

Danmarks
Tekniske
Universitet



02135 Introduction to Cyber System - Assignment 4

Implementation of an Internet-of-Things node using the Adafruit Feather Huzzah32 WiFi board and MicroPython

AUTHORS

Group 8

Jianan Xu - s204698

Rune Bjerre Clausen- s204702

Simona Tican - s204703

December 1, 2021

Contents

1	Introduction	1
2	Task 1	1
3	Task 2	1
4	Task 3	2
5	Task 4	3
6	Task 5	4
6.1	Server	4
6.2	Client	5

1 Introduction

In this report we will show the implementation of an Internet-of-Things node using the Adafruit Feather Huzzah32 WiFi board using MicroPython. The five tasks designed for this work will take us through the very simple setup of an HTML based HTTP web-server into the more complex implementation of a web API from where we can interact with the board, though a client python program. These layered tasks help us understand the steps of the setup on a more profound level.

2 Task 1

The first task was deigned to give a brief explanation of the program that sets up the Huzzah32 as WiFi access point and starts a simple Web server (HTML based) reporting the status of the board pins. Following the code provided, we introduced all the necessary comments in order to get better understanding of the whole functionality of the program.

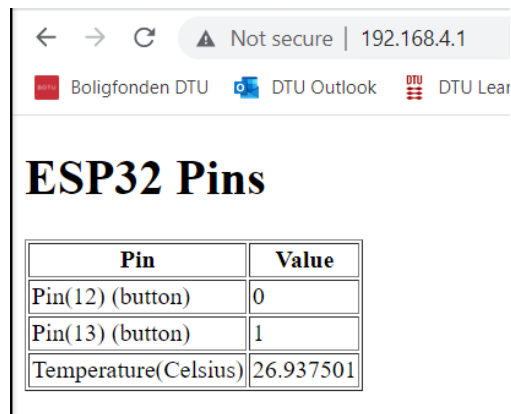
3 Task 2

In the second task we made some minor changes in the code and then used `ampy` to run it in Huzzah32 as a server. The PC connected to the WiFi network ESP32-JRS can access the server from a browser at the address 192.168.4.1:80, then it will show an HTML web page that reports the status of the buttons and the temperature sensor.

This was done by implementation of functions from assignment 3 to convert sensor input to a temperature in degree Celsius, as well as making clear definitions of the pins used. In our case, Pin12 and Pin13 were configured as inputs for monitoring two buttons. The other change was to convert a single row of the table in the HTML to two rows for the buttons and another row for the sensor:

```
while True:
    ...
    row_button = ['<tr><td> %s </td><td> %d </td></tr>' % (str(p) + ' (button)', p.value()) for p in pins]
    sensor.readfrom_mem_into(address, temp_reg, data)
    row_temp = ['<tr><td> %s </td><td> %f </td></tr>' % ('Temperature(Celsius)', temp_c(data))]
    response = html % ('\n'.join(row_button) + '\n'.join(row_temp))
    ...
```

What we end up with is then this pretty HTML page:



Pin	Value
Pin(12) (button)	0
Pin(13) (button)	1
Temperature(Celsius)	26.937501

Figure 1: Task 2 HTML webpage

Note that the `pin(12) (button)` was pushed at the time the page was loaded, hence the value was 0.

4 Task 3

The third task required us to define a Web API for interrogating the states of the pins and the temperature sensor. As stated in the conditions, the response of a certain query should be JSON-formatted information. The program created had the aim to distinguish a normal Web request with an empty resource path from a Web API request with a specific resource path.

This created two sub-tasks to overcome:

- To create the API map that is a simple request-response messages system.
- To extract the request path, and match the path with the corresponding JSON response.

The first part:

A dictionary `api` was created as the API map, in which the key is a request path and the corresponding value is JSON response(s). This was done by using the `json` package in Python, specifically the `json.dump()` function that converts the Python objects into appropriate JSON objects.

```
while True:
    ...
    api = {"/pins/": json.dumps(["pin12", "pin13"]),
           "/pins/pin12/": json.dumps(machine.Pin(12, machine.Pin.IN).value()),
           "/pins/pin13/": json.dumps(machine.Pin(13, machine.Pin.IN).value()),
           "/sensors/": json.dumps(["temperature_sensor"]),
           "/sensors/temperature_sensor/": json.dumps(temp_c(data))}
    ...
```

The second part:

The first line in request message is called request line, which consists of the method, the **request path** and the protocol version. For example, when we enter the URI <http://192.168.4.1/pins/pin13/> in a browser, the request line received by the server will be `b'GET /pins/pin13/ HTTP/1.1\r\n'`. We are interested in the request path, so request path should be extracted firstly. Next, if the request path is empty, the same HTML web page as previous task will be shown. If the request path exists in the API map, then the status line `HTTP/1.1 200 OK` and the corresponding JSON response(message body) will be sent to the client(browser). Otherwise, the status line `HTTP/1.1 404 Not Found` is sent.

```
while True:
    ...
    get_request = cl_file.readline().decode('ascii') # convert the request line to normal python string
    try:
        path = get_request.split()[1] # split 'get_request' by whitespace, and get the resource path
    except:
        break
    # Make every path end in '/'
    if path[-1] != '/':
        path = path + '/'
    while True:
        ...
        else:
            try:
                response = "HTTP/1.1 200 OK\r\n" + "\r\n" + api[path]
            except:
                response = "HTTP/1.1 404 Not Found" # If no such path, return "404 NOT FOUND"
        ...
```

5 Task 4

In this task, we still used the server implemented in the task 3, but we need access the server from a *control program* instead of a browser. The *monitor program* enable to log the board temperature every 0.5 seconds for 10 seconds, which is set at the beginning. Secondly, a text with the header **Time,Temperature** is created.

Thirdly, the temperature is logged by a while loop. `request.get(URI)` is called to send GET request to the server every 0.5 seconds. It is the "same" as entering a [URI](#) in a browser in task 3. The temperature responded by server is written in the `t-T.txt`, where time and temperature in each line are separated by comma. The variable time is 0 before the loop starts, and is incremented by 0.5 each cycle. The while loop will stop when `time>10`. Fourthly, the `t-T.txt` file is read as CSV file. Finally, the 'temperature vs. time' is plotted by using `matplotlib`.

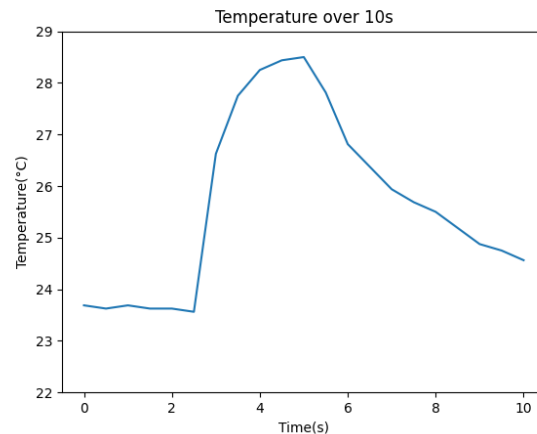


Figure 2: Task 4 temperature vs. time plot

6 Task 5

In task 5, a user can read the status of the buttons by sending **GET** request and control the LEDs by sending **PUT** request. When checking the status of buttons, 1 will be shown in the console if the button is pressed, otherwise 0 will be shown. Additionally, there are two color modes. For the **Mode Purple**, the green LED turns on and the NeoPixel shows purple. For the **Mode Yellow**, the green LED also turns on but the NeoPixel shows yellow. If you choose **Turn off**, only the red LED turns on. The program also supports returning and exiting.

6.1 Server

We made some changes to the code based on task 3. As previous tasks, Pin12 and Pin13 are configured as inputs for two buttons. Besides, Pin25, Pin26 and Pin21 are configured as outputs for red LED, green LED and NeoPixel respectively. The API map built in task 3 was also extended in this task. Differently, in order to control the LEDs, each extended request path corresponds to an operation instead of a JSON response. The matching and operations are done by the `handle_put(mode)` function. Next, we extracted the request path. If the method was **PUT**, then match the request path with the corresponding operation. If the method was **GET**, then match the request path with the corresponding JSON response as the same as task 3.

```
def handle_put(mode):  
    if mode == 'on_yellow':  
        led_red.value(0)  
        led_green.value(1)  
        global np  
        np[0] = (178, 80, 0)  
    elif mode == 'on_purple':
```

```

...
else:
...

while True:
...
get_request = cl_file.readline().decode('ascii') # convert the request line to normal python string
try:
    request = get_request.split()[0]
    path = get_request.split()[1] # split 'get_request' by whitespace, and get the resource path
except:
    break
# Distinguish the types of requests, GET or PUT
if request == 'PUT':
    handle_put(path.split('/')[1])
    cl.send('HTTP/1.1 200 OK\r\n')
    cl.close()
    continue
else:
    pass
...

```

6.2 Client

The *control program* consists of two parts. In the function definition part, `master_switch()` and `color_switch()` are defined to get the status of Pin12 and Pin13 respectively, in which the `request.get(URI)` is called. Besides, `on_purple()`, `on_yellow()` and `fun_off()` are defined to show **Mode Purple**, **Mode Yellow** and **Turn off** respectively. In each of these three functions, `request.put(URI)` is called with the corresponding *URI* as its argument.

Then the user interface is implemented by using `consolemenu` package. Function defined above will be called in this part.

```

├─ 1 - Check Button Status
│   ├── 1 - Master Switch(Pin13)
│   ├── 2 - Color Switch(Pin12)
│   └── 3 - Return to Task5_Client
├─ 2 - Color mode
│   ├── 1 - Mode Purple
│   ├── 2 - Mode Yellow
│   └── 3 - Return to Task5_Client
├─ 3 - Turn off
└─ 4 - Exit

```

Figure 3: Task 5 user interface

Note that the *URI* represents the URI used in GET request and the *URI* represents the URI used in PUT request.