FREEDM MQTT DOCUMENTATION

1. Introduction 3

2. Overview of MQTT Protocol 3

2.1 MQTT Example 4

3. MQTT protocol in FREEDM Energy Cell 5

4. Building FREEDM MQTT project 7

5. Running FREEDM MQTT 7

6. Implementation 8

6.1 Device Profile 9

6.2 Configuration 10

7. Data Flow Mechanisms 10

8. Example JSON file 13

# Introduction

MQTT stands for **Message Queueing Telemetry Transport**. It is a publish/subscribe, extremely simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks. The design principles are to minimize network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery. These principles also turn out to make the protocol ideal of the emerging **“machine-to-machine” (M2M)** or **“Internet of Things”** world of connected devices, and for mobile applications where bandwidth and battery power are at a premium.

MQTT developed by IBM and recently (November 2014) was formalized as an open standard managed by OASIS (Organization for the Advancement of Structured Information Standards). MQTT provides a simple pub-sub model designed for resource constrained devices, typically embedded systems. While it’s simple to implement, it still supports three qualities of service: fire-and- forget(unreliable), at least once (to ensure a message is sent a minimum of one time) and exactly once. It also provides a hierarchical topic based message passing.

# Overview of MQTT Protocol

1. MQTT is described as a **pub-sub** messaging protocol that uses TCP/IP under the hood. Any given MQTT network consists of one or more clients and a single broker.
2. All the clients on a network communicate with each other via a ***Broker***. As such, all the clients are decoupled from each other and are connected only to the broker.
3. An MQTT client could be any device, varying in computational power from a microcontroller to a server, that can run any implementation of the MQTT library on it. **The client** could be a *publisher*, *subscriber* or both.
4. ***Publishers*** tag their messages with topics and transmit them to the broker.
5. ***Subscribers*** let the broker know about the message topics they are interested in.
6. It’s the ***Broker*** that receives all the messages, filters them based on their respective topics and then forwards each message to appropriate subscribers. A powerful implementation of broker can handle subscription of thousands of clients. The broker is also responsible for handling authentication and security over the network.
7. [This link](http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html) is the complete specification of the protocol and [this link](http://www.hivemq.com/blog/mqtt-essentials-wrap-up) provides a comprehensive overview of the protocol from an implementation perspective. Moreover, one can also use TLS/SSL (Transport Layer Security/ Secure Sockets Layer) protocol for encryption purposes if required.
8. MQTT also provides username/password based authentication mechanism for the clients to connect over the network.

## MQTT Example

Following example illustrates a IOT scenario of usage of MQTT protocol.

1. Consider a simple thermostat system for home that uses an MQTT network as shown in *Figure 1: Example of MQTT network using hierarchical topics to organize messages* Clients A, B, C and D are connected to a central broker, and are agnostic about each other’s presence.
2. Client A is a temperature sensor that publishes the real time temperature value to the network using the topic *Home/Sensors/Temperature*.
3. Client B is subscribed to it because it manages the cooling system and uses the temperature sensor’s value for control purposes. Client B is also a publisher for the fan speed parameter which being acquired by a sensor, is published under the topic *Home/Sensors/FanSpeed*.
4. Client C is basically a data logger and is therefore interested in all the sensor readings available over the network. As such, it subscribes to the broker using a wild card topic using an ‘\*’. as *Home/Sensors/∗*. This basically tells the broker that Client C is interested in all the data published under the topic of ’Sensors’.
5. Client D being a remote display is interested in only displaying the temperature data and therefore subscribes specifically to the *topic Home/Sensors/Temperature*.

This is makes the topic based pub-sub communication not only scalable by decoupling clients but is also organized and hierarchical.

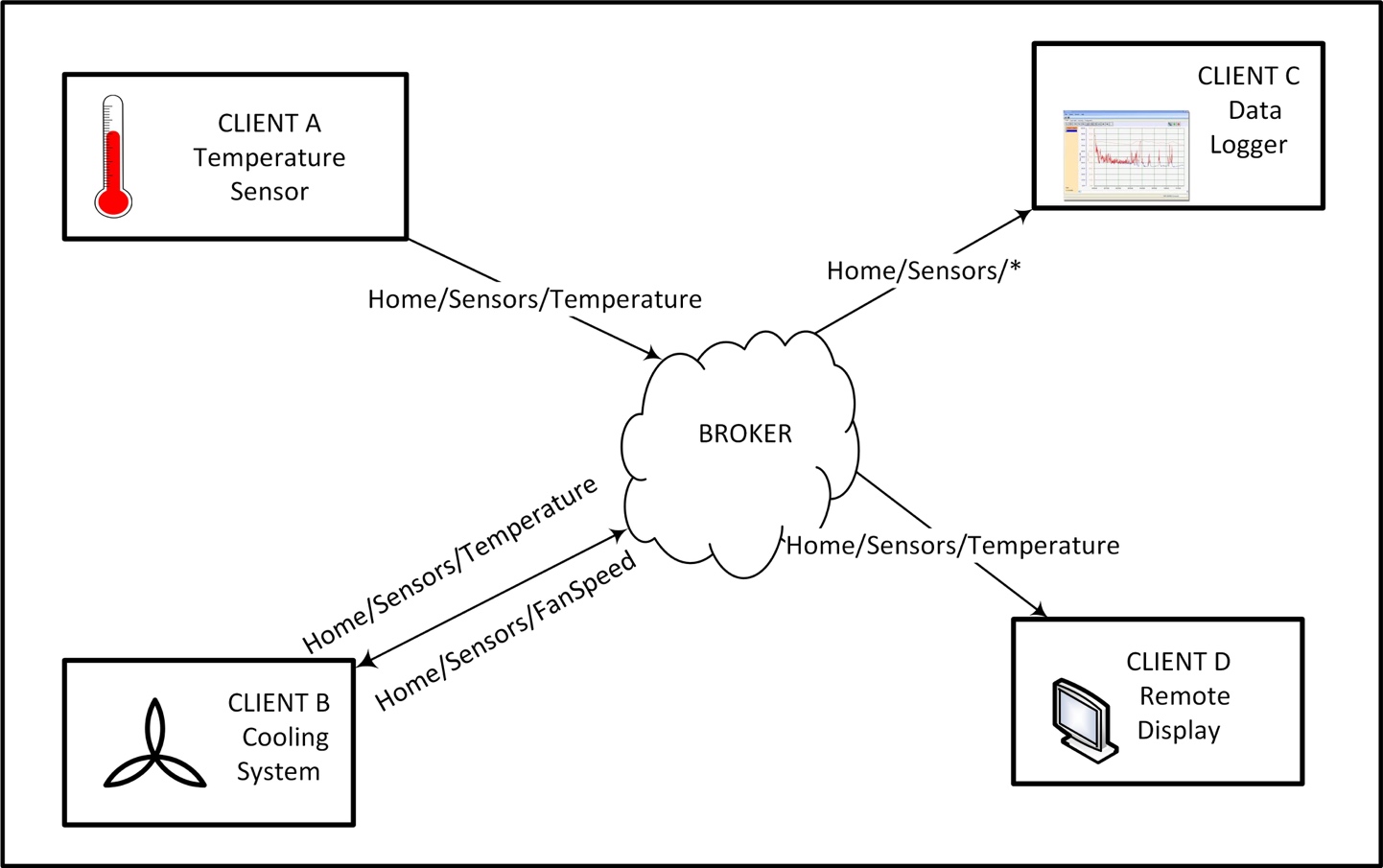
Figure 1: Example of MQTT network using hierarchical topics to organize messages

Figure 1: Example of MQTT network using hierarchical topics to organize messages

# MQTT protocol in FREEDM Energy Cell

In the context of the FREEDM Energy Cell, we can think of all the possible devices within the Energy Cell as MQTT clients which network via a broker as shown in *Figure 2*. The broker, essentially being software driven, could be actually running on any of the client devices as well to reduce the device count on the network. So a possible configuration could be as follows,

1. All the DRERs, DESDs, FIDs etc. connected to a given SST can have an instance of MQTT client running on them.
2. Now since all of them need to communicate at least with the SST, even the SST shall have an instance of MQTT client running on it.
3. Moreover, the broker could be running from SST itself as a background process.

Another possible configuration could be as shown on the right hand side of *Figure 2*.

1. An SST directly networks with a Smart Home in a small community.
2. Here we show an architecture that further exploits the choice of MQTT by having a tiered networking structure.
3. Internal to the home, all the smart sensors and appliances could be communicating to the HEMS via MQTT-SN or MQTT as applicable. We assume the presence of an appropriate Gateway device to couple the two networks.
4. And then the HEMS as a single entity could network with the SST to arrive at an optimal power usage operation.

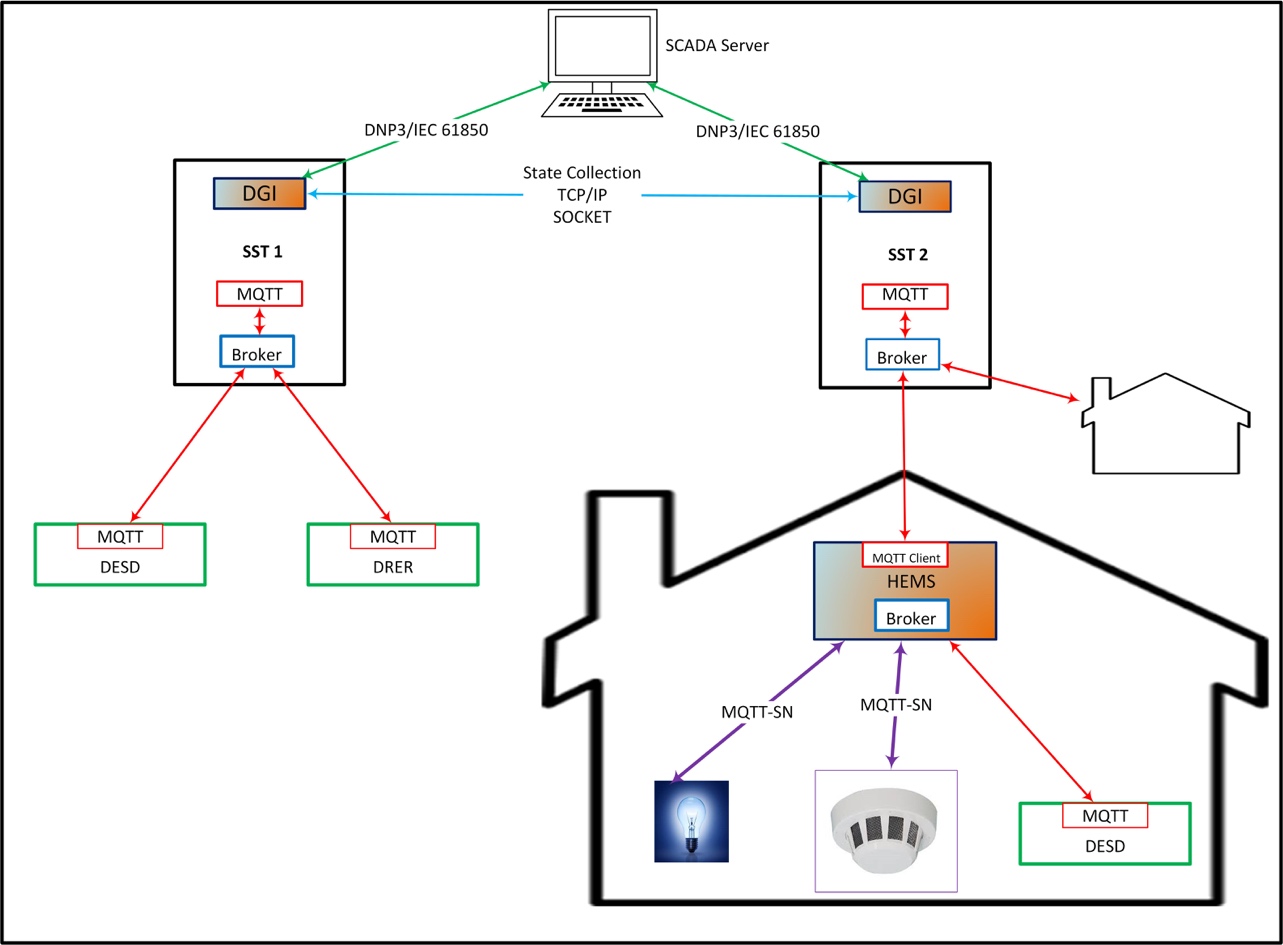


Figure 2: Multi-Tiered Network Architecture of the FREEDM System

Looking at a possible overall FREEDM communication architecture in *Figure 2*, the communication across SSTs is handled using TCP/IP sockets as managed by DGI. The SSTs could also need to communicate to a SCADA server or any central command center. Since this falls within the ambit of a distribution network, a protocol like DNP3 or IEC 61850 is ideal for this purpose.

# Building FREEDM MQTT project

1. You need python 2.7 installed on your computer. Clone the following repository.

git clone https://github.ncsu.edu/FREEDM2015/GEH.git

1. Build and install freedmqtt.

sudo make

The repository has all the files necessary. It even has Python to C/C++ interface files called Pyapi, which could be used in case of C/C++ applications like MODBUS to use FREEDM MQTT.

# Running FREEDM MQTT

Go to GEH directory and enter following command for more details,

python MQTT\_server.py –h

Usage: MQTT -d <device> [options]

-d --device <device> The name of the Device You are Running this Client on

Options:

-b --broker <127.0.0.1> The IP that the MQTT Broker is Running on

-p --port <1883> The Port the Broker is running on

-h --help Print this Usage table

Usage: MQTT -d <device> [options]

-d --device <device> The name of the Device You are Running this Client on

Options:

-b --broker <127.0.0.1> The IP that the MQTT Broker is Running on

-p --port <1883> The Port the Broker is running on

-h --help Print this Usage table

Example:

python MQTT\_server.py -d SST\_1 -b 127.0.0.1

# Implementation

We have used open source tools for our implementation of MQTT. The broker services were provided using Mosquitto (this can be any other Broker), whereas Client services were built on top of the Python implementation of Eclipse-Paho-project.

The implementation structure of FREEDM MQTT is as follows,

--- MQTT\_server.py

--- freedmqtt

--- DeviceProfile.py

--- MQTTClient.py

--- Server.py

--- Pyapi (C++ source folder)

1. **MQTT\_server.py:** This is like main file. It spawns a native process (process related to the device on which MQTT is running) and creates an instance of Device Profile which maintains the JSON file.
2. **DeviceProfile.py:** This handles maintaining data required, read and write of data to JSON file.
3. **MQTTClient.py:** Handles parsing and publishing messages/commands based on the device type, type of topics it is subscribed and type topics it has to publish.
4. **Server.py:** Handles spawning of a native process and processes of the clients to which native device is subscribed to. It also handles instantiating corresponding DeviceProfiles.
5. **Pyapi:** Python to C/C++ interface files from application to DeviceProfile. Use the source and header files in your application.

## 6.1 Device Profile

MQTT doesn’t specify any data model for messaging as a part of it’s standard. It’s completely up to the application to specify the message format and it is assumed that the receiver is aware of this message format. We devised a simple data model for prototyping purposes which can be used by any of the small scale distributed entities within the Energy Cell.

1. We start with listing all the static and dynamic attributes of interest for the device in a spreadsheet in a predefined format. This spreadsheet is called as the **Profile or Device Profile** of the device.
2. The device attributes could be either static (such as Device Name, Model, Version etc.) or dynamic (Active Power, Temperature, Discharge Capacity etc.). Static attributes do not influence the run-time behavior of the device whereas dynamic attributes either control or report the run-time behavior of the device.
3. Furthermore, the dynamic attributes could either be analog or digital in nature. And they could as well be a status report (OUT) of the device or a command (IN) to the device by one of its peers in the network. Thus we come up with further classification of device attributes as AIN, AOUT, DIN and DOUT.
4. Static attributes are correspondingly tagged by a field called DEV CHAR (device characteristics). Different attributes belonging to the same tag are identified by their indices (starting with 0). The said representation borrows heavily from the DNP3 representation of data points.
5. This spreadsheet also lists the default, minimum and maximum values for all the attributes which are used at start up (for initialization) and run time (for boundary check). A sample device profile made for the DESD is shown in Figure 3. The device profile is stored locally on the device’s BBB or equivalent board.

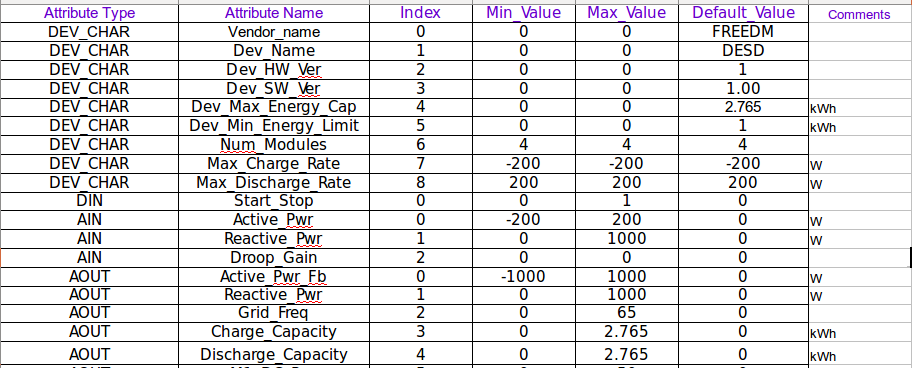


Figure : Sample Device Profile spreadsheet

## 6.2 Configuration

1. Location of Device Profile spread sheet will be ***$(GEH\_DIRECTORY)/conf/profile/.*** Make sure Device Profile spreadsheet for the given device is located in that location with the name specific to the device. For example, SST\_1.xls, DESD\_4.xls etc.
2. The file in which current device is supposed to communicate is listed in the file ***$(GEH\_DIRECTORY)/conf/<current\_devicename>.conf***
3. Make sure to add all the list of devices with which current device is supposed to communicate with.

# Data Flow Mechanisms

1. While the Device Profile excel sheet provides an easy to manage interface for the programmer to add or remove data points from the device’s object model, it is not used directly during run time as working with spreadsheets at run time could be computationally intensive.
2. Instead, we use a data format called JSON (JavaScript Object Notation) to manage the data points at run time. JSON defines a format for specifying an order collection of name/value pairs which is light weight (in terms of parsing) and easy to read/write (in terms of implementation as well as form human visibility). Such a file interface to manage data points is required because at run time the MQTT client as well as any native applications on the device could read/write into the data points.
3. The JSON file then serves as a consistent means of storing the state of the device across all the applications running on it. A set of API routines were developed to read/write specific data attributes from the JSON file (handled by ***DeviceProfile.py***) with inbuilt checks for atomic access (using file lock mechanisms), write protection and boundary checking. Any application, including the MQTT Client, would be using these routines to access the data attributes thereby avoiding any race conditions.

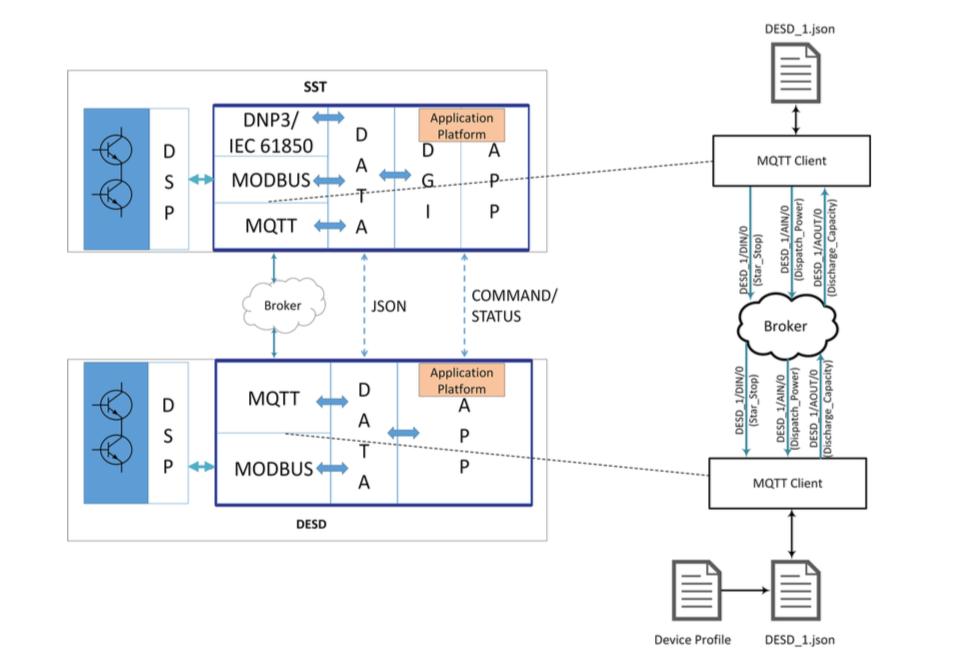


Figure : Data Flow between SST and DESD using MQTT

Figure 4 describes the data flow path across devices and within their respective controllers.

1. For any given device (like for the SST and DESD shown), there could be multiple applications, such as the MQTT Client, the Modbus implementation or any other native application (App) that could be the producer or consumer of data points stored in the JSON file (device attributes).
2. While the Modbus code deals with translation and transfer of data points within the device, MQTT Client handles the transfer of data points external to the device.
3. We can generically call the DSPs as controllers for power electronics and BBB as an Application Platform (which could possibly be any device with an operating system and a networking communication stack).

Consider the device communication between just the DESD and an SST for simplicity.

1. To begin with the DESD, at the simplest level, the DESD expects to receive commands for start/stop, power dispatch etc. from the SST and the SST receives the residual energy left in the DESD as a status update from it.
2. All of these data points are listed in the Device Profile for DESD. For example,
   1. start/stop being a binary value is considered a DIN because it is received as an external Digital Input.
   2. Similarly, power command to the DESD is an AIN data point (Analog Input).
   3. Also the residual energy calculated by the DESD based on it’s BMS system is quantified as an AOUT (Analog Output) data point.
3. At the beginning of program execution, the DESD’s application platform (BBB in our case) translates the Device Profile (which is a spreadsheet) into an equivalent JSON format.
4. This JSON file is also published over the network so that the devices that are interested in controlling or monitoring the DESD can do so based on the data points published therein.
5. Any native application that wants to read the AIN/DIN data points or write to AOUT/DOUT data points would be using the previously mentioned API routines developed to handle the JSON file in order to do so.
6. The Modbus application would need to periodically poll for any updates in the data points that need to be transferred to the DSPs and as well write to those data points which need to be updated based on the data received from the DSPs.
7. The MQTT Client would be tracking any changes that happen on the JSON file and if any of the AOUT/DOUT data points is updated by Modbus or any other native application, then the MQTT Client shall push it over the network to the broker with full hierarchy of the data point as the message topic.
   1. For example, if it wants to publish the quantity of residual energy left in the DESD over the network, it would publish the data point called ‘Discharge Capacity’ which is enumerated as an AOUT with index 0 in the Device Profile, as a message with topic ‘DESD 1/AOUT/0’. The actual name of the quantity is not required as a part of the message topic because the type (AOUT) and enumeration index (0) are enough to uniquely identify the data point and its source. (DESD 1).
8. From the perspective of the application platform on SST, as soon as the DESD application platform joins the MQTT network, it receives DESD’s Device Profile in the form of JSON file because it would have subscribed for the same with the broker.
9. The SST’s application platform would also have another JSON file which it would have generated by itself based on it’s own unique Device Profile. However, that JSON file is not relevant to the discussion here because it plays no role in communicating with the DESD.
10. As such, any device that controls another device, shall have it’s JSON file in addition to it’s own JSON file.
11. The MQTT Client on SST has a two-fold task: one, to subscribe to the relevant AOUT/DOUT topics of DESD so that anything that the DESD publishes for those data points is received by the SST and second, to publish to DESD’s AIN/DIN data points in order to control it.
12. The static characteristics of the DESD, under the category of DEV CHAR are not subscribed to because they are not updated/published during run time, instead they are read directly from the JSON file received at the beginning of the communication.

In this way, MQTT Clients on all the devices connected to the network essentially seek to synchronize the data points for each device via updates in their respective JSON files. Applications running natively on the devices can then read or write to the data points by just updating the JSON file’s corresponding data field. They are agnostic to the underlying communication network.

# Example JSON file

The following example JSON files encodes DeviceProfile.py object with the relevant data.

{"py/object": "freedmqtt.DeviceProfile.DeviceProfile", "DIN": [py/object":

{""freedmqtt.DeviceProfile.din", "index": 0, "name": "Start\_Stop", "maximum":

{"1.0, "value": 0, "minimum": 0.0}], "AOUT": [py/object":

{""freedmqtt.DeviceProfile.aout", "index": 0, "name": "Active\_Pwr\_Fb",

{""maximum": 1000.0, "value": 0.0, "minimum": -1000.0}], "DOUT": [],

{""Name\_Hash": Active\_PwrIN": "AIN/0", "Start\_StopIN": "DIN/0",

{""Max\_Charge\_Rate": "DEV\_CHAR/7", "Num\_Modules": "DEV\_CHAR/6", "Dev\_Name":

{""DEV\_CHAR/1", "Dev\_Max\_Energy\_Cap": "DEV\_CHAR/4", "Active\_Pwr\_FbOUT":

{""AOUT/0", "Dev\_Min\_Energy\_Limit": "DEV\_CHAR/5", "Dev\_SW\_Ver": "DEV\_CHAR/3",

{""Vendor\_name": "DEV\_CHAR/0", "Max\_Discharge\_Rate": "DEV\_CHAR/8", "Dev\_HW\_Ver":

{""DEV\_CHAR/2"}, "Topic\_Hash": DEV\_CHAR/8": "Max\_Discharge\_Rate", "DIN/0":

{""Start\_Stop", "AIN/0": "Active\_Pwr", "DEV\_CHAR/2": "Dev\_HW\_Ver", "DEV\_CHAR/1":

{""Dev\_Name", "DEV\_CHAR/0": "Vendor\_name", "DEV\_CHAR/7": "Max\_Charge\_Rate",

{""DEV\_CHAR/6": "Num\_Modules", "DEV\_CHAR/5": "Dev\_Min\_Energy\_Limit",

{""DEV\_CHAR/4": "Dev\_Max\_Energy\_Cap", "DEV\_CHAR/3": "Dev\_SW\_Ver", "AOUT/0":

{""Active\_Pwr\_Fb"}, "file\_1": "json/LVSST\_1.json", "flag": 0, "AIN":

{"[py/object": "freedmqtt.DeviceProfile.ain", "index": 0, "name": "Active\_Pwr",

{""maximum": 1000.0, "value": 0.0, "minimum": -1000.0}], "DEV\_CHAR":

{"[py/object": "freedmqtt.DeviceProfile.dev\_char", "index": 0, "name":

{""Vendor\_name", "value": "FREEDM"}, py/object":

{""freedmqtt.DeviceProfile.dev\_char", "index": 1, "name": "Dev\_Name", "value":

{""LVSST"}, py/object": "freedmqtt.DeviceProfile.dev\_char", "index": 2, "name":

{""Dev\_HW\_Ver", "value": 1.0}, py/object": "freedmqtt.DeviceProfile.dev\_char",

{""index": 3, "name": "Dev\_SW\_Ver", "value": "1.00"}, py/object":

{""freedmqtt.DeviceProfile.dev\_char", "index": 4, "name": "Dev\_Max\_Energy\_Cap",

{""value": 0.0}, py/object": "freedmqtt.DeviceProfile.dev\_char", "index": 5,

{""name": "Dev\_Min\_Energy\_Limit", "value": 0.0}, py/object":

{""freedmqtt.DeviceProfile.dev\_char", "index": 6, "name": "Num\_Modules",

{""value": 1.0}, py/object": "freedmqtt.DeviceProfile.dev\_char", "index": 7,

{""name": "Max\_Charge\_Rate", "value": -1000.0}, py/object":

{""freedmqtt.DeviceProfile.dev\_char", "index": 8, "name": "Max\_Discharge\_Rate",

{""value": 1000.0}], "write\_time": py/object": "datetime.datetime",

{""\_\_reduce\_\_": [py/type": "datetime.datetime"}, ["B98HFQ0kCgHOCA=="]]},

{""initialized": 1, "start\_time": py/object": "datetime.datetime", "\_\_reduce\_\_":

{"[py/type": "datetime.datetime"}, ["B98HFQ0jBgdctQ=="]]}, "change": ["Error"],

{""native": 1}