HAN smart EV charging pole An IoT network with MQTT protocol

Pham Minh Nhat
Student number: 614723

August 9, 2020

Embedded System Engineering
HAN University of Applied Sciences

Contents

Chapter 1

Introduction

1.1 Background

Since 2013, Alfen, TU Delf, HAN, Aliander, WES and Bredenoord have cooperated in project CSGrip. One target of this project is to balance the generated electricity and the consumed electricity at any moment. The model which is used to tested the control algorithm contains 3 parts: Solar panels as energy generators, an energy container to store generated electricity and two controllable EV charging poles which consume the energy[1].

In 2019, a student group has done the fundamental steps to implement the communication within Modbus network and MQTT network[4]. Data from energy meters in a Modbus network can be read and sent to other nodes in a MQTT network. There was also a .txt file to save measured results through time.

1.2 About this document

This document includes 6 chapters. Chapter 2 specifies the targets of the project. In chapter 3: Design, the conceptual and technical aspects of the system are described. Chapter 4 explains the realization phase of the project and chapter 5 is a record of tests being done. Finally, chapter 6 discusses the result of the project.

Chapter 2

Requirements

This project inherits from the student project in 2019. Its targets are divided into 3 groups: Modbus network, MQTT network and database.

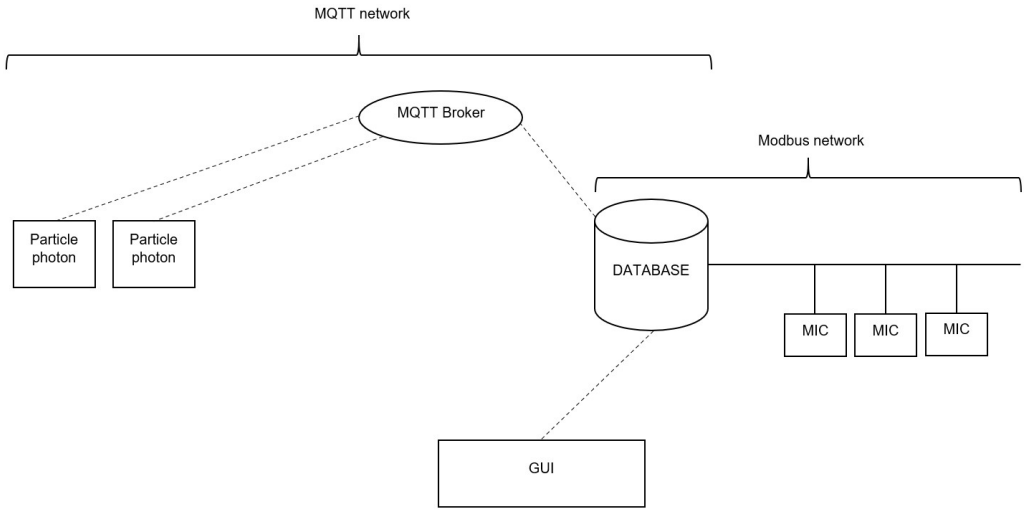

Figure 2.1: Conceptual framework

2.1 Modbus network

In a real system, there are many noise sources that can influence the voltage in the cable. Messages can be corrupted as the sequence of these noise and the program can be suspended by errors (missing bits, value out of range, etc). However, This system is expected to run continuously. Hence, filters to get rid of corrupted data is essential.

The previous student group work with MIC2 Mk-II energy meter. The system, in fact, uses MIC energy meters, which have different registers and data format. Therefore, a new package need to be written based on the old one.

Calculations using data from registers need to done to obtain the value of current, voltage, power and frequency.

2.2 MQTT network

MQTT is a lightweight messaging protocol. In this project, wireless communication between microcontroller is done using MQTT.

Following data need to be sent with MQTT:

From Raspberry Pi to Particle Photon: measured results, user list.

From Particle Photon to other controllers: charging time, which chargers are in used, who is using the charger, current and voltage of the charger.

Communication must be done frequently to keep every controller up to date with the database.

2.3 Database

SQLite database stores the user list and all measurement data.

2.4 User interface

The graphical user interface (GUI) should display the value of voltage, current, power and frequency of each energy meter over time.

The GUI is a website that is accessible for devices having internet connection.

Chapter 3

Design

3.1 Conceptual design

In previous phases, Raspberry Pi was chosen to be the master of Modbus network[4] and Particle Photon kits have been already used in charging poles.

Data can be stored within either an "online" database on cloud or an "offline" database on a machine. A separated sever is too much for the small scale of this project, whose database only includes tables of users and measured results. To minimize the cost and the complexity of the system, the SQL database is installed on the Raspberry Pi. Raspberry Pi 3 Model B provides multiple USB ports. Memory sticks can be connected to these ports to extend the storage space of the Pi.

Data is published to different topics on the MQTT broker. In this designed, topics are treated as "name tags". Data are processed in different ways based on their topic that they belong to.

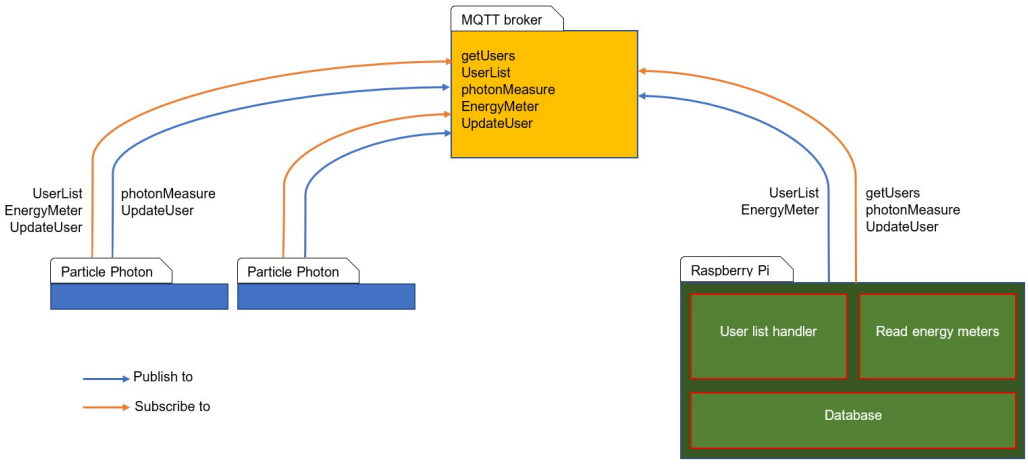

Figure 3.1: Conceptual design

3.2 Detailed design

Raspberry Pi has an operating system and a powerful processor (in the embedded world). These factors allow the Pi to run multiple tasks in parallel (multi-threading). Take advantage of this feature, two python scripts were written to execute two (mostly) irrelevant series of actions. The behavior of two scripts are described in the following diagram.

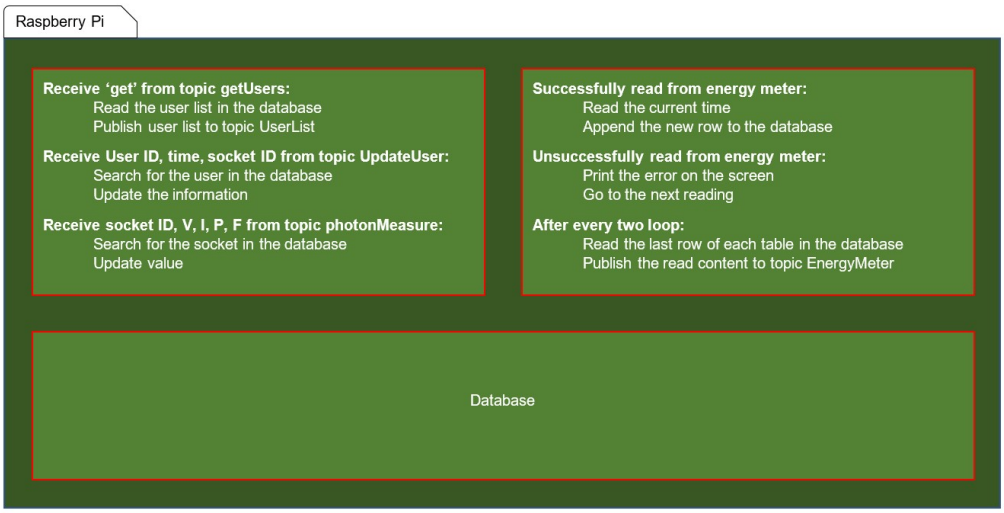

Figure 3.2: Two processes running on the Raspberry Pi

The database is also divided into two separated databases. One stores the data from energy meters and the other stores data relating to the user.

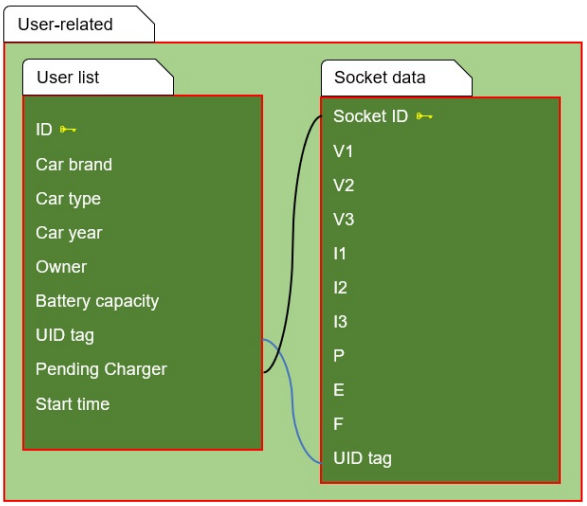

Figure 3.3: This databse is used by the User list handler

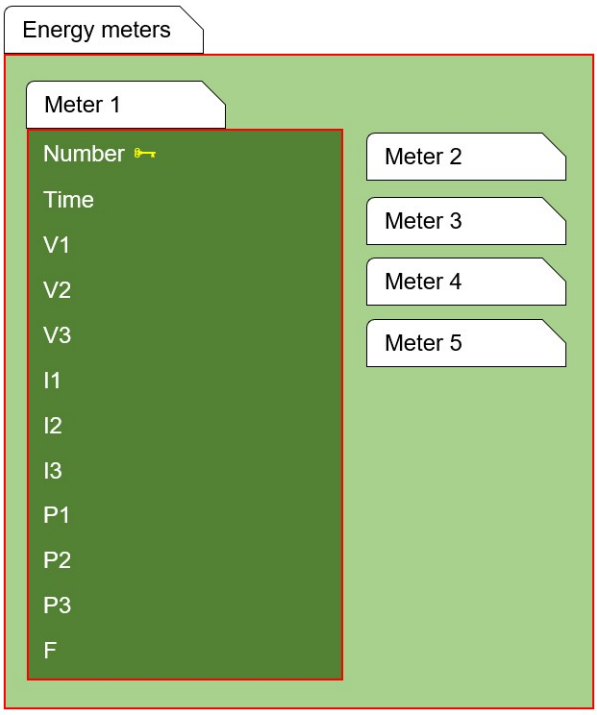

Figure 3.4: This database stores measured results from Modbus network

To create a database on Linux machine, MySQL and SQLite are two popular solutions. MySQL operates using server-client model. The server is the machine that the database is in, all devices that want to use the database (clients) must communicate with the server through a protocol. However, in this design, the database and programs which use it are in the same machine (Raspberry Pi). Hence, SQLite, which does not need the server-model structure, is used.

Like the Raspberry Pi, Particle Photon also needs to handle message from different MQTT topic. In fact, Particle Photon was programmed to use MySQL database with http calls. Because the database has been replaced in this phase, those calls must also be replaced by the MQTT functions while the rest of the program remains the same. In short, when the Raspberry Pi or other Particle Photon publishes a message, the board will process it and save the data within an array or a struct. More details will be discussed in the next chapter.

Chapter 4

Realization

4.1 Reading energy meters

The core of the package is Modbus-reading functions discussed in another document[4]. This report only explains how those functions are used to build a package for reading MIC energy meters.

According to the Modbus manual of MIC[2], following formulas is used to calculate the voltage, current, power and frequency:

$$V = R_x \times \frac{PT_1}{PT_2 \times 10} \tag{4.1}$$

$$I = R_x \times \frac{CT_1}{5000} \tag{4.2}$$

$$P = R_x \times \frac{PT_1}{PT_2} \times \frac{CT_1}{5} \tag{4.3}$$

$$F = \frac{R_x}{100} \tag{4.4}$$

PT_1, PT_2 and CT_1 are called voltage transformer data and current transformer data. All of them can be read from meter's registers. R_x is the value read from voltage, current, power and frequency registers respectively. The addresses of these registers are on pages 17 to 19 of the Modbus manual[2].

As explained in chapter 2, filters are added to determine the false data when reading from registers. Take the simplified version of function readPT1() as an example:

```
#Error codes
Data_error = -3
CRC_error  = -2
Trans_error = -1
No_error   = 0

def readPT1(self):
   #Send the request for reading a register...

   #Receive data
   GPIO.output(self.__Control, GPIO.LOW)
   cnt = 0
   data_left = ser.inWaiting()
   while (data_left == 0):
      #wait for data
      cnt=cnt+1
      if (cnt < 50000): #wait for maximum 5 seconds
         sleep(0.0001)
         data_left = ser.inWaiting()
      else:
         print("Transmitting error: Time out")
         return Trans_error

   #Read data from serial bus and write them to received_data...

   #Check if the data is correct
   if (ord(received_data[0]) != self.__Address):
      print("Transmitting error: Data corrupted")
      return Data_error
   if (len(received_data) != 9):
      print("Transmitting error: Data corrupted")
      return Data_error

   #Check the CRC code
   crc_cal = hex(crc16(received_data[:7]))
   crc_Rx = hex(struct.unpack('H',received_data[7:])[0])

   if crc_cal == crc_Rx:
      self.__PT1 = float(struct.unpack('I',
          received_data[6:2:-1])[0])
      return No_error
   else:
```

```
42        print("Transmitting error: Incorrect CRC")
43        return CRC_error
```

After sending the request, line 14 suspends the program. If there is no reply after 5 seconds of waiting, the function will return a time out error. In some cases, data can be received with a missing part. Since the length of the message is an input of CRC calculation, an incomplete message will interrupt the program. This behavior is unwanted in this system, which is supposed to run continuously for a long time. Hence, the second and the third filter in line 27 and 30 were brought out to return the error code if the first byte is wrong or the message is incomplete and keep the loop running. The final check is the CRC code. The CRC code is calculated with the received data. If the message is received correctly, the result should match the two last bytes of the message. Reading different parameters uses the same technique with different register addresses.

PT_1, PT_2 and CT_1 only need to be read once. Ideally, those are read at the start of the program, outside the infinite loop. However, there is always the probability that there is no successful read after many attempt to read the value. On the other hand, a conditional loop without a limit will become an unwanted infinite loop if one energy meters is not working and never answers the request. Therefore, the most robust option is reading PT_1, PT_2 and CT_1 in every loop, before reading other registers. Though it takes more time to read and calculate all necessary values, this solution can keep the program running and minimize the consequence of a failed read. In a special circumstances, if registers are read incorrectly while CRC codes are still match, there will be only one row of the database effected by this error.

```
1  #Read PT1, PT2, CT1:
2  readingPT1 = meter1.readPT1()
3  readingPT2 = meter1.readPT2()
4  readingCT1 = meter1.readCT1()
5  #If there is no error, then continue
6  if((readingPT1+readingPT2+readingCT1)==0):
7     #Read PHASE VOLTAGE
8     #Read PHASE CURRENT
9     #Read PHASE POWER
10    #Read FREQUENCY
```

The program only reads voltage, current, power and frequency when reading PT_1, PT_2 and CT_1 gives no error.

4.2 Communication between micro-controllers

In a MQTT network, subscribers of a topic will be "waken up" if the publisher pushes a text message to that topic. The message then can be read by subscribers. However, data that need to be transferred are not always just text. They can be numbers, arrays or combinations of words and numbers. The key of the communication is a data format that all nodes in the network can process and get the necessary information. Base on the fact that messages in this project include only alphabet characters and numbers, a special character is used to mark the start and the end of each data filed. For example, The particle photon publishes measured value from the energy meter inside the charging pole in this format:
V1%V2%V3%I1%I2%I3%P%E%F%Time stamp%Socket ID%User ID
This message is read by the following code snippet:

```
def photonMeasure_callback(client, userdata, message):
   data = message.payload
   index = []
   for i in range(len(data)):
      if (data[i] == '%'):
      index.append(i)
   V1 = float(data[:index[0]])
   V2 = float(data[index[0]+1:index[1]])
   V3 = float(data[index[1]+1:index[2]])
   I1 = float(data[index[2]+1:index[3]])
   I2 = float(data[index[3]+1:index[4]])
   I3 = float(data[index[4]+1:index[5]])
   P = float(data[index[5]+1:index[6]])
   E = float(data[index[6]+1:index[7]])
   F = float(data[index[7]+1:index[8]])
   Time = int(data[index[8]+1:index[9]])
   SocketID = int(data[index[9]+1:index[10]])
   UserID = data[index[10]+1:index[11]]
```

The for loop scans through the message and saves the indices of percent signs ('%') in the message. Thanks to these indices, the message is sliced into small segments that contains information about measured results, time, socket and user ID. At the time this document is being written, all communications between microcontroller in the network are done using the same technique, the same data format.

```
void callback(char* topic, byte* payload, unsigned int length) {
   if (strcmp(topic, "HANevse/UserList")==0) {
```

```
3       getUsers_callback(payload, length);
4     }
5     else if (strcmp(topic, "HANevse/UpdateUser")==0) {
6       getUpdate_callback(payload, length);
7     }
8     if (strcmp(topic, "HANevse/EnergyMeter")==0) {
9       getMeasure_callback(payload, length);
10    }
11    time_t time = Time.now();
12    DEBUGPORT.print("MQTT>\tCallback function is called at: ");
13    DEBUGPORT.println(Time.format(time, TIME_FORMAT_DEFAULT));
14  }
```

The received messages are sorted by their topic, as described in the snippet above. In total, there are 5 MQTT topics used in this system (Figure 3.1). Topic "getUsers" has only one subscriber, the Raspberry Pi. When a Particle Photon starts working, it publishes a message to this topic to request for the database from the Pi. This action allows each Particle Photon has its own database stored locally. The Photon uses this database to check the user ID and store charging information. By doing this, the system can still work if the internet is not stable and the Photon is disconnected from the network.

Every time a Particle Photon updates its data, it publishes the same information to topic "UpdateUser" to synchronize all database. If the connection is lost, the synchronization may be done with the next message. The measured data from charging poles, which have nothing to do with authorizing a user to charge the car is sent only to the Raspberry Pi via topic "photonMeasure".

On the opposite side, the Pi frequently publishes the last row of its measured data to topic named "EnergyMeter". The Particle Photon reads information of the grid from this topic to sets the current of its charging pole.

4.3 Database

If all read functions of a meter return no error, a new row will be added to the associated table in the energy meters database. For example:

```
cur.execute("INSERT INTO meter1(Time, V1, V2, V3, I1, I2, I3, P1,
    P2, P3, F) VALUES(?,?,?,?,?,?,?,?,?,?,?)",
         (int(round(time.time())),
         meter1._MIC1__V1, meter1._MIC1__V2, meter1._MIC1__V3,
         meter1._MIC1__I1, meter1._MIC1__I2, meter1._MIC1__I3,
         meter1._MIC1__P1, meter1._MIC1__P2, meter1._MIC1__P3,
         meter1._MIC1__F))
con.commit()
```

The time written to the database is the system time at the moment the database is updated. In most case, the system time is an integer presenting the number of seconds has elapsed since 00:00:00 UTC on 1/1/1970 (leap seconds are ignored). However, time.time() method returns a float number, which describes the time at millisecond-level. As mentioned, because not all system handle the time this way, the fractional part is removed from the read using round() and int(). This modification help prevent conflict in an IoT system with multiple types of devices.

The user database is treated in a different way. The tables are updated when a callback function is executed. If the update_callback() is called, the UserID, PendingCharger and StartTime are read from the MQTT message. The program will search for the row which have the same ID and update the charger and time data:

```
cur.execute("UPDATE list SET PendingCharger=? WHERE Id=?",
    (PendingCharger, UserId))
cur.execute("UPDATE list SET StartTime=? WHERE Id=?", (StartTime,
    UserId))
```

If the photonMeasure_callback() is called, it will read all value from the MQTT message and add a new row to the socket database:

```
cur.execute("INSERT INTO photonMeasure(UserID, SocketID, V1, V2,
    V3, I1, I2, I3, P, E, F, Time) VALUES(?,?,?,?,?,?,?,?,?,?,?,?)",
   (UserID, SocketID, V1, V2, V3, I1, I2, I3, P, E, F, Time))
```

4.4 User interface with Node-RED

Node-RED is a flow-based development tool for visual programming, which means the program is done mostly by dragging and connecting blocks.

It also provides a dash board that can display charts, gauges, tables, buttons etc. Node-RED editor and dashboard can be reached through an IP address, makes it suitable for creating a user interface for this project.

Node-RED sever runs on the Raspberry Pi and its editor and dashboard can access via the IP address of the Pi. A static public IP address is given to the Raspberry Pi therefore the dashboard and the online editor can be access from internet via `http://80.113.19.27:1880/ui` and `http://80.113.19.27:1880` respectively.

To allow only admin to modify and deploy the Node-RED flow, username and password are required to log into the editor. The default combination is:

Username: admin
Password: password

Steps to change username and password are describe in the appendix.

The data flow in this project is as follows:

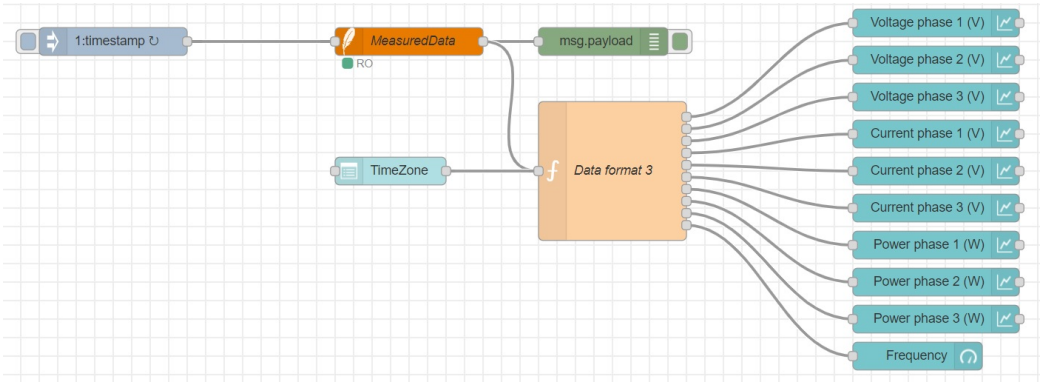

Figure 4.1: Node-RED data flow

Node "MeasuredData" sends a SQLite command to read 20 latest rows of the database. This command is executed once per 30 seconds when receives the trigger signal from node "timestamp". By doing this, data displayed on the GUI are updated. The output of the "MeasuredData" is a JSON message. This message is processed by the function node named "Data format <n>" (n can be 1, 2, 3, 4 or 5). The output of the function is 10 JSON messages which are read by display nodes. The "TimeZone" is a drop box which allows user to select which time zone should be used by the charts. The available options are GMT 0, GMT +1 and GMT +2.

Figure 4.2: The Node-RED dashboard

Because the system includes 5 energy meters, there are 5 similar data flows (including flow in Figure 4.1) for visualizing the read from 5 meters. The results of these flows are displayed in 5 tabs of the GUI. The dashboard also has the sixth tab which displays the latest measured data of meter 3, 4 and 5.

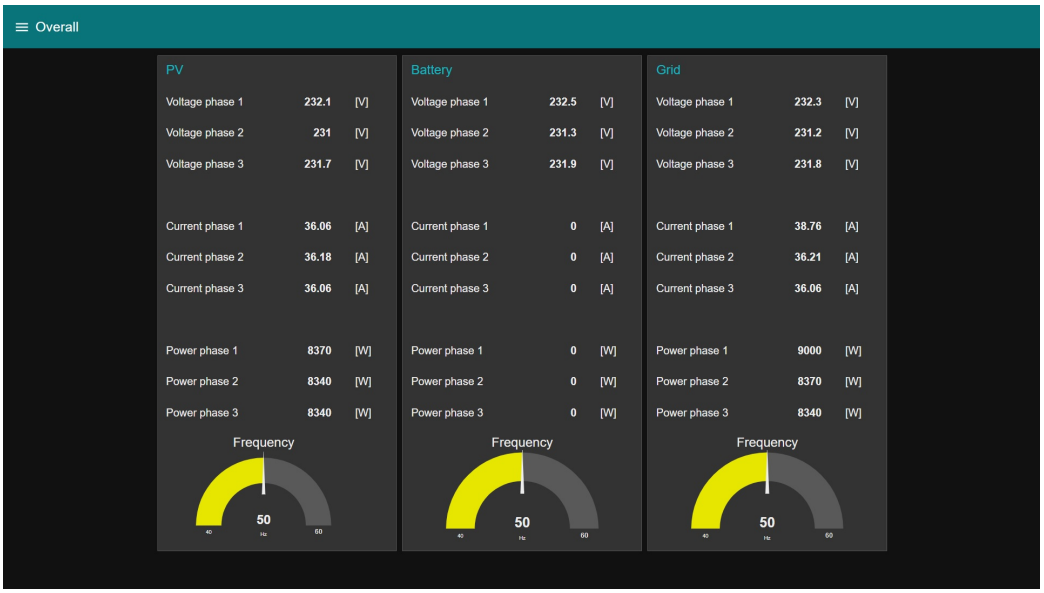

Figure 4.3: Latest data are shown on the dashboard

Chapter 5

Test results

5.1 Reading from Modbus network

The script can run for weeks without interruption. However, sometimes the read of power phase 2 of the grid gives negative value, the reason of this error has not been found. The absolute value looks correct (approximates the multiplication of voltage and current of the same line) with an extra minus. This behavior lasts for a while, which is not possible if this is an error caused by noise.

The script needed to be manually restarted when the electricity was cut off. To deal with similar events, file rc.local was edited to run the script whenever the Pi is booted. The detail is described in the appendix.

5.2 Database

The problem is found when the Particle Photon tried to receive long message like the database. The MQTT.h library for Particle Photon can only receive message with maximum 512 bytes[3]. It is not certain whether other MQTT libraries also have similar limit. To adapt to this situation, the UID check function was temporarily removed from the Particle Photon. Everyone can use the charging poles. However, if a card is present, the controller still sends its ID to the Raspberry Pi.

Chapter 6

Conclusion and Recommendation

Thanks to the communication between Raspberry Pi and Particle Photons, charging poles now can operate in sustainable charging mode. Two scripts can work for a long time without any serious errors which can stop the programs. Data are stored in SQL databases and are displayed on a real-time GUI. Overall, all requirements are met. However, there are always room for improvement:

The current message format is simple, but is not easy to process. There is no way to read a specific field of the message without scanning the whole string. Therefore the format is one thing that need to be improved. JSON format is recommended because it is popular, has supported packages for many programming languages. Data in JSON message always go in pair (name-value) hence it is easy to find the wanted part of data in a message.

The Raspberry Pi has a complex operating system to control its peripherals, which makes it more sensitive with the unexpected loss of energy. Power outage has a chance to corrupt the data stored in the SD card and the memory stick. To protect the data, an UPS circuit for the Raspberry Pi is a good option to detect the voltage drop and safely turn off the Pi.

Because the python scripts on Raspberry Pi communicate with the broker frequently, network problem can halt the scripts. The code needs to be improved to resume the task when the internet connection is recovered.

If possible, MQTT.h has to be replaced by another library which has no limit of the message length.

NodeRED (including the dashboard) is running on port 1880. To access the site, the port number is mandatory (for

example: `sampledomain.com:1880`). By default, address without port number is bound to port 40 or 443. If possible, the NodeRED server should be moved to those ports.

Two table photonMeasure and userList have one data field in common (UIDtag). With this column, a link between two table can be created to reduce the manual work when processing data.

Practice shows that the USB stick is broken after a few weeks. This can be the result of overwriting to the memory chips continuously during a long time. Because this seems to be a hardware problem, the only solutions is backing up the data in the memory stick regularly to avoid the loss of all data.

Appendix A

Securing Node-RED

Username and password can be set (or changed) by editing the setting file (settings.js). To enable user authentication on the online editor, the **adminAuth** property must be uncommented. The property can look like this:

```
adminAuth: {
   type: "credentials",
   users: [
      {
         username: "admin",
         password:
             "$2a$08$zZWtXTja0fB1pzD4sHCMyOCMYz2Z6dNbM6tl8sJogENOMcxWV9DN.",
         permissions: "*"
      },
      {
         username: "nhatpham",
         password:
             "$2b$08$wuAqPiKJlVN27eF5qJp.RuQYuy6ZYONW7a/UWYxDTtwKFCdB8F19y",
         permissions: "read"
      }
   ]
}
```

There can be more than one user and each account may has different permission. In the snippet above, admin has all permissions while nhatpham has read-only access to the editor. The password is hashed using bcrypt algorithm. To generate a password hash, following command is used:

```
node-red admin hash-pw
```

Appendix B

Run commands at boot

Any commands (including running a script) can be added to file rc.local to be executed at boot. This file must be edited with root using any text editor. For example:

```
1 sudo nano /etc/rc.local
```

Commands are added bellow the comment part. If the script run an infinite loop, an ampersand must be appended to the end of the command. Without this, the loop will run on the front ground and halt the booting process. The file must end with line *exit 0*.

Appendix C

Source code

The code for particle photon can be found here: `https://github.com/MN-Pham/HAN-EV-ParticleCode`. This repository has two branches: The master branch is the program having a database stored locally in the Particle Photon, the other is the program whose database-related functions were removed.

The energy meter reader can be found here: `https://github.com/MN-Pham/Modbus-HAN-ev/tree/master/Modbus`

The script that manages the user list can be found here: `https://github.com/MN-Pham/Modbus-HAN-ev/tree/master/userID`

References

[1] Alliander (2018). Han ev charger technical reference. https://github.com/HanlectoraatMR/HAN-Smart-Electric-Charging-Station/blob/master/Docs/20180412

[2] DEIF A/S (2018). Modbus communication manual. https://github.com/HanlectoraatMR/HAN-Smart-Electric-Charging-Station/tree/master/Datasheet.

[3] hirotakaster (2017). Mqtt for photon, spark core. https://github.com/hirotakaster/MQTT.

[4] Pham Minh Nhat, Cristian Batog, Kalin Dyankov (2019). Smart ev charging pole-project report.