le fichier CSV n’existe pas encore,
// on crée l’en-tête du fichier.

  Serial.println("Pret. Wi-Fi: CHAMPY — URL: http://192.168.4.1/");
// Message final de démarrage.
}

// ====== Loop ======
void loop(){
// Fonction répétée en continu par l’ESP32.
```

```
  server.handleClient();
// Traite les éventuelles requêtes HTTP reçues depuis le navigateur.

  uint32_t now=millis();
// Récupère le temps écoulé depuis le démarrage en millisecondes.

  if(now-lastMs<INTERVAL_MS) return;
// Si 5 secondes ne sont pas encore passées depuis le dernier cycle, on
sort.

  lastMs=now;
// Met à jour le temps du dernier cycle.

  // BME280
  T=bme.readTemperature();
// Lit la température.

  H=bme.readHumidity();
// Lit l'humidité relative.

  P=bme.readPressure()/100.0f;
// Lit la pression en pascals puis la convertit en hPa.

  // SGP30
  sgp.setHumidity(absoluteHumidityForSGP30(T,H));
// Envoie au SGP30 l'humidité absolue calculée à partir du BME280.
// Cela améliore la qualité de ses mesures.

  if(sgp.IAQmeasure()){
    ECO2=sgp.eCO2;
    TVOC=sgp.TVOC;
  }
// Si la mesure qualité d'air réussit,
// on récupère eCO2 et TVOC.

  else {
    Serial.println("SGP30 lecture KO");
  }
// Sinon on affiche un message d'erreur.

  // Sol
  SOIL_RAW = readAnalogAveraged(SOIL_PIN);
// Lit la sonde de sol plusieurs fois et calcule la moyenne.

  SOIL_PCT = soilRawToPercent(SOIL_RAW);
```

```
// Convertit la valeur brute en pourcentage.

  // pH (info)
  PH_RAW = readAnalogAveraged(PH_PIN);
// Lit la sonde pH plusieurs fois et calcule la moyenne.

  PH_V   = adcRawToVolt(PH_RAW);
// Convertit la valeur brute en volts.

  PH     = phFromVoltage(PH_V);
// Convertit la tension en pH estimé.

  // Ventilo (CO2 uniquement) — déclenché par eCO2 avec hystérésis
  bool needOn = (ECO2 >= CO2_ON_PPM);
// Vrai si le CO2 dépasse le seuil haut.

  bool canOff = (ECO2 <= CO2_OFF_PPM);
// Vrai si le CO2 redescend sous le seuil bas.

  bool targetFan = fanOn;
// On part de l'état actuel du ventilateur.

  if(needOn)      targetFan = true;
// Si besoin d'aérer, on veut ON.

  else if(canOff) targetFan = false;
// Si le CO2 est redescendu, on veut OFF.

  if(targetFan != fanOn){
    setFan(targetFan);
  }
// Si l'état souhaité diffère de l'état actuel, on change l'état.

  else {
    writeFanPin(targetFan);
  }
// Sinon on réécrit l'état actuel sur la broche pour sécuriser le
matériel.

  // Brumisateur — hystérésis + anti-cliquetis (min ON/OFF)
  bool mistNeedOn = (!isnan(H) && H <= HUM_ON_PCT);
// Si l'humidité est valide et faible, on a besoin du brumisateur.

  bool mistCanOff = (!isnan(H) && H >= HUM_OFF_PCT);
// Si l'humidité est valide et assez haute, on peut l'arrêter.
```

```
  uint32_t dwell = now - mistLastChangeMs;
// Calcule depuis combien de temps le brumisateur est dans son état
actuel.

  if(!mistOn && mistNeedOn && dwell>=MIST_MIN_OFF_MS){
    setMist(true);
    Serial.printf("MIST ON (H=%.1f%%)\n",H);
  }
// Si le brumisateur est OFF, qu'il faut l'allumer,
// et qu'il a été assez longtemps éteint, alors on l'allume.

  else if(mistOn && mistCanOff && dwell>=MIST_MIN_ON_MS){
    setMist(false);
    Serial.printf("MIST OFF (H=%.1f%%)\n",H);
  }
// Si le brumisateur est ON, qu'on peut l'éteindre,
// et qu'il a été assez longtemps allumé, alors on l'éteint.

  else {
    writeMistPin(mistOn);
  }
// Sinon on maintient son état matériel actuel.

  // LOG SD
  appendCsvRow();
// Enregistre les mesures et les états dans le fichier CSV.

  // DEBUG
  Serial.printf("SD=%s | CO2=%u (on>=%u off<=%u) FAN=%s | H=%.1f MIST=%s |
pH=%.2f | soil=%u(%.1f%%)\n",
    sdOK?"OK":"KO", ECO2, CO2_ON_PPM, CO2_OFF_PPM, fanOn?"ON":"OFF",
    H, mistOn?"ON":"OFF", PH, SOIL_RAW, SOIL_PCT);
// Affiche un résumé complet dans le moniteur série.
}
```