MicroPython and WiFi part 2

By jean-claude.feltes@education.lu

Thonny is used as IDE for all examples, see
http://staff.ltam.lu/feljc/electronics/uPython/ESP8266&uPython_01.pdf

2.1. The WiFi connection

The best way is to use the mini library wifi_connect.py described in part 1, available here:

http://staff.ltam.lu/feljc/electronics/uPython/wifi_connect.py

With this we can connect to an existing WiFi network or create our own accesspoint.
Copy the file to the ESP file system.

Let's say we want an accesspoint.
To do this at boot time we put the following code into main.py:

```
from wifi_connect import *
ssid = 'MicroPython-AP'
password = '123456789'
ip = '192.168.179.1'
ip = create_accesspoint(ssid, password, ip)
print(ip)
```

The main.py file is executed at boot time and we get a message over the serial connection in Thonny:

```
Setting ipinfo  ('192.168.179.1', '255.255.255.0', '192.168.179.1', '8.8.8.8')
Access point ready!
('192.168.179.1', '255.255.255.0', '192.168.179.1', '8.8.8.8')
192.168.179.1
```

From now on the accesspoint is ready to be connected by a PC, a tablet or a phone.

We can test with ping from the command line:

```
 ping 192.168.179.1
PING 192.168.179.1 (192.168.179.1) 56(84) bytes of data.
64 bytes from 192.168.179.1: icmp_seq=1 ttl=255 time=56.2 ms
64 bytes from 192.168.179.1: icmp_seq=2 ttl=255 time=1.52 ms
. . .
```

All the following examples assume that a WiFi connection is set up in the main.py file at boot time.

2.2. The MicroWebserv library

There are several web server libraries.

I used the one by Jean-Christophe Bos available here:

https://github.com/jczic/MicroWebSrv

There are some nice tutorials:

https://www.rototron.info/raspberry-pi-esp32-micropython-web-server-tutorial/

https://www.youtube.com/watch?v=xscBwC1SrF4

As described in the installation guide, we should copy microWebSrv.py tho the root folder of the ESP file system. Jean-Christophe suggests to create a folder '/www' in the ESP file system, where the HTML, CSS and Javascript files we need to display a webpage can be created or copied.

In Thonny, we can create a new folder by klicking on the small button to the right of "MicroPython device":

We can change to this folder by klicking it in the file menu or by "magic command" %cd www in the shell window.

To come back to the upper folder, use %cd .. in the shell window.

2.3. An absolutely minimalistic file server

```
from microWebSrv import MicroWebSrv
 print("Starting server")
srv = MicroWebSrv(webPath='www/')
srv.Start()
```

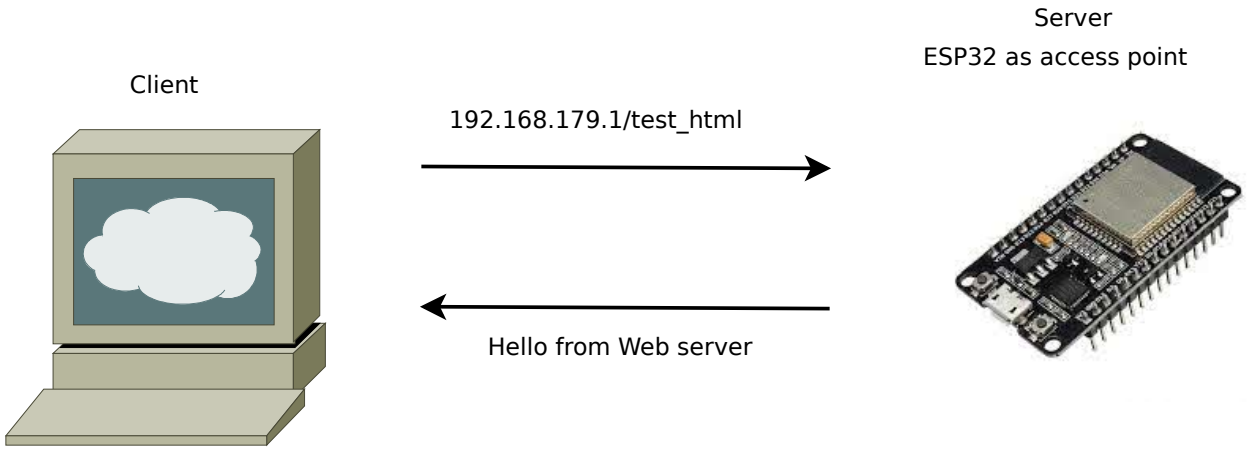
This bit of code is enough to allow viewing or downloading of files from the www folder on the ESP.

(Not all files though, but text files, PDF, HTML etc., see documentation)

2.4. Hello from the server

The following example is the “Hello world” of the server.

It responds to a HTML request:




```
from microWebSrv import MicroWebSrv

htmlstring = '<h1>HELLO from Web server !</h1>'

def _httpHandlerDHTGet(httpClient, httpResponse):


    httpResponse.WriteResponseOk(
        contentType = 'text/html',
        contentCharset = 'UTF-8',
        content = htmlstring
        )


routeHandlers = [ ( "/test_html", "GET",  _httpHandlerDHTGet ) ]
srv = MicroWebSrv(routeHandlers = routeHandlers)
srv.Start(threaded=False)
```

Here is the result:

2.5. Sending HTML pages

For bigger webpages it is better to separate the HTML data from the code.

We put a file www/hello2.html containing the HTML text into the www folder of the ESP.
This file contains another Hello message, with style formatting:

```
<!DOCTYPE html>
<html>
    <head>
        <style>

            body {
             background-color: #FFFFB0;
             font-family: Verdana, sans-serif;
             font-size: 100%;
             }
            h1 {
             font-size: 200%;
             color: red;
             text-align: center;
             }

            p {
             color: black;
             }
            pre {
                color: blue;
                }
        </style>

    </head>

    <body>
        <h1>Hello from server</h1>
        <p>This HTML document resides in the www folder of the ESP</p>
        <p>It is sent by the function httpResponse.WriteResponseFile</p>

        <pre>
            MicroWebSrv is a micro HTTP Web server that supports WebSockets,
html/python language templating and routing handlers,
            for MicroPython (principally used on ESP32 and Pycom modules.
            Now supports all variants of Pyboard D-series from the makers of
Micropython)
        </pre>


    </body>
</html>
```

(Never mind, it could be any html page.)

The Python code now uses the WriteResponseFile function:

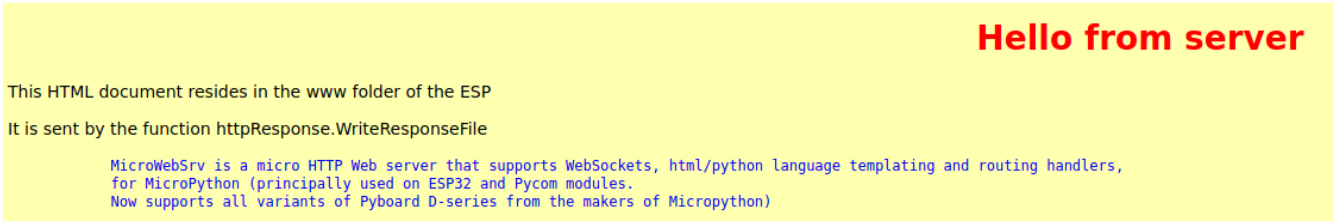
```
from microWebSrv import MicroWebSrv
```

```
def _httpHandlerDHTGet(httpClient, httpResponse):

    httpResponse.WriteResponseFile(
        'www/hello2.html',
        contentType = 'text/html',
        )

print("Starting server")
routeHandlers = [ ( "/hello", "GET",  _httpHandlerDHTGet ) ]
srv = MicroWebSrv(routeHandlers = routeHandlers)
srv.Start(threaded=False)
```

The result is a bit more colorful:



2.6. Problems?

- If the py program says it cannot import MicrWebsrv, verify that you start it in the correct folder (‘/’). If you are currently in the www folder, the import cannot be found
- If there is any difficulty to change folders, the “magic commands” work in the Thonny shell:

```
%cd www
% cd ..
```

- You have the possibility to store files on the ESP device or on the computer.
  Be sure to have the right version in your editor.
- When experimenting, it is a good idea to do a reset (<Ctrl>-D) to avoid getting the error message ‘EADDR IN USE’.
- Always confirm that a connection exists (with ping), before looking into the server code.

2.7. Playing around …

A modified program shows info about the client and uses Threading, so other things can be done after starting the server:

```
from microWebSrv import MicroWebSrv

def print_info(httpClient):
    print()
    print (httpClient.GetRequestMethod(), ' from ',
```

```
          httpClient.GetIPAddr(), ':', httpClient.GetPort())

def _httpHandlerDHTGet(httpClient, httpResponse):

    print_info(httpClient)

    httpResponse.WriteResponseFile('www/hello2.html',
        contentType = 'text/html', )

print("Starting server")
routeHandlers = [ ( "/", "GET",  _httpHandlerDHTGet ) ]
srv = MicroWebSrv(routeHandlers = routeHandlers)
srv.Start(threaded=True)

import time
while True:
    time.sleep(1)
    print('*', end ='')
```

Output after some seconds and after the server has been contacted:

Starting server

```
***
GET  from  192.168.179.2 : 58422
********
```

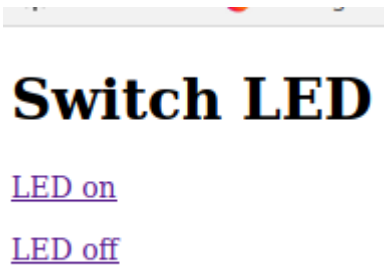
2.8. Switching a LED over WiFi (via HTTP request)

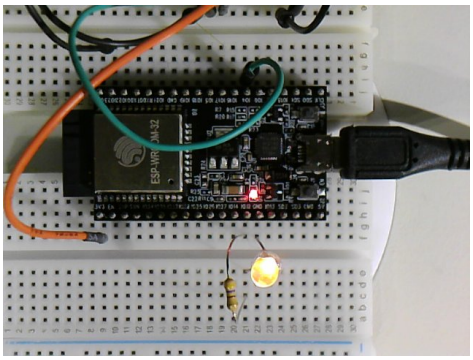
I'll try to reproduce the example described here with an ESP8266 programmed in C using the Arduino IDE:

http://staff.ltam.lu/feljc/electronics/arduino/WiFi_01.pdf

For this I have used an ESP32 with a LED connected to IO12 (over a 470 Ohm resistance).

We have a webpage with two links for switching on or off.





This webpage is defined by the following HTML code:

```
content = """\
    <!DOCTYPE html>
         <head>
            <meta charset="UTF-8" />
            <title>Switch LED</title>
        </head>
        <body>
            <h1>Switch LED</h1>
            <p><a href = /led_on> LED on </a></p>
            <p><a href = /led_off> LED off </a></p>

        </body>
    </html>
    """
```

We define three route handler functions, one for ‘/’, one for ‘led_on’ and one for ‘led_off’.

To avoid problems we always resend the whole webpage.

This is done by the following functions:

```
def send_webpage(httpResponse):
    httpResponse.WriteResponseOk( headers         = None,
                                  contentType     = "text/html",
                                  contentCharset = "UTF-8",
                                  content         = content )

def _httpHandler_root(httpClient, httpResponse) :
    send_webpage(httpResponse)


def _httpHandler_on(httpClient, httpResponse) :
    led.on()
    send_webpage(httpResponse)

def _httpHandler_off(httpClient, httpResponse) :
    led.off()
    send_webpage(httpResponse)
```

The main program imports the web server and machine library and defines the pin for the LED.
After this, the three route handlers are defined and the server is started:

```
from microWebSrv import MicroWebSrv
from machine import Pin
led = Pin(12, Pin.OUT)

print('Starting server')

routeHandlers = [
    ( "/",  "GET",  _httpHandler_root ),
    ( "/led_on",  "GET",  _httpHandler_on ),
    ( "/led_off",  "GET",  _httpHandler_off ),
]
```

```
srv = MicroWebSrv(routeHandlers=routeHandlers)
srv.Start(threaded=True)

import time
time.sleep(0.2)
print('Server started: ', srv.IsStarted())
```

Now we can switch the LED on and off by clicking ‘LED on’ or ‘LED off’ on the webpgage.

What about the latency?
It seems to be a little bit longer than for the C program, but I am not sure, as I did not directly compare. Anyway, it seems too long for critical operations, like controlling a robot.

To have a better idea, I used **Wireshark** to analyze the transmission (using a filter to only display the TCP traffic on wlan0):

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.179.2	192.168.179.1	TCP	74	48796 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2658304145 TSecr=0 WS=128
2	0.003686843	192.168.179.1	192.168.179.2	TCP	58	80 → 48796 [SYN, ACK] Seq=0 Ack=1 Win=5744 Len=0 MSS=1436
3	0.003719339	192.168.179.2	192.168.179.1	TCP	54	48796 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0
4	0.003847296	192.168.179.2	192.168.179.1	HTTP	439	GET /led_off HTTP/1.1
5	0.249301814	192.168.179.1	192.168.179.2	TCP	54	80 → 48796 [ACK] Seq=1 Ack=386 Win=5359 Len=0
6	0.307321906	192.168.179.1	192.168.179.2	TCP	71	80 → 48796 [PSH, ACK] Seq=1 Ack=386 Win=5359 Len=17 [TCP segment of a reassembled PDU]
7	0.307358757	192.168.179.2	192.168.179.1	TCP	54	48796 → 80 [ACK] Seq=386 Ack=18 Win=64223 Len=0
8	0.321788058	192.168.179.1	192.168.179.2	TCP	94	80 → 48796 [PSH, ACK] Seq=18 Ack=386 Win=5359 Len=40 [TCP segment of a reassembled PDU]
9	0.321809744	192.168.179.2	192.168.179.1	TCP	54	48796 → 80 [ACK] Seq=386 Ack=58 Win=64183 Len=0
10	0.325423288	192.168.179.1	192.168.179.2	TCP	75	80 → 48796 [PSH, ACK] Seq=58 Ack=386 Win=5359 Len=21 [TCP segment of a reassembled PDU]
11	0.325454714	192.168.179.2	192.168.179.1	TCP	54	48796 → 80 [ACK] Seq=386 Ack=79 Win=64162 Len=0
12	0.334062534	192.168.179.1	192.168.179.2	TCP	85	80 → 48796 [PSH, ACK] Seq=79 Ack=386 Win=5359 Len=31 [TCP segment of a reassembled PDU]
13	0.334083921	192.168.179.2	192.168.179.1	TCP	54	48796 → 80 [ACK] Seq=386 Ack=110 Win=64131 Len=0
14	0.344187224	192.168.179.1	192.168.179.2	TCP	73	80 → 48796 [PSH, ACK] Seq=110 Ack=386 Win=5359 Len=19 [TCP segment of a reassembled PDU]
15	0.344211119	192.168.179.2	192.168.179.1	TCP	54	48796 → 80 [ACK] Seq=386 Ack=129 Win=64112 Len=0
16	0.350944560	192.168.179.1	192.168.179.2	TCP	56	80 → 48796 [PSH, ACK] Seq=129 Ack=386 Win=5359 Len=2 [TCP segment of a reassembled PDU]
17	0.350968362	192.168.179.2	192.168.179.1	TCP	54	48796 → 80 [ACK] Seq=386 Ack=131 Win=64110 Len=0
18	0.360031748	192.168.179.1	192.168.179.2	HTTP	375	HTTP/1.1 200 OK (text/html)
19	0.360057077	192.168.179.2	192.168.179.1	TCP	54	48796 → 80 [ACK] Seq=386 Ack=452 Win=63789 Len=0
20	0.360231051	192.168.179.2	192.168.179.1	TCP	54	48796 → 80 [FIN, ACK] Seq=386 Ack=452 Win=63789 Len=0
21	0.362285226	192.168.179.1	192.168.179.2	TCP	54	80 → 48796 [ACK] Seq=452 Ack=387 Win=5358 Len=0
22	0.369174602	192.168.179.1	192.168.179.2	TCP	54	80 → 48796 [FIN, ACK] Seq=452 Ack=387 Win=5358 Len=0
23	0.369212963	192.168.179.2	192.168.179.1	TCP	54	48796 → 80 [ACK] Seq=387 Ack=453 Win=63789 Len=0

If I interprete this correctly, the latency would be about 370ms, which corresponds to my feeling.

Naturally it is possible to beautify the HTML page to have nice knobs for switching, or anything you want …