IoT Lab 1

Communicating with WiFi and IoT servers

In this lab we are going to build communication links for directly connected terminal. In this case we use WiFi radio to connect to an available access point.

1.1 WiFi tools to connect/disconnect and scan the WiFi networks

Below is an example **`wifi_tools.py`** module for MicroPython on the ESP32 that provides three functions to manage WiFi connections:

1. **`connect_WiFi(ssid,passwd)`**: Connect to a WiFi network using the provided SSID and password.
2. **`disconnect_WiFi()`**: Disconnect from the currently connected WiFi network.
3. **`scan_WiFi()`**: Scan for available WiFi networks and return a list of tuples with network information.

Note:

- Make sure you have WiFi enabled and that you're running on an ESP32 with WiFi capability.
- Replace `ssid` and `passwd` with your own credentials when using the `connect` function.

```
import network
import time

def connect_WiFi(ssid, passwd, timeout=10):
    """
    Connect to the given WiFi network using the specified SSID and password.
    :param ssid: The SSID of the WiFi network.
    :param passwd: The password of the WiFi network.
    :param timeout: Maximum time in seconds to wait for connection.
    :return: True if connected, False otherwise.
    """
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    wlan.config(txpower=8.5)  # or even 5.0
    # If already connected to the same network, return True
    if wlan.isconnected() and wlan.config('essid') == ssid:
        print(wlan.ifconfig())
        return True
    # Connect to the given SSID
    wlan.connect(ssid, passwd)
    # Wait for connection or timeout
    start = time.time()
    while not wlan.isconnected():
        if time.time() - start > timeout:
            return False
        time.sleep(1)
    print(wlan.ifconfig())
    return True

def disconnect_WiFi():
    """
    Disconnect from the currently connected WiFi network.
    """
    wlan = network.WLAN(network.STA_IF)
    if wlan.isconnected():
        wlan.disconnect()

def scan_WiFi():
    """
    Scan for available WiFi networks.
    :return: A list of tuples containing network information:
             (ssid, bssid, channel, RSSI, authmode, hidden)
    """
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    return wlan.scan()

# test of functions - comment for separete use
print(scan_WiFi())
if connect_WiFi("PhoneAP","smartcomputerlab"):
    print("WiFi connected")
else:
    print("WiFi not connected")
```

Example Usage :

```
from wifi_tools import *

# Replace with your own WiFi credentials
SSID = "PhoneAP"
PASSWORD = "smartcomputerlab"

# Scan available networks
networks = scan_WiFi()
print("Available networks:")
for net in networks:
    ssid, bssid, channel, RSSI, authmode, hidden = net
    print("SSID:", ssid.decode('utf-8'), "| RSSI:", RSSI)

if connect_WiFi(SSID, PASSWORD):
    print("Connected to WiFi:", SSID)
    print("Network config:", network.WLAN(network.STA_IF).ifconfig())
else:
    print("Failed to connect to WiFi:", SSID)

# Disconnect from current WiFi network
disconnect_WiFi()
print("Disconnected from WiFi")
```

Note:

- **`network.WLAN(network.STA_IF).ifconfig()`** prints the current network configuration: IP, subnet, gateway, and DNS.
- Ensure that the SSID and password are correct and that the access point is in range.
- Adjust `timeout` in **`connect_WiFi()`** if you need a longer wait for the connection.

Program execution:

```
MPY: soft reboot
Connected to WiFi: PhoneAP
Network config:('192.168.222.252','255.255.255.0','192.168.222.85','192.168.222.85')
Available networks:
SSID: PhoneAP | RSSI: -18
SSID: DIRECT-G8M2070 Series | RSSI: -63
SSID: VAIO-MQ35AL | RSSI: -67
SSID: Livebox-08B0 | RSSI: -68
SSID: SFR_2C80 | RSSI: -85
SSID: Livebox-C6E0 | RSSI: -85
SSID: Papa tango Charly | RSSI: -89
SSID:  | RSSI: -90
Disconnected from WiFi
```

1.1.1 Reading network connection parameters

Here is a MicroPython function for the ESP32 that retrieves the Wi-Fi configuration details, such as the IP address, subnet mask, gateway, and DNS server, after successfully connecting to an access point.

```
import network

def connect_to_wifi(ssid, password):
    wlan = network.WLAN(network.STA_IF)  # Initialize station interface
    wlan.active(True)  # Activate Wi-Fi interface
    print(f"Connecting to Wi-Fi network: {ssid}")
    wlan.disconnect()  # disconnect if already connected
    wlan.connect(ssid, password)
    while not wlan.isconnected():
        pass

    print("Connected to Wi-Fi!")
    print("Network configuration:")
    config = wlan.ifconfig()  # Get network configuration as a tuple
    config_dict = {
        "IP Address": config[0], "Subnet Mask": config[1],
        "Gateway": config[2], "DNS Server": config[3]
    }
    # Display configuration
    for key, value in config_dict.items():
        print(f"{key}: {value}")

    return config_dict

# Example usage
```

```
if __name__ == "__main__":
    # Replace with your Wi-Fi credentials
    SSID = 'Livebox-08B0'
    PASSWORD = 'G79ji6dtEptVTPWmZP'
    config = connect_to_wifi(SSID, PASSWORD)
```

How It Works:

1. **Wi-Fi Activation**:
    - The **`network.WLAN(network.STA_IF)`** initializes the ESP32 as a station (client).
    - The `wlan.active(True)` activates the station interface.
2. **Connect to Access Point**:
    - The **`wlan.connect(ssid, password)`** function attempts to connect to the specified Wi-Fi network.
3. **Wait for Connection**:
    - The **`while not wlan.isconnected()`** loop ensures the code waits until the ESP32 is successfully connected.
4. **Retrieve Configuration**:
    - Once connected, `wlan.ifconfig()` returns a tuple containing:
        - **IP Address**: The assigned IP address.
        - **Subnet Mask**: The subnet mask of the network.
        - **Gateway**: The default gateway address.
        - **DNS Server**: The primary DNS server.
5. **Return Configuration**:
    - The function organizes the network configuration into a dictionary for easy access and printing.

Execution run:

```
MPY: soft reboot
Connecting to Wi-Fi network: Livebox-08B0
Connected to Wi-Fi!
Network configuration:
DNS Server: 192.168.1.1
Gateway: 192.168.1.1
IP Address: 192.168.1.61
Subnet Mask: 255.255.255.0
```

1.2 Using UDP sockets to send/receive datagrams (packets)

Here are two MicroPython programs for the ESP32 to send and receive simple UDP datagrams using the **`socket`** module.

1.2.1 UDP Sender

This program sends a simple UDP message to a specified IP and port.

```
import socket
import time
from wifi_tools import *
SSID = 'Livebox-08B0'
PASSWORD = 'G79ji6dtEptVTPWmZP'
# Configuration
UDP_IP = "192.168.1.61"  # Replace with the receiver's IP address
UDP_PORT = 8899          # Replace with the desired port
MESSAGE = "Hello, ESP32C3!"
# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
try:
    while True:
        connect_WiFi(SSID,PASSWORD)
        print(f"Sending message: {MESSAGE}")
        sock.sendto(MESSAGE.encode('utf-8'), (UDP_IP, UDP_PORT))
        time.sleep(5)  # Send every 2 seconds
        disconnect_WiFi()
except KeyboardInterrupt:
    time.sleep(5)          # to allow stop
    print("UDP sender stopped.")
finally:
    sock.close()
```

1.2.2 UDP Receiver

This program listens for incoming UDP messages on a specific port.

```
import socket
from wifi_tools import *
SSID = 'Livebox-08B0'
PASSWORD = 'G79ji6dtEptVTPWmZP'
# Configuration
UDP_IP = "192.168.1.61"  # Listen on all available interfaces
UDP_PORT = 8899     # Port to listen on
# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((UDP_IP, UDP_PORT))
print(f"Listening for UDP messages on port {UDP_PORT}...")
try:
    connect_WiFi(SSID,PASSWORD)
    while True:
        data, addr = sock.recvfrom(1024)  # Buffer size of 1024 bytes
        print(f"Received message: {data.decode('utf-8')} from {addr}")
except KeyboardInterrupt:
    print("UDP receiver stopped.")
finally:
    sock.close()
```

Notes

- **Port Numbers**: Use the same UDP port on both sender and receiver.
- **IP Address**: Ensure the receiver's IP is correct in the sender's code.
- **Buffer Size**: Adjust the buffer size (`1024`) in the receiver to accommodate your data size.

To do:

Use `sensors.py` module to prepare the message with sensor's data before sending it to the receiver.

Write the same sender program with deepsleep mode to introduce `low_power` stage.

```
import socket
from machine import deepsleep
import time
from wifi_tools import *
from sensors import *
SSID = 'Livebox-08B0'
PASSWORD = 'G79ji6dtEptVTPWmZP'
UDP_IP = "192.168.1.61"  # Replace with the receiver's IP address
UDP_PORT = 8899          # Replace with the desired port
MESSAGE = "Hello, ESP32C3!"
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
try:
    connect_WiFi(SSID,PASSWORD)
```

```
    lumi,temp,humi = sensors(8,9)
    message="Lumi:"+str(lumi)+", Temp:"+str(temp)+" ,Humi:"+str(humi)
    print(f"Sending message:" + message)
    sock.sendto(message.encode('utf-8'), (UDP_IP, UDP_PORT))
    time.sleep(5)  # Send every 2 seconds
    disconnect_WiFi()
except KeyboardInterrupt:
    time.sleep(2)
    print("UDP sender stopped.")
finally:
    sock.close()
    time.sleep(2)
    print("Going to deepsleep")
    deepsleep(10*1000)
```

Attention:

When we use `deepsleep()` we have to save the program as `main.py`

1.3 Reading time from NTP server

1.3.1 NTP (Network Time Protocol)

Function:

NTP is a networking protocol designed to synchronize the clocks of computers and devices over a network. It ensures that the time on various systems is accurate and consistent, which is crucial for distributed systems, logging, and time-sensitive applications.

Key Features:

1. **Time Synchronization**: NTP synchronizes the device's clock with highly accurate time servers.
2. **Hierarchy of Servers**: NTP operates in a hierarchy where Stratum 0 servers (e.g., atomic clocks) provide time to Stratum 1 servers, which in turn serve Stratum 2 servers, and so on.
3. **Error Correction**: NTP adjusts for clock drift using algorithms to improve accuracy.
4. **Redundancy**: Typically connects to multiple servers for redundancy and reliability.

How It Works:

1. The client sends a request to an NTP server.
2. The server responds with a timestamp indicating the current time.
3. The client calculates the round-trip delay and adjusts its clock accordingly.

NTP Server

An NTP server provides time information to NTP clients. It can act as:

- **Primary Server**: Directly synchronized with reference clocks (e.g., GPS, atomic clocks).
- **Secondary Server**: Synchronizes with other NTP servers and serves time to its clients.

Use Cases of NTP:

1. **System Synchronization**: Ensures accurate timestamps in distributed systems, logs, and transactions.
2. **Security**: Accurate time is critical for cryptographic protocols like SSL/TLS and Kerberos.
3. **IoT Devices**: IoT systems often rely on NTP to maintain consistent timing across devices.
4. **Financial Systems**: Precise timekeeping is vital for trading and auditing.
5.

1.3.1 Reading time from NTP server – always in `high_power` state

In this section we use NTP (Network time protocol) and server to receive wall time value. Note that after the first reception via WiFi connection, the board uses its internal timer to follow the time line.
Below is an example **`main.py`** that uses the **`wifi_tools.py`** module to:

1. Connect to a Wi-Fi network.
2. Synchronize with an NTP server only once at the start.
3. In an infinite loop, every 10 seconds, read and print the current hour, minute, and second without re-synchronizing.

Prerequisites:

- Ensure **`wifi_tools.py`** is on your ESP32.
- Set the correct Wi-Fi credentials in the code.

```
import time
import ntptime
from wifi_tools import *
# Replace with your Wi-Fi credentials
SSID = 'Livebox-08B0'
PASSWORD = 'G79ji6dtEptVTPWmZP'
# Connect to WiFi
if connect_WiFi(SSID, PASSWORD):
    print("Connected to WiFi:", SSID)
else:
    print("Failed to connect to WiFi:", SSID)
```

```
    # If cannot connect to WiFi, can't access NTP. Stop here.
    while True:
        pass
# Synchronize time with NTP server once
try:
    ntptime.settime()
    print("Time synchronized with NTP server.")
except OSError as e:
    print("Failed to synchronize time:", e)

# Now enter an infinite loop where we print the current time every 10 seconds
while True:
    current_time = time.localtime()
    hour = current_time[3]
    minute = current_time[4]
    second = current_time[5]
    print("Current UTC Time: {:02d}:{:02d}:{:02d}".format(hour, minute, second))
    # Wait 10 seconds before the next print
    time.sleep(10)
```

Explanation:

1. **WiFi Connection**:
   Calls **`connect_WiFi()`** from **`wifi_tools.py`** to connect to the provided SSID and password.
2. **One-Time NTP Synchronization**:
   Uses **`ntptime.settime()`** once to synchronize the device's RTC.
3. **Infinite Loop**:
   - Every 10 seconds, reads the current local time (which is in UTC if you haven't adjusted otherwise).
   - Extracts hour, minute, and second from **`time.localtime()`** return value.
   - Prints the time in a formatted manner.
   - Does **not** call **`ntptime.settime()`** again, so it does not re-synchronize. The time relies on the ESP32's RTC after the initial synchronization.

Note:

- If you need local time (not UTC), you'll have to adjust the returned time accordingly.
- The ESP32's RTC should remain reasonably accurate for short periods; for long durations, you may want periodic re-synchronization, but that's not requested here.

1.3.2 Reading time from NTP server – `high_power` and `low_power` states

Below is an example **`main.py`** script incorporating all the requested features:

1. Uses RTC memory to track the cycle count.
2. On the first cycle (when cycle = 0), connects to WiFi, synchronizes time via NTP, and then disconnects.
3. On subsequent cycles, it does not reconnect to WiFi or resynchronize time.
4. Reads the current time (hour, minute, second) from the RTC.
5. Displays the time on an SSD1306 OLED display for 3 seconds.
6. After 3 seconds, turns off the display.
7. Goes into deep sleep for 6 seconds, then repeats.

Prerequisites:

- Ensure **`wifi_tools.py`** is on the device.
- Ensure **`ssd1306.py`** driver is on the device (commonly available from MicroPython libraries).
- Adjust the SDA and SCL pin numbers for I2C as needed.
- Adjust the display size (128x64 is assumed below).

```
import time
import ntptime
import machine
from wifi_tools import *
from machine import Pin, I2C
from ssd1306 import SSD1306_I2C
# Initialize RTC
rtc = machine.RTC()
# Retrieve cycle counter from RTC memory
```

```
rtc_mem = rtc.memory()
if len(rtc_mem) == 0:
    cycle = 0
else:
    cycle = int(rtc_mem.decode())

if cycle == 0:
    # First cycle: Connect to WiFi and sync time
    SSID = 'Livebox-08B0'
    PASSWORD = 'G79ji6dtEptVTPWmZP'
    if connect_WiFi(SSID, PASSWORD):
        try:
            ntptime.settime()
            print("Time synchronized with NTP server.")
        except OSError as e:
            print("Failed to synchronize time:", e)
        disconnect_WiFi()
        print("Disconnected from WiFi.")
    else:
        print("Failed to connect to WiFi:", SSID)
        # Continue anyway; the time may not be correct without NTP sync
else:
    # Subsequent cycles: Do not connect to WiFi or re-sync time
    print("No WiFi connection this cycle. Using previously set RTC time.")
# Read current time from RTC
current_time = time.localtime()
hour = current_time[3]
minute = current_time[4]
second = current_time[5]
print("Current UTC Time: {:02d}:{:02d}:{:02d}".format(hour, minute, second))
# Initialize the OLED display
# Adjust frequency and dimensions as per your display specs
i2c = I2C(scl=Pin(9), sda=Pin(8), freq=400000)
oled = SSD1306_I2C(128, 64, i2c)
# Display the current time
oled.fill(0)
oled.text("Time (UTC):", 0, 0)
oled.text("{:02d}:{:02d}:{:02d}".format(hour, minute, second), 0, 16)
oled.show()
# Keep the display on for 3 seconds
time.sleep(3)
# Power off the display
oled.poweroff()
print("Display powered off.")
# Increment cycle counter and store in RTC memory
cycle += 1
rtc.memory(str(cycle).encode())
# Deep sleep for 6 seconds
print("Entering deep sleep for 6 seconds...")
time.sleep_ms(1000)  # small delay before sleep
machine.deepsleep(6000)
```

Explanation:

- The code checks **`rtc.memory()`** to determine which cycle it is on.
- If **`cycle == 0`**, it connects to WiFi and tries to synchronize time using NTP (**`ntptime.settime()`**).
- On subsequent cycles, it relies on the RTC's already set time.
- The current time is displayed on the SSD1306 OLED for 3 seconds.
- After that, the display is powered off using **`oled.poweroff()`**.
- The cycle counter is incremented and saved back to RTC memory.
- The device then enters deep sleep for 6 seconds, after which it restarts and runs the code again.
- On the next run, **`cycle`** is now > 0, so it skips the WiFi connection and NTP sync steps.
- The process repeats infinitely.

To do

1. Analyze the codes of the above two examples.
2. Try to understand the role of RTC memory (RAM) associated to the ULP unit and processor.
3. Below is a simple example generated yo illustrate independently the mechanism of RTC memory and its role within a cycle with deepsleep mode.

Mini-project:

Complete the above IoT architecture by sending time-stamped packets over UDP datagrams to the distant receiver

1.4 Understanding RTC memory (and timer)

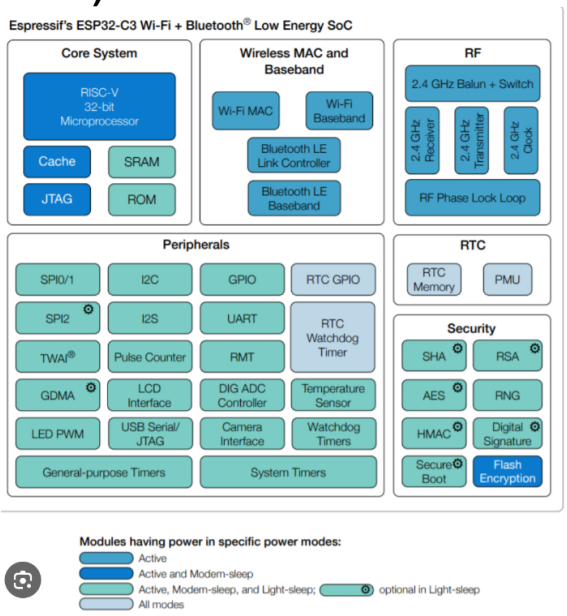


Fig 1.1 ESP32C3 SoC with **RTC** components including: RTC timer/watchdog, RTC memory, and RTC GPIO

Below we have simple example of using RTC memory on the ESP32 with MicroPython to keep track of how many times the device has woken from deep sleep. The counter is stored in RTC memory, which persists across deep sleep cycles.

What the code does:

1. Tries to read a counter value from RTC memory.
2. If there's no value (first run), it sets the counter to 0.
3. Prints the current counter value.
4. Increments the counter and writes it back to RTC memory.
5. Goes into deep sleep for 6 seconds.
6. On wake-up, the cycle repeats, and the counter is incremented again.

```
import machine
import time

# Get the RTC object
rtc = machine.RTC()
# Read the current memory contents
rtc_mem = rtc.memory()
if len(rtc_mem) == 0:
    # If empty, this is the first cycle
    cycle = 0
else:
    # Convert stored bytes to integer
    cycle = int(rtc_mem.decode())
# Print the current cycle count
print("Current cycle count:", cycle)
# Increment the cycle counter
cycle += 1

# Save the new cycle count to RTC memory
rtc.memory(str(cycle).encode())

# Sleep for 6 seconds
print("Going to deep sleep for 6 seconds...")
time.sleep_ms(1000)  # small delay before entering sleep
machine.deepsleep(6000)
```

Explanation:

- **RTC Memory**: `rtc.memory()` returns a `bytes` object. We decode it to a string and then convert to `int`.
- **First Cycle**: If `rtc.memory()` is empty, we assume it's the first cycle and start from zero.
- **Update Cycle**: We print and then increment the cycle count. We write it back to RTC memory by converting it to a string and then to `bytes`.
- **Deep Sleep**: We call `machine.deepsleep(6000)` to put the ESP32 into deep sleep mode for **6 seconds**. When it wakes up, **it restarts from** `main.py`, and thus the cycle is repeated, with the updated counter read from RTC memory.

To do

Complete the above example of RTC counter with display showing the value of the counter during 3 seconds in `high_power` **stage**.
Use the following module to display the counter value:

```
from machine import Pin, I2C, RTC, deepsleep
from ssd1306 import SSD1306_I2C
import time

def display_count(sda, scl, counter, dur):
    i2c = I2C(scl=Pin(scl), sda=Pin(sda), freq=400000)
    oled = SSD1306_I2C(128, 64, i2c)
    oled.fill(0)
    oled.text("RTC Counter", 0, 0)
    oled.text("Count: "+ str(counter), 0, 16)
    oled.show()
    time.sleep(dur)
    oled.poweroff()

# Get the RTC object
rtc = RTC()
# Read the current memory contents
rtc_mem = rtc.memory()
if len(rtc_mem) == 0:
    # If empty, this is the first cycle
    cycle = 0
else:
    # Convert stored bytes to integer
    cycle = int(rtc_mem.decode())
# Print the current cycle count
print("Current cycle count:", cycle)
# Increment the cycle counter
cycle += 1
display_count(8,9,cycle,3)
# Save the new cycle count to RTC memory
rtc.memory(str(cycle).encode())
print("Going to deep sleep for 6 seconds...")
time.sleep_ms(1000)  # small delay before entering sleep
deepsleep(6000)
```

1.5 Sending and receiving data from MQTT server/broker

For **IoT** devices, **MQTT** (Message Queuing Telemetry Transport) is a popular protocol due to its lightweight, **publish-subscribe nature,** which makes it well-suited for resource-constrained devices. There are several MQTT server (broker) options and services that we can use for our IoT projects: **broker.emqx.io**, **`test.mosquitto.org`**, ..

MQTT operates in three main Quality of Service (QoS) levels that define the guarantee of delivery for a message between the publisher and the subscriber. These operational modes are essential for managing the trade-off between performance and reliability in IoT communications.

1.5.1 MQTT Operational Modes (QoS Levels)

1. **QoS Level 0: "At most once" (Fire-and-Forget)**
    - The publisher sends the message once and does not store it. (**`PUB`**)
    - The broker delivers the message once to the subscriber without acknowledgment (**`SUB`**).
2. **QoS Level 1: "At least once**
    - The publisher sends the message and waits for an acknowledgment (**`PUBACK`**) from the broker.
    - If no acknowledgment is received, the message is resent.
    - The broker stores the message and delivers it to the subscriber, waiting for an acknowledgment.
3. **QoS Level 2: "Exactly once"**
    - The publisher sends a message (**`PUBREL`**) and waits for a release acknowledgment from the broker (**`PUBREC`**).
    - The broker stores the message and sends an acknowledgment to the publisher.
    - The broker delivers the message to the subscriber and waits for an acknowledgment (**`PUBCOMP`**).
    - The broker sends a completion acknowledgment to the publisher.

In our case we are going to use only **Level 0**.

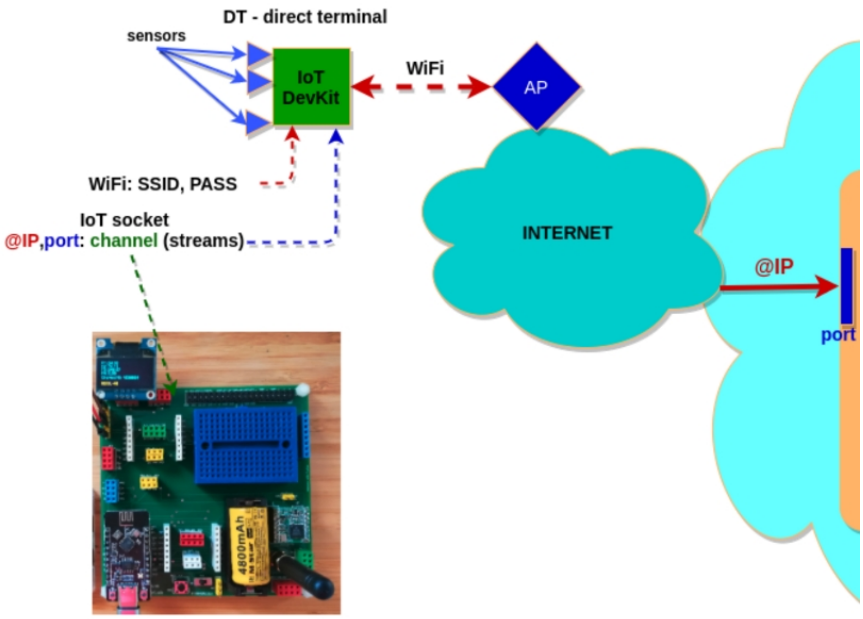


Fig 1.2 Sending message to MQTT broker (topic)

1.5.2 MQTT topics and messages

MQTT messages are delivered to the **topics** that are subscribed to and managed by the **broker**. In our system, these topics are represented by **numeric identifiers** corresponding to **IoT socket channel numbers**.
In the context of MQTT, an **IoT socket** is defined by the combination of:

- The **IP address** of the MQTT server,
- The **service port** (typically **1883**),
- The **topic (channel) number**.
-

This structure allows sensor data and control messages to be efficiently routed between IoT devices and the corresponding application channels.

For example:

```
# MQTT Configuration
MQTT_BROKER = 'broker.emqx.io'      # IP address
MQTT_PORT = 1883                    # port number
MQTT_TOPIC = '145879'               # channel number
```

1.5.3 Sending (publishing) MQTT messages: H (`high_power` stage) only mode

The following example uses the **`umqtt.simple`** module to establish communication with a specified **MQTT server (broker)** and to **publish a message**.
Note that the **topic name** can be interpreted as the **channel number**. The message payload itself may be composed of multiple fields (e.g., **`f1:, f2:, f3:`**), each carrying the value of a corresponding sensor data stream.
Below is the **MicroPython code for the ESP32**, which performs the following steps:

1. Connects to a **Wi-Fi access point** using the **`wifi_tools.py`** module.
2. Reads sensor data from:
    - **MAX44009** (luminosity),
    - **SHT21** (temperature and humidity).
3. Publishes the collected sensor data as an **MQTT message** to the configured topic.

This example assumes that:

- A **Wi-Fi helper module** (e.g., **`wifi_tools.py`**) is available for network connectivity,
- A **`sensors.py` module** is available to interface with the MAX44009 and SHT21 sensors.

```
from machine import Pin, I2C
import ubinascii
import machine
from wifi_tools import *
from sensors import *
import time
from umqtt.simple import MQTTClient

# WiFi credentials
SSID = 'PhoneAP'
PASSWORD = 'smartcomputerlab'

# MQTT broker details
MQTT_BROKER = "broker.emqx.io"  # Replace with your broker address
MQTT_PORT = 1883
MQTT_CLIENT_ID = ubinascii.hexlify(machine.unique_id())  # Unique client ID
MQTT_TOPIC = 'esp32c3/sensor_data'  # Replace with your topic

# Initialize MQTT client
client = MQTTClient(MQTT_CLIENT_ID, MQTT_BROKER, port=MQTT_PORT)

def connect_mqtt():
    """Connect to the MQTT broker."""
    try:
        client.connect()
        print("Connected to MQTT broker.")
    except Exception as e:
        print("Failed to connect to MQTT broker:", e)
```

```
def disconnect_mqtt():
    client.disconnect()
    print("Disconnected from MQTT broker.")

def publish_sensor_data():
    """Publish sensor data to MQTT broker."""
    luminosity, temperature, humidity = sensors(sda=8, scl=9)

    if luminosity is not None and temperature is not None and humidity is not None:
        message = {
            "luminosity": luminosity,
            "temperature": temperature,
            "humidity": humidity
        }
        client.publish(MQTT_TOPIC, str(message))
        print("Published:", message)
    else:
        print("Failed to publish sensor data.")

# Main function
def main():
    # Initialize WiFi and connect to access point
    res=connect_WiFi(SSID, PASSWORD)
    if res==True:
        print("WiFi connected")
    time.sleep(1)
    connect_mqtt()
    try:
        # Publish sensor data every 10 seconds
        while True:
            publish_sensor_data()
            time.sleep(10)
    finally:
        # Disconnect from MQTT and WiFi on exit
        disconnect_mqtt()
        time.sleep(1)
        disconnect_WiFi()

# Run the main function
main()
```

To do:

Provide your WiFi credentials are, for example, "**PhoneAP**" and "**smartcomputerlab**".

The broker address is: '**broker.emqx.io**'

Test your example.

Terminal printout:

```
start
Network config: ('192.168.222.67', '255.255.255.0', '192.168.222.85', '192.168.222.85')
already connected
Network config: ('192.168.222.67', '255.255.255.0', '192.168.222.85', '192.168.222.85')
Connected to MQTT broker.
Warning: I2C(-1, ...) is deprecated, use SoftI2C(...) instead
Published: {'humidity': 45.07498, 'luminosity': 30.6, 'temperature': 22.20891}
Warning: I2C(-1, ...) is deprecated, use SoftI2C(...) instead
Published: {'humidity': 47.35617, 'luminosity': 27.54, 'temperature': 22.13383}
```

The published messages may be seen with mosquitto application running on PC or MyMQTT on Android.

```
$ mosquitto_sub -h broker.emqx.io -t esp32/sensor_data
{'humidity': 44.85373, 'luminosity': 30.6, 'temperature': 22.29471}
{'humidity': 44.67825, 'luminosity': 30.6, 'temperature': 22.29471}
{'humidity': 44.7164, 'luminosity': 33.66, 'temperature': 22.26254}
{'humidity': 45.84555, 'luminosity': 33.66, 'temperature': 22.27325}
```

1.5.4 Sending (publishing) MQTT messages with `high_power` and `low_power` stages and `delta` parameter

The following example extends the previous code by introducing **two key mechanisms** that significantly reduce the **power (current) consumption** of the terminal node.

- **The first mechanism** is the introduction of a **low-power stage** using **deep-sleep mode**. This allows the device to remain inactive for extended periods while consuming minimal energy.
- **The second mechanism** is based on a **delta parameter**. This parameter is used to eliminate unnecessary transmission phases when the difference between the current sensor value and the last transmitted value is smaller than a predefined threshold.

For example, consider a delta value of **0.1**. If the last transmitted temperature was **20.14 °C** and the current measured temperature is **20.20 °C**, the difference (**0.06 °C**) is below the delta threshold. In this case, there is no need to transmit the new value. By avoiding the transmission phase, typically the most energy-intensive operation, the system achieves **significant energy savings**.

```
---------------------------------------------------------------------------------------------------------------------------------------------------
from machine import Pin, I2C, deepsleep
import ubinascii
import machine
from wifi_tools import *
from sensors import sensors
import time
from umqtt.simple import MQTTClient
# WiFi credentials
SSID = 'PhoneAP'
PASSWORD = 'smartcomputerlab'
# MQTT broker details
MQTT_BROKER = 'broker.emqx.io'  # Replace with your broker address
MQTT_PORT = 1883
MQTT_CLIENT_ID = ubinascii.hexlify(machine.unique_id())  # Unique client ID
MQTT_TOPIC = 'esp32/sensor_data'  # Replace with your topic
# Initialize MQTT client
client = MQTTClient(MQTT_CLIENT_ID, MQTT_BROKER, port=MQTT_PORT)
rtc = machine.RTC()

def connect_mqtt():
    """Connect to the MQTT broker."""
    try:
        client.connect()
        print("Connected to MQTT broker.")
    except Exception as e:
        print("Failed to connect to MQTT broker:", e)

def disconnect_mqtt():
    """Disconnect from the MQTT broker."""
    client.disconnect()
    print("Disconnected from MQTT broker.")

def publish_sensor_data(luminosity, temperature, humidity):
    """Publish sensor data to MQTT broker."""
    if luminosity is not None and temperature is not None and humidity is not None:
        message = {
            "luminosity": luminosity,
            "temperature": temperature,
            "humidity": humidity
        }
        client.publish(MQTT_TOPIC, str(message))
        print("Published:", message)
    else:
        print("Failed to publish sensor data.")
# Main function
def main():
    delta=0.1
    # Retrieve cycle counter from RTC memory
    rtc_mem = rtc.memory()
    if len(rtc_mem) == 0:
        stemp = 20.0
    else:
        stemp = float(rtc_mem.decode())
    luminosity, temperature, humidity = sensors(sda=8, scl=9)
    if (abs(stemp-temperature)> delta):
        rtc.memory(str(temperature).encode())
        # Initialize WiFi and connect to access point
        connect_WiFi(SSID, PASSWORD)
        connect_mqtt()          # Connect to MQTT
        time.sleep(1)
```

```
        publish_sensor_data(luminosity, temperature, humidity)
        time.sleep(1)
        disconnect_mqtt() # Disconnect from MQTT and WiFi on exit
        time.sleep(1)
        disconnect_WiFi()
        print("message sent")
    else:
        print("message not sent")

    deepsleep(10*1000)

# Run the main function
main()
```

1.5.5 Receiving (and displaying) sensor data from MQTT broker

The **MQTT receiver** on a **direct terminal** operates **only during high-power stages**, as it requires the Wi-Fi interface and processor to be active.

Below is an example of **MicroPython code for an ESP32** that:

- Connects to a **Wi-Fi access point**,
- Subscribes to an **MQTT topic** on the **`broker.emqx.io`** MQTT broker,
- Displays received messages on an **SSD1306 OLED display**.

This example uses the following components and libraries:

- The **`wifi_tools.py`** module to manage Wi-Fi connectivity,
- The **`umqtt.simple`** library for MQTT communication,
- The **`ssd1306`** library to drive the OLED display.

```
from machine import Pin, I2C
from wifi_tools import connect, disconnect, scan
from umqtt.simple import MQTTClient
from ssd1306 import SSD1306_I2C
from time import sleep_ms
# WiFi credentials
SSID = 'PhoneAP'
PASSWORD = 'smartcomputerlab'
# MQTT broker details
MQTT_BROKER = 'broker.emqx.io'
MQTT_PORT = 1883
MQTT_TOPIC = b'sensor/readings'  # Topic as bytes

# Initialize I2C and SSD1306 OLED display (SDA=Pin 8, SCL=Pin 9)
i2c = I2C(0, scl=Pin(9), sda=Pin(8), freq=400000)
oled = SSD1306_I2C(128, 64, i2c)

# Function to display messages on the OLED
def display_message(message):
    oled.fill(0)  # Clear display
    oled.text("MQTT Message:", 0, 0)
    oled.text(message, 0, 20)
    oled.show()

# MQTT message callback function
def mqtt_callback(topic, msg):
    print("Received message:", msg)
    display_message(msg.decode())  # Display message on OLED

# Initialize WiFi and connect to access point
connect("PhoneAP","smartcomputerlab")
# Initialize MQTT client
client = MQTTClient('esp32_client', MQTT_BROKER, port=MQTT_PORT)

try:
    # Connect to MQTT broker
    client.set_callback(mqtt_callback)
    client.connect()
    print("Connected to MQTT broker.")
    # Subscribe to the topic
    client.subscribe(MQTT_TOPIC)
    print("Subscribed to topic:", MQTT_TOPIC)
    # Main loop to check for messages
    while True:
        # Wait for incoming messages
```

```
        client.wait_msg()
        # Short delay between checks to avoid high CPU usage
        sleep_ms(500)
except Exception as e:
    print("Error:", e)
finally:
    # Disconnect from MQTT broker and WiFi
    client.disconnect()
    wifi.disconnect()
    print("Disconnected from MQTT broker and WiFi.")
```

Explanation of the Code

1. **WiFi Connection**:
    - The **`connect_WiFi()`** function from **`wifi_tools.py`** connects to the specified WiFi network.
2. **MQTT Setup**:
    - The MQTT broker is set to **`broker.emqx.io`** on port **1883**.
    - The MQTT topic **`sensor/readings`** is defined, and the client subscribes to this topic.
3. **Display Function**:
    - **`display_message(message)`**: Clears the OLED display, then shows the received message on it.
4. **MQTT Callback**:
    - **`mqtt_callback(topic, msg)`**: Triggered when a message arrives on the subscribed topic.
    - Decodes the message and calls `display_message` to display it on the OLED.
5. **Main Loop**:
    - Calls **`client.wait_msg()`** to listen for new messages on the subscribed topic.
    - A delay (**`sleep_ms(500)`**) prevents the loop from consuming too much CPU.
6. **Error Handling and Cleanup**:
    - The code handles errors by disconnecting the MQTT and WiFi connections in the **`finally`** block.

To do

The code works out of the box. The received message is displayed on the SSD1306 screen.
Send a message from the PC terminal with **`mosquitto_pub`** or from your smartphone via **MyMQTT application**.
The following is terminal printout that corresponds to the message sent from host computer with **`mosquitto`** client:

```
~$mosquitto_pub -h broker.emqx.io -t esp32c3/sensor_data -m Temp:21.5
```

Terminal printout:

```
MPY: soft reboot
Connected to MQTT broker.
Subscribed to topic: esp32c3/sensor_data
Received message: b'Temp:21.5'
```

1.6 Sending sensor's data (streaming) to ThingSpeak server

An **MQTT broker service** provides message routing between publishers and subscribers, but it does **not** offer a built-in **database** for long-term storage of IoT data, nor a **user interface** for visualizing data streams.
To address these limitations, we use a **complete IoT server platform** such as **ThingSpeak**.
ThingSpeak was originally released as an open-source project and is now also available as a commercial cloud service at **`ThingSpeak.com`**, operated by **MathWorks**.

For small projects, ThingSpeak offers a **free account** with limited resources, including:

- Up to **10 channels**,
- A maximum of **256 stored entries per channel**,
- A maximum data write frequency of **50 mHz** (i.e., a minimum update period of **20 seconds**).

After creating and initializing a ThingSpeak account, you can define a **new channel** and assign it a custom name to organize your sensor data.

According to Wikipedia, *MathWorks, Inc.* is an American privately held corporation that specializes in **mathematical computing software**. Its major products include **MATLAB** and **Simulink**, which are widely used for data analysis, modeling, and simulation.

What is ThingSpeak?

ThingSpeak is a cloud-based Internet of Things (IoT) platform designed for collecting, storing, analyzing, and visualizing data from various IoT devices, sensors, and applications. Developed by MathWorks, ThingSpeak integrates easily with MATLAB for data analysis, making it a popular choice among researchers, hobbyists, and professionals for prototyping and small to medium-scale IoT deployments.

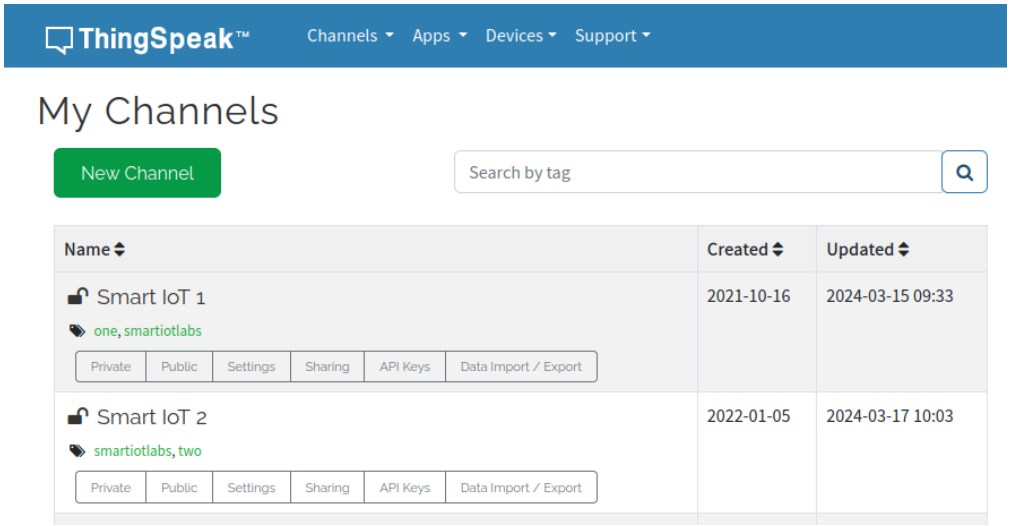
Key features include:

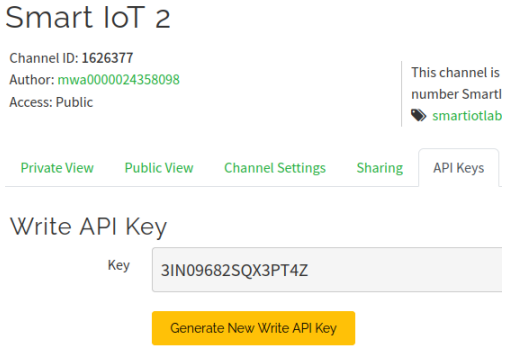
1. **Data Ingestion**: ThingSpeak allows you to send sensor readings or data points to its cloud service using simple HTTP requests or MQTT.
2. **Data Storage**: Collected data is stored in channels (like tables or containers) where each channel can have multiple fields to represent different data streams (e.g., temperature, humidity, pressure).
3. **Visualization**: You can view your sensor data in real-time using built-in charts, plots, and gauges. The platform also supports time-based data visualization.
4. **Data Analysis**: ThingSpeak integrates with MATLAB, enabling you to run MATLAB code directly within ThingSpeak for advanced analysis, processing, and decision-making.
5. **Alerts and Actions**: You can set up triggers (based on data conditions) that send alerts or perform actions, such as posting to social media, sending emails, or controlling devices.

How to Use ThingSpeak to Keep Sensor Data:

1. **Create a ThingSpeak Account**:
    - Go to https://thingspeak.com/ and sign up for a free account.
    - A free account allows limited channels and data rates, which is often sufficient for small projects.
2. **Create a Channel**:
    - After logging in, create a new channel.
    - A channel is like a container for your sensor data and can have up to 8 fields. For example, field1 might store temperature data, field2 might store humidity, and so forth.
    - Name your channel and give it a description. Enable the fields you need and name them appropriately (e.g., "Temperature (°C)").
3. **Obtain the API Keys**:
    - Each channel has unique Write and Read API Keys.
    - The Write API Key is used to send data from your device to ThingSpeak.
    - The Read API Key (or making the channel public) is used to retrieve data for visualization or other applications.

4. **Send Data to ThingSpeak**:
    - With the Write API Key, your device (e.g., an ESP32, Raspberry Pi, or Arduino with a sensor) can send HTTP POST or GET requests to ThingSpeak's endpoint.
    - The endpoint typically looks like:
      `https://api.thingspeak.com/update?api_key=YOUR_WRITE_KEY&field1=VALUE1&field2=VALUE2...`
    - Every time your device successfully sends a request with sensor values, ThingSpeak stores the new data point in your channel.
5. **Visualize Your Data**:
    - Once data is stored, you can go to your channel's page on ThingSpeak and view automatically generated charts for each field.
    - You can also embed these visualizations in your own website or retrieve the data using JSON endpoints for custom dashboards.
6. **Perform Analysis (Optional)**:
    - Use the MATLAB Analysis and MATLAB Visualizations apps within ThingSpeak to run code on your stored data.
    - For example, you can calculate moving averages, detect anomalies, or convert raw readings into more meaningful metrics.
2. **Set Up Triggers and Actions**:
    - With the React app on ThingSpeak, you can monitor your data and trigger actions when certain conditions are met.
    - For instance, send an email alert if the temperature exceeds a threshold, or publish a message to social media.





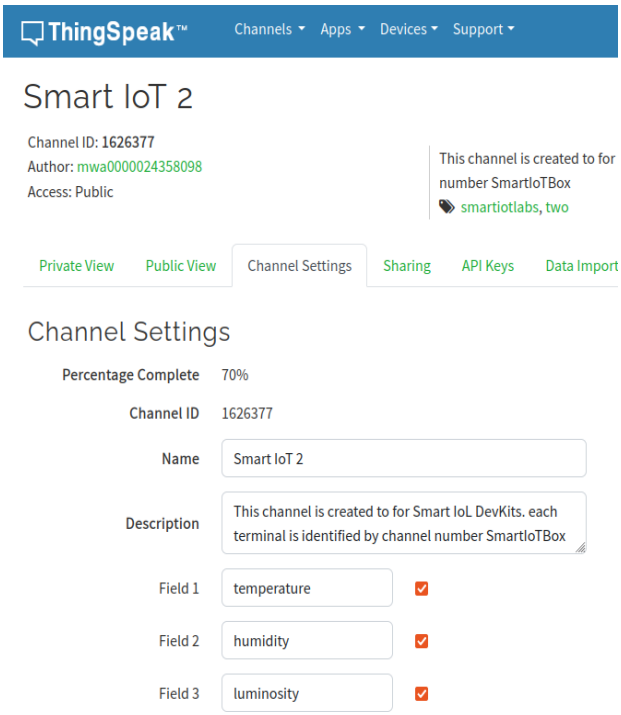


Fig.1.3 ThingSpeak: new channel creation (`1626377`) with 3 fields and its write API key: `3IN09682SQX3PT4Z1`

1.6.1 Sending (writing) data to ThingSpeak

To send sensor data to the ThingSpeak server using MicroPython, you can follow the code below. We are using the **`urequests`** library to send HTTP requests. The code includes channel number and **`AP_KEY`** to write/read the channel (fields) content.

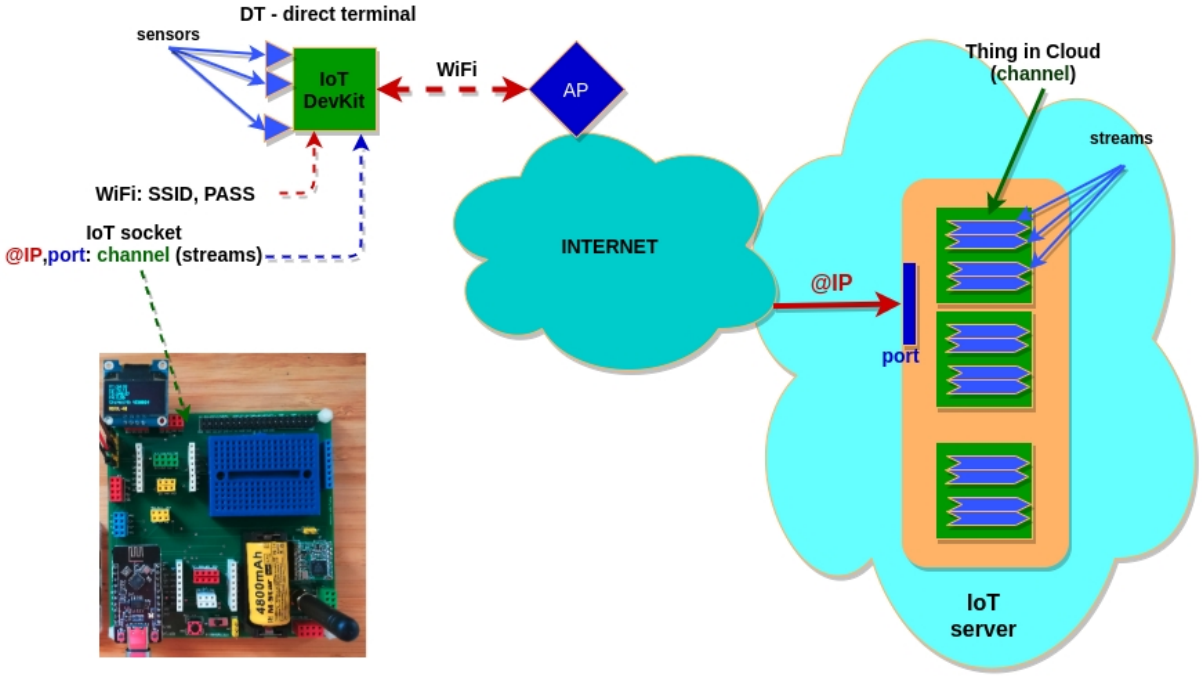


Fig.1.4 Identification (mapping) scheme of Direct Terminal on **ThinSpeak** server: WiFi and ThingSpeak **parameters**.

Below is the MicroPython code for an ESP32 to send sensor data from **MAX44009** (luminosity) and **SHT21** (temperature and humidity) sensors to **ThingSpeak**. This code uses the **`wifi_tools.py`** module to handle WiFi connectivity, then sends data to ThingSpeak via an HTTP GET request.

```
from machine import Pin, I2C, deepsleep
import ubinascii, urequests
import machine
from wifi_tools import *
from sensors import sensors
import time
# WiFi credentials
SSID = 'PhoneAP'
PASSWORD = 'smartcomputerlab'
# Initialize MQTT client
rtc = machine.RTC()

# Function to send data to ThingSpeak
def send_data_to_thingspeak(lumi, temp, humi):
    try:
        sf1="&field1="+str(lumi); sf2="&field2="+str(temp); sf3="&field3="+str(humi)
        url = "https://107.23.148.232/update?key=YOX31M0EDKO0JATK"+sf1+sf2+sf3
        response = urequests.get(url)
        response.close()
        print("Data sent to ThingSpeak:", lumi, temp, humi)
    except Exception as e:
        print("Failed to send data:", e)

def main():
    delta=0.1
    rtc_mem = rtc.memory()
    if len(rtc_mem) == 0:
        stemp = 20.0
    else:
        stemp = float(rtc_mem.decode())

    luminosity, temperature, humidity = sensors(sda=8, scl=9)
    if (abs(stemp-temperature)> delta):
        rtc.memory(str(temperature).encode())
        connect_WiFi(SSID, PASSWORD)
        print("WiFi connected")
        send_data_to_thingspeak(luminosity, temperature, humidity)
        time.sleep(1)
        disconnect_WiFi()

    deepsleep(15*1000)

# Run the main function
main()
```

Explanation of the Code

1. **WiFi Connection**:
    - The **`connect_WiFi()`** function uses **`wifi_tools.py`** to connect to the WiFi network using the provided SSID and password.
2. **Sensor Initialization**:
    - The MAX44009 (luminosity) and SHT21 (temperature and humidity) sensors are connected over I2C (SDA on Pin 8, SCL on Pin 9). They are read with **`lumi,temp,humi=sensors(sda=8,scl=9)`** function predefined in **`sensors.py`** module
3. **Sending Data to ThingSpeak**:
    - The `send_data_to_thingspeak` function constructs a URL to send data using an HTTP GET request. The URL includes:
        - The **API write key** (to authenticate with ThingSpeak).
        - **Field parameters** (`field1`, `field2`, `field3`) which ThingSpeak uses to map the sensor data to specific fields in the channel.
    - The `urequests.get()` method sends the request, and after the response is received, the connection is closed to free up resources.
4. **Main Loop**:
    - Reads sensor values from MAX44009 and SHT21 sensors.
    - Calls `send_data_to_thingspeak` to upload the values to ThingSpeak.
    - Pauses for 15 seconds between data uploads, which aligns with ThingSpeak's free-tier update rate limit.

Important Notes

- **Field Mapping (streams)**:
  - Ensure the ThingSpeak channel has been set up with appropriate fields (e.g., Field 1 = Luminosity, Field 2 = Temperature, Field 3 = Humidity).
- **ThingSpeak Update Interval**:
  - ThingSpeak's free tier allows updates every 15 seconds. Adjust the `time.sleep(15)` interval if you are using a paid plan or need less frequent updates.
- **API Key**:
  - Keep your ThingSpeak API key secure, and do not share it publicly.

Execution:

```
MPY: soft reboot
('192.168.45.136', '255.255.255.0', '192.168.45.115', '192.168.45.115')
WiFi connected
Data sent to ThingSpeak: 220.32 24.01072 49.95779
ESP-ROM:esp32c3-api1-20210207
Build:Feb  7 2021
rst:0x5 (DSLEEP),boot:0xc (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:1
load:0x3fcd5820,len:0xf28
load:0x403cc710,len:0x944
load:0x403ce710,len:0x2b1c
entry 0x403cc710
ESP-ROM:esp32c3-api1-20210207
Build:Feb  7 2021
rst:0x5 (DSLEEP),boot:0xc (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:1
load:0x3fcd5820,len:0xf28
load:0x403cc710,len:0x944
load:0x403ce710,len:0x2b1c
entry 0x403cc710
('192.168.45.136', '255.255.255.0', '192.168.45.115', '192.168.45.115')
WiFi connected
Data sent to ThingSpeak: 220.32 23.97855 47.89786
..
```

To do

Test the above example with ThingSpeak. (**First prepare <u>your</u> ThingSpeak channel and write key.**)

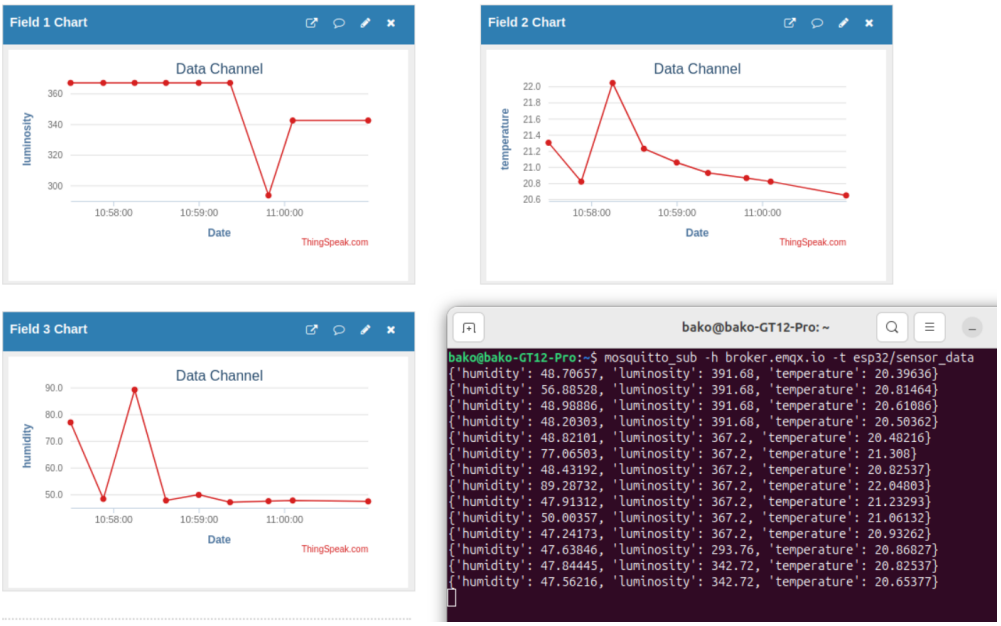


Fig.1.5 Combined results of sending data to ThingSpeak channel and MQTT topic

1.6.2 Receiving data from ThingSpeak

The following the MicroPython code for an ESP32 to read **temperature**, **humidity**, and **luminosity** data from **ThingSpeak**.

This code uses:

- The **`wifi_tools.py`** module to handle WiFi connectivity.
- The HTTP GET request to retrieve data from a specific ThingSpeak channel.

```
import urequests
import time
from wifi_tools import *
from display_sensors import *
# WiFi credentials
SSID = 'PhoneAP'
PASSWORD = 'smartcomputerlab'
# ThingSpeak API details
THINGSPEAK_CHANNEL_ID = '1538804'          # Replace with your ThingSpeak channel ID
THINGSPEAK_READ_API_KEY = '20E9AQVFW7Z6XXOM'       # Replace with your ThingSpeak read API key
THINGSPEAK_URL = f'https://api.thingspeak.com/channels/{THINGSPEAK_CHANNEL_ID}/feeds.json'

# Function to read data from ThingSpeak
def read_data_from_thingspeak():
    try:
        url = f"{THINGSPEAK_URL}?api_key={THINGSPEAK_READ_API_KEY}&results=1"
        response = urequests.get(url)
        data = response.json()
        response.close()
        # Extract the latest feed (last entry)
        feed = data['feeds'][0]
        lumi = feed['field1']; temp = feed['field2'];humi = feed['field3']
        print("Temperature:", temp); print("Humidity:", humi)
        print("Luminosity:", lumi);
        #display_sensors(8,9,lumi,temp,humi,5)  # 0 if continuous
        return lumi,temp, humi
    except Exception as e:
        print("Failed to retrieve data:", e)
        return None, None, None

def main():
    disconnect_WiFi()
    time.sleep(0.5)
    connect_WiFi(SSID,PASSWORD)
    while True:
        # Read and print the data from ThingSpeak
        luminosity, temperature, humidity = read_data_from_thingspeak()
        if temperature and humidity and luminosity:
            print(f"Read from ThingSpeak - Luminosity: {luminosity}, Temperature: {temperature},
Humidity: {humidity}")
        # Delay before the next read (e.g., every 15 seconds)
        time.sleep(15)

main()
```

Explanation :

1. **WiFi Connection**:
   - **`connect_WiFi()`** function from **`wifi_tools.py`** to connect to the WiFi network.
2. **Reading Data from ThingSpeak**:
   - **`read_data_from_thingspeak`** constructs a URL to retrieve data from ThingSpeak, using the **Channel ID** and **Read API Key**.
   - The URL includes **`results=1`**, so only the most recent data entry is returned.
   - **Parsing JSON Response**:
     - **`data['feeds'][0]`** accesses the latest feed entry.
     - **`field1`**, **`field2`**, and **`field3`** correspond to **temperature**, **humidity**, and **luminosity** respectively. Modify these mappings if your channel fields differ.
3. **Main Loop**:
   - Connects to WiFi, retrieves, and prints sensor values every 15 seconds, which is typical for ThingSpeak's free-tier update interval.

Important Notes

- **Field Mapping (streams)**:
  - Ensure your ThingSpeak channel fields correspond to luminosity (**`field1`**) temperature (**`field2`**), humidity (field3), and) as configured here. Adjust the code if necessary.
- **ThingSpeak Read Limit**:
  - The free tier allows reads every 15 seconds; you may increase the sleep time if necessary.

Execution:

```
MPY: soft reboot
('192.168.45.136', '255.255.255.0', '192.168.45.115', '192.168.45.115')
Temperature: 23.09
Humidity: 31.83
Luminosity: 73.44
Read from ThingSpeak - Luminosity: 73.44, Temperature: 23.09, Humidity: 31.83
Temperature: 23.09
Humidity: 31.83
Luminosity: 73.44
Read from ThingSpeak - Luminosity: 73.44, Temperature: 23.09, Humidity: 31.83
```

To do:

To complete the code we have to provide your WiFi credentials and ThingSpeak channel number and read key.
Develop the same example adding OLED display (SSD1306)

```
from machine import Pin, SoftI2C
from machine import Pin, SoftI2C
from ssd1306 import SSD1306_I2C
import time

def display_sensors(sda, scl, luminosity, temperature, humidity, duration):
    # Initialize I2C bus
    i2c = SoftI2C(scl=Pin(scl), sda=Pin(sda), freq=400000)
    # Adjust display width/height if different. Common sizes: 128x64 or 128x32.
    # Assuming a 128x64 display:
    width = 128;  height = 64
    # Initialize the OLED display
    oled = SSD1306_I2C(width, height, i2c)

    # Clear the display
    oled.fill(0)
    # Write text to display
    oled.text("Sensor readings", 0, 0)
    oled.text("Lumi: "+ str(luminosity), 0, 16)
    oled.text("Temp: "+ str(temperature), 0, 32)
    oled.text("Humi: "+ str(humidity), 0, 48)
    oled.show()
    if duration!=0:
        time.sleep(duration)
        oled.poweroff()
```

1.6.3 Receiving last time (record) from ThingSpeak

The following example shows how to get the time/data value for the last registered record in ThingSpeak channel. Note , that for a **public** channel, you need only to provide the **channel number**. If the channel is private you also have to provide the read key.

```
-----------------------------------------------------------------------------------------------------
import urequests, network, time
from wifi_tools import *
SSID = 'PhoneAP'
PASSWORD = 'smartcomputerlab'
THINGSPEAK_CHANNEL_ID = '1538804'          # Replace with your ThingSpeak channel ID
THINGSPEAK_API_KEY = '20E9AQVFW7Z6XXOM'    # Replace with your ThingSpeak read API key
# Function to fetch the last record from ThingSpeak
def fetch_last_record():
    url = f"https://api.thingspeak.com/channels/{THINGSPEAK_CHANNEL_ID}/feeds.json?results=1"
    print(f"Fetching last record from {url}")
    try:
        response = urequests.get(url)
        if response.status_code == 200:
            data = response.json()
            feeds = data.get("feeds", [])
            if feeds:
                last_record = feeds[0]
                created_at = last_record.get("created_at", "Unknown")
                print(f"Last record timestamp: {created_at}")
                return created_at
            else:
                print("No records found in the channel.")
        else:
            print("Failed to fetch data. Status code:", response.status_code)
    except Exception as e:
        print("Error fetching data:", e)
    return None
# Main program
def main():
    connect_WiFi(SSID,PASSWORD)
    last_timestamp = fetch_last_record()
    if last_timestamp:
        print("The last record was created at:", last_timestamp)
    else:
        print("Could not retrieve the last record.")

main()
-----------------------------------------------------------------------------------------------------
```

I**nstructions**

1. **Replace the placeholders**:
   - **`your_wifi_ssid`**: Your Wi-Fi network name.
   - **`your_wifi_password`**: Your Wi-Fi network password.
   - **`your_read_api_key`**: Your ThingSpeak read API key.
   - **`your_channel_id`**: Your ThingSpeak channel ID.
2. **Upload the script** to your ESP32 using an IDE like Thonny or uPyCraft.
3. **Run the program**. It will connect to the Wi-Fi, fetch the last record from your ThingSpeak channel, and print the timestamp of the last record.
4.

Make sure the ESP32 has internet access and your ThingSpeak channel is properly configured with data for accurate results.

```
-----------------------------------------------------------------------------------------------------------------------------------------------
```

Execution result:

```
>>> %Run -c $EDITOR_CONTENT
MPY: soft reboot
('192.168.45.136', '255.255.255.0', '192.168.45.115', '192.168.45.115')
Fetching last record from https://api.thingspeak.com/channels/1538804/feeds.json?results=1
Last record timestamp: 2025-03-13T08:28:47Z
The last record was created at: 2025-03-13T08:28:47Z
-----------------------------------------------------------------------------------------------------
```

To do

Integrate the time reading from ThingSpeak channel to the previous example with sensor values readings.

1.7 Building Low Power Protocol (adaptive) for Direct Terminal

In this section, we introduce **additional features** for building a **low-power protocol** that sends data directly to a **ThingSpeak server** via a Wi-Fi connection.

The protocol relies on several configurable parameters:

- **Cycle**: defined by a **`base_cycle (`**in seconds) and a **`max_cycle`** factor
- **Delta**: defined by **`min_delta`** and **`max_delta`**
- **Threshold**: defined by lower (**`t_low`**) and upper (**`t_high)`** limits

Cycle Parameter

The **cycle parameter** defines the duration of the **low-power stage**. The initial **`base_cycle`** value is fixed either in the program code or stored in **non-volatile memory (NVS)** (for example, 10 seconds). The effective cycle duration is obtained by multiplying the **`base_cycle`** by a **cycle factor**.

The **cycle factor**:

- Starts at **1**,
- Evolves dynamically based on the behavior of the sensor data.

If there are **two consecutive high-power stages without transmission**, the cycle factor is **multiplied by 2** and stored in **low-power RTC SRAM**.
This updated value is then used in the next operational cycle. Since the **main SRAM is powered off during deep sleep**, storing the cycle factor in RTC memory is essential.
Conversely, if there are **two consecutive high-power stages with transmission**, the cycle factor is **divided by 2**, reducing the sleep duration and increasing responsiveness.

Delta Parameter

The **delta parameter**, when applied to a temperature sensor, represents the difference between the **last transmitted value** and the **current measured value**.

The initial **`delta`** is set to **`max_delta`** (for example, 0.1). During extended periods without transmission, the **`delta`** value may gradually decrease toward **`min_delta`**, improving measurement sensitivity. Conversely, during periods with frequent transmissions, the **`delta`** value may increase toward **`max_delta`**, reducing unnecessary updates.

Threshold Parameter

The **threshold parameter** defines acceptable bounds for the sensor value using **`t_low`** and **`t_high`**. When the current sensor value crosses either threshold, the system must:

- Transmit the data immediately during the high-power stage,
- Reduce the cycle factor to shorten the next low-power stage.

This ensures rapid response to critical or abnormal conditions.

Adaptive Behavior and Memory Management

The **adaptive adjustment** of both the cycle factor and delta parameter allows:

- An **exponential extension** of low-power stage duration when conditions are stable,
- Fine-tuning of delta values to achieve maximum measurement precision when needed.

For correct operation, the **current cycle factor and delta values** must be stored in **low-power RTC SRAM**, which retains its contents during the SoC's deep-sleep mode.
However, RTC SRAM loses its contents when the device is completely powered off.

Therefore, **meta-parameters** such as **`base_cycle`**, **`max_cycle, min_delta, max_delta, t_low,`** and **`t_high`** must be stored in **non-volatile internal memory (NVS)** to ensure persistence across power cycles.

1.7.1 Adaptive sender code

The following is the program sending "adaptively" the sensor data to ThingSpeak server.

```
-----------------------------------------------------------------------------------------------
import time, ustruct, ubinascii, urequests
from machine import I2C, Pin, freq, deepsleep
from sensors import sensors
from wifi_tools import *
from rtc_tools import *
from nvs_tools import *

# WiFi credentials
SSID = 'PhoneAP'
PASS = 'smartcomputerlab'

nvs_key="param"                             # key to NVS records
led = Pin(3, Pin.OUT)

def send_wifi_data(wkey,lumi,temp,humi):
    try:
        sf1="&field1="+str(lumi); sf2="&field2="+str(temp); sf3="&field3="+str(humi)
        url = "https://thingspeak.com/update?key="+wkey.decode()+sf1+sf2+sf3
        response = urequests.get(url)
        response.close()
        print("Data sent to ThingSpeak:", lumi, temp, humi)
    except Exception as e:
        print("Failed to send packet:", e)

def main():
    global cdef; global ts_chan; global ncycle
    #freq(20000000)
    print("Reading ts from internal EEPROM...")
    len,ts_rparam = read_nvs_ts(nvs_key)
    if len:
        ts_chan,ts_wkey=ustruct.unpack("i16s",ts_rparam)
        print("len:",len,"ts_chan:",ts_chan,"ts_wkey:",ts_wkey.decode())
    print("Reading pow from internal EEPROM...")
    len,pow_rparam = read_nvs_power(nvs_key)
    if len:
        cdef,cmax,dmin,dmax,tlow,thigh=ustruct.unpack("2i4f",pow_rparam)
        print("len:",len,", cdef:",cdef,", cmax:",cmax,", dmin:",dmin,", dmax:",dmax,",
tlow:",tlow,", thigh:",thigh)

    while True:
        ncycle,npos,nneg= rtc_load_param()
        ssens= rtc_load_sensor(); sdelta= rtc_load_delta()
        print("ncycle:" +str(ncycle));
        lumi, temp, humi = sensors(sda=8, scl=9)
        print("Luminosity:", lumi, "lux")
        print("Temperature:", temp, "C")
        print("Humidity:", humi, "%")
        print("current: "+str(temp)+" saved: "+str(ssens));   # sensor is temperature
        print(dmin,dmax,sdelta);
        if abs(ssens-temp)>sdelta or temp>thigh or temp<tlow :  # testing delta and thresholds
            print("data to SEND")
            rtc_store_sensor(temp)
            led.on()
            if npos :
                if ncycle > 2:
                    ncycle= int(ncycle/2)
                else:
                    if sdelta< dmax:
                        sdelta = sdelta*2         # new delta
                        rtc_store_delta(sdelta)

            npos=npos+1; nneg=0  # positive and negative counters
            rtc_store_param(ncycle,npos,nneg)
            print(ts_wkey,lumi,temp,humi)
            if connect_WiFi("PhoneAP", "smartcomputerlab"):
                print("WiFi connected")
                send_wifi_data(ts_wkey,lumi, temp, humi)
                print("data packet sent")
                led.off()
            else:
                print("WiFi not connected")
        else:
            print("data packet NOT sent")
            if nneg :
                if ncycle < cmax:
                    ncycle = int(ncycle*2)           # maximum factor 64
```

```
                else :
                    if sdelta> dmin:
                        sdelta = sdelta/2
                        rtc_store_delta(sdelta)

            npos=0; nneg=nneg+1
            rtc_store_param(ncycle,npos,nneg)
            # waiting for ACK frame
        time.sleep(0.1)
        print(ncycle*cdef)
        print(sdelta)
        deepsleep(ncycle*cdef*1000)               # 10*1000 miliseconds

# Run the main program
main()
```

Execution – one cycle:
Loading the program to main SRAM from EEPROM (**`SPI – FLASH_BOOT`**)

```
ESP-ROM:esp32c3-api1-20210207
Build:Feb  7 2021
rst:0x5 (DSLEEP),boot:0xe (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:1
load:0x3fcd5820,len:0xf28
load:0x403cc710,len:0x944
load:0x403ce710,len:0x2b1c
entry 0x403cc710
```

Reading from internal EEPROM – NVS: thingspeak **write keyword** and **meta-parameters**

```
Reading ts from internal EEPROM...
len: 20 ts_chan: 1234 ts_wkey: YOX31M0EDKO0JATK
Reading pow from internal EEPROM...
len: 24 , cdef: 1 , cmax: 64 , dmin: 0.01 , dmax: 0.2 , tlow: 16.0 , thigh: 26.0
```

Cycle factor and reading sensors and saved sensor value:

```
ncycle:32
Luminosity: 146.88 lux
Temperature: 24.92236 C
Humidity: 51.25479 %
current: 24.92236 saved: 24.80438
```

Preparing data to send (or not!)

```
0.01 0.2 0.1
data to SEND
b'YOX31M0EDKO0JATK' 146.88 24.92236 51.25479
('192.168.45.90', '255.255.255.0', '192.168.45.115', '192.168.45.115')
WiFi connected
Data sent to ThingSpeak: 146.88 24.92236 51.25479
data packet sent
```

Calculating new cycle factor and delta value:

```
32
0.1
..
```

Going to **deepsleep** (**`low_power`** **stage**)

The following are **two screenshots** from PPK2 to show the evolution of the cycle factor (**cycle_base = 1s**).

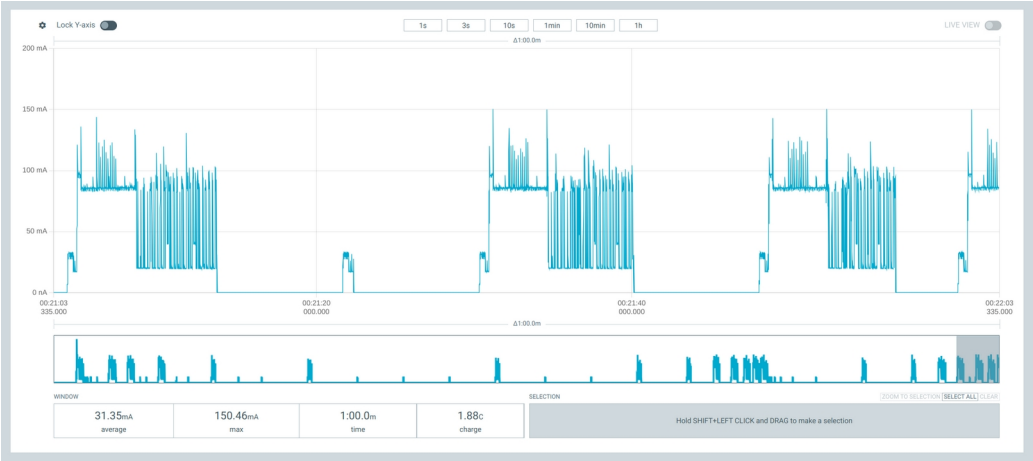


Fig.1.6a Power consumption evolution in a **30min window** (**minimap**) with **adaptive cycle factor** from 1 to 64. The delta value decreases (exponentially to **`min_delta`**) when the **`max_cycle`** factor is attained **two times in a row** and there is no transmission.

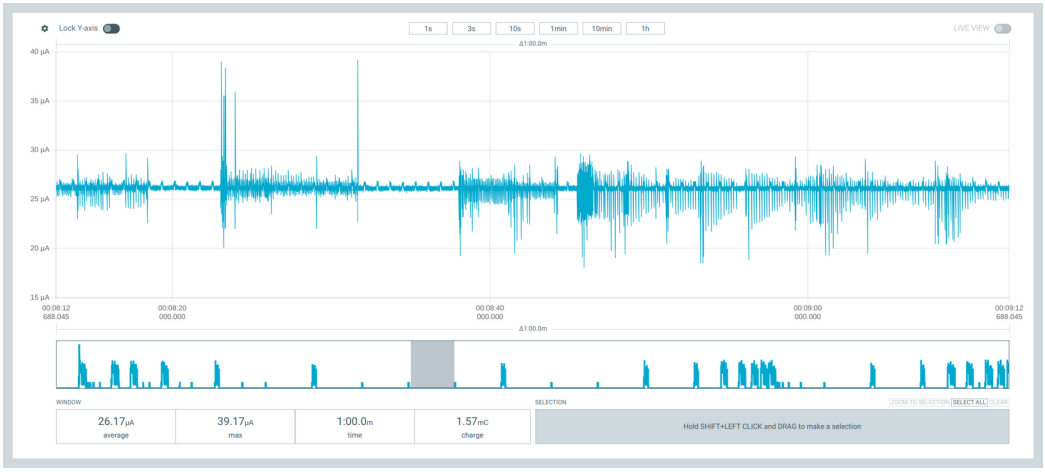


Fig 1.6b Power consumption during **`low_power stage (average current ~26µA)`**. Note very high power consumption required for WiFi connection and communication during the **`high_power stages`** **with transmission**.

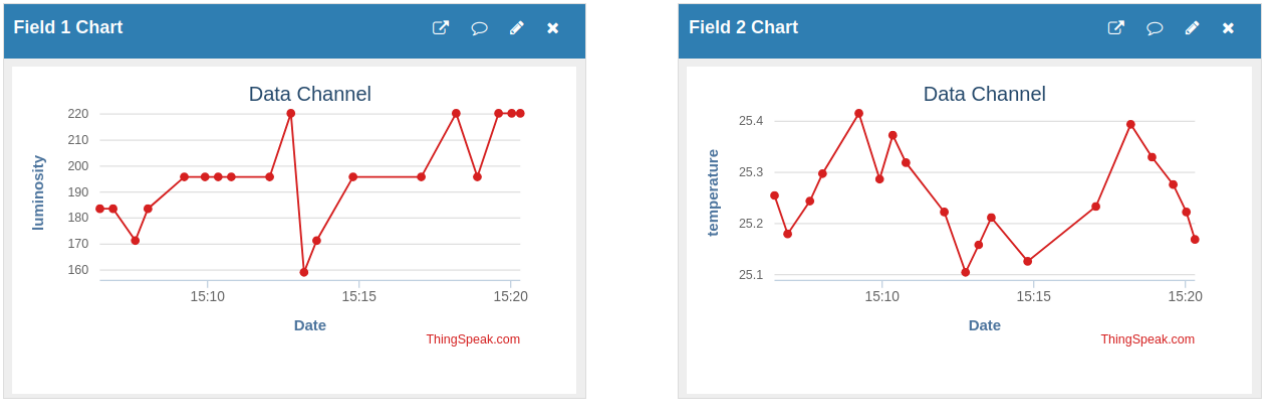


Fig 1.6c ThingSpeak diagrams : the luminosity and **temperature** points – evolution related to the execution cycles above.

Important note:

SoC frequency reduction is not allowed for WiFi communication, it must stay at 160MHz.

1.7.2 The tools:

In the above code we exploit the RTC memory as well as the NVS memory. To use these resources we need some additional functionalities. They are provided in **`rtc_tools.py`** and **`nvs_tools.py`** modules as follows:

```
# rtc_tools.py
import machine
import ustruct

rtc = machine.RTC()
# Function to store four integer/float values in RTC memory
def rtc_store_param(cycle, pos, neg):  # these values define next cycle length factor
    data = rtc.memory()
    c,p,n,s,d = ustruct.unpack('3i2f', data);
    data = ustruct.pack('3i2f',cycle,pos,neg,s,d)
    rtc.memory(data)

def rtc_load_param():
    # Retrieve the packed data from RTC memory
    c=1; p=0; n=0; s=20.0; d=0.1
    data = rtc.memory()
    if not data:
        data = ustruct.pack('3i2f',c,p,n,s,d)
        rtc.memory(data)
    # Unpack the integers from the byte array
    c,p,n,s,d = ustruct.unpack('3i2f', data);
    return c, p, n

def rtc_store_delta(delta):       # stores last sent sensor and delta values
    data = rtc.memory()
    c,p,n,s,d = ustruct.unpack('3i2f', data);
    data = ustruct.pack('3i2f',c,p,n,s,delta)
    rtc.memory(data)

def rtc_load_delta():                  # loads stored sensor and delta values, or default init
    c=0; p=0; n=0; s=20.0; d=0.1
    data = rtc.memory()
    if not data:
        data = ustruct.pack('3i2f',c,p,n,s,d)
        rtc.memory(data)
    c,p,n,s,d = ustruct.unpack('3i2f', data);
    return d

def rtc_store_sensor(sensor):     # stores last sent sensor and delta values
    data = rtc.memory()
    c,p,n,s,d = ustruct.unpack('3i2f', data);
    data = ustruct.pack('3i2f',c,p,n,sensor,d)
    rtc.memory(data)

def rtc_load_sensor():                  # loads stored sensor and delta values, or default init
    c=0; p=0; n=0; s=20.0; d=0.1
    data = rtc.memory()
    if not data:
        data = ustruct.pack('3i2f',c,p,n,s,d)
        rtc.memory(data)
    c,p,n,s,d = ustruct.unpack('3i2f', data);
    return s
```

```
# nvs_tools.py
import machine, ustruct
import esp32

# Function to write data to internal flash memory using NVS (Non-Volatile Storage)
def write_nvs_ts(key, value):
    nvs_key = esp32.NVS("thingspeak")  # Open the NVS namespace "thingspeak"
    nvs_key.set_blob(key, value)  # Store a byte array (blob) with a key
    nvs_key.commit()  # Commit the changes

# Function to read data from internal flash memory using NVS
def read_nvs_ts(key):
    nvs_key = esp32.NVS("thingspeak")  # Open the NVS namespace "thingspeak"
    try:
        buff = bytearray(32)
        value = nvs_key.get_blob(key,buff)  # Retrieve the byte array (blob) using the key
        return value,buff
    except OSError:
        print(f"Key '{key}' not found in EEPROM.")
        return None
```

```
# Function to write data to internal flash memory using NVS (Non-Volatile Storage)
def write_nvs_power(key, value):
    nvs_key = esp32.NVS("power")  # Open the NVS namespace "thingspeak"
    nvs_key.set_blob(key, value)  # Store a byte array (blob) with a key
    nvs_key.commit()  # Commit the changes

# Function to read data from internal flash memory using NVS
def read_nvs_power(key):
    nvs_key = esp32.NVS("power")  # Open the NVS namespace "thingspeak"
    try:
        buff = bytearray(32)
        value = nvs_key.get_blob(key,buff)  # Retrieve the byte array (blob) using the key
        return value,buff
    except OSError:
        print(f"Key '{key}' not found in EEPROM.")
        return None
```

To do:

Modify the code fragment that decides about the sending or not the data packet. In this modification build separate tests for delta and thresholds (for example):

```
   if abs(ssens-temp)>sdelta :  # testing delta and thresholds
        print("data to SEND")
        rtc_store_sensor(temp)
        led.on()
        if npos :
            if ncycle > 2:
                ncycle= int(ncycle/2)
            else:
                if sdelta< dmax:
                    sdelta = sdelta*2         # new delta
                    rtc_store_delta(sdelta)

        npos=npos+1; nneg=0  # positive and negative counters
        rtc_store_param(ncycle,npos,nneg)
        print(ts_wkey,lumi,temp,humi)
        connect_send_espnow(ts_chan,ts_wkey,lumi,temp,humi)
        print("data packet sent");led.off()

    elif temp>thigh or temp<tlow :
        ncycle=1; npos=0; nneg=0; rtc_store_param(ncycle,npos,nneg)
        sdelta = dmin; rtc_store(sdelta)
        print(ts_wkey,lumi,temp,humi)
        connect_send_espnow(ts_chan,ts_wkey,lumi,temp,humi)
        print("urgent data packet sent");led.off()

    else:
        print("data packet NOT sent")
        if nneg :
            if ncycle < cmax:
                ncycle = int(ncycle*2)           # maximum factor 64
            else :
                if sdelta> dmin:
                    sdelta = sdelta/2
                    rtc_store_delta(sdelta)

        npos=0; nneg=nneg+1
        rtc_store_param(ncycle,npos,nneg)
..
```
