IoT NetLab 1

Using UDP,TCP and Web Sockets

In this lab we are going to implement basic internet transport protocols such as UDP, TCP or WebSockets used extensively for IoT messaging.

1.1 UDP sockets – sending and receiving datagrams

UDP socket routines enable simple IP communication using the user datagram protocol (UDP).
The User Datagram Protocol (UDP) runs on top of the Internet Protocol (IP) and was developed for applications that do not require reliability, acknowledgment, or flow control features at the transport layer. This **simple protocol** provides transport layer addressing in the form of UDP ports and an optional checksum capability. UDP is a very simple protocol.

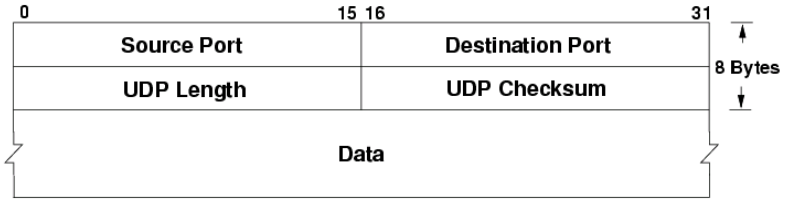


Fig 1.1 UDP datagram with header

Messages, so called **datagrams**, are sent to other hosts on an IP network without the need to set up special transmission channels or data paths beforehand. The UDP socket only needs to be opened for communication. It listens for incoming messages and sends outgoing messages on request.

On the given IP network the **datagrams** my be sent in **unicast** or **broadcast** mode.

1.1.1 UDP client

The following example contains UDP client code that send simple datagrams in **broadcast** mode on **this** network. Note that the data have constant values. They are prepared in the **`union`** structure:

```
union
{
  uint8_t frame[16];
  float sensor[4];
  char mess[16];
} sdp, rdp; // send and receive packets
```

The same union is used at the UDP receiver side.

```
#include <WiFi.h>
#include <WiFiUdp.h>
/* WiFi network name and password */

const char * ssid= "Livebox-08B0";
const char * pwd = "G79ji6dtEptVTPWmZP";
const char * udpAddress = "192.168.1.255";
const int udpPort = 1234;

//create UDP instance
WiFiUDP udp;

union
{
  uint8_t frame[16];
  float sensor[4];
  char mess[16];
} sdp, rdp; // send and receive packets
```

```
void setup()
{
  Serial.begin(115200);
  WiFi.begin(ssid, pwd);
  Serial.println("");
  // Wait for connection
  while (WiFi.status()!=WL_CONNECTED){delay(500);Serial.print(".");}
  Serial.println("");
  Serial.print("Connected to ");
  Serial.println(ssid);
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
  //This initializes udp and transfer buffer
  udp.begin(udpPort);
}

void loop()
{
  //data will be sent to server
  Serial.println("Send unicast/broadcast packet ");
  udp.beginPacket(udpAddress, udpPort);
  sdp.sensor[0]=0.1;sdp.sensor[1]=0.2;sdp.sensor[2]=0.3;sdp.sensor[3]=0.4;
  udp.write(sdp.frame, 16);
  udp.endPacket();
  memset(rdp.frame,0,16);
  //processing incoming packet, must be called before reading the buffer
  udp.parsePacket();
  //receive response from server, it will be ACK
  if(udp.read(rdp.frame,16)==16)
  {
    Serial.print("Received from server: ");
    Serial.println(rdp.mess);
  }
  //Wait for 2 seconds
  delay(2000);
}
```

1.1.2 Synchronous UDP server

```
#include <WiFi.h>
#include <WiFiUdp.h>
/* WiFi network name and password */
const char * ssid= "Livebox-08B0";
const char * pwd = "G79ji6dtEptVTPWmZP";

const char * udpAddress = "192.168.1.255";
const int udpPort = 1234;
//create UDP instance
WiFiUDP udp;

union
{
  uint8_t frame[16];
  float sensor[4];
  char mess[16];
} rdp,sdp;

void setup()
{
  Serial.begin(115200);
  //Connect to the WiFi network
  WiFi.begin(ssid, pwd);
  Serial.println("");
  // Wait for connection
  while (WiFi.status()!=WL_CONNECTED)
  {
      delay(500);Serial.print(".");
  }
  Serial.println("");
  Serial.print("Connected to ");
  Serial.println(ssid);
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
  //This initializes udp and transfer buffer
  udp.begin(udpPort);
}
```

```
void loop()
{
  memset(rdp.frame,0,16);
  //processing incoming packet, must be called before reading the buffer
  udp.parsePacket();
  if(udp.read(rdp.frame,16)==16) // read packet from client
  {
    Serial.printf("\nRecv data:%f,%f,%f,%f\
n",rdp.sensor[0],rdp.sensor[1],rdp.sensor[2],rdp.sensor[3]);
    //Wait for 1 second
    delay(1000);
    Serial.print("Send ACK packet ");
    udp.beginPacket(udpAddress, udpPort);
    strcpy(sdp.mess,"ACK data");
    udp.write(sdp.frame,16);
    udp.endPacket();  // send packet to client
  }
}
```

1.1.3 Synchronous UDP server with `SoftAP`

```
#include <WiFi.h>
#include <WiFiUdp.h>
/* WiFi network name and password */
//const char * ssid= "Livebox-08B0";
//const char * pwd = "G79ji6dtEptVTPWmZP";
const char * ssid= "ESP32_AP";
const char * pwd = "smartcomputerlab";
const int udpPort = 9999;
//create UDP instance
WiFiUDP udp;
union
{
  uint8_t frame[16];
  float sensor[4];
  char mess[16];
} rdp,sdp;

void setup()
{
  Serial.begin(115200);
  //Connect to the WiFi network
  WiFi.softAP(ssid, pwd);
  Serial.print("\nServer SoftIP: ");
  Serial.println(WiFi.softAPIP());
  udp.begin(udpPort);
}

void loop()
{
  memset(rdp.frame,0,16);
  //processing incoming packet, must be called before reading the buffer
  udp.parsePacket();
  //receive response from server, it will be HELLO WORLD
  if(udp.read(rdp.frame,16)==16)
  {
    IPAddress remaddr;
    remaddr=udp.remoteIP();
    uint16_t remport;
    remport=udp.remotePort();
    Serial.println(remaddr);
    Serial.printf("\nRecv data:%f,%f,%f,%f\
n",rdp.sensor[0],rdp.sensor[1],rdp.sensor[2],rdp.sensor[3]);
    //Wait for 1 second
    delay(1000);
    Serial.print("Send ACK packet ");
    //udp.beginPacket(udpAddress, udpPort);
    udp.beginPacket(remaddr, remport);
    strcpy(sdp.mess,"ACK data");
    udp.write(sdp.frame,16);
    udp.endPacket();
  }
}
```

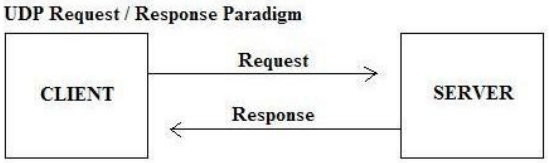
1.1.4 Synchronous UDP Client for server with with `SoftAP`

```
#include <WiFi.h>
#include <WiFiUdp.h>
/* WiFi network name and password */
//const char * ssid= "Livebox-08B0";
//const char * pwd = "G79ji6dtEptVTPWmZP";

const char * ssid= "ESP32_AP";
const char * pwd = "smartcomputerlab";

// IP address to send UDP data to.
// it can be ip address of the server or
// a network broadcast address like below
const char * udpAddress = "192.168.4.255";
const int udpPort = 9999;
//create UDP instance
WiFiUDP udp;
union
{
  uint8_t frame[16];
  float sensor[4];
  char mess[16];
} sdp, rdp; // send and receive packets

void setup()
{
  Serial.begin(115200);
  //Connect to the WiFi network
  WiFi.begin(ssid, pwd);
  Serial.println("");
  // Wait for connection
  while (WiFi.status()!=WL_CONNECTED){delay(500);Serial.print(".");}
  Serial.println("");
  Serial.print("Connected to ");
  Serial.println(ssid);
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
  //This initializes udp and transfer buffer
  udp.begin(udpPort);
}

void loop()
{
  //data will be sent to server
  Serial.println("Send unicast/broadcast packet ");
  udp.beginPacket(udpAddress, udpPort);
  sdp.sensor[0]=0.1;sdp.sensor[1]=0.2;sdp.sensor[2]=0.3;sdp.sensor[3]=0.4;
  udp.write(sdp.frame, 16);
  udp.endPacket();
  memset(rdp.frame,0,16);
  //processing incoming packet, must be called before reading the buffer
  udp.parsePacket();
  //receive response from server, it will be ACK
  if(udp.read(rdp.frame,16)==16)
  {
    Serial.print("Received from server: ");
    Serial.println(rdp.mess);
  }
  //Wait for 2 seconds
  delay(2000);
}
```

To do

1. Analyze the codes and execute them
2. Add the sensor (s) at the client side
3. Add the OLED display at the server side

1.2.1 TCP sockets – establishing connections and sending/receiving data (segments)



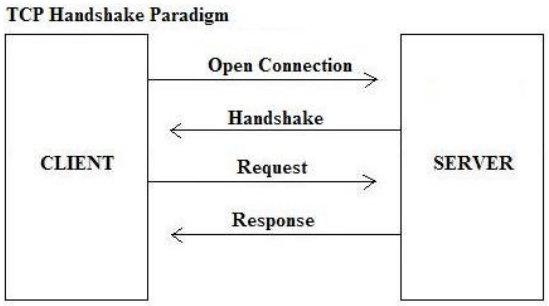


Fig 1.2 TCP **segment** with header

UDP protocol **is connection-less** protocol where the data messages – datagrams may be sent immediately after the initialization of the WiFi connection.

TCP is connection-oriented protocol where the data may be sent after the establishment of a "connection". This connection is identified by the **Initial Sequence Number**.

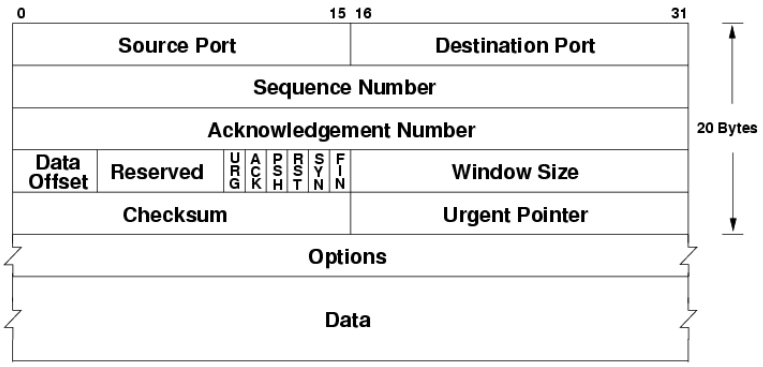


Fig 1.3 TCP **segment** with header

1.2.1.1 TCP server

```
#include <WiFi.h>
const char* ssid = "Livebox-08B0"; // Enter SSID
const char* password = "G79ji6dtEptVTPWmZP"; // Enter Password
/* create a server and listen on port 8088 */
WiFiServer server(8088);
IPAddress local_IP(192, 168, 1, 131);
IPAddress gateway(192, 168, 1, 1);
IPAddress subnet(255, 255, 255, 0);

union
{
  uint8_t frame[16];
  float sensor[4];
  char mess[16];
} sdp, rdp; // send and receive packets

void setup()
{
  Serial.begin(115200);
  Serial.println("\nConnecting to ");
  Serial.println(ssid);
  /* connecting to WiFi */
  WiFi.begin(ssid, password);
  /*wait until ESP32 connect to WiFi*/
  while(WiFi.status()!=WL_CONNECTED)
    {
```

```
      delay(500);Serial.print(".");
    }
  Serial.println("\nConnected to ");
  WiFi.config(local_IP, gateway, subnet);
  Serial.print("Server IP address: ");
  Serial.println(WiFi.localIP());
  /* start Server */
  server.begin();
  delay(400);
  Serial.println("Waiting for client");
}

void loop()
{
uint8_t data[30];
  /* listen for client */
  WiFiClient client = server.available();
  if (client)
  {
    Serial.println("new client");
    /* check client is connected */
    while (client.connected())
    {
      if (client.available())
      {
        int len = client.read(rdp.frame, 16);
        if(len==16)
        {
          Serial.print("client sent: ");
          Serial.println(rdp.sensor[0]);
          strcpy(sdp.mess,"ACK data");
          delay(1000);
          client.write(sdp.frame,16);
        }
      }
    }
  }
}
```

1.2.2.2 TCP client

```
#include <WiFi.h>
/* change ssid and password according to yours WiFi*/
const char* ssid = "Livebox-08B0"; //Enter SSID
const char* password = "G79ji6dtEptVTPWmZP"; //Enter Password
const char* host = "192.168.1.131";
const int port = 8088;

union
{
  uint8_t frame[16];
  float sensor[4];
  char mess[16];
} sdp, rdp; // send and receive packets

void setup()
{
  Serial.begin(115200);
  Serial.print("Connecting to ");
  Serial.println(ssid);
  /* connect to your WiFi */
  WiFi.begin(ssid, password);
/* wait until ESP32 connect to WiFi*/
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected with IP address: ");
  Serial.println(WiFi.localIP());
}

void loop()
{
uint8_t data[30];
int len=0;
```

```
  delay(5000);
  Serial.print("\nConnecting to ");
  Serial.println(host);
  /* Use WiFiClient class to create TCP connections */
  WiFiClient client;
  if (!client.connect(host, port))
  {
    Serial.println("connection failed");
    return;
  }
  Serial.println("connection OK - client sends data: ");
  sdp.sensor[0]=0.1; sdp.sensor[1]=0.2; sdp.sensor[2]=0.3; sdp.sensor[3]=0.4;
  client.write(sdp.frame,16);
  delay(2000);
  len = client.read(rdp.frame,16);
  Serial.println(len);
  if(len==16)
    {
    Serial.print("server sent: ");
    Serial.println(rdp.mess);
    }
  client.stop();
}
```

1.2.2.3 TCP server with `SoftAP`

```
#include <WiFi.h>
const char * ssid= "ESP32_AP";
const char * pwd = "smartcomputerlab";

WiFiServer server(8088); // create a server and listen on port 8088

union
{
  uint8_t frame[16];
  float sensor[4];
  char mess[16];
} sdp, rdp; // send and receive packets

void setup()
{
  Serial.begin(115200);
  WiFi.softAP(ssid, pwd);
  Serial.print("\nServer SoftIP: ");
  Serial.println(WiFi.softAPIP());
  /* start Server */
  server.begin();
  delay(400);
  Serial.println("Waiting for client");
}

void loop()
{
uint8_t data[30];
  /* listen for client */
  WiFiClient client = server.available();
  if (client)
  {
    Serial.println("new client");
    /* check client is connected */
    while (client.connected())
    {
      if (client.available())
      {
        int len = client.read(rdp.frame, 16);
        if(len==16)
        {
          Serial.print("client sent: ");
          Serial.println(rdp.sensor[0]);
          strcpy(sdp.mess,"ACK data");
          delay(1000);
          client.write(sdp.frame,16);
        }
      }
    }
  }
}
```

1.2.2.4 TCP client for server with `SoftAP`

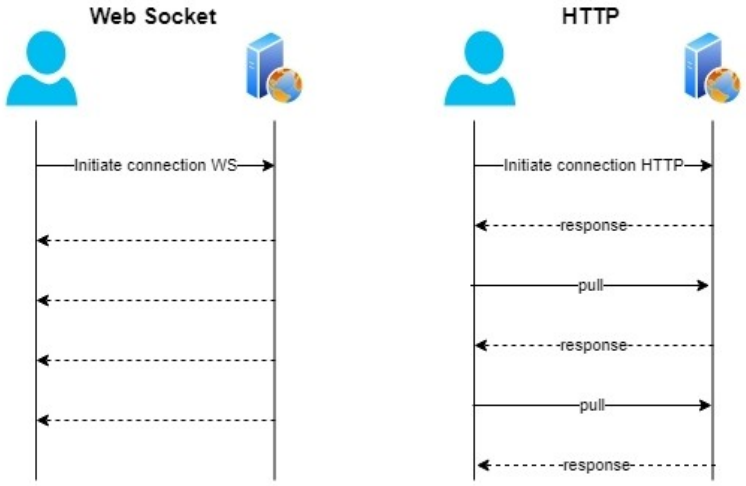
```
#include <WiFi.h>
/* change ssid and password according to yours WiFi*/
//const char * ssid= "Livebox-08B0";
//const char * pwd = "G79ji6dtEptVTPWmZP";

const char * ssid= "ESP32_AP";
const char * pwd = "smartcomputerlab";
const char* host = "192.168.4.1";
const int port = 8088;

union
{
  uint8_t frame[16];
  float sensor[4];
  char mess[16];
} sdp, rdp; // send and receive packets

void setup()
{
  Serial.begin(115200);
  //Connect to the WiFi network
  WiFi.begin(ssid, pwd);
  Serial.println("");
  // Wait for connection
  while (WiFi.status()!=WL_CONNECTED){delay(500);Serial.print(".");}
  Serial.println("");
  Serial.print("Connected to ");
  Serial.println(ssid);
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
  //This initializes udp and transfer buffer
}

void loop()
{
uint8_t data[30];
int len=0;
  delay(5000);
  Serial.print("\nConnecting to ");
  Serial.println(host);
  /* Use WiFiClient class to create TCP connections */
  WiFiClient client;
  if (!client.connect(host, port))
  {
    Serial.println("connection failed");
    return;
  }
  // This will send the data to the server
  Serial.println("connection OK - client sends data: ");
  sdp.sensor[0]=0.1; sdp.sensor[1]=0.2; sdp.sensor[2]=0.3; sdp.sensor[3]=0.4;
  client.write(sdp.frame,16);
  delay(2000);
  len = client.read(rdp.frame,16);
  Serial.println(len);
  if(len==16)
    {
    Serial.print("server sent: ");
    Serial.println(rdp.mess);
    }
  client.stop();
}
```

To do

1. Analyze the codes and execute them
2. Add the sensor (s) at the client side
3. Add the OLED display at the server side

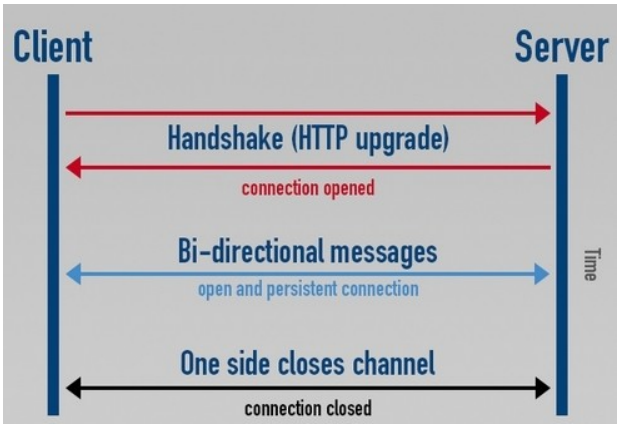
1.3. Web Socket

So the first question is what a web socket is. Web socket is a communication protocol over TCP that enables bidirectional communication between client and server.



As you can see in the diagram above, the client initiate connection over WebSocket or secure WebSocket (**wss),** and then the server can send back messages to the client. With HTTP protocol we do not need to do any pulling. Once the connection is established, the server can send as many requests as it likes.

The greatest benefit of using Web Socket instead of HTTP is performance boost as we do not need to establish a new connection. What is more, we do not need to introduce any sort of pulling mechanism.



Bi-directional Data Exchange/Transfer
By transmitting data in both directions simultaneously through a single connection instead of two, you get to reduce unnecessary network traffic and delay.

Publish/Subscribe Event platform

Send messages to and from a server and receive event-driven responses without having to poll the server for a reply.

1.3.1 Basic Usage of ArduinoWebsockets for ESP32

1.3.1.1 Client

Creating a client and connecting to a server:

```
WebsocketsClient client;
client.connect("ws://your-server-ip:port/uri");
```

Sending a message:

```
client.send("Hello Server!");
```

Waiting for messages:

```
client.onMessage([](WebsocketsMessage msg){
    Serial.prinln("Got Message: " + msg.data());
});
```

1.3.1.2 Server

Creating a server and listening for connections:

```
WebsocketsServer server;
server.listen(8080);
```

Accepting connections:

```
WebsocketsClient client = server.accept();
// handle client as described before :)
```

1.3.2 Full Examples

1.3.2.1 Minimal Esp32 Websockets Client

This sketch:

1. Connects to a WiFi network
2. Connects to a Websockets server
3. Sends the websockets server a message ("`Hello Server`")
4. Prints all incoming messages while the connection is open

```
#include <ArduinoWebsockets.h>
#include <WiFi.h>
const char* ssid = "Livebox-08B0"; // Enter your SSID
const char* password = "G79ji6dtEptVTPWmZP"; // Enter your Password
const char* websockets_server_host = "192.168.1.132"; //Enter server adress
const uint16_t websockets_server_port = 8080; // Enter server port

using namespace websockets;
WebsocketsClient client;

union
{
  char message[17];
  uint8_t frame[17];
  float sensor[4];
} sdf;

void setup() {
    Serial.begin(115200);
    // Connect to wifi
    WiFi.begin(ssid, password);
    // Wait some time to connect to wifi
    for(int i = 0; i < 10 && WiFi.status() != WL_CONNECTED; i++) {
        Serial.print(".");
        delay(1000);
    }
    Serial.println();
```

```
    Serial.print("client connected to WiFi");
    // Connect to server
    client.connect(websockets_server_host, websockets_server_port, "/");
    // Send a message - data
    sdf.sensor[0]=0.1;sdf.sensor[1]=0.2;sdf.sensor[2]=0.3;sdf.sensor[3]=0.4;
    client.send(sdf.message,17);
}

void loop() {
    client.connect(websockets_server_host, websockets_server_port, "/");
    // Send a message - data
    sdf.sensor[0]=0.1;sdf.sensor[1]=0.2;sdf.sensor[2]=0.3;sdf.sensor[3]=0.4;
    client.send(sdf.message,17);
    Serial.print("client sent data");
    client.close();
    delay(3000);
}
```

1.3.2.2 Minimal Esp32 Websockets Server

This sketch:

1. Connects to a WiFi network
2. Starts a websocket server on port 80
3. Waits for connections
4. Once a client connects, it wait for a message from the client
5. Eventually sends an "**ACK**" message to the client
6. Closes the connection and goes back to step 3

```
#include <ArduinoWebsockets.h>
#include <WiFi.h>
const char* ssid = "Livebox-08B0"; // Enter your SSID
const char* password = "G79ji6dtEptVTPWmZP"; // Enter your Password
IPAddress local_IP(192, 168, 1, 132);  // set static values on your network
IPAddress gateway(192, 168, 1, 1);
IPAddress subnet(255, 255, 255, 0);
using namespace websockets;
int sport=8080;

WebsocketsServer server;

union
{
  uint8_t frame[17];
  char message[17];
  float sen[4];
} sdf;

void setup() {
  Serial.begin(115200);
  // Connect to wifi
  WiFi.begin(ssid, password);
  // Wait some time to connect to wifi
  for(int i = 0; i < 15 && WiFi.status() != WL_CONNECTED; i++) {
      Serial.print(".");
      delay(1000);
  }
  Serial.println("\nServer connected");
  WiFi.config(local_IP, gateway, subnet);
  Serial.printf("Server port: %d and IP address: ",sport);
  Serial.println(WiFi.localIP());
  Serial.println("");
  server.listen(sport);
  Serial.print("Is server live? ");
  Serial.println(server.available());
}

void loop() {
  WebsocketsClient client = server.accept();
  if(client.available()) {
    WebsocketsMessage msg = client.readBlocking();
    // log
    Serial.println("\nGot Message: ");
    msg.data().toCharArray(sdf.message,17);
```

```
    Serial.printf("Sensors: %2.2f,%2.2f,%2.2f,%2.2f",sdf.sen[0],sdf.sen[1],sdf.sen[2],sdf.sen[3]);
    client.send("ACK message");
    // close the connection
    client.close();
  }
  delay(1000);
}
```

To do

1. Analyze the codes and execute them
2. Add the sensor (s) at the client side
3. Add the OLED display at the server side
4. Use **`SoftAP`** for server and client connecting to this server

IoT NetLab 2

Low level WiFi

2.0 Introduction

In this lab we are going to deepen our practice of WiFi protocols starting with the analysis of WiFi frames and the code for beacon generator.

Fig 2.1 WiFi frame with control fields and 3 (4) MAC adresses

```
typedef struct {
  unsigned frame_ctrl:16;
  unsigned duration_id:16;
  uint8_t addr1[6]; /* receiver address */
  uint8_t addr2[6]; /* sender address */
  uint8_t addr3[6]; /* filtering address */
  unsigned sequence_ctrl:16;
  uint8_t addr4[6]; /* optional */
} wifi_ieee80211_mac_hdr_t;

typedef struct {
  wifi_ieee80211_mac_hdr_t hdr;
  uint8_t payload[0]; /* network data ended with 4 bytes csum (CRC32) */
} wifi_ieee80211_packet_t;
```

The above picture shows the format of WiFi frame , it consists of the **header** and the **payload** part.
The **header** contains the **control** field, the **duration** field, the place for **four MAC addresses** , and the **sequence control** field.

Type Value B3..B2	Type Description	Subtype Value B7 .. b4	Subtype Description
00	Management	0000	Association Request
00	Management	0001	Association Response
00	Management	0010	Reassociation Request
00	Management	0011	Reassociation Response
00	Management	0100	Probe Request
00	Management	0101	Probe Response
00	Management	0110	Timing Advertisement
00	Management	0111	Reserved
00	Management	1000	Beacon
00	Management	1001	ATIM
00	Management	1010	Disassociation
00	Management	1011	Authentication
00	Management	1100	Deauthentication
00	Management	1101	Action
00	Management	1110	Action No Ack (NACK)
00	Management	1111	Reserved

Fig 2.2 WiFi Management MAC frames

2.1.1 WiFi frame (`beacon`) generator

The complete header of the MAC frame is coded as follows:

```
uint8_t beacon_raw[] = {
  0x80,0x00,      // 0-1: Frame Control: subtype 80 - beacon, type 00 MNGM
  0x00,0x00,      // 2-3: Duration
  0xff,0xff,0xff,0xff,0xff,0xff, // 4-9: Destination address (broadcast)
  0xba,0xde,0xaf,0xfe,0x00,0x06, // 10-15: Source address
  0xba,0xde,0xaf,0xfe,0x00,0x06, // 16-21: BSSID
  0x00,0x00,              // 22-23: Sequence / fragment number
  0x00,0x01,0x02,0x03,0x04, 0x05, 0x06, 0x07,  // 24-31: Timestamp
  0x64,0x00,              // 32-33: Beacon interval
  0x31,0x04,              // 34-35: Capability info
  0x00,0x00,      // 36-38: SSID parameter set, 0x00:length:content
  0x01,0x08,0x82,0x84,0x8b,0x96,0x0c,0x12,0x18,0x24, // 39-48: Supported rates
  0x03,0x01,0x01,      // 49-51: DS Parameter set, current channel 1 (= 0x01),
  0x05,0x04,0x01,0x02,0x00,0x00,       // 52-57: Traffic Indication Map
};
```

The **names** of the proposed **ssid** are prepared in `my_ssid[]` table. They are attached to the header before sending the frame on the line.

```
char *my_ssids[] = {
  "01 Hello from SmartComputerLab",
  "02 Hello from SmartComputerLab",
  "03 Hello from SmartComputerLab",
  "04 Hello from SmartComputerLab",
};
```

The constant indicating the position of the selected fields in the raw frame are give below:

```
#define BEACON_SSID_OFFSET 38
#define SRCADDR_OFFSET 10
#define BSSID_OFFSET 16
#define SEQNUM_OFFSET 22
#define TOTAL_LINES (sizeof(my_ssids) / sizeof(char *))
```

`spam_task` is running in the infinite `for(;;)` loop.
Before generating a new frame it waits for the `100/TOTAL_LINES/portTICK_PERIOD_MS` period.
The initial place of the frame is `beacon_rick[200]` table of bytes `(uint8_t)`. This table is filled in with `beacon_raw` header followed by the name of **ssid**.

```
void spam_task(void *pvParameter) {
uint8_t line = 0;
// Keep track of beacon sequence numbers on a per line-basis
uint16_t seqnum[TOTAL_LINES] = { 0 };
for (;;) {
  vTaskDelay(100/TOTAL_LINES/portTICK_PERIOD_MS);
  printf("%i %i %s\r\n",strlen(my_ssids[line]),TOTAL_LINES,my_ssids[line]);
  uint8_t beacon_rick[200];
  memcpy(beacon_rick,beacon_raw,BEACON_SSID_OFFSET-1);
  beacon_rick[BEACON_SSID_OFFSET-1]=strlen(my_ssids[line]);
  memcpy(&beacon_rick[BEACON_SSID_OFFSET],my_ssids[line],
strlen(my_ssids[line]));
  memcpy(&beacon_rick[BEACON_SSID_OFFSET+strlen(my_ssids[line])],
&beacon_raw[BEACON_SSID_OFFSET],sizeof(beacon_raw)-BEACON_SSID_OFFSET);
```

The last byte of source address and **BSSID** is the line number to **emulate the multiple broadcasting Access Points**.

```
    beacon_rick[SRCADDR_OFFSET+5]=line;
    beacon_rick[BSSID_OFFSET+5]=line;
```

The **sequence number** field is calculated as follows:

```
beacon_rick[SEQNUM_OFFSET] = (seqnum[line] & 0x0f) << 4;
beacon_rick[SEQNUM_OFFSET + 1] = (seqnum[line] & 0xff0) >> 4;
seqnum[line]++;
if(seqnum[line] > 0xfff) seqnum[line] = 0;
```

Finally the frame-beacon is **sent** by:

```
esp_wifi_80211_tx(WIFI_IF_AP,beacon_rick,sizeof(beacon_raw)+ strlen(my_ssids[line]),false);
```

The new line number (ssid) is incremented :

```
if (++line >= TOTAL_LINES) line = 0;
```

At some point, the task execution and the esp32 itself go into **deep sleep** state. This solution allows for a lower power consumption of the device. There are no operations in the main **loop() task.**

```
if(count<0)
    {
    esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
    esp_deep_sleep_start(); count=30;
    }
    else count--;
  }
}
```

The **setup()** function initializes the required functions (drivers) dealing with the WiFi interface:

```
void setup(void) {
  Serial.begin(9600);
  nvs_flash_init();
  tcpip_adapter_init();
  wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
  esp_event_loop_init(event_handler, NUL);
  esp_wifi_init(&cfg);
  esp_wifi_set_storage(WIFI_STORAGE_RAM);
```

We initialize a **dummy AP** to specify a channel and get WiFi hardware into a mode where we can send the actual fake beacon frames.

```
esp_wifi_set_mode(WIFI_MODE_AP);
esp_wifi_start();
esp_wifi_set_ps(WIFI_PS_NONE);
```

Finally we launch the **`spam_task`**.

```
  xTaskCreate(&spam_task, "spam_task", 2048, NULL, 5, NULL);
}

void loop()
{}
```

2.1.1.1 Complete code

```
#include "freertos/FreeRTOS.h"
#include "esp_wifi.h"
#include "esp_wifi_types.h"
#include "esp_system.h"
#include "esp_event.h"
#include "esp_event_loop.h"
#include "nvs_flash.h"
#include "driver/gpio.h"
#define uS_TO_S_FACTOR 1000000  /* Conversion factor to seconds */
#define TIME_TO_SLEEP  15       /* Time ESP32 will go to sleep (in seconds) */

esp_err_t esp_wifi_80211_tx(wifi_interface_t ifx, const void *buffer, int len, bool en_sys_seq);

uint8_t beacon_raw[] = {
  0x80, 0x00,       // 0-1: Frame Control: subtype 80 - beacon, type 00 MNGM
  0x00, 0x00,       // 2-3: Duration
  0xff, 0xff, 0xff, 0xff, 0xff, 0xff,       // 4-9: Destination address (broadcast)
  0xba, 0xde, 0xaf, 0xfe, 0x00, 0x06,       // 10-15: Source address
  0xba, 0xde, 0xaf, 0xfe, 0x00, 0x06,       // 16-21: BSSID
  0x00, 0x00,               // 22-23: Sequence / fragment number
  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,   // 24-31: Timestamp
  0x64, 0x00,               // 32-33: Beacon interval
  0x31, 0x04,               // 34-35: Capability info
  0x00, 0x00,  // 36-38: SSID parameter set, 0x00:length:content
  0x01,0x08,0x82,0x84,0x8b,0x96,0x0c,0x12,0x18,0x24, // 39-48: Supported rates
```

```
  0x03, 0x01, 0x01,    // 49-51: DS Parameter set, current channel 1 (= 0x01),
  0x05, 0x04, 0x01, 0x02, 0x00, 0x00,  // 52-57: Traffic Indication Map
};

char *my_ssids[] = {                   // dummy access points
  "01 Hello from SmartComputerLab",
  "02 Hello from SmartComputerLab",
  "03 Hello from SmartComputerLab",
  "04 Hello from SmartComputerLab",
};

#define BEACON_SSID_OFFSET 38
#define SRCADDR_OFFSET 10
#define BSSID_OFFSET 16
#define SEQNUM_OFFSET 22
#define TOTAL_LINES (sizeof(my_ssids) / sizeof(char *))

esp_err_t event_handler(void *ctx, system_event_t *event) {
  return ESP_OK;
}

int count=120;

void spam_task(void *pvParameter) {
  uint8_t line = 0;
  // Keep track of beacon sequence numbers on a per-songline-basis
  uint16_t seqnum[TOTAL_LINES] = { 0 };
  for (;;) {
    vTaskDelay(100 / TOTAL_LINES / portTICK_PERIOD_MS);
    printf("%i %i %s\r\n", strlen(my_ssids[line]), TOTAL_LINES, my_ssids[line]);
    uint8_t beacon_rick[200];
    memcpy(beacon_rick, beacon_raw, BEACON_SSID_OFFSET - 1);
    beacon_rick[BEACON_SSID_OFFSET - 1] = strlen(my_ssids[line]);
    memcpy(&beacon_rick[BEACON_SSID_OFFSET], my_ssids[line], strlen(my_ssids[line]));
    memcpy(&beacon_rick[BEACON_SSID_OFFSET + strlen(my_ssids[line])],
&beacon_raw[BEACON_SSID_OFFSET], sizeof(beacon_raw) - BEACON_SSID_OFFSET);
    // Last byte of source address/BSSID will be line number
    beacon_rick[SRCADDR_OFFSET + 5] = line;
    beacon_rick[BSSID_OFFSET + 5] = line;
    // Update sequence number
    beacon_rick[SEQNUM_OFFSET] = (seqnum[line] & 0x0f) << 4;
    beacon_rick[SEQNUM_OFFSET + 1] = (seqnum[line] & 0xff0) >> 4;
    seqnum[line]++;
    if (seqnum[line] > 0xfff)
      seqnum[line] = 0;
    esp_wifi_80211_tx(WIFI_IF_AP, beacon_rick, sizeof(beacon_raw) + strlen(my_ssids[line]), false);
    if (++line >= TOTAL_LINES)
      line = 0;
  if(count<0)
    {
    esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
    esp_deep_sleep_start(); count=30;
    }
    else count--;
  }
}

void setup(void) {
  Serial.begin(9600);
  nvs_flash_init();
  tcpip_adapter_init();
  wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
  ESP_ERROR_CHECK(esp_event_loop_init(event_handler, NULL));
  ESP_ERROR_CHECK(esp_wifi_init(&cfg));
  ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
  // Init dummy AP
  ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_AP));
  ESP_ERROR_CHECK(esp_wifi_start());
  ESP_ERROR_CHECK(esp_wifi_set_ps(WIFI_PS_NONE));
  xTaskCreate(&spam_task, "spam_task", 2048, NULL, 5, NULL);
}

void loop()
{
}
```

To do:

1. Analyze the above code
2. Modify the name(s) of dumb Access Points generated by the program

2.2 WiFi sniffer

In this section we are going to build WiFi sniffer that operates in promiscuous mode. The sniffer scans all WiFi channels (13 in France) for the presence of MAC frames.
When it captures the frame it analyzes the header content looking for the sender and receiver addresses. Then it manages a counter that stores different addresses.


```
typedef struct {
  unsigned frame_ctrl:16;
  unsigned duration_id:16;
  uint8_t addr1[6]; /* receiver address */
  uint8_t addr2[6]; /* sender address */
  uint8_t addr3[6]; /* filtering address */
  unsigned sequence_ctrl:16;
  uint8_t addr4[6]; /* optional */
} wifi_ieee80211_mac_hdr_t;

typedef struct {
  wifi_ieee80211_mac_hdr_t hdr;
  uint8_t payload[0]; /* network data ended with 4 bytes csum (CRC32) */
} wifi_ieee80211_packet_t;
```

The above picture shows the format of WiFi frame , it consists of the **header** and the **payload** part.
The **header** contains the **control** field, the **duration** field, the place for **four MAC addresses** , and the **sequence control** field.

2.2.1 WiFi interface functions and sniffer client elements

The WiFi packets are captured by the WiFi interface operating in **promiscuous** mode set by:

```
esp_wifi_set_promiscuous(true);
```

The **initialization** of the WiFi interface is done by the following functions:

```
void wifi_sniffer_init(void)
{
  nvs_flash_init();
  tcpip_adapter_init();
  ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );
  wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
  ESP_ERROR_CHECK( esp_wifi_init(&cfg) );
  ESP_ERROR_CHECK( esp_wifi_set_country(&wifi_country) ); /* set country for channel range [1,13] */
  ESP_ERROR_CHECK( esp_wifi_set_storage(WIFI_STORAGE_RAM) );
  ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_NULL) );
  ESP_ERROR_CHECK( esp_wifi_start() );
  esp_wifi_set_promiscuous(true);
  esp_wifi_set_promiscuous_rx_cb(&wifi_sniffer_packet_handler);
}
```

When a frame arrives the **`wifi_sniffer_packet_handler`** function is evoked. Note that in this case only **management** frames are selected.

```
void wifi_sniffer_packet_handler(void* buff, wifi_promiscuous_pkt_type_t type)
{
  if (type != WIFI_PKT_MGMT)
    return;

  const wifi_promiscuous_pkt_t *ppkt = (wifi_promiscuous_pkt_t *)buff;
  const wifi_ieee80211_packet_t *ipkt = (wifi_ieee80211_packet_t *)ppkt->payload;
  const wifi_ieee80211_mac_hdr_t *hdr = &ipkt->hdr;
..
```

The header part is received in the **`ipkt->hdr`** part of the **`buff`**(er). The sniffed **channel** and the received **signal strength** are available in **`ppkt->rx_ctrl.channel`** and **`ppkt->rx_ctrl.rssi`** .

Our program analysis the received header and compares the source address (**hdr->addr2**) with the already received frames with the same address.
Only the **new address is retained and stored** in the MAC table **uint8_t tmac[512][6];** that is a part of the **udp union** and **pack struct** presented below.
The additional bytes **[6]** and **[7]** of each table element are used to store the corresponding **RSSI value** and **channel number**.
The additional bytes **[6]** and **[7]** of each table element are used to store the corresponding **RSSI value** and **channel number**.

```
union
  {
  uint8_t frame[4120]; // 512*(7) + 2*6 + 3*4 =
  struct
    {
    int ti; int minRSSI; int maxRSSI;
    uint8_t minmac[6];
    uint8_t maxmac[6];
    uint8_t tmac[512][8];  // -RSSI , channel
    } pack;
  } udp;     // UDP packet
```

The value of **udp.pack.ti** is a counter that indicates the number of different **MAC** addresses, **RSSI**, and channel numbers stored in the **tmac[]** table.

The execution of the whole program is activated by the **wifi_sniffer_init();** in the **setup()** function and synchronized by the following code in the **loop**() task.

```
vTaskDelay(WIFI_CHANNEL_SWITCH_INTERVAL/portTICK_PERIOD_MS);
wifi_sniffer_set_channel(channel);
channel = (channel % WIFI_CHANNEL_MAX) + 1;
```

The program modifies the **number** of the sniffed channel and indicates the **channel switch interval that is calculated as:**

WIFI_CHANNEL_SWITCH_INTERVAL/portTICK_PERIOD_MS

The collected MAC addresses are refreshed cyclically to follow the evolution of the presence of the devices in the given area that is delimited by the signal strength (from **-40** to **-120**) provided in the initial phase of the execution of the program.

2.2.2 Complete code

```
#include "freertos/FreeRTOS.h"
#include "esp_wifi.h"
#include "esp_wifi_types.h"
#include "esp_system.h"
#include "esp_event.h"
#include "esp_event_loop.h"
#include "nvs_flash.h"
#include "driver/gpio.h"
#define LED_GPIO_PIN                    22
#define WIFI_CHANNEL_SWITCH_INTERVAL  (500)
#define WIFI_CHANNEL_MAX               (13)
uint8_t level = 0, channel = 1;

static wifi_country_t wifi_country = {.cc="FR", .schan = 1, .nchan = 13};

typedef struct {
  unsigned frame_ctrl:16;
  unsigned duration_id:16;
  uint8_t addr1[6]; /* receiver address */
  uint8_t addr2[6]; /* sender address */
  uint8_t addr3[6]; /* filtering address */
  unsigned sequence_ctrl:16;
  uint8_t addr4[6]; /* optional */
} wifi_ieee80211_mac_hdr_t;

typedef struct {
  wifi_ieee80211_mac_hdr_t hdr;
  uint8_t payload[0]; /* network data ended with 4 bytes csum (CRC32) */
} wifi_ieee80211_packet_t;
```

```
static esp_err_t event_handler(void *ctx, system_event_t *event);
static void wifi_sniffer_init(void);
static void wifi_sniffer_set_channel(uint8_t channel);
static const char *wifi_sniffer_packet_type2str(wifi_promiscuous_pkt_type_t type);
static void wifi_sniffer_packet_handler(void *buff, wifi_promiscuous_pkt_type_t type);

esp_err_t event_handler(void *ctx, system_event_t *event)
{
  return ESP_OK;
}

void wifi_sniffer_init(void)
{
  nvs_flash_init();
  tcpip_adapter_init();
  ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );
  wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
  ESP_ERROR_CHECK( esp_wifi_init(&cfg) );
  ESP_ERROR_CHECK( esp_wifi_set_country(&wifi_country) ); /* set country for channel range [1, 13]
*/
  ESP_ERROR_CHECK( esp_wifi_set_storage(WIFI_STORAGE_RAM) );
  ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_NULL) );
  ESP_ERROR_CHECK( esp_wifi_start() );
  esp_wifi_set_promiscuous(true);
  esp_wifi_set_promiscuous_rx_cb(&wifi_sniffer_packet_handler);
}

void wifi_sniffer_set_channel(uint8_t channel)
{
  esp_wifi_set_channel(channel, WIFI_SECOND_CHAN_NONE);
}

const char * wifi_sniffer_packet_type2str(wifi_promiscuous_pkt_type_t type)
{
  switch(type) {
  case WIFI_PKT_MGMT: return "MGMT";
  case WIFI_PKT_DATA: return "DATA";
  default:
  case WIFI_PKT_MISC: return "MISC";
  }
}

uint8_t tmac[100][6]; int ti=0; int match=0;
uint8_t minmac[6],maxmac[6];

bool clearmac()
{
  int i=0;
  for(i=0;i<ti;i++) memset(tmac[i],0x00,6);
  ti=0;
  return 0;
}

bool cmpmac(uint8_t *mac1,uint8_t *mac2)
{
  int i=0;
  for(i=0;i<6;i++) if(mac1[i]!=mac2[i]) return false;
  return true;
}

void wifi_sniffer_packet_handler(void* buff, wifi_promiscuous_pkt_type_t type)
{
  if (type != WIFI_PKT_DATA)
    return;
  const wifi_promiscuous_pkt_t *ppkt = (wifi_promiscuous_pkt_t *)buff;
  const wifi_ieee80211_packet_t *ipkt = (wifi_ieee80211_packet_t *)ppkt->payload;
  const wifi_ieee80211_mac_hdr_t *hdr = &ipkt->hdr;
  printf("PACKET TYPE=%s, CHAN=%02d, RSSI=%02d,"
    " ADDR1=%02x:%02x:%02x:%02x:%02x:%02x,"
    " ADDR2=%02x:%02x:%02x:%02x:%02x:%02x,"
    " ADDR3=%02x:%02x:%02x:%02x:%02x:%02x\n",
    wifi_sniffer_packet_type2str(type),
    ppkt->rx_ctrl.channel,
    ppkt->rx_ctrl.rssi,
    /* ADDR1 */
    hdr->addr1[0],hdr->addr1[1],hdr->addr1[2],
```

```
    hdr->addr1[3],hdr->addr1[4],hdr->addr1[5],
    /* ADDR2 */
    hdr->addr2[0],hdr->addr2[1],hdr->addr2[2],
    hdr->addr2[3],hdr->addr2[4],hdr->addr2[5],
    /* ADDR3 */
    hdr->addr3[0],hdr->addr3[1],hdr->addr3[2],
    hdr->addr3[3],hdr->addr3[4],hdr->addr3[5]
  );
  //printf(" number addr2 =%d\n",ti);

 match=0;
 for(int j=0;j<ti;j++)
   {
    if(cmpmac((uint8_t *)hdr->addr2,tmac[j])) match=1;
    else continue;
   }
 if(match==0)
   {
    if(ti==99)ti=0;
    memcpy(tmac[ti],(uint8_t *)hdr->addr2,6);
    if(hdr->addr1[0]!= 0xFF) ti++;  // we may exclude broadcast frames
   }
   printf("  addr2 number=%d\n",ti);
}

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 5 as an output.
  Serial.begin(115200);
  delay(10);
  wifi_sniffer_init();
  pinMode(LED_GPIO_PIN, OUTPUT);
}

void loop() {
  //Serial.print("inside loop");
  delay(1000); // wait for a second
  if (digitalRead(LED_GPIO_PIN) == LOW)
    digitalWrite(LED_GPIO_PIN, HIGH);
  else
    digitalWrite(LED_GPIO_PIN, LOW);
  vTaskDelay(WIFI_CHANNEL_SWITCH_INTERVAL/portTICK_PERIOD_MS);
  wifi_sniffer_set_channel(channel);
  channel = (channel % WIFI_CHANNEL_MAX) + 1;
}
```

The terminal output:

```
..
PACKET TYPE=DATA, CHAN=01, RSSI=-81, ADDR1=01:80:c2:00:00:13, ADDR2=78:81:02:31:08:b0,
ADDR3=78:81:02:31:08:b0
  addr2 number=10
PACKET TYPE=DATA, CHAN=01, RSSI=-44, ADDR1=78:81:02:31:08:b0, ADDR2=c8:c9:a3:d1:96:60,
ADDR3=78:81:02:31:08:b0
  addr2 number=10
PACKET TYPE=DATA, CHAN=01, RSSI=-83, ADDR1=ff:ff:ff:ff:ff:ff, ADDR2=78:81:02:31:08:b0,
ADDR3=c8:c9:a3:d1:96:60
  addr2 number=10
PACKET TYPE=DATA, CHAN=01, RSSI=-45, ADDR1=78:81:02:31:08:b0, ADDR2=c8:c9:a3:d1:96:60,
ADDR3=78:81:02:31:08:b0
  addr2 number=10
...
PACKET TYPE=DATA, CHAN=01, RSSI=-51, ADDR1=78:81:02:31:08:b0, ADDR2=c8:c9:a3:d1:96:60,
ADDR3=ff:ff:ff:ff:ff:ff
  addr2 number=10
PACKET TYPE=DATA, CHAN=01, RSSI=-45, ADDR1=78:81:02:31:08:b0, ADDR2=c8:c9:a3:d1:96:60,
ADDR3=78:81:02:31:08:b0
  addr2 number=10
```

To do

1. Analyze the code
2. Mod**ify** the type of frames to be analyzed (displayed)
3. Add an OLED display to show the number of different active stations (**`addr2`** with **DATA** frames)

IoT NetLab 3

Direct WiFi (ESP-NOW) and Long Range WiFi

3.0 Introduction

ESP-NOW is a protocol developed by **Espressif**, which enables multiple devices to communicate with one another without using high level WiFi protocols . The protocol is similar to the low-power 2.4GHz wireless connectivity that is often deployed in wireless mouses. So, the pairing between devices is needed prior to their communication. After the pairing is done, the connection is secure and peer-to-peer, with no handshake being required.

This means that after pairing a device with each other, the connection is persistent. In other words, if suddenly one of your boards loses power or resets, when it restarts, it will automatically connect to its peer to continue the communication.

ESP-NOW supports the following features:

- Encrypted and un-encrypted unicast communication;
- Mixed encrypted and un-encrypted peer devices;
- Up to 250-byte payload can be carried;
- Sending callback function that can be set to inform the application layer of transmission success or failure.

ESP-NOW technology also has the following limitations:

- Limited encrypted peers. 10 encrypted peers at the most are supported in **`STA`** mode; 6 at the most in **`SoftAP`** or **`SoftAP + STA`**;
- Multiple unencrypted peers are supported, however, their total number should be less than 20, including encrypted peers;
- Payload is limited to 250 bytes.

In simple words, ESP-NOW is a fast communication protocol that can be used to exchange small messages (up to 250 bytes) between ESP32 boards.

ESP-NOW is very versatile and you can have one-way or two-way communication in different setups.

ESP-NOW protocol allows us to send the **WiFi frames identified** by the **MAC** address. The following code shows how to get the MAC address of the board.

```
#include "WiFi.h"
void setup()
{
  Serial.begin(9600);
  WiFi.mode(WIFI_MODE_STA);
  Serial.println();
  Serial.print("My MAC address is: ");
  Serial.println(WiFi.macAddress());
}

void loop()
{
}
```

3.1 Simple server-client example

Our first example shows how to send and receive the ESP-NOW packets. The sender (Terminal) captures two values from DHT22 sensor: temperature and humidity and sends them in a ESP-NOW packet

3.1.1 The sender

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

// REPLACE WITH YOUR RECEIVER MAC Address
uint8_t broadcastAddress[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
// Structure example to send data
// Must match the receiver structure
union
  {
  uint8_t  frame[16];
  int sensor[4];
  } pack;

esp_now_peer_info_t peerInfo;

// callback when data is sent
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
  Serial.print("\r\nLast Packet Send Status:\t");
  Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" : "Delivery Fail");
}
void setup() {
  // Init Serial Monitor
  Serial.begin(115200);

  // Set device as a Wi-Fi Station
  WiFi.mode(WIFI_STA);
  // Init ESP-NOW
  if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
  }
  // Once ESPNow is successfully Init, we will register for Send CB to
  // get the status of Trasnmitted packet
  esp_now_register_send_cb(OnDataSent);
  // Register peer
  memcpy(peerInfo.peer_addr, broadcastAddress, 6);
  peerInfo.channel = 0;
  peerInfo.encrypt = false;
  // Add peer
  if (esp_now_add_peer(&peerInfo) != ESP_OK){
    Serial.println("Failed to add peer");
    return;
  }
}
void loop() {
  // Set values to send
  pack.sensor[0]= random(1,20);
  pack.sensor[1]= random(1,90);
  // Send message via ESP-NOW
  esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) pack.frame, 16);
  if (result == ESP_OK) {
    Serial.println("Sent with success");
  }
  else {
    Serial.println("Error sending the data");
  }
  delay(2000);
}
```

3.1.2 The receiver

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

union
  {
  uint8_t  frame[16];
  int sensor[4];
```

```
  } pack;

// callback function that will be executed when data is received
void OnDataRecv(const uint8_t *mac, const uint8_t *buff, int len)
{
  memcpy(pack.frame, buff,len);
  Serial.print("Bytes received: ");
  Serial.println(len);
  Serial.print("Int: ");
  Serial.println(pack.sensor[0]);
  Serial.print("Int: ");
  Serial.println(pack.sensor[1]);
  Serial.println();
}

void setup() {
  // Initialize Serial Monitor
  Serial.begin(115200);
  // Set device as a Wi-Fi Station
  WiFi.mode(WIFI_STA);
  // Init ESP-NOW
  if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
  }

  // Once ESPNow is successfully Init, we will register for recv CB to
  // get recv packer info
  esp_now_register_recv_cb(OnDataRecv);
}

void loop() {
}
```

3.1.3 Building receiver-gateway node to `ThingSpeak`

The following example presents the code to create a receiver-gateway node with two WiFi modes, one for the reception of the ESP-NOW packets (AP mode) , and one for the transmission of the received data via WiFi connection to a given Access Point (STA mode).

The sender-terminal code is similar to the preceding example (29.1.1). It is completed with a function that determines the WiFi channel number used by the gateway-receiver to communicate with the Access Point.

The **radio channel used by WiFi and ESP-NOW must the same**.

```
#include <esp_now.h>
#include <WiFi.h>
#include "ThingSpeak.h"

WiFiClient  client;
QueueHandle_t queue;

const char * ssid= "Livebox-08B0";
const char * pwd = "G79ji6dtEptVTPWmZP";
//static const char* ssid     = "PhoneAP";
//static const char* password = "smartcomputerlab";

// callback function that will be executed when data is received
void OnDataRecv(const uint8_t *mac, const uint8_t *buff, int len)
{
  union
  {
  uint8_t  frame[16];
  int sensor[4];
  } pack;

  memcpy(pack.frame, buff,len);
  Serial.print("Bytes received: ");
  Serial.println(len);
  Serial.print("Int: ");
  Serial.println(pack.sensor[0]);
  Serial.print("Int: ");
  Serial.println(pack.sensor[1]);
```

```
  Serial.println();
  xQueueReset(queue);
  xQueueSend(queue, pack.frame, 0);
}

void setup() {
  // Initialize Serial Monitor
  Serial.begin(115200);Serial.println();
  // Set device as a Wi-Fi Station for WiFi and AP for ESP-NOW
  WiFi.mode(WIFI_AP_STA);
  WiFi.begin(ssid, pwd);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.print(".");
  }
  Serial.println(); Serial.println("WiFi connected");
  ThingSpeak.begin(client);  // Initialize ThingSpeak
  queue = xQueueCreate(10,16);  // 10 slots for 4 integer

  // Init ESP-NOW
  if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
  }
  // Once ESPNow is successfully Init, we will register for recv CB to
  // get recv packer info
  esp_now_register_recv_cb(OnDataRecv);
}

void loop() {
  union
  {
  uint8_t  frame[16];
  int sensor[4];
  } rpack;

  xQueueReceive(queue, rpack.frame, portMAX_DELAY);
  ThingSpeak.setField(1, rpack.sensor[0]);
  ThingSpeak.setField(2, rpack.sensor[1]);
  // write to the ThingSpeak channel
  int x = ThingSpeak.writeFields(1626377,  "3IN09682SQX3PT4Z" );
  if(x == 200){
    Serial.println("Channel update successful.");
  }
  else{
    Serial.println("Problem updating channel. HTTP error code " + String(x));
  }
  delay(16000);
}
```

To do

1. Test the presented examples.
2. Add physical sensor to the terminal (ESP-NOW sender node)
3. Complete the above applications with the OLED screen attached to the Gateway and Terminal nodes.

3.1.5 Building receiver-gateway node to an MQTT broker

The following code example implements receiver-gateway (WiFi-MQTT) node. The received ESP-NOW packets are re-transmitted to the MQTT server "5.196.95.208" or test.mosquitto.org.
We use the same AP node for the receiver (ThingSpeak to ESP-NOW) node. Note that different AP may be used only if the WiFi channel number is the same (in our case channel number 6).

```
#include <esp_now.h>
#include <WiFi.h>
#include <MQTT.h>

WiFiClient  net;
MQTTClient client;

QueueHandle_t queue;

const char * ssid= "Livebox-08B0";
const char * pwd = "G79ji6dtEptVTPWmZP";
//static const char* ssid     = "PhoneAP";
//static const char* password = "smartcomputerlab";

const char* mqttServer = "broker.emqx.io";


void OnDataRecv(const uint8_t *mac, const uint8_t *buff, int len)
{
  union
  {
  uint8_t  frame[16];
  int sensor[4];
  } pack;

  memcpy(pack.frame, buff,len);
  Serial.print("Bytes received: ");
  Serial.println(len);
  Serial.print("Int: ");
  Serial.println(pack.sensor[0]);
  Serial.print("Int: ");
  Serial.println(pack.sensor[1]);
  Serial.println();
  xQueueReset(queue);
  xQueueSend(queue, pack.frame, 0);
}

void messageReceived(String &topic, String &payload) {
  Serial.println("incoming: " + topic + " - " + payload);
  // send LoRa message depending on topic
}


void connect() {
  char cbuff[128];
  Serial.print("checking wifi...");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");
    delay(1000);
  }
  Serial.print("\nconnecting...");
  while (!client.connect("IoT.GW1")) {
    Serial.print("."); delay(1000);
  }
  Serial.println("\nIoT.GW1 - connected!");
  client.subscribe("/esp32_GW1/Test");
  delay(1000);

}


void setup() {
  // Initialize Serial Monitor
  Serial.begin(115200);Serial.println();
  // Set device as a Wi-Fi Station for WiFi and AP for ESP-NOW
  WiFi.mode(WIFI_AP_STA);
  WiFi.begin(ssid, pwd);
  while (WiFi.status() != WL_CONNECTED) {
```

```
    delay(1000);
    Serial.print(".");
  }
  delay(200);
  Serial.println();
  Serial.println("WiFi connected");
  queue = xQueueCreate(10,16);  // 10 slots for 4 integer

  // Init ESP-NOW
  if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
  }
  // Once ESPNow is successfully Init, we will register for recv CB to
  // get recv packer info
  esp_now_register_recv_cb(OnDataRecv);
  client.begin(mqttServer, net);
  client.onMessage(messageReceived);
  connect();
}

unsigned long lastMillis = 0;

void loop()
{
  union
  {
  uint8_t  frame[32];
  int sensor[4];
  } rpack;

  char buff[64];

 // client.loop();
  delay(10);  // <- fixes some issues with WiFi stability
  xQueueReceive(queue, rpack.frame, portMAX_DELAY);
  Serial.print("\nconnecting...");
  if (!client.connected()) { connect(); }
    if (millis() - lastMillis > 5000) {   // publish a message every 5 seconds
    lastMillis = millis();
    sprintf(buff,"Temp:%d, Humi:%d\n",rpack.sensor[0],rpack.sensor[1]);
    client.publish("/esp32_GW1/Test",buff);
    Serial.println("in the loop");
    delay(2000);
    }
}
```

The code is tested with the mosquitto_sub command on Ubuntu host.

```
bako@bako-SH67H3:~$ mosquitto_sub -h "broker.emqx.io" -t /esp32_GW1/Test
Temp:56, Humi:0
Temp:67, Humi:0
Temp:30, Humi:0
Temp:33, Humi:0
Temp:85, Humi:0
```

3.1.6 Long Range WiFi-Lora

```
// master.ino

#include <Arduino.h>
#include <WiFi.h>
#include <WiFiUdp.h>
#include <esp_wifi.h>

const char* ssid = "LongRange";//AP ssid
const char* password = "smartcomputerlab";//AP password

const char* ssidRouter = "Livebox-08B0";//STA router ssid
const char* passwordRouter = "G79ji6dtEptVTPWmZP";//STA router password

WiFiUDP udp;
```

```
void setup() {
    pinMode(22, OUTPUT);//builtin Led, for debug
    digitalWrite(22, HIGH);
    Serial.begin(115200);
    Serial.println( "Master" );
    //first, we start STA mode and connect to router
    WiFi.mode( WIFI_AP_STA );
    WiFi.begin(ssidRouter,passwordRouter);

    //Wifi connection
    while (WiFi.status() != WL_CONNECTED)
    {
      delay(500);
      Serial.print(".");
    }
    Serial.println("Router WiFi connected");
    Serial.print("IP address: ");
    Serial.println(WiFi.localIP());
    //second, we start AP mode with LR protocol
    //This AP ssid is not visible whith our regular devices
    WiFi.mode( WIFI_AP );//for AP mode
    //here config LR mode
    int a= esp_wifi_set_protocol( WIFI_IF_AP, WIFI_PROTOCOL_LR );
     if (a==0)
    {
      Serial.println(" ");
      Serial.print("Return = ");
      Serial.print(a);
      Serial.println(" , Mode LR OK!");
    }
    else//if some error in LR config
    {
      Serial.println(" ");
      Serial.print("Return = ");
      Serial.print(a);
      Serial.println(" , Error in Mode LR!");
    }
    WiFi.softAP(ssid, password);
    Serial.println( WiFi.softAPIP() );
    Serial.println("#");//for debug
    delay( 5000 );
    digitalWrite(22, LOW);
    udp.begin( 8888 );
}

void loop()
{
    udp.beginPacket( { 192, 168, 4, 255 }, 8888 );//send a broadcast message
    udp.write( 'b' );//the payload
    digitalWrite(22, !digitalRead(25));

    if ( !udp.endPacket() ){
        Serial.println("NOT SEND!");
        delay(100);
        ESP.restart(); // When the connection is bad, the TCP stack refuses to work
    }
    else{
          Serial.println("SEND IT!!");
    }
    delay( 1000 );//wait a second for the next message
}
```

```
// slave.ino
```

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

const char* ssid = "LongRange";//AP ssid
const char* password = "smartcomputerlab";//AP password

const char* ssidRouter = "Livebox-08B0";//STA router ssid
const char* passwordRouter = "G79ji6dtEptVTPWmZP";//STA router password

WiFiUDP udp;

const char *toStr( wl_status_t status ) {
    switch( status ) {
    case WL_NO_SHIELD: return "No shield";
    case WL_IDLE_STATUS: return "Idle status";
    case WL_NO_SSID_AVAIL: return "No SSID avail";
    case WL_SCAN_COMPLETED: return "Scan compleded";
    case WL_CONNECTED: return "Connected";
    case WL_CONNECT_FAILED: return "Failed";
    case WL_CONNECTION_LOST: return "Connection lost";
    case WL_DISCONNECTED: return "Disconnected";
    }
    return "Unknown";
}

void setup() {
    Serial.begin(115200);
    Serial.println( "Slave" );
    pinMode(22, OUTPUT);//bultin Led, for debug
    //first, we start STA mode and connect to router
    WiFi.mode( WIFI_AP_STA );
    WiFi.begin(ssidRouter,passwordRouter);
    //Wifi connection
    while (WiFi.status() != WL_CONNECTED)
    {
      delay(500);
      Serial.print(".");
    }
    Serial.println("Router WiFi connected");
    Serial.print("IP address: ");
    Serial.println(WiFi.localIP());
    //We start STA mode with LR protocol
    //This ssid is not visible whith our regular devices
    WiFi.mode( WIFI_STA );//for STA mode
    //if mode LR config OK
    int a= esp_wifi_set_protocol( WIFI_IF_STA, WIFI_PROTOCOL_LR );
    if (a==0)
    {
      Serial.println(" ");
      Serial.print("Return = ");
      Serial.print(a);
      Serial.println(" , Mode LR OK!");
    }
    else//if some error in LR config
    {
      Serial.println(" ");
      Serial.print("Return = ");
      Serial.print(a);
      Serial.println(" , Error in Mode LR!");
    }

    WiFi.begin(ssid, password);//this ssid is not visible
    //Wifi connection, we connect to master
    while (WiFi.status() != WL_CONNECTED)
    {
      delay(500);
      Serial.print(".");
    }
    Serial.println("WiFi connected");
    Serial.print("IP address: ");
    Serial.println(WiFi.localIP());
    delay(5000);
    udp.begin( 8888 );
```

```
}

void loop() {
     //problems whith connection
    if ( WiFi.status() != WL_CONNECTED )
    {
        Serial.println( "|" );
        int tries = 0;
        WiFi.begin( ssid, password );
        while( WiFi.status() != WL_CONNECTED ) {
            tries++;
            if ( tries == 5 )
                return;
            Serial.println( toStr( WiFi.status() ) );
            delay( 1000 );
        }
        Serial.print( "Connected " );
        Serial.println( WiFi.localIP() );
    }
    //if connection OK, execute command 'b' from master
    int size = udp.parsePacket();
    if ( size == 0 )
        return;
    char c = udp.read();
    if ( c == 'b' ){
        digitalWrite(22, !digitalRead(22));//toggle Led
        Serial.println("RECEIVED!");
        Serial.println(millis());
    }
    udp.flush();
}
```