Week 8 Lab 5: Weather Station

Important: I strongly recommend Windows users to install and use WSL for this lab (https://docs.microsoft.com/en-us/windows/wsl/install-win10). Especially for the second part of this lab with the use of MQTT and Mosquitto. The windows version of this MQTT broker has multiple peculiarities and we do not suggest using it in this lab.

Introduction

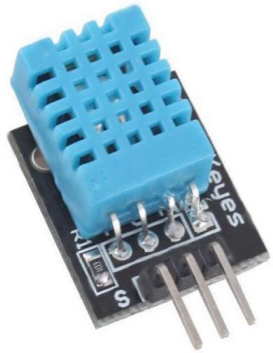
In this lab, we will build a weather station consisting of an IoT device and a server running on our laptop. The system will collect weather data, send it to a server and then make decisions based on the temperature and humidity. It should then "open or close" a window (using the servo) to simulate climate control. We have five major steps to build our first IoT system:

i. Recap: Humidity and Temperature Sensor.
ii. MQTT: We look at how to set up MQTT connections.
iii. MQTT on ESP32
iv. Power Management.
v. Bringing all these components together

This is a long lab, so please start early so that you can complete it on time. Every part of this lab is important in helping you with your project, so please do not skip anything.

Recap: Humidity and Temperature Sensor

Ok, cast your mind back to lab 1. We are going to get DHT11 sensor back up for today's lab.



There are three pins on the DHT11, **DATA**, **VCC**, and **GND**. Connect **GND** and **VCC** on the DHT11, to **GND** and **3.3V** on the ESP32. Connect the **DATA** pin to **16** on your ESP32. If you need the library, go to **Sketch > include Library > Manage Library.**

Search "DHT Sensor library" by AdaFruit. You will be asked to install the "AdaFruit Unified Sensor", click "Install all". Grab the example code and test it out.

Open `File > Example > DHT Sensor Library > DHT_Unified_Sensor`

You will need to make some small changes:

1. DHTPIN should be defined as 16
2. DHTTYPE should be defined as DHT11

Try running the program on your ESP32. In the serial monitor, you will be able to see the temperature and humidity as shown below:

```
Temperature: 27.30°C
Humidity: 52.90%
```

Try blowing on the sensor to increase the humidity or covering it with your hands to change the temperature.

Message Queuing Telemetry Transport (MQTT)

We need to send data between our ESP32 and laptop, or more generally between our IoT device and the cloud. For the first part we will just build a simple MQTT publisher and subscriber.

1. Setting Up the MQTT Broker and Client

   We will use Mosquitto for our MQTT Broker running on your laptop. Open a terminal, and Enter the following commands:

   Install Mosquitto and its clients.

   (Ubuntu, including Windows 10 WSL with Ubuntu)

   ```
   sudo apt-get install mosquitto mosquitto-clients
   pip install paho-mqtt
   ```

   (MacOS)

   ```
   brew install mosquitto
   pip install paho-mqtt
   ```

   MacOS automatically installs mosquitto-clients so there's no need to do so manually Paho MQTT is the Python library for communicating with the MQTT Broker.

2. MQTT Topics and Messages

   MQTT messages are organized into topics. A topic can be of the form "X/Y", where "X" is some category, and "Y" is a particular topic in the category.

   Mosquitto comes with two very useful programs:

   mosquitto_sub: Subscribes and listens for messages for a particular topic.

   mosquitto_pub: Publishes messages to a particular topic.

   To see how these work:

   (a) (Ubuntu) In the first terminal, type the following to start Mosquitto:

   ```
   mosquitto
   ```

   (MacOS) in the first terminal, type the following to start Mosquitto:

   ```
   brew services start mosquitto
   ```

   Note: If Mosquitto has already been started you will get an error message.

   (b) Open up two more Ubuntu or MacOS terminals.
   (c) In one terminal, start up a subscriber by typing:

```
mosquitto_sub –h localhost –t test/abc
```

This subscriber is only interested in the topic test/abc. We will call this the "subscriber terminal".

(d) In the second terminal publish a message to topic test/abc. We will call this terminal the "publisher terminal".

```
mosquitto_pub –h localhost –t test/abc –m "Hello world"
```

You will see "Hello world" appear in the subscriber terminal.

(e) Again in the publisher terminal we publish a new message to topic test/def:

```
 mosquitto_pub –h localhost –t test/def –m "Hello!"
```

This time you will notice that the message DOES NOT appear in the subscriber terminal.

(f) Go back to the subscriber window in part (b) above. Press CTRL-C to exit the mosquitto_sub program. Restart it with:

```
mosquitto_sub –t test/# -h localhost
```

(g) Now in the publisher terminal in part (e), type:

```
mosquitto_pub –t test/abc –h localhost –m "TEST!"
```

You will see that it appears in the subscriber window. Likewise:

```
mosquitto_pub –t test/xyz –h localhost –m "TEST TEST!"
```

You will see that this again appears in the subscriber terminal even though test/xyz is a new topic.

This is because the subscriber has subscribed to test/#. The "#" means that it is interested in every topic under test/, which is why it receives all the messages.

This is all very interesting but now we need to build a python script so we can integrate MQTT into our IoT server.

3. Writing MQTT Clients

We will now use our first python MQTT client (mqtt.py).

a. Our Paho MQTT Client Code

```
import paho.mqtt.client as mqtt

from time import sleep

def on_connect(client, userdata, flags, rc):

    print("Connected with result code: " + str(rc))
    client.subscribe("hello/#")


def on_message(client, userdata, message):

    print("Received message " + str(message.payload))


client = mqtt.Client()

client.on_connect = on_connect

client.on_message = on_message

client.connect("localhost", 1883, 60)

client.loop_forever()
```

The first callback is called when Paho MQTT connects to the MQTT broker:

```
def on_connect(client, userdata, flags, rc):

    print("Connected with result code: " + str(rc))
    client.subscribe("hello/#")
```

The parameters for on_connect are as follows:

Parameter	**Description**
client	Instance of the Paho MQTT client library.
userdata	Private user data that can be set in the MQTT Client() constructor. Not used here.
flags	A dictionary containing flags returned by the broker. Not used here.
rc	Result code: 0 – OK 1 – Connection refused, incorrect protocol version. 2 – Connection refused, invalid client identifier.

	3 – Connection refused, service not available. 4 – Connection refused, bad user name and password. 5 – Connection refused, not authorized.

In our on_connect code we simply subscribe to the “hello/#” topic, by calling client.subscribe.

Our second callback is shown below. It is called whenever a new message comes in.

```
def on_message(client, userdata, message):
    print("Received message " + str(message.payload.decode('utf-8')))
```

The most important parameter is msg, which contains the message topic and contents (payload).

Now we are ready to set up the Paho MQTT library. We call the Client() constructor, then point it to the on_connect and on_message callbacks:

```
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
```

We then connect to the broker, which, since we are running this on the laptop, will be at localhost.

```
client.connect("localhost", 1883, 60)
client.loop_forever()
```

The broker runs by default at port 1883 (the second parameter). The third parameter (60) tells the client the interval between “keepalive” messages, which keep the connection to the broker alive.

Finally, we call “loop_forever” so that our program doesn’t exit, and we can wait for messages.

b. Running Our Client Code

We will now run our code to see what it does.

i. Ensure that mosquitto is running. Start mosquitto if it is not (see Step 2).
ii. Have at least two other terminals open.
iii. Run our program in one terminal:

```
python mqtt.py
```

iv. Now in the other window, send a message over:

mosquitto_pub –h localhost –t hello/world –m "Hello!!"

v. If all goes well you should see:

```
Connecting
Connected with result code 0
hello/world b'Hello!'
```

Task 1. Notice that there is a stray "b" attached to our Hello! message. Using Google or otherwise, explain what this "b" means and why it is present in our output. *(2 marks)*

Task 2. Using Google or otherwise, modify your program so that this "b" no longer appears. Paste your code into the lab report. *(3 marks)*

c. Publishing Messages

Paho-MQTT allows us to publish messages using the "publish" call:

```
client.publish(topic, payload)
```

To see how to use this, use pub.py with the following code:

```
import paho.mqtt.client as mqtt
from time import sleep
def on_connect(client, userdata, flags, rc):
    print("Connected with result code: " + str(rc))
    print("Waiting for 2 seconds")
    sleep(2)
    print("Sending message.")
    client.publish("hello/world", "This is a test")
```

```
client = mqtt.Client()
client.on_connect = on_connect
client.connect("localhost", 1883, 60)
client.loop_forever()
```

This code should be fairly self-explanatory.

Now:

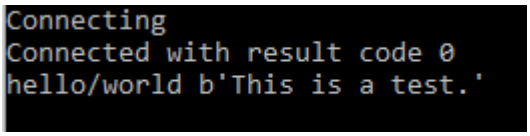
i. Ensure that Mosquitto is running.
ii. Have two terminals open. In one, start the mqtt.py program that we wrote earlier:

```
python mqtt.py
```

iii. In the other, start this program:

```
python pub.py
```

iv. If all goes well, after two seconds pub.py will send a message over to mqtt.py and on mqtt.py you will see:



Note: Here the 'b' still appears. It should not appear in your version.

We will now integrate this with a client running on the ESP32.

Note, due to changes in default security options in MQTT, only localhost is allowed. So to connect from the outside world you will need to add two lines to the mosquitto.conf file on the laptop server:

allow_anonymous true

listener 1883

MQTT on our ESP32

MQTT clients can be quite lightweight and this makes them suitable to run on low power microcontrollers. There are many libraries you can play around with but we will use the library **ESP32MQTTClient**, a wrapper around the ESP32 MQTT library. Install it now in the Ardunio IDE.

Now, open the example, ESP32MQTTClient > **HelloToMyself** example.

You will need to put in your WiFi credentials:

```
const char *ssid = "ssid";
const char *pass = "passwd";
```

Then specify the MQTT server/broker that is running on your laptop. Replace foo.bar with your ip address.

```
char *server = "mqtt://foo.bar:1883";
```

Finally, you will need to specify the topic that you will publish to, let's use "hello/esp":

```
char *publishTopic = "hello/esp";
```

Now run your program on your ESP32, and mqtt.py on your laptop. Both your laptop and ESP32 need to be connected to the same network. You should now see in your mqtt python script:

```
Received message Hello: 1

Received message Hello: 2

Received message Hello: 3

Received message Hello: 4
```

Great, now we can send data from our ESP32, but what about in the other direction? Note that there is also a **subscribeTopic** function in the ESP32 program. Can you now get your pub.py script to send data to your ESP32?

Task 3. Modify your pub.py python script to send some data to your ESP32. Detail your changes in your lab report. *(2 marks)*

Let's now send something more useful that hello world messages. Modify your ESP32 to send the humidity and temperature readings from your DHT11 to your MQTT script running on your laptop. You can use the topics "weather/temp" and "weather/humidity". Modify your mqtt.py script to print these values.

Task 4. Modify your ESP32 sketch and to send back temperature and humidity readings. You can use the topics "weather/temp" and "weather/humidity". Paste your code into the lab report. *(3 marks)*

Bonus task: Can you figure out how to get the MQTT server to hold onto the messages for a topic until the ESP32 device reconnects?

Power Management

Our IoT devices need to conserve power when possible and the microcontroller (the **ESP32)** has five power modes:

- **Active mode:** The chip radio is powered up. The chip can receive, transmit, or listen.
- **Modem-sleep mode:** The CPU is operational and the clock is configurable. The Wi-Fi/Bluetooth are disabled.
- **Light-sleep mode:** The CPU is paused. The RTC memory and RTC peripherals, as well as the ULP coprocessor are running. Any wake-up events (MAC, SDIO host, RTC timer, or external interrupts) will wake up the chip.
- **Deep-sleep mode:** Only the RTC memory and RTC peripherals are powered up. Wi-Fi and Bluetooth connection data are stored in the RTC memory. The ULP coprocessor is functional.
- **Hibernation mode:** The internal 8 MHz oscillator and ULP coprocessor are disabled. The RTC recovery memory is powered down. Only one RTC timer on the slow clock and certain RTC

GPIOs are active. The RTC timer or the RTC GPIOs can wake up the chip from the Hibernation mode.

We will now check out the power modes and test some ourselves.

First, let's look at the documentation of the ESP32.

Task 5. What is the average power consumption of the ESP32 when in Deep Sleep mode, modem-sleep, and when it is in active mode (you can assume it is transmitting/receiving wifi)? Assuming an ideal battery of 200mAh, how long would the device last in each state? **Hint:** look at the tech sheet. *(2 marks)*

Now, let's try out a Deep Sleep on the ESP32. Create a new sketch from **Examples > ESP32> DeepSleep > TimerWakeUp**

This example uses a timer to wake up your ESP32 device after putting it to deep sleep (all other part of the chips are powered off except the timer). There are multiple other options for how to wake up your device for example using an external line or using the touch sensors. Have a play around.

Note that after the device wakes up from Deep Sleep, it is as if it has rebooted. It will run the setup routine again. Now, let's try writing a program that uses deep sleep ourselves.

Task 6. Write a program (sketch) that connects to wifi, sends an MQTT message, prints something to the serial, and then puts the ESP32 device into deep-sleep for 20 seconds. Explain and include this program in your report. *(3 marks)*

Bringing it all together

We now have most of the components we need already for our Weather Station. We have our embedded system which can collect and send data from our DHT sensors data using MQTT. We have our server scripts which can display the data on our laptop. Two main things are missing: a classifier and actuation.

For our classifier we are just going to use a very simple model. Namely a rules-based model:

If the temperature if above 30 degrees, it is too hot, open the window, and if it is below 25 degrees (too cold) close the window. If it is between 30 and 25, partially open the window.

Task 8. Write a function in our mqtt.py python script to classify the temperature and send back the classification result. *(1 mark)*

For actuation, we will use our servo to act as a motor attached to a "window opener". If the weather is too hot, we can open the window, and if it gets too cold, we can close the window. You should use the MQTT topic to trigger the window opening.

Task 8. Write a program (sketch) that reads in the DHT sensors, connects to wifi, publishes the DNT sensor readings, waits for a classifier response, actuates appropriately, and then puts the ESP32 device back into deep-sleep for 20 seconds. Explain and include this program in your report. *(4 marks)*

Wow, so there you have it. Our first IoT system. In your project you will have to have a much more complex model but this is good enough for our usecase.

Bonus task: Get your device working off a battery! Note that the batteries I gave you will turn off after about 30 seconds of inactivity so you will have to wake your device up and use some power to keep the battery on.

Bonus bonus task: Try collecting data from multiple ESP32s from all your project group mates using the same MQTT server. Make some devices be window openers and some weather stations.

Appendix:

Connecting to the NUSWifi
It possible to connect your ESP32 to the NUS_STU network, and we have given you an example program to do so (NUS_STU_ESP.ino).

This will involve putting your username and password in plain text in the source code of your program. This is a big security risk so do be careful. Don't send these files to anyone or push them to github.

3D Printed Enclosure
We have also included a 3D printed enclosure for you to try out, modify as the base for your project, and print at Makers. After you finish the above tasks take a look! You can find them in canvas, under Lab 5.