Programmation avancée

connexion Wifi & TCP avec esp32/esp8266

Détails

Écrit par stéphane Domas

Catégorie : Programmation avancée (/index.php/menu-lpsil/objets-connectes)

Publication : 18 novembre 2009

Affichages : 552

1°/ Wifi

- les esp8266/32 intègre nativement un support bluetooth et wifi, ce qui n'est pas le cas de l'ATmega328
- De plus, ils peuvent être programmés pour agir comme un client wifi, un point d'accès (AP) wifi, ou bien les deux :
  - Un client se connecte à un AP et reçoit en retour une adresse IP,
  - Un AP fournit des connexions Wifi et des IPs aux clients, mais n'a pas forcément d'IP
  - La combinaison des eux permet à l'AP d'avoir une IP et de partciper aux communications entre µC.

- Un sketch nécessitant d'utiliser le Wifi doit forcément inclure le fichier .h contenant les déclarations des fonctions/classes/constantes/... liées au Wifi.
- Malheureusement, le nom de ce fichier n'est pas standardisé et il change selon le µC.
- Par exemple, pour un arduino et esp32, il faut inclures `WiFi.h`, et pour les esp8266, c'est `ESP8266WiFi.h`

1.1°/ client Wifi

- Généralement, on écrit une fonction qui sera appelée dans setup().
- Cette fonction essaie à l'infini d'établir une connexion à un point d'acces dont on connait le SSID et le mot de passe.

Exemple (esp8266) :

```
#include <ESP8266WiFi.h>
...
void setupWifi() {

  const char *ssid = "ssid_wifi_ap"; // the ssid of the AP
  const char *password = "pass_ap"; // the password of the AP

  // comment 2 following lines if not needed
  WiFi.setAutoConnect(false);  // see comments
  WiFi.setSleepMode(WIFI_NONE_SLEEP); // always active => big consumption

  WiFi.mode (WIFI_STA); // setup as a wifi client
  WiFi.begin(ssid,password); // try to connect
  while (WiFi.status() != WL_CONNECTED) { // check connection
    delay(500);
    Serial.print(".");
  }
  // debug messages
  Serial.print("Connected, IP address: ");
  Serial.println(WiFi.localIP());
}
```

Remarques :

- `setAutoConnect(false)` permet d'éviter que le µC se connecte automatiquement au dernier AP qu'il a utilisé. En effet, ses informations sont stockées en mémoire flash et par défaut, le µC tente lors du boot de s'y connecter. Au mieux, cela ralentit la connexion à un nouvel AP, mais dans certains cas, cela empêche totalement cette connexion. Il est donc conseillé de désactiver cette connexion automatique quand le µC est nomde et va souvent changer d'AP. On la laisse active uniquement s'il se connecte toujours au même AP.
- `setSleepMode(WIFI_NONE_SLEEP)` permet de garder le circuit Wifi toujours actif. Ce n'est pas le mode par défaut où ce circuit peut parfois être mis en pause. Ces pauses impliquent que des réceptions peuvent être manquées par le µC, ce qui nécessite de renvoyer les données. Dans certaines applications, cette réémission n'est pas acceptable, d'où la fait de garder le wifi toujours actif. En revanche, ce mode consomme beaucoup d'énergie électrique ce qui pose problème si le µC est alimenté sur batterie/pile (cf. section sur la consommation)

1.2°/ Point d'accès + client Wifi access

- Comme un AP agit comme un serveur DHCP, il faut obligatoirement fixer un adresse et un masque de sous-réseau, ainsi qu'une passerelle pour que l'AP soit en mesure d'attribuer des IPs pour ce sous-réseau.
- Dans l'exemple ci-dessous, on suppose que l'AP fournit des adresses de classe C, comme 192.168.0.XXX
- Il faut également définir un canal Wifi.

Apparté :

- La norme Wifi 2.4GHz définit 13 canaux, correspondant à 13 fréquences autour de 2.4GHz.
- Si deux AP sont proches l'un de l'autre, cela risque de provoquer des interférences s'ils utilisent des fréquences trop proche. Dans ce cas, des paquets de données sont corrompus et doivent être rémis.
- Pour régler ce type de problème, on utilise des canaux séparé d'au moins 4 en valeur.
- Par exemple, dans une même pièce, on peut mettre jusqu'à 4 AP, avec comme canaux 1, 5, 9 et 13.

Exemple (esp8266) :

```
#include <ESP8266WiFi.h>
...
void setupAP() {
  IPAddress local_IP(192,168,0,1);
  IPAddress gateway(192,168,0,1);
  IPAddress subnet(255,255,255,0);

  WiFi.mode(WIFI_AP_STA); // AP + client
  int channel = 1; // choose between 1 and 13
  const char *ssid = "ssid_wifi_ap";
  const char *password = "pass_ap";

 // CAUTION: on some µC, 2 following lines may be exchanged to work
  WiFi.softAP(ssid, password, channel);
  WiFi.softAPConfig(local_IP, gateway, subnet);

  // debug messages
  IPAddress myIP = WiFi.softAPIP();
  Serial.print("AP IP address: ");
  Serial.println(myIP);
}
```

Remarques :

- en tant qu'AP, un esp8266/32 peut accepter au maximum 4 clients, mais il est déconseillé d'atteindre ce nombre, surtout si l'AP est lui-même un client est doit faire quelque chose d'autre. Dans ce cas, le µC peut vite être surchargé et planter.
- il est donc conseillé d'utiliser un µC comme AP uniquement dans des applications où seulement 2 µC "nomades" doivent communiquer.

2°/ Connexion et communication TCP

- Une fois qu'un µC possède une IP, il est possible de créer une connexion TCP
- A µC can act as a TCP client, or a TCP server., soit comme un client, soit comme un serveur.

2.1°/ client TCP

- Généralement, on écrit une fonction chargée de demander la connexion et qui stocke l'état de cette connexion das une variable globale.
- Comme il est fréquent qu'une connexion TCP soit interrompue, cette fonction est plutôt appelée si nécessaire au début de la fonction loop(), pour être sur de pouvoir communiquer dans la suite de l'exécution.
- Cette fonction doit utiliser une instance de la classe `WiFiClient`, qui contient une méthode `connect()` pour demander la connexion.
- Cette classe contient également des méthodes :
  - `available()` pour tester si des octets peuvent être lus,
  - `write()` pour écrire 1 seul octet,
  - `read()` pour lire un seul octet,
  - `println()` pour écrire une ligne de texte, transformée en une suite d'octets avant d'être envoyée.
- Malheureusement, il n'existe pas de fonction pour lire directement une ligne de texte. Il faut l'écrire soi-même.

Exemple (esp32) :

```
#include <WiFi.h>
...
WiFiClient client;
byte connState;
const char* ipServ = "192.168.0.1";
int portServ = 12345;
byte buf[1024];
int idx = 0;
...
String readLine() {
  char buffer[1024];
  memset(buffer,0,1024);    // reset buffer
  int index = 0;
  char c;
  while(true) {
    while (!client.available()) {} // wait for smthg to read
    while ((index < 1023) && (client.available())) {
      c = client.read();
      if (c == '\n') return String(buffer); // end-of-line reached => return
      buffer[index++] = c; // store the char
    }
    // prevent buffer overflow: return the whole buffer even if no \n encountered
    if (index == 1023) return String(buffer);
  }
}

void setup() {
  Serial.begin(115200);
  connState = 0;
  // establish the wifi connection
  ...
}

bool doConnection() {
  if (client.connect(ipServ, portServ)) {
    connState = 1;
    Serial.println("connected to server")
    // uncomment next line  for esp8266: disable buffering small mesg
    // client.setNoDelay(true);
    return true;
  }
  else {
    connState = 0;
    Serial.println("connection failed")
    return false;
}

void loop() {
  if (connState == 0) {
    if (! doConnection()) { delay(1000); }
  }
```

```
  if (connState == 1) {
    client.write(10); // write a byte equal to 10
    client.println("10"); // write 3 bytes: ascii code of 1, ascii code of 0, ascii code of new
    // wait something from the server
    while (!client.available()) {} // see comments below
    // wait something from the server
    String msg = readLine();
    Serial.println(msg);
}
```

Remarques :

- contrairement à un OS classique, lire sur une socket grâce à la fonction `WiFiClient.read()` n'est pas bloquant. S'il n'y a rien à lire, la fonction renvoie donc une valeur invalide.
- C'est pourquoi il faut toujours vérifier que quelque chose est disponible AVANT de lire. Pour cela, on utilise `available()`.
- Cependant, il est possible d'attendre qu'au moins un octet soit disponible à la lecture avec une boucle "bizarre" : `while(!client.available()) {}`. Sur un processeur classique, cela ne peut pas se faire sinon on le charge à 100%. Mais sur un µC, cela fonctionne parfaitement et c'est même la seule façon de faire.

2.2°/ serveur TCP

- Pour attendre une connexion, le µC doit utiliser une instance de `WiFiServer`, dont le constructeur permet de fixer le port de connexion.
- Cette classe contient une méthode `available()` qui teste s'il y a une demande de connexion client.
- Attention : cette fonction n'est pas bloquante comme sur un OS classique. Elle renvoie directement null si aucune demande n'est faite. Il faut donc l'appeler en boucle.
- Dès qu'une demande est accepté, cette fonction renvoie un objet instance de `WiFiClient` qui permet de communiquer avec le client, comme vu en section 7.1

Exemple (esp32) :

```
#include <WiFi.h>
...
WiFiServer server(12345);
WiFiClient client;
byte connState;
byte buf[1024];
int idx = 0;
...

void setup() {
  Serial.begin(115200);
  connState = 0;
  // establish the wifi connection
  ...
}

void waitConnection() {
  client = server.available(); // test if there is an incoming connection and if yes, accept it
  if (client) {
    connState = 1;
    Serial.println("client connected")
    // special for esp8266: disable buffering small mesg
    client.setNoDelay(true);
  }
}

void loop() {
  if (connState == 0) {
    waitConnection();
  }
  if (connState == 1) {
    // wait something from the client
    while (!client.available()) {}
    // example on how to read what is available in a simple buffer
    while (client.available()) {
      buf[idx++] = client.read(); // read & store a byte
    }
    Serial.println(String(buf)); // print the content of buf as a string
    for(int i=0;i<idx;i++) buf[i] = 0; // reset buf
  }
}
```