UE INF-IoT
« Internet of Things »

Sujet de TP : Transmission de données issues de capteurs vers un serveur distant via Wi-Fi

L'objectif de cette séance de travaux pratiques est d'étudier comment un système piloté par un micro-contrôleur ESP32 (ou équivalent) peut utiliser le Wi-Fi pour transmettre des données capturées localement (température, humidité, luminosité, etc.) vers un serveur distant.

1 Aperçu du micro-contrôleur ESP32

Le micro-contrôleur ESP32 existe sous différents conditionnements, et est intégré dans un grand nombre de platines de développement, telle que la platine DOIT-DEVKIT représentée dans la figure 1. Toutes les caractéristiques de ce micro-contrôleur sont bien sûr détaillées dans sa fiche technique [1].

L'une des caractéristiques intéressantes de l'ESP32 est qu'il intègre un transceiver Wi-Fi ainsi qu'un transceiver Bluetooth. Il peut donc communiquer avec d'autres équipements par voie radio sans qu'il faille lui adjoindre de circuit supplémentaire pour ce faire.

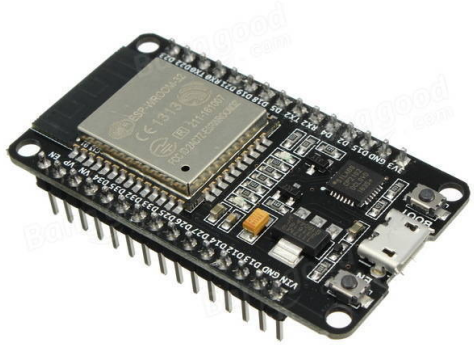

FIGURE 1 – Platine de développement DOIT-DEVKIT-V1 basée sur un micro-contrôleur ESP32-WROOM-32

2 Montage

Aucun montage n'est requis dans un premier temps : vous vous contenterez d'utiliser une platine supportant un ESP32, et le programme que vous allez développer enverra des données fictives vers un serveur distant.

Dans un second temps, vous ajouterez quelques capteurs à votre platine afin que les données envoyées au serveur soient cette fois des données réelles.

3 Développement d'un programme de test

3.1 Objectif

En utilisant l'éditeur Visual Studio Code avec l'extension PlatformIO, vous allez développer un petit programme permettant :

1. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

1. de scanner les canaux Wi-Fi afin de détecter les SSID des points d'accès (AP) environnants ;
2. lorsque le SSID du réseau visé est détecté, de tenter de se connecter au point d'accès correspondant ;
3. lorsque la connexion Wi-Fi est établie, de consulter périodiquement un ou plusieurs capteurs afin de produire des données ;
4. lorsque de nouvelles données sont produites, de les adresser par TCP à un serveur distant.

Le code permettant d'atteindre ces quatre objectifs principaux devra être organisé afin de lui assurer le maximum de robustesse. Par exemple, il ne sert à rien de scanner les canaux Wi-Fi lorsque la platine est déjà connectée à un point d'accès. Inversement, il ne sert à rien de consulter l'état des capteurs si l'on n'a pas de connectivité Wi-Fi (puisqu'on est alors incapable de transmettre les données issues de ces capteurs). De même, si la connection établie avec un point d'accès est interrompue, alors le programme doit se remettre à chercher un point d'accès pour tenter de s'y connecter.

Le programme sera développé en utilisant le Framework Arduino, et notamment la librairie *arduino-esp32*, conçue spécifiquement pour piloter l'ESP32 dans le cadre du Framework Arduino [2].

3.2 Démarche

Pour vous faire gagner du temps, le squelette du programme que vous devez réaliser est mis à votre disposition. Dans ce squelette les fonctions `setup()` et `loop()` sont déjà définies, ainsi que quelques fonctions annexes dont vous aurez besoin.

La fonction `loop()` gère le cycle de vie du programme : à chaque tour de boucle on s'assure que la connexion Wi-Fi est établie. Si ce n'est pas le cas, on lance une procédure de connexion Wi-Fi. Cette procédure n'est toutefois lancée que si la dernière tentative n'est pas trop récente : il serait contre-productif d'inonder un point d'accès Wi-Fi en lui envoyant des rafales de demandes de connexion.

Lorsqu'une connexion Wi-Fi est établie, et seulement dans ce cas, on effectue périodiquement l'acquisition de données en consultant un ou plusieurs capteurs. Si des données sont effectivement produites (souvenez vous que la consultation d'un capteur peut parfois échouer), alors on tente de les transmettre vers un serveur distant. Cette transmission s'effectue via une session TCP, qui est ouverte sur demande et fermée sitôt après la fin de la transmission des données.

Les données sont produites par la fonction `get_samples()`, qui retourne ces données sous la forme d'une simple chaîne de caractères C (ou `NULL` si l'acquisition de données a échoué). Dans un premier temps le code de cette fonction retourne des données « bidon » constituées de l'identifiant du capteur (ici remplacé par l'adresse MAC de la platine) et d'une valeur de température fictive. Lorsque votre programme sera opérationnel vous pourrez faire en sorte que les données produites soient en fait des données réelles issues, par exemple, de capteurs de température. Il vous faudra pour ce faire reprendre le code d'un TP précédent et le combiner avec celui du TP actuel.

La transmission des données vers le serveur distant s'effectue dans la fonction `send()`. La librairie *arduino-esp32* fournit une classe `WiFiClient`, qui comme son nom ne l'indique pas (!) implémente un code client TCP utilisable dans le cadre d'une connexion Wi-Fi. En instanciant cette classe, l'ouverture d'une session TCP, l'émission et/ou la réception de données via cette session TCP, et la fermeture de session s'effectuent sans difficulté majeure. Dans le cas présent une nouvelle session TCP doit être ouverte, utilisée, puis fermée à chaque fois que de nouvelles données sont acquises via la fonction `get_samples()`.

3.3 Quelques remarques

— Les paramètres utilisés pour la transmission (e.g., SSID et mot de passe du réseau visé, adresse et n° de port du serveur distant, intervalles d'attente entre deux connexions Wi-Fi ou entre deux acquisitions de données) sont fixés « en dur » dans le code, au début du programme. Vous devrez les ajuster en fonction de vos besoins. Idéalement de tels paramètres devraient être stockés en mémoire Flash, ce qui permettrait de les modifier sans avoir à recompiler le programme.

— La fonction `start_WiFi_connection()` est invoquée pour essayer d'établir une connexion avec un point d'accès environnant. Le code de cette fonction devrait être décomposé comme suit :
  — on scanne les réseaux environnants ;
  — on vérifie avec `checkSSID()` que le SSID du réseau visé a été détecté (sinon on abandonne) ;
  — on extrait avec `getChannel()` le n° de canal sur lequel se trouve l'AP ayant ce SSID ;
  — on tente une connexion avec cet AP, en spécifiant le SSID, le mot de passe, et le canal utilisé [3].

2. `https://github.com/espressif/arduino-esp32`
3. Il est théoriquement possible de tenter une connexion sans spécifier le canal choisi, mais l'expérience montre que cela se solde très

```
Enabling mode WIFI_STA
mac=24:0A:C4:81:A5:7C

Attempting to connect to D008_CYBERLAB
Scanning... D008_CYBERLAB found!
Sending association request
Wi-Fi connected
IP address: 10.220.47.132

Connecting to server: OK
Sending data: id=24:0A:C4:81:A5:7C temp=22.9
Closing connection

Connecting to server: OK
Sending data: id=24:0A:C4:81:A5:7C temp=23.1
Closing connection
```

FIGURE 2 – Illustration de l'interaction avec le programme via la console

souvent par un échec.

Squelette du programme

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

#define LED_WiFi  LED_BUILTIN

// ========================================================================

WiFiClient client;
String mac;


#define NETWORK_SSID    "D008_CYBERLAB"
#define NETWORK_PASSWD  "********"
#define SERVER_ADDR     "10.0.0.1"
#define SERVER_PORT     6000

// ========================================================================

wl_status_t WiFi_status = WL_NO_SHIELD;

unsigned long WiFi_last_attempt               = 99999;
static unsigned long WiFi_attempt_interval    = 10000;

unsigned long last_sampling                   = 99999;
static unsigned long sampling_interval        = 10000;

// ========================================================================
// WI-FI MANAGEMENT
// ========================================================================

// ========================================================================
// Set up the Wi_Fi client
//
// - Enable the LED that will display the connectivity status
// - Enable mode WIFI_STA and display the MAC addresses
// - Set the client's timeout to 5 seconds
void setup_WiFi_client() {

  Serial.println("TO BE COMPLETED");
}

// ========================================================================
// Return 'true' if 'ssid' has been detected during the last scan.
// Return 'false' otherwise.
bool checkSSID(String ssid) {

  Serial.println("TO BE COMPLETED");
}

// ========================================================================
// Return the channel number used by network 'ssid', as detected during
// the last scan.  Return 0 if 'ssid' was not detected.
int getChannel(String ssid) {

  Serial.println("TO BE COMPLETED");
}

// ========================================================================
// Check the current WiFi status (i.e., connected or not connected).
// When a connection or disconnection is detected, display information in
```

```
61 // the serial console, and switch the LED on or off accordingly.
62 // When a disconnection is detected, stop the WiFiClient if it was connected.
63 void check_WiFi_status() {
64
65   Serial.println("TO BE COMPLETED");
66 }
67
68 // ============================================================================
69 void start_WiFi_connection() {
70
71   Serial.println("TO BE COMPLETED");
72 }
73
74 // ============================================================================
75 // DATA ACQUISITION MANAGEMENT
76 // ============================================================================
77
78 // ============================================================================
79 // Try to get new samples, and return these samples as a C string (or NULL
80 // if no samples were obtained)
81 char *get_samples() {
82
83   // Dummy implementation: should be replaced by real data sampling
84   static char data[256];
85
86   unsigned long now = millis();
87   float temp = 18.0 + (now % 100) / 10.0;
88   strcpy(data, "id=");
89   strcat(data, mac.c_str());
90   strcat(data, " temp=");
91   sprintf(data+strlen(data), "%.1f", temp);
92
93   return data;
94 }
95
96 // ============================================================================
97 // Send data to the remote server (via a TCP session)
98 void send(char *data) {
99
100   Serial.print("Connecting to server: ");
101   if (client.connect(SERVER_ADDR, SERVER_PORT)) {
102
103     Serial.println("OK");
104     Serial.print("Sending data: ");
105     Serial.println(data);
106
107     client.println(data);
108
109     Serial.println("Closing connection");
110     client.stop();
111   }
112   else {
113     Serial.println("failed");
114   }
115   Serial.println();
116 }
117
118 // ============================================================================
119 // MAIN FUNCTIONS
120 // ============================================================================
121
122
123 void setup() {
```

```
124
125   Serial.begin(115200);
126
127   setup_WiFi_client();
128
129   Serial.println(); Serial.println();
130 }
131
132 // ============================================================================
133 void loop() {
134
135   unsigned long now = millis();
136
137   // Check if the Wi-Fi status has changed
138   check_WiFi_status();
139
140   // If the Wi-Fi is currently "not connected"
141   if (WiFi_status != WL_CONNECTED) {
142     // If the last connection attempt was long ago, let us make another attempt
143     if (now - WiFi_last_attempt > WiFi_attempt_interval) {
144       WiFi_last_attempt = now;
145       start_WiFi_connection();
146     }
147     // There is no need to go further until the Wi-Fi connection is established
148     return;
149   }
150
151   // If the last data acquisition was long ago, let us start another one
152   if (now - last_sampling > sampling_interval) {
153     last_sampling = now;
154
155     // Try to get new samples
156     char *data = get_samples();
157
158     // If new samples have been obtained, send these samples away
159     if (data != NULL)
160       send(data);
161   }
162 }
```