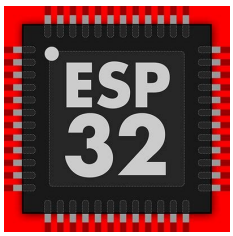
CHARLES CARNUS
LYCÉE TECHNIQUE
ENSEIGNEMENT SUPÉRIEUR



ESP32

TP dans l'IDE Arduino

Prof : Kamal Boudjelaba

30 janvier 2023

B T S S N

LTP Charles Carnus
Avenue de Saint Pierre - Bourran - 12000 RODEZ, France

IR–EC

Table des matières

1. Introduction

1.1 Installation de la librairie ESP32 dans l'IDE Arduino

La procédure d'installation est disponible à partir de ce lien ou copier le lien suivant : https://docs.espressif.com/projects/arduino-esp32/en/latest/installing.html

1.2 Les broches de l'ESP32

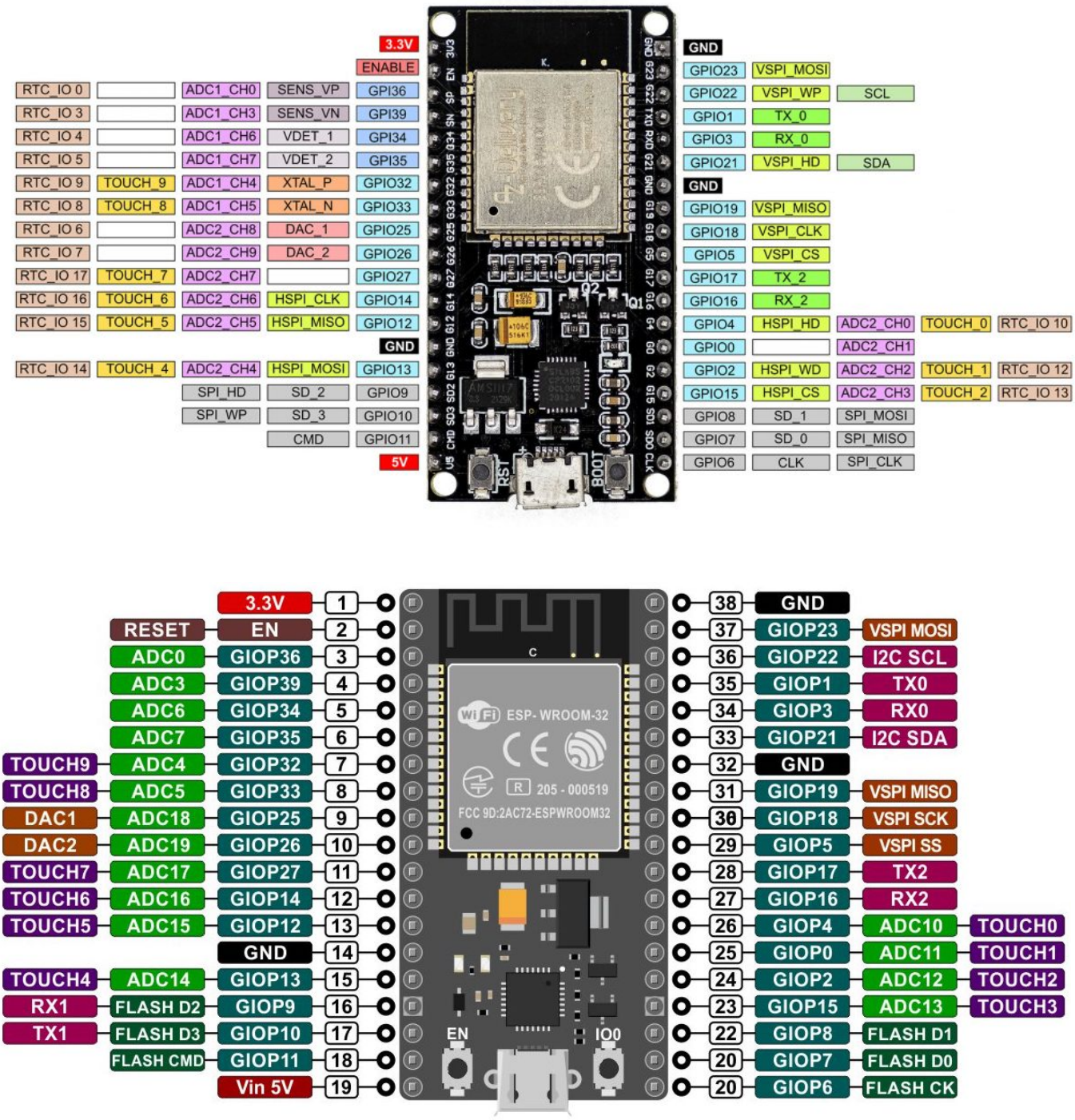


Figure 1. Les pins de l'ESP32

Si la carte ESP32 possède les pins `GPIO6, GPIO7, GPIO8, GPIO9, GPIO10, GPIO11`, il ne faut surtout pas les utiliser car ils sont reliés à la mémoire flash de l'ESP32 : si on les utilise, l'ESP32 ne fonctionnera plus. C'est pour cette raison que ces pins ne sont pas présents sur certains modèles de cartes ESP32.

GPIO	Noms possibles
6	SCK/CLK
7	SDO/SD0
8	SDI/SD1
9	SHD/SD2
10	SWP/SD3
11	CSC/CMD

Table 1. Les broches reliées à la mémoire Flash de l'ESP32

GPIO	INPUT	OUTPUT	Commentaires
0	OUI (Pullup interne)	OUI	Doit être à 0V pendant le FLASH
1 (TX0)	NON	OUI	Communication UART avec le PC
2	OUI (Pulldown interne)	OUI	Doit être à 0V pendant le FLASH
3 (RX0)	OUI	NON	Communication UART avec le PC
4 (D4)	OUI	OUI	
5 (D5)	OUI	OUI	
6	NON	NON	Connecté au flash SPI intégré
7	NON	NON	Connecté au flash SPI intégré
8	NON	NON	Connecté au flash SPI intégré
9	NON	NON	Connecté au flash SPI intégré
10	NON	NON	Connecté au flash SPI intégré
11	NON	NON	Connecté au flash SPI intégré
12 (MTDI) (D12)	OUI (Pulldown interne)	OUI	Doit être à 0V pendant le BOOT
13 (D13)	OUI	OUI	
14 (D14)	OUI	OUI	
15 (MTDO) (D15)	OUI (Pullup interne)	OUI	Startup log si à 3.3V
16 (RX2)	OUI	OUI	Pas dispo sur les WROVER
17 (TX2)	OUI	OUI	Pas dispo sur les WROVER
18 (D18)	OUI	OUI	
19 (D19)	OUI	OUI	
21 (D21)	OUI	OUI	
22 (D22)	OUI	OUI	
23 (D23)	OUI	OUI	
25 (D25)	OUI	OUI	
26 (D26)	OUI	OUI	
27 (D27)	OUI	OUI	
32 (D32)	OUI	OUI	
33 (D33)	OUI	OUI	
34 (D34)	OUI	NON	Pas de pullup/pulldown interne
35 (D35)	OUI	NON	Pas de pullup/pulldown interne
36 (VP)	OUI	NON	Pas de pullup/pulldown interne
39 (VN)	OUI	NON	Pas de pullup/pulldown interne
EN	NON	NON	Relié au bouton EN (ESP32 Reset)

Table 2. I/O possibles pour les pins de l'ESP32

2. TP1 : Afficher l'adresse MAC de l'ESP32

```
1
2 #include "WiFi.h" // ou #include <WiFi.h>
3
4 void setup() {
5   Serial.begin(115200);
6   WiFi.mode(WIFI_MODE_STA);
7   // On peut ajouter un délai : delay(2000);
8   Serial.println(WiFi.macAddress());
9 }
10
11 void loop() {
12 }
```

3. TP2 : Réseaux Wi-Fi avec l'ESP32

La librairie WiFi.h (installée automatiquement) permet d'utiliser facilement les fonctionnalités Wi-Fi de la carte ESP32. L'ESP32 possède 2 modes WiFi possibles :

STATION (WIFI_STA) : Le mode Station (STA) est utilisé pour connecter le module ESP32 à un point d'accès Wi-Fi. L'ESP32 se comporte comme un ordinateur qui serait connecté au réseau du lycée (ou au wifi Carnus) et peut accéder à Internet. L'ESP32 peut se comporter en tant que client, c'est-à-dire faire des requêtes aux autres appareils connectés sur le réseau ou en tant que serveur, c'est-à-dire que d'autres appareils connectés sur le réseau vont envoyer des requêtes à l'ESP32. Dans les 2 cas, l'ESP32 peut accéder à Internet.

AP (Access Point) (WIFI_AP) : En mode Access Point, l'ESP32 se comporte comme un réseau Wi-Fi (un peu comme le partage de connexion avec le smartphone) : d'autres appareils peuvent s'y connecter dessus. Dans ce mode, l'ESP32 n'est relié à aucun autre réseau et n'est donc pas connecté à Internet. Ce mode est plus gourmand en calcul et en énergie (la carte ESP32 va chauffer) puisque l'ESP32 doit simuler un routeur Wi-Fi complet (Soft AP). La latence et le débit seront moins bons qu'avec le réseau du lycée (ou une box classique).

Remarque 3.1 :

L'ESP32 est par défaut en mode STATION.

Pour le choix du mode :

— En général, on utilise le mode STATION. On pourra accéder à internet pour récupérer des informations d'API, avoir un serveur "domotique" avec des capteurs ...

— On utilise en général le mode AP provisoirement pour rentrer les paramètres de connexion du réseau Wi-Fi (SSID + MDP). On peut également l'utiliser pour avoir un réseau séparé du réseau du lycée (ou de la salle) mais il est non relié à Internet.

3.1 Connexion à un réseau Wi-Fi

```
1
2 #include <WiFi.h>
3
4 const char* ssid = "NomDuRéseau";
5 const char* password = "MotDePasse";
6
7 void setup(){
8     Serial.begin(115200);
9     delay(1000);
10
11     WiFi.mode(WIFI_STA); //Optionnel
12     WiFi.begin(ssid, password);
13     Serial.println("\nConnexion");
14
15     while(WiFi.status() != WL_CONNECTED){
16         Serial.print(".");
17         delay(100);
18     }
19
20     Serial.println("\nConnecté au réseau Wi-Fi");
21     Serial.print("IP locale ESP32 : ");
22     Serial.println(WiFi.localIP());
23 }
24
25 void loop(){}
```

Explication du code :

— Inclure la librairie `WiFi.h`

— On rentre le nom du réseau et son mot de passe

— On met l'ESP32 en mode STATION avec la fonction `WiFi.mode(WIFI_STA);`

— L'ESP32 essaye de se connecter au réseau WiFi à l'aide de la fonction `WiFi.begin(ssid, password);`

— La connexion n'est pas instantanée. Il faut donc regarder régulièrement l'état de la connexion : tant que l'ESP32 n'est pas connecté au réseau, on reste bloqué dans la boucle while. On ajoute un petit délai pour éviter de regarder en permanence l'état.

— Une fois que l'on est connecté, on affiche l'adresse IP locale de l'ESP32 sur ce réseau.

3.2 Obtenir des informations sur le réseau Wi-Fi

On peut obtenir des informations sur le réseau une fois que l'on est connecté sur celui-ci :

— La puissance du signal WiFi (RSSI) avec la fonction `WiFi.RSSI()`

— L'adresse MAC du réseau WiFi avec `WiFi.BSSIDstr()` ou `WiFi.macAddress()`

— L'adresse IP locale de l'ESP32 attribuée par le serveur DHCP du réseau `WiFi WiFi.localIP()`

— L'adresse IP locale du réseau WiFi (passerelle) avec `WiFi.gatewayIP()`

— Le masque de sous-réseau avec `WiFi.subnetMask()`

Le code ci-dessous affiche toutes ces informations :

```
1
2  #include <WiFi.h>
3
4  const char* ssid = "NomDuRéseau";
5  const char* password = "MotDePasse";
6
7  void get_network_info(){
8      if(WiFi.status() == WL_CONNECTED) {
9          Serial.print("[*] Information sur le réseau : ");
10         Serial.println(ssid);
11
12         Serial.println("[+] BSSID : " + WiFi.BSSIDstr());
13         Serial.print("[+] Gateway IP : ");
14         Serial.println(WiFi.gatewayIP());
15         Serial.print("[+] Subnet Mask : ");
16         Serial.println(WiFi.subnetMask());
17         Serial.println((String)"[+] RSSI : " + WiFi.RSSI() + " dB");
18         Serial.print("[+] ESP32 IP : ");
19         Serial.println(WiFi.localIP());
20     }
21 }
22
23 void setup(){
24     Serial.begin(115200);
25     delay(1000);
26
27     WiFi.begin(ssid, password);
28     Serial.println("\nConnexion");
29
30     while(WiFi.status() != WL_CONNECTED){
31         Serial.print(".");
32         delay(100);
33     }
34
35     Serial.println("\nConnecté au réseau Wi-Fi");
36     get_network_info();
37 }
38
39 void loop(){}
```

3.3 Débugger les problèmes de connexion

Regarder le statut de la connexion

On peut connaître le statut de la connexion WiFi avec la fonction WiFi.status(). Cette fonction renvoie un entier en fonction de l'état actuel de la connexion.
Les statuts possibles sont :

- `WL_IDLE_STATUS` : C'est le statut par défaut avant d'essayer de se connecter à un réseau.
- `WL_SCAN_COMPLETED` : Le scan des réseaux WiFi est terminé.
- `WL_NO_SSID_AVAIL` : L'ESP32 n'arrive pas à trouver le nom du réseau WiFi. Soit le réseau est trop loin de l'ESP32, soit le nom (SSID) du réseau est incorrect.
- `WL_CONNECT_FAILED` : L'ESP32 n'arrive pas à se connecter au réseau WiFi désigné.
- `WL_CONNECTION_LOST` : La connexion WiFi avec le réseau est perdue. Si cette erreur se répète c'est peut être un problème d'alimentation de l'ESP32.
- `WL_CONNECTED` : L'ESP32 est connecté au réseau WiFi.
- `WL_DISCONNECTED` : L'ESP32 est déconnecté du réseau WiFi.

```
1
2 #include <WiFi.h>
3
4 const char* ssid = "NomDuRéseau";
5 const char* password = "MotDePasse";
6
7 String get_wifi_status(int status){
8     switch(status){
9         case WL_IDLE_STATUS:
10        return "WL_IDLE_STATUS";
11        case WL_SCAN_COMPLETED:
12        return "WL_SCAN_COMPLETED";
13        case WL_NO_SSID_AVAIL:
14        return "WL_NO_SSID_AVAIL";
15        case WL_CONNECT_FAILED:
16        return "WL_CONNECT_FAILED";
17        case WL_CONNECTION_LOST:
18        return "WL_CONNECTION_LOST";
19        case WL_CONNECTED:
20        return "WL_CONNECTED";
21        case WL_DISCONNECTED:
22        return "WL_DISCONNECTED";
23    }
24 }
25
26 void setup(){
27     Serial.begin(115200);
28     delay(1000);
29     int status = WL_IDLE_STATUS;
30     Serial.println("\nConnexion");
31     Serial.println(get_wifi_status(status));
32     WiFi.begin(ssid, password);
33     while(status != WL_CONNECTED){
34         delay(500);
35         status = WiFi.status();
36         Serial.println(get_wifi_status(status));
37     }
38
39     Serial.println("\nConnecté au réseau Wi-Fi");
40     Serial.print("IP locale ESP32 : ");
41     Serial.println(WiFi.localIP());
42 }
43
44 void loop(){}
```

Redémarrer l'ESP32

Parfois, pour une raison inconnue, l'ESP32 peut ne pas arriver temporairement à se connecter au réseau WiFi. La meilleure solution est de dire qu'au bout de n secondes si l'ESP32 ne s'est toujours pas connecté au WiFi, on redémarre l'ESP32. Il suffit d'ajouter un timeout et d'utiliser la fonction `ESP.restart()` pour redémarrer l'ESP32 depuis le code.
Voici un exemple qui permet de redémarrer l'ESP32 au bout de 10 s s'il n'est toujours pas connecté au WiFi.

```
1
2 #include <WiFi.h>
3
4 #define CONNECTION_TIMEOUT 10
5
6 const char* ssid = "NomDuRéseau";
7 const char* password = "MotDePasse";
8
9 void setup(){
10     Serial.begin(115200);
11     delay(1000);
12
13     WiFi.mode(WIFI_STA); //Optional
14     WiFi.begin(ssid, password);
15     Serial.println("\nConnexion");
16     int timeout_counter = 0;
17
18     while(WiFi.status() != WL_CONNECTED){
19         Serial.print(".");
20         delay(200);
21         timeout_counter++;
22         if(timeout_counter >= CONNECTION_TIMEOUT*5){
23         ESP.restart();
24         }
25     }
26
27     Serial.println("\nConnecté au réseau Wi-Fi");
28     Serial.print("IP locale ESP32 : ");
29     Serial.println(WiFi.localIP());
30 }
31
32 void loop(){}
```

Attribuer une adresse IP fixe

L'adresse IP locale de l'ESP32 a été attribuée automatiquement par le serveur DHCP. L'inconvénient (ou avantage en fonction des cas) est que l'adresse IP est dynamique : elle peut changer. Cela peut devenir gênant si par exemple, dès que l'on redémarre l'ESP32 (ou que le bail du DHCP est expiré) l'adresse IP change, alors qu'on a un serveur web qui tourne sur l'ESP32. Il faudrait à chaque fois retrouver l'IP de l'ESP32. On peut pallier ce problème en fixant l'adresse IP de l'ESP32 sur le réseau. Il faut utiliser pour cela la fonction
`WiFi.config(ip, dns, gateway, subnet)`
Les paramètres à renseigner sont :

- IP : L'adresse IP que l'on souhaite attribuer.
- DNS : Service qui fait le lien entre une url et une IP. Par défaut, on utilise le serveur DNS du routeur ; donc on indique la même adresse que le routeur
- GATEWAY : C'est l'adresse IP du routeur
- SUBNET : Masque de sous-réseau

Dans cet exemple, en utilisant le partage WiFi d'un téléphone, l'adresse IP du routeur est 192.168.43.1 . On choisit d'avoir comme IP statique 192.168.43.42 pour l'ESP32.

```
#include <WiFi.h>

const char* ssid = "NomDuRéseau";
const char* password = "MotDePasse";

IPAddress ip(192, 168, 43, 42);
IPAddress dns(192, 168, 43, 1);
IPAddress gateway(192, 168, 43, 1);
IPAddress subnet(255, 255, 255, 0);

void setup(){
  Serial.begin(115200);
  delay(1000);

  WiFi.config(ip, gateway, subnet, dns);
  WiFi.begin(ssid, password);
  Serial.println("\nConnexion");

  while(WiFi.status() != WL_CONNECTED){
    Serial.print(".");
    delay(100);
    }

    Serial.println("\nConnecté au réseau Wi-Fi");
    Serial.print("IP locale ESP32 : ");
    Serial.println(WiFi.localIP());
}

void loop(){}
```

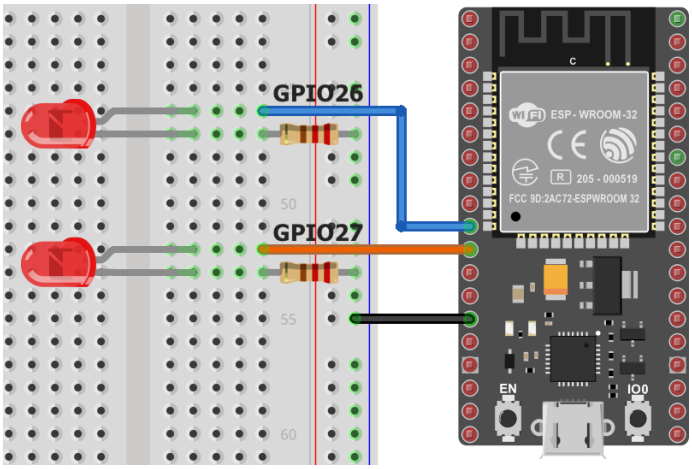
4. TP3 : Serveur Web

On veut créer un serveur Web autonome avec un ESP32 qui contrôle deux LEDs. Le serveur Web est accessible à l'aide d'un navigateur sur le réseau local.

— Le serveur Web doit contrôler deux LEDs connectées aux GPIO26 et GPIO27 de l'ESP32

— On peut accéder au serveur Web en tapant l'adresse IP locale de l'ESP32 sur un navigateur

— En cliquant sur les boutons du serveur Web, on peut modifier instantanément l'état de chaque LED



Début

```
#include <WiFi.h>

const char* ssid = "NomDuRéseau";
const char* password = "MotDePasse";

// Port 80
WiFiServer server(80);

// Variable pour sauvegarder la requête HTTP
String header;

String output26State = "off";
String output27State = "off";

const int output26 = 26;
const int output27 = 27;

unsigned long currentTime = millis();
unsigned long previousTime = 0;
const long timeoutTime = 2000;

void setup() {
  Serial.begin(115200);
  pinMode(output26, OUTPUT);
  pinMode(output27, OUTPUT);
  digitalWrite(output26, LOW);
  digitalWrite(output27, LOW);

  // Connexion au réseau Wi-Fi
  Serial.print("Connecté au réseau ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  // Affichage de l'adresse IP et démarrage du serveur web
  Serial.println("");
  Serial.println("Connecté en WiFi.");
  Serial.println("Adresse IP : ");
  Serial.println(WiFi.localIP());
  server.begin();
}

void loop() {
  WiFiClient client = server.available();  // Ecoute les clients entrants

  if (client) {  // Si un nouveau client se connecte,
    currentTime = millis();
    previousTime = currentTime;
    Serial.println("Nouveau Client.");
    String currentLine = "";
    while (client.connected() && currentTime - previousTime <= timeoutTime) {
      currentTime = millis();
      if (client.available()) {
        char c = client.read();
        Serial.write(c);
        header += c;
```

Suite

```
        //header += c;
        if (c == '\n') {
          if (currentLine.length() == 0) {
            client.println("HTTP/1.1 200 OK");
            client.println("Type de contenu : text/html");
            client.println("Connexion : fermée");
            client.println();

            if (header.indexOf("GET /26/on") >= 0) {
              Serial.println("GPIO 26 on");
              output26State = "on";
              digitalWrite(output26, HIGH);
            } else if (header.indexOf("GET /26/off") >= 0) {
              Serial.println("GPIO 26 off");
              output26State = "off";
              digitalWrite(output26, LOW);
            } else if (header.indexOf("GET /27/on") >= 0) {
              Serial.println("GPIO 27 on");
              output27State = "on";
              digitalWrite(output27, HIGH);
            } else if (header.indexOf("GET /27/off") >= 0) {
              Serial.println("GPIO 27 off");
              output27State = "off";
              digitalWrite(output27, LOW);
            }

            // Affichage de la page HTML
            client.println("<!DOCTYPE html><html>");
            client.println("<head><meta name=\"viewport\" content=\"width=device-width,
                  initial-scale=1\">");
            client.println("<link rel=\"icon\" href=\"data:,\">");
            // CSS : boutons on/off
            client.println("<style>html { font-family: Helvetica; display: inline-block;
                  margin: 0px auto; text-align: center;}");
            client.println(".button { background-color: #4CAF50; border: none;
                  color: white; padding: 16px 40px;");
            client.println("text-decoration: none; font-size: 30px; margin: 2px;
                  cursor: pointer;}");
            client.println(".button2 {background-color: #555555;}</style></head>");

            // En-tête page Web
            client.println("<body><h1>ESP32 Web Server</h1>");

            // Affichage de l'état actuel et les boutons ON/OFF pour GPIO 26
            client.println("<p>GPIO 26 - State " + output26State + "</p>");
            // Si output26State est éteint, il affiche le bouton ON
            if (output26State == "off") {
              client.println("<p><a href=\"/26/on\"><button class=\"button\">
                    ON</button></a></p>");
            } else {
              client.println("<p><a href=\"/26/off\"><button class=\"button button2\">
                    OFF</button></a></p>");
            }

            // Affichage de l'état actuel et les boutons ON/OFF pour GPIO 27
            client.println("<p>GPIO 27 - State " + output27State + "</p>");
            // Si output27State est éteint, il affiche le bouton ON
            if (output27State == "off") {
              client.println("<p><a href=\"/27/on\"><button class=\"button\">
                    ON</button></a></p>");
            } else {
              client.println("<p><a href=\"/27/off\"><button class=\"button button2\">
                    OFF</button></a></p>");
            }
            client.println("</body></html>");
```

Fin

```
            //client.println("</body></html>");

            client.println();
            break;
          } else {
            currentLine = "";
          }
        } else if (c != '\r') {
          currentLine += c;
        }
      }
    }
    // Effacer la variable d'en-tête
    header = "";
    // Fermer la connexion
    client.stop();
    Serial.println("Client déconnecté.");
    Serial.println("");
  }
}
```

— Après avoir téléversé le code, ouvrir le moniteur série sur un débit de 115200 bauds. L'ESP32 se connecte au Wi-Fi et affiche l'adresse IP de l'ESP sur le moniteur série. Copier cette adresse IP.

— Pour accéder au serveur Web, ouvrir un navigateur et coller l'adresse IP.

— Sur le moniteur série, on peut voir ce qui se passe en arrière-plan. L'ESP reçoit une requête HTTP d'un nouveau client (dans notre cas, le navigateur).

— On peut maintenant tester si le serveur Web fonctionne correctement. Cliquer sur les boutons pour contrôler les LEDs.
En même temps, on peut voir dans le moniteur série ce qui se passe en arrière-plan. Par exemple, lorsqu'on clique sur le bouton pour activer le GPIO 26, ESP32 reçoit une requête sur l'URL /26/on. Lorsque l'ESP32 reçoit cette demande, il allume la LED attachée au GPIO 26 et met à jour son état sur la page Web.

— Le bouton pour GPIO 27 fonctionne de manière similaire. Vérifier qu'il fonctionne correctement.

5. TP4 : Bluetooth (classique) ESP32

L'ESP32 prend en charge à la fois les normes Classic Bluetooth v4.2 et Bluetooth Low Energy (BLE).
Le Bluetooth est une technologie de communication sans fil fonctionnant dans la bande de fréquences ISM (industrielle, scientifique et médicale) de 2,4 GHz. La portée du Bluetooth est courte (jusqu'à 100 m).
Le SoC ESP32 intègre à la fois le contrôleur de liaison Bluetooth (ou gestionnaire de liaison) et la bande de base dans son silicium. Étant donné que le Wi-Fi et le Bluetooth fonctionnent à la même fréquence ISM de 2,4 GHz, la radio Wi-Fi et la radio Bluetooth partagent la même antenne dans ESP32. Si on regarde le brochage du SoC ESP32, il n'y a qu'une seule broche pour la connexion à l'antenne (LNA_IN). L'ESP32 prend en charge à la fois le Bluetooth classique (Classic BT) et le Bluetooth Low Energy (BLE) qui peuvent être configurés avec BLUEDROID Bluetooth Stack.
L'ESP32 Bluetooth prend en charge trois types d'interface de contrôleur hôte (HCI) : les interfaces UART, SPI et VHCI (Virtual HCI) (une seule peut être utilisée à la fois et UART est la valeur par défaut).

Le Bluetooth classique est la topologie de réseau point à point conçue pour une communication sans fil un à un entre un maître et un esclave. Même si plusieurs appareils esclaves peuvent être connectés à un seul maître, un seul esclave peut communiquer activement avec le maître. Les claviers et souris Bluetooth fonctionnent avec la technologie Bluetooth classique, on peut citer aussi le transfert de fichiers entre deux appareils (deux téléphones portables ou un ordinateur portable et un téléphone portable).

BLE ou Bluetooth Low Energy est conçu pour un fonctionnement à faible consommation d'énergie et développé avec les applications IoT comme cible principale. La spécification Bluetooth 4.0 a ajouté la fonctionnalité BLE et

est principalement utilisée dans les appareils fonctionnant sur batterie comme les montres, les appareils audio, les trackers de santé, les moniteurs de fitness et les balises de données.

La pile Bluetooth BLUEDROID communique avec le contrôleur Bluetooth via VHCI (Virtual Host Controller Interface) et fournit en même temps des API pour l'application utilisateur.
Cela signifie qu'après avoir reçu des données d'un appareil Bluetooth sans fil, le contrôleur Bluetooth de l'ESP32 transfère ces données au processeur de l'ESP32 via une communication série. De même, afin d'envoyer des données via Bluetooth, le processeur de l'ESP32 transmet les données au contrôleur Bluetooth à l'aide de l'interface série.

```
#include "BluetoothSerial.h"

/* Vérifie si les configurations Bluetooth sont activées dans le SDK */
/* Sinon, il faut recompiler le SDK */
#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth n'est pas activé! Lancer "make menuconfig" et l'activer
#endif

BluetoothSerial SerialBT;

void setup() {
  Serial.begin(115200);
  // Si aucun nom n'est donné, 'ESP32' par défaut est appliqué
  // Si on souhaite donner un autre nom à l'appareil Bluetooth ESP32, alors
  // on spécifie le nom comme argument SerialBT.begin("monESP32Bluetooth");
  SerialBT.begin();
  Serial.println("Bluetooth démarré ! Prêt à appairer (coupler) ...");
}

void loop() {
  if (Serial.available()) {
    SerialBT.write(Serial.read());
  }
  if (SerialBT.available()) {
    Serial.write(SerialBT.read());
  }
  delay(20);
}
```

— On utilise la bibliothèque « BluetoothSerial » dédiée pour transmettre et recevoir des données.

— On crée un objet de classe 'BluetoothSerial' et commence la communication en utilisant la fonction 'begin ()'.

— On initialise également la communication série normale avec un débit en bauds de 115200.

— Ensuite, dans la fonction loop, on lit les données de BluetoothSerial et les affiche sur le moniteur série et on lit les données du moniteur série et on les écrit sur BluetoothSerial. Lorsqu'on écrit des données sur BluetoothSerial, l'application Bluetooth Terminal sur le téléphone reçoit les données et les imprime sur l'application. Lorsqu'on saisit des données dans l'application et qu'on les envoie via Bluetooth, le BluetoothSerial lira ces données et seront affichées sur le moniteur série.

— Après avoir télévisé le programme, ouvrir le moniteur série puis, activer le Bluetooth sur le smartphone et rechercher les appareils Bluetooth. On doit voir une liste des "Appareils jumelés" et des "Appareils disponibles" et parmi les appareils disponibles, sélectionner « ESP32 ». Connecter le smartphone à l'ESP32 via le Bluetooth.

— Maintenant, ouvrir l'application "Serial Bluetooth Terminal" sur le téléphone et cliquer sur les trois barres horizontales dans le coin supérieur gauche de l'écran.
Sélectionner l'onglet "Appareils" et sélectionner ESP32 dans la liste.
Cliquer sur l'icône "Lien" en haut pour se connecter au périphérique Bluetooth ESP32. L'application affichera l'état "Connexion à ESP32..." lors de la connexion et si la connexion réussit, elle affichera "Connecté".

— En bas se trouve un espace pour saisir les données à transférer via Bluetooth. Taper quelque chose et cliquer sur le bouton envoyer. Les données envoyées sont répercutées sur l'application. Ces données sont envoyées à ESP32 via Bluetooth et sont reçues par la fonction BluetoothSerial read(). Puisque on transmet ces informations au port série, on peut voir les données s'afficher sur le moniteur série.

— De même, on peut envoyer des données d'ESP32 vers le smartphone. Taper simplement quelques données dans le moniteur série et cliquer sur envoyer. Ces données sont envoyées via Bluetooth au smartphone via la fonction BluetoothSerial write(). L'application du terminal Bluetooth série lira ces données et les imprimera sur l'application.

5.1 LED contrôlée par Bluetooth via ESP32

En utilisant l'application précédente, on modifie légèrement le code pour implémenter une LED contrôlée par Bluetooth à l'aide d'ESP32. Le but est de voir à quel point il est facile de contrôler les broches GPIO de l'ESP32 en envoyant et en interprétant les données de Bluetooth. Pour garder les choses simples, on transmet « 1 » et « 0 » depuis l'application pour téléphone mobile à l'aide des touches de macro. J'ai attribué '1' pour M1 et '0' pour M2. On peut comparer les données reçues avec les caractères « 1 » et « 0 » ou leur équivalent décimal en ASCII, c'est-à-dire 49 et 48.
Lorsque "1" est reçu, la LED connectée au GPIO 2 s'allume et si "0" est reçu, la LED s'éteint. De toute évidence, la LED n'est qu'une représentation de la broche GPIO allumée et éteinte. On peut encore améliorer cette application en utilisant un relais contrôlé par Bluetooth via l'ESP32.

```
#include <BluetoothSerial.h>

#define ledPIN 2
BluetoothSerial SerialBT;
byte BTData;

/* Vérifie si les configurations Bluetooth sont activées dans le SDK */
#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth n'est pas activé! Lancer "make menuconfig" et l'activer
#endif

void setup() {
  pinMode(ledPIN, OUTPUT);
  Serial.begin(115200);
  SerialBT.begin();
  Serial.println("Bluetooth démarré! Prêt à coupler ...");
}

void loop() {
  if (SerialBT.available()) {
    BTData = SerialBT.read();
    Serial.write(BTData);
  }

  /* Si le caractère reçu est 1, alors allumer la LED */
  /* On peut également comparer les données reçues avec l'équivalent décimal */
  /* 48 pour 0 et 49 pour 1 */
  /* si(BTData == 48) ou si(BTData == 49) */
  if (BTData == '1') {
    digitalWrite(ledPIN, HIGH);
  }

  /* Si le caractère reçu est 0, éteindre la LED */
  if (BTData == '0') {
    digitalWrite(ledPIN, LOW);
  }
}
```

6. TP5 : Utiliser PWM dans ESP32

En utilisant PWM (Pulse Width Modulation, Modulation de Largeur d'Impulsions) dans l'ESP32, on peut contrôler la luminosité de la LED, définir la position d'un servomoteur et régler la vitesse d'un moteur à courant continu.

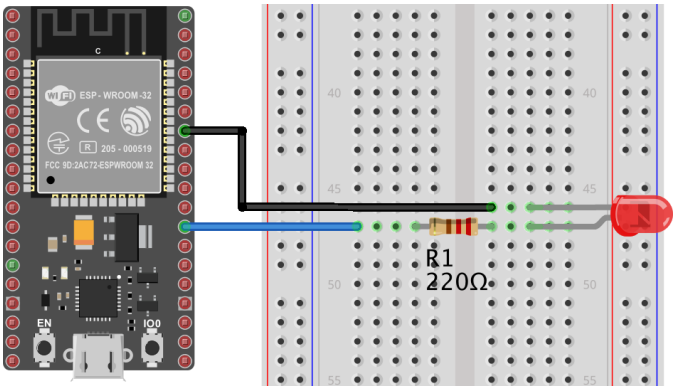
6.1 Contrôleur PWM LED ESP32 (LEDC)

Le périphérique LEDC de l'ESP32 se compose de 16 canaux PWM capables de générer des formes d'onde indépendantes, principalement pour le contrôle des LED RGB, mais peut également être utilisé à d'autres fins. 16 canaux PWM indépendants, divisés en groupe de deux avec 8 canaux par groupe. Résolution programmable entre 1 bit et 16 bits. La fréquence de l'onde PWM dépend de la résolution du PWM. Augmente/diminue automatiquement le rapport cyclique sans intervention du processeur.
Le tableau suivant montre quelques fréquences et résolutions PWM couramment utilisées.

Clock Source for LEDC	LEDC PWM Frequency	PWM Resolution
80 MHz APB_CLK	1 kHz	16 bits
80 MHz APB_CLK	5 kHz	14 bits
80 MHz APB_CLK	10kHz	13 bits
8 MHz RTC8M_CLK	1 kHz	13 bits
8 MHz RTC8M_CLK	8 kHz	10 bits
1 MHz REF_TICK	1 kHz	10 bits

Table 3. PWM dans l'ESP32

Le montage est simple, la luminosité d'une LED connectée à une broche GPIO d'ESP32 augmentera et diminuera progressivement à plusieurs reprises.



On peut utiliser n'importe quelle broche GPIO pour émettre le signal PWM. Donc, on utilise GPIO 16, qui est également la broche RX UART2. Ensuite, on doit configurer le canal LEDC à l'aide de la fonction "ledcSetup". Le premier argument est le canal. Toute valeur comprise entre 0 et 15 peut être donnée comme canal. L'argument suivant est la fréquence. On peut fournir n'importe quelle fréquence, mais pour plus de commodité, on définira la fréquence sur 5 KHz. De plus, on doit définir la résolution du PWM. Cette valeur doit être un nombre compris entre 1 et 16. On opte pour une résolution de 10 bits. Pour le reste des paramètres, voir le code suivant.

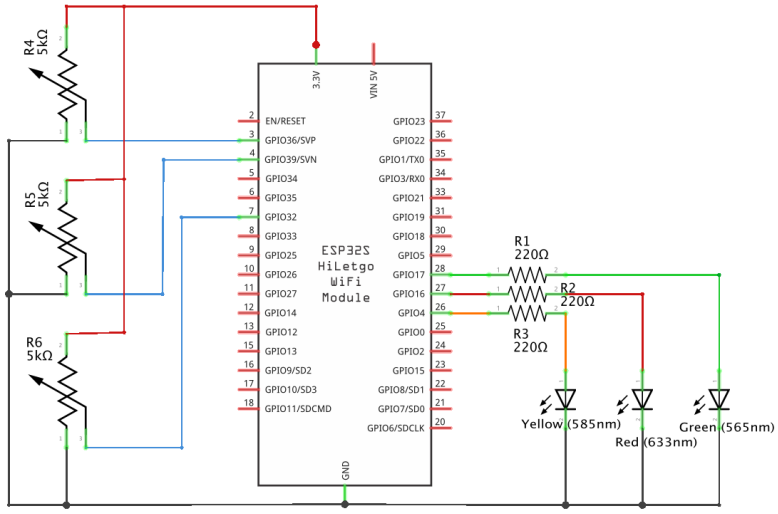
```
const int LEDPin = 16; /* GPIO16 */

int dutyCycle;
/* Définition des propriétés PWM */
const int PWMFreq = 5000; /* 5 KHz */
const int PWMChannel = 0;
const int PWMResolution = 10;
const int MAX_DUTY_CYCLE = (int)(pow(2, PWMResolution) - 1);
void setup() {
  ledcSetup(PWMChannel, PWMFreq, PWMResolution);
  /* Attacher le canal LED PWM à la broche GPIO */
  ledcAttachPin(LEDPin, PWMChannel);
}
void loop() {
  /* Augmenter la luminosité des LED avec PWM */
  for (dutyCycle = 0; dutyCycle <= MAX_DUTY_CYCLE; dutyCycle++) {
    ledcWrite(PWMChannel, dutyCycle);
    delay(3);
  }
  /* Diminuer la luminosité des LED avec PWM */
  for (dutyCycle = MAX_DUTY_CYCLE; dutyCycle >= 0; dutyCycle--) {
    ledcWrite(PWMChannel, dutyCycle);
    delay(3);
  }
}
```

6.2 ESP32 PWM avec ADC

L'une des caractéristiques importantes de PWM dans ESP32 est que les 16 canaux peuvent être configurés indépendamment, c'est-à-dire que chaque canal peut avoir sa propre résolution et sa propre fréquence. Pour le démontrer, utilisons le périphérique ADC et ajustons indépendamment le rapport cyclique de trois canaux PWM LEDC différents en tournant un potentiomètre. Trois potentiomètres 5 kΩ sont connectés à trois broches d'entrée ADC d'ESP32. Sur la base de la sortie de l'ADC, nous définirons le rapport cyclique de trois canaux PWM, qui sont configurés avec des paramètres différents. Pour faciliter la compréhension, on connecte trois LED : ROUGE, VERT et JAUNE à trois broches GPIO. Ces trois broches GPIO sont attachées à trois canaux LEDC PWM différents et chaque canal est initialisé avec sa propre fréquence et sa propre résolution.

LED	Pin GPIO	Canal PWM	Fréquence PWM	Résolution PWM
ROUGE	GPIO16	0	5000 (5 kHz)	12
VERT	GPIO17	2	8000 (8 kHz)	13
JAUNE	GPIO4	4	10000 (10 kHz)	14

Un autre point important à retenir est que la résolution de l'ADC de l'ESP32 est de 12 bits. Nous devons donc mapper cela avec soin sur la résolution PWM, pour obtenir la gamme complète de contrôle.


R4
5kΩ
R5
5kΩ
R6
5kΩ
3.3V
VIN 5V
GND
ESP32S
HiLetgo
WiFi
Module
EN/RESET
GPIO36/SVP
GPIO39/SVN
GPIO34
GPIO35
GPIO32
GPIO33
GPIO25
GPIO26
GPIO27
GPIO14
GPIO12
GPIO13
GPIO9/SD2
GPIO10/SD3
GPIO11/SDCMD
GPIO23
GPIO22
GPIO1/TX0
GPIO3/RX0
GPIO21
GPIO19
GPIO18
GPIO5
GPIO17
GPIO16
GPIO4
GPIO0
GPIO2
GPIO15
GPIO8/SD1
GPIO7/SD0
GPIO6/SDCLK
R1
220Ω
R2
220Ω
R3
220Ω
Yellow (585nm)
Red (633nm)
Green (565nm)

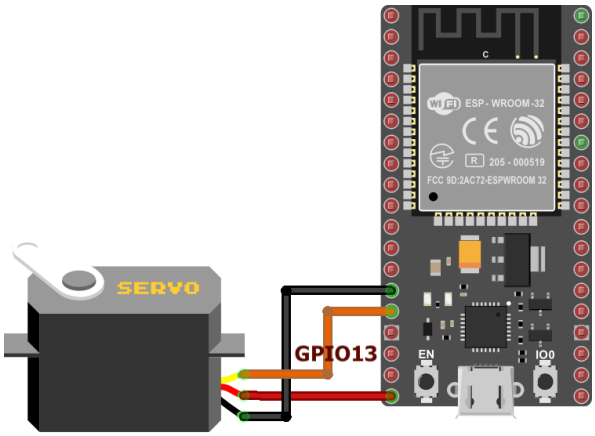
```
const int rougeLEDPin = 16;    /* GPIO16 */
const int vertLEDPin = 17;     /* GPIO17 */
const int jauneLEDPin = 4;     /* GPIO4 */
uint16_t rougeDutyCycle;
uint16_t vertDutyCycle;
uint16_t jauneDutyCycle;

const int rougePWMFreq = 5000; /* 5 kHz */
const int rougePWMChannel = 0;
const int rougePWMResolution = 12;
const int ROUGE_MAX_DUTY_CYCLE = (int)(pow(2, rougePWMResolution) - 1);

const int vertPWMFreq = 8000; /* 8 kHz */
const int vertPWMChannel = 2;
const int vertPWMResolution = 13;
const int VERT_MAX_DUTY_CYCLE = (int)(pow(2, vertPWMResolution) - 1);

const int jaunePWMFreq = 10000; /* 10 kHz */
const int jaunePWMChannel = 4;
const int jaunePWMResolution = 14;
const int JAUNE_MAX_DUTY_CYCLE = (int)(pow(2, jaunePWMResolution) - 1);

const int ADC_RESOLUTION = 4095; /* 12-bits */


void setup() {
  Serial.begin(115200);
  ledcSetup(rougePWMChannel, rougePWMFreq, rougePWMResolution);
  ledcSetup(vertPWMChannel, vertPWMFreq, vertPWMResolution);
  ledcSetup(jaunePWMChannel, jaunePWMFreq, jaunePWMResolution);
  ledcAttachPin(rougeLEDPin, rougePWMChannel);
  ledcAttachPin(vertLEDPin, vertPWMChannel);
  ledcAttachPin(jauneLEDPin, jaunePWMChannel);
}
void loop() {
  /* Lire l'entrée analogique à partir de trois entrées ADC */
  rougeDutyCycle = analogRead(A0);
  vertDutyCycle = analogRead(A3);
  jauneDutyCycle = analogRead(A4);
  /* Mapper la sortie ADC sur le cycle de service maximal possible */
  //rougeDutyCycle = map(rougeDutyCycle, 0, ADC_RESOLUTION, 0, ROUGE_MAX_DUTY_CYCLE);
  vertDutyCycle = map(vertDutyCycle, 0, ADC_RESOLUTION, 0, VERT_MAX_DUTY_CYCLE);
  jauneDutyCycle = map(jauneDutyCycle, 0, ADC_RESOLUTION, 0, JAUNE_MAX_DUTY_CYCLE);
  /* Définir la sortie PWM du canal avec le rapport cyclique souhaité */
  ledcWrite(rougePWMChannel, rougeDutyCycle);
  ledcWrite(vertPWMChannel, vertDutyCycle);
  ledcWrite(jaunePWMChannel, jauneDutyCycle);

  Serial.println("ROUGE -- VERT -- JAUNE");
  Serial.print(rougeDutyCycle);
  Serial.print(" -- ");
  Serial.print(vertDutyCycle);
  Serial.print(" -- ");
  Serial.print(jauneDutyCycle);
  Serial.print("\n");

  delay(1000);
}
```

7. TP6 : Serveur Web pour contrôler un servomoteur

On va créer un serveur Web avec l'ESP32 qui contrôle la position de l'arbre d'un servomoteur à l'aide d'un curseur.

7.1 Connexion du servomoteur à l'ESP32

Les servomoteurs ont trois fils : alimentation, masse et signal. L'alimentation est généralement rouge, le GND est noir ou marron et le fil de signal est généralement jaune, orange ou blanc.



7.2 Installation de la librairie ESP32_Arduino_Servo_Library

La bibliothèque de servomoteurs Arduino ESP32 facilite le contrôle d'un servomoteur avec l'ESP32, à l'aide de l'IDE Arduino.

— Télécharger la librairie ESP32_Arduino_Servo_Library. Vous devriez avoir un dossier .zip dans votre dossier Téléchargements Décompressez le dossier .zip et vous devriez obtenir le dossier ESP32-Arduino-Servo-Library-Master

— Renommez votre dossier ESP32-Arduino-Servo-Library-Master en ESP32_Arduino_Servo_Library

— Déplacez le dossier ESP32_Arduino_Servo_Library vers le dossier des bibliothèques d'installation de votre IDE Arduino

— Rouvrez votre IDE Arduino

```ino
#include <Servo.h>

Servo myservo;

int pos = 0;

void setup() {
  myservo.attach(13);
}

void loop() {
  for (pos = 0; pos <= 180; pos += 1) {    // passe de 0 degrés à 180 degrés
                                           // par pas de 1 degré
    myservo.write(pos);  // indique au servo d'aller à la position dans la variable 'pos'
    delay(15);           // attend 15ms que le servo atteigne la position
  }
  for (pos = 180; pos >= 0; pos -= 1) {  // passe de 180 degrés à 0 degrés
    myservo.write(pos);                  // indique au servo d'aller à la position dans
                                         // la variable 'pos'
    delay(15);                           // attend 15ms que le servo atteigne la position
  }
}
```

7.3 Création du serveur Web

Maintenant que vous savez comment contrôler un servo avec l'ESP32, créons le serveur Web pour le contrôler (en savoir plus sur la création d'un serveur Web ESP32).

— Le serveur Web que nous allons construire : Contient un curseur de 0 à 180, que vous pouvez régler pour contrôler la position de l'arbre du servo

— La valeur actuelle du curseur est automatiquement mise à jour dans la page Web, ainsi que la position de l'arbre, sans qu'il soit nécessaire d'actualiser la page Web. Pour cela, nous utilisons AJAX pour envoyer des requêtes HTTP à l'ESP32 en arrière-plan

— L'actualisation de la page Web ne modifie pas la valeur du curseur, ni la position de l'arbre.

Page HTML

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="data:,">
  <style>
    body {
      text-align: center;
      font-family: "Trebuchet MS", Arial;
      margin-left:auto;
      margin-right:auto;
    }
    .slider {
      width: 300px;
    }
  </style>
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
</head>
<body>
  <h1>ESP32 with Servo</h1>
  <p>Position: <span id="servoPos"></span></p>
  <input type="range" min="0" max="180" class="slider" id="servoSlider"
  onchange="servo(this.value)"/>
  <script>
    var slider = document.getElementById("servoSlider");
    var servoP = document.getElementById("servoPos");
    servoP.innerHTML = slider.value;
    slider.oninput = function() {
      slider.value = this.value;
      servoP.innerHTML = this.value;
    }
    $.ajaxSetup({timeout:1000});
    function servo(pos) {
      $.get("/?value=" + pos + "&");
      {Connection: close};
    }
  </script>
</body>
</html>
```

Maintenant, on doit inclure le code HTML précédent dans le sketch et faire tourner le servomoteur en conséquence.

Début

```
#include <WiFi.h>
#include <Servo.h>

Servo myservo;

static const int servoPin = 13;

const char* ssid = "NomDuRéseau";
const char* password = "MotDePasse";

WiFiServer server(80);

String header;

String valueString = String(5);
int pos1 = 0;
int pos2 = 0;

unsigned long currentTime = millis();
unsigned long previousTime = 0;
const long timeoutTime = 2000;

void setup() {
  Serial.begin(115200);

  myservo.attach(servoPin);

  Serial.print("Connecté au réseau ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.println("Connecté au WiFi.");
  Serial.println("Adresse IP : ");
  Serial.println(WiFi.localIP());
  server.begin();
}

void loop() {
  WiFiClient client = server.available();

  if (client) {
    currentTime = millis();
    previousTime = currentTime;
    Serial.println("New Client.");
    String currentLine = "";
    while (client.connected() && currentTime - previousTime <= timeoutTime) {
      currentTime = millis();
      if (client.available()) {
        char c = client.read();
        Serial.write(c);
        header += c;
        if (c == '\n') {
          if (currentLine.length() == 0) {
            client.println("HTTP/1.1 200 OK");
            client.println("Type de contenu :text/html");
            client.println("Connexion: fermée");
            client.println();

            client.println("<!DOCTYPE html><html>");
            client.println("<head><meta name=\"viewport\" content=\"width=device-width,
                  initial-scale=1\">");
            client.println("<link rel=\"icon\" href=\"data:,\">");
            client.println("<style>body { text-align: center; font-family: \"Trebuchet MS\",
                  Arial; margin-left:auto; margin-right:auto;}");
```

Fin

```
            //client.println("<style>body { text-align: center; font-family: \"Trebuchet MS\",
            //      Arial; margin-left:auto; margin-right:auto;}");
            client.println(".slider { width: 300px; }</style>");
            client.println("<script src=\"https://ajax.googleapis.com/ajax/libs/jquery/
                  3.3.1/jquery.min.js\"></script>");

            client.println("</head><body><h1>ESP32 with Servo</h1>");
            client.println("<p>Position: <span id=\"servoPos\"></span></p>");
            client.println("<input type=\"range\" min=\"0\" max=\"180\"
                  class=\"slider\" id=\"servoSlider\" onchange=\"servo(this.value)\"
        value=\"" + valueString + "\"/>");

            client.println("<script>var slider = document.getElementById(\"servoSlider\");");
            client.println("var servoP = document.getElementById(\"servoPos\");
                  servoP.innerHTML = slider.value;");
            client.println("slider.oninput = function() { slider.value = this.value;
                        servoP.innerHTML = this.value; }");
            client.println("$.ajaxSetup({timeout:1000}); function servo(pos) { ");
            client.println("$.get(\"/?value=\" + pos + \"&\"); {Connection: close};}</script>");

            client.println("</body></html>");

            if (header.indexOf("GET /?value=") >= 0) {
              pos1 = header.indexOf('=');
              pos2 = header.indexOf('&');
              valueString = header.substring(pos1 + 1, pos2);

              //Rotation servo
              myservo.write(valueString.toInt());
              Serial.println(valueString);
            }
            client.println();
            break;
          } else {
            currentLine = "";
          }
        } else if (c != '\r') {
          currentLine += c;
        }
      }
    }
    header = "";
    client.stop();
    Serial.println("Client déconnecté.");
    Serial.println("");
  }
}
```

Test du serveur Web

- Copier l'adresse IP ESP32 qui s'affiche sur le moniteur série.
- Ouvrir le navigateur, coller l'adresse IP ESP pour aller sur la page Web créée précédemment.
- Déplacer le curseur pour contrôler le servomoteur.
- Dans le moniteur série, on peut également voir les requêtes HTTP envoyée à l'ESP32 lorsqu'on déplace le curseur.