LoRaWAN Evaluation

Master Project

Paul Spooren

2021-09-27

Contents

1 Introduction

This project evaluates the *LoRa* frequency modulation and the *LoRaWAN* stack as an alternative for existing sensor setups using the technologies *ZigBee* and *GSM*. The current scope is to build a simple stack to collect sensor data end visualize them inside a web interface.

Within this work modular nodes were designed allowing to measure a variety of environmental metrics, including temperature, rain fall, humidity and sea level.

Software and hardware used on sensor nodes are documented to allow easy replication of the setup. Further multiple recommendations regarding software libraries and hardware are given.

All work happened in collaboration with the *MESH LAB* at Oahu, Hawaii, supervised by Dr. Brian Glazer and Dr. Edo Biagioni.

> Note: This manual is work in progress and may receive updates over time. A list of changes will be attached to the bottom with corresponding dates.

1.1 Motivation

Based on the architecture of LoRaWAN, described in the TheThingsNetwork section, it allows a separation of sensor nodes and base stations. In simple terms this could be compared to a free cellular network where every user brings their own phone while the cellular base stations are managed by an organization with the public interest of connectivity. With LoRaWAN it is possible for researchers to deploy their sensor nodes while other entities, like universities or even states, handle the signal coverage.

The existing setup aimed to upgrade and extend uses both cellular *GSM* and *ZigBee*. While the formers base stations are managed by third parties it involves a recurring fee of usage, making large scale deployments less feasible. *ZigBee* requires researchers to manage their own gateways since there is no roaming standard or protocol for multiple users to share a base station.

LoRaWAN solves this by introducing a software stack which allows multiple entities to share base stations but also allows easy deployment of base stations contributing to the network as a whole.

Within this project existing sensor setups where *rebuild* to use LoRaWAN instead of *GSM* or *ZigBee*, reporting the same results while using a different medium of transportation. Collected data is stored in the same database as existing nodes, theoretically allowing a migration over time without changing work flows.

1.2 Aim

This work focuses unlike related scientific papers on a reproducible manual of said sensor setups. Multiple papers in the Related Work section prove the usability of LoRaWan as a concept and specifically for the *Internet of Things* and *sensor network* use case.

Specific steps to deploy a *sensor network* are described to combine available resources to a working, low-cost live monitoring system with possible data science pipelines.

1.3 Structure

Within this document no specifics of the LoRaWAN protocol are covered. For technical details on the LoRa frequency modulation or the LoRaWAN stack the Related Work section gives further information.

In the Hardware used microcontrollers and LoRaWAN gateways are covered. Additional documentation is provided on the custom designed *Printed circuit board* (PCB) as well as the used outdoor casing.

The Software section describes all code and tooling used and created within this project. Multiple sub-sections describe the full setup in detail and should be followed in that order.

Within the brief Resources section the advantages of the *Cayenne Low Power Protocol* for data transmission as well as an overview of used sensors for multiple environmental metrics.

1.4 Related Work

Using LoRaWAN for sensor networks is not a new idea and multiple papers describe a variety of experiments[1]. Also the technology of LoRa frequency modulation as well as the LoRaWAN stack was discussed in multiple publications[2].

This projects is similar to work of *Davec et al*[3] where they proof the feasibility of sensors over large distances gathering environmental metrics, in their case specifically temperature and soil moisture.

Next to academic publications this work is based on free documentation available online, notably the Arduino Reference and the many code examples by *HelTec Inc.* within their CubeCell repository.

Low-cost weather stations based on Arduino with custom circuit boards were also described by the OpenWeatherStation project and DFRobot, however neither uses LoRaWAN and is therefore more bound to local installations.

Complete setups from specialized vendors like Onset Computer Corporation offer professional weather stations, however come at a much higher price tag as solutions within this project.

2 Hardware

Within this section all hardware like microcontrollers, LoRaWAN base stations and outdoor casing are described.

2.1 Microcontroller

This section describes the used microcontrollers and development boards used within this setup. The microcontrollers is essentially an extremely small computer with hardware inputs and outputs. In simple terms, sensors are attached to the inputs and a LoRaWAN radio frequency module is connected to the outputs.

Using microcontrollers comes in two flavors, either as development boards with populated connectors, for instance a USB port and attached LEDs or as a plain module, which requires designing of a custom PCB. The

former is used for rapid prototyping while the latter runs on an optimized PCB which only contains required functionality.

To simplify the development only microcontrollers compatible with the Arduino framework were considered. The framework abstracts talking to low level hardware components and offers a simple C API to control the microcontrollers behaviour. Additionally this allows to migrate code from one device to another with minimal code modifications sine both devices understand the same API.

Due to low price and using the LoRaWAN reference implementation the *CubeCell* product line from the vendor *Heltec Automation*. Both development boards *CubeCell Dev-Board* (`htcc-ab01`) and *CubeCell Dev-Board Plus* (`htcc-ab02`) as well as the plain *Module Plus* (`htcc-am02`) were used.

2.1.1 Heltec CubeCell Dev-Board (Plus)

Figure 1: Dev Board

Figure 2: Dev Board Plus

The development boards shown on the right both allow to connect sensors via *GPIO Pins* as well as *I2C* or serial console. Only the bigger *Plus* versions (bottom) contains two *Analog to Digital converter* (ADC) which are required for some sensor like the *VH400* moisture sensor.

Additionally the *Plus* version comes with an attached LCD display which can be used for debugging or additional information during setup. No long time tests were performed to evaluate the UV resistance of the display.

For outdoor setups they come with a solar charge controller and battery connector which allows the usage without a constant power supply.

Both boards use the same microcontroller and can run the same code.

Flashing is trivial by using the USB connection and the Platformio framework.

- Dev-Board Pinout
- Dev-Board Plus Pinout

2.1.2 Heltec CubeCell Module Plus

Figure 3: Module Plus

The *Module Plus* allows (and requires) to use custom PCB and therefore only runs parts that are required. For production deployments the nodes would not need a LCD screen for debugging or a USB port for flashing. These components can simply be removed in the PCB design which reduces the number or parts, thereby cost and complexity.

Dev-Board Plus and *Module Plus* use the same microcontroller and therefore support the same connections.

More information on the PCB required for the *Module Plus* is available in the PCB section.

- Module Plus Pinout

2.2 Gateways

While the Microcontrollers with an attached LoRaWAN radio frequency module mostly send data, the receiving part is called gateway. Essentially it is a device with a LoRaWAN compatible receiver attached as well as an Internet connection. Received packages are forwarded to a brooker. More details on that are available in the LoRaWAN section

Within this project the following three gateways were used as described below. This was partly necessary as LoRaWAN coverage with TheThingsNetwork broker were limited within the area of research, Honolulu, Ohau, Hawaii. At other locations institutions or private entities may already provide coverage free of charge. Using the TTNMapper it's possible to see a local coverage map and determine if a self maintained gateway is required.

2.2.1 The Things Indoor Gateways

Figure 4: Indoor Gateway

This gateway allows simple operation since it's developed by the same people as the used LoRaWAN broker. It was used during local development and allows quick migration between different locations. While not waterproof, using additional casing and attaching an external antenna it can also be used as a outdoor gateway. For mobile development it is possible to power the gateway via a USB-C cable.

A complete install and maintenance guide is available in the vendors documentation and is not replicated here.

2.2.2 Heltec HT-M00

Figure 5: HT-M00

Just as the microcontrollers Heltec offers a development gateway as well. At the time of writing this is the cheapest development gateway available with a price of about $40. The upstream documentation guides through the installation process of the gateway into TheThingsNetwork broker. For mobile development it is possible to power the gateway via a USB-C cable.

2.2.3 Mikrotik wAP LR9 kit

Figure 6: wAP LR9 kit

Lastly the outdoor LoRaWAN gateway from the vendor Mikrotik (in 900MHz or 860MHz) offers the, at the time of writing, cheapest outdoor gateway. With *Power over Ethernet* (PoE) the installation is possible with a single Ethernet cable without requiring (ideally elevated) deployment location.

2.3 Printed Circuit Board

This section briefly describes the PCBs designed within this project. While breadboards are often used for prototyping with microcontroller, real deployments should use custom PCBs since they offer a more reliable setup.

The project scope required a rain gauge sensor as well as a temperature sensor to transmit measurements over LoRaWAN to a online database. With the below described PCB it is possible to attach a *Heltec HTCC-AB02* as well as the two sensors within minutes.

2.3.1 *Rain Gauge* PCB

As mentioned above the two values *temperature* and *rain fall* were to be collected and send to an online database. First prototypes using a breadboard and a *Heltec HTCC-AB02* worked out as expected, so the same exact circuit

was transfered in a PCB. On the right side a picture of the PCB is shown. The labels `RAIN_GAUGE`, `SOLAR_IN` and `ONE_WIRE_IN` contain screw terminals which allow easy attaching of the sensor cables.

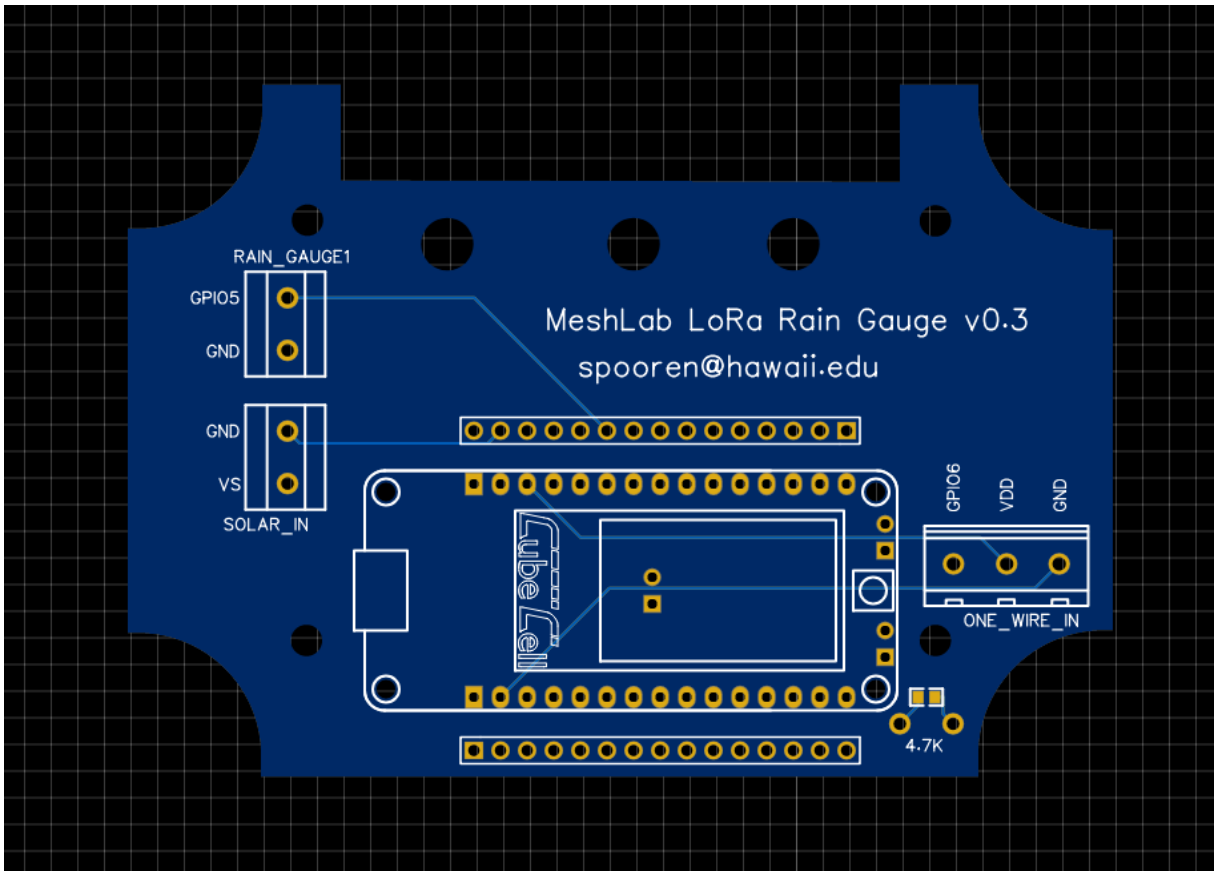

Figure 7: Custom Rain Gauge Design

Below is a printed and mostly assembled (missing 4.7k resistor) board. The *CubeCell board* can be directly attached to the pin headers, the labeled screw terminals allow a simple connection of sensors. Next to the attached pin headers is a second row of pins which can be used for developing to connect any other GPIO which might be of interest in the future.

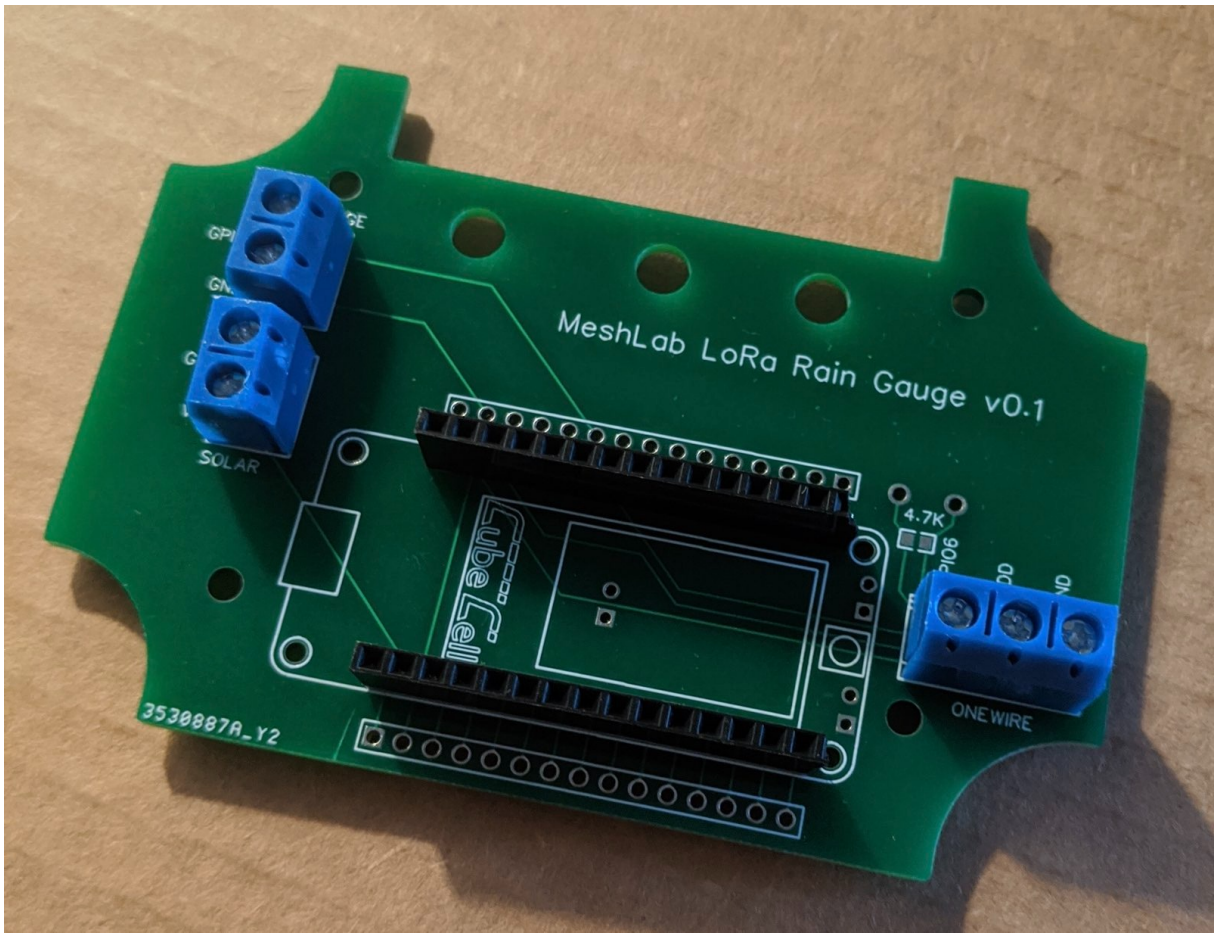

Figure 8: Custom Rain Gauge Assembled

An EasyEda project file is available here for customization or ordering online.

Additionally the *BOM*, *Gerber* and *Pick & Place* files are available:

- BOM_PCB_rain-box_htcc-ab02.csv
- Gerber_PCB_rain-box_htcc-ab02.zip
- PickAndPlace_PCB_rain-box_htcc-ab02.csv

The PCB has a custom cutting so it fits perfectly into the outdoor case PTK-18420-C by *Bud Industries, Inc.*. It is important to order the C version since a **clear cover** is required to allow solar charging. Using cable glands it's possible to connect external sensors. The picture below shows two possible positions for 7mm holes to attach PG7 cable glands. Silicon was added around the cable glands for better long term water resistance.

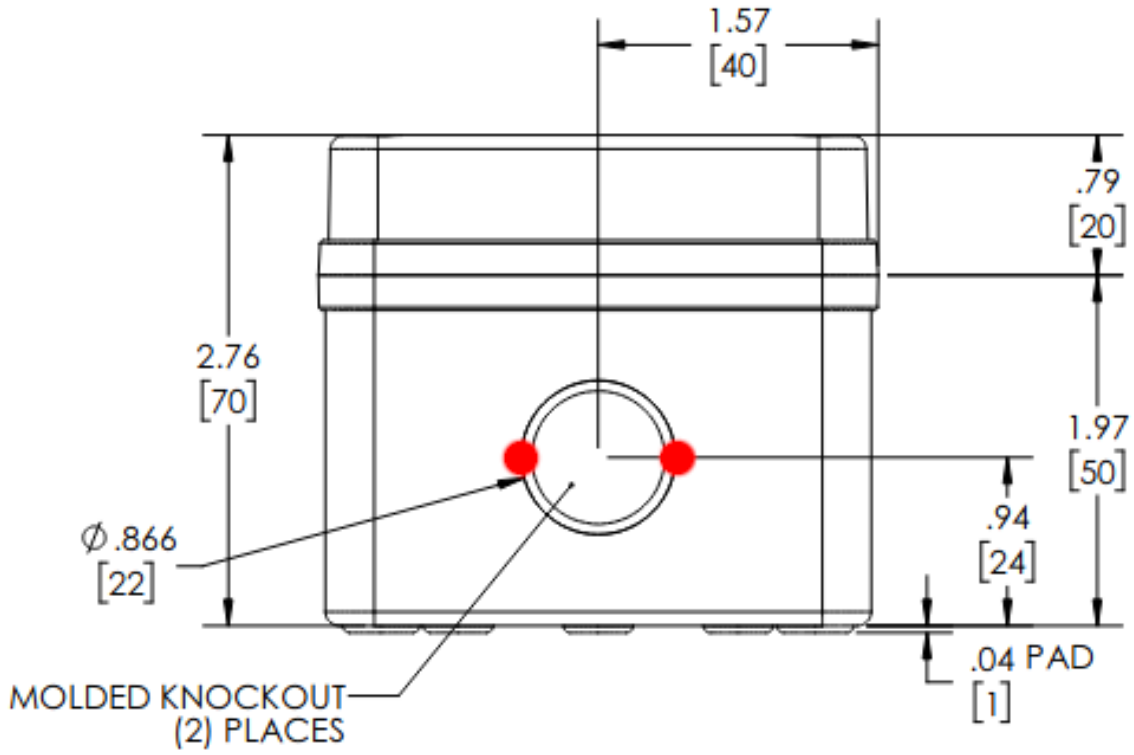

Figure 9: Drill Example for PTK-18420-C Box

The full box specifications are available on the vendors website or as an attached resource.

It is possible to use the *Pick and Place* file to order all parts soldered on or solder all components manually. In that case a `4.7K` resistor should be added on the bottom left instead of a flat resistor.

A picture of an assembled *Rain Box* is shown below. The picture shows four different setups of the rain gauges.

- The **left box** has a HOBO RG3 rain gauge
- The **middle box** has only a DS18B20 temperature sensor is connected inside the box. This was done to measure the maximum daily temperature inside the box.
- The **right box** has both a temperature sensor and a MISOL rain guage attached. By the time of writing (2021-09-16) the official vendor website is unreachable.
- In **front** is a PCB board without a box showing a connected *Heltec HTCC-AB02*.

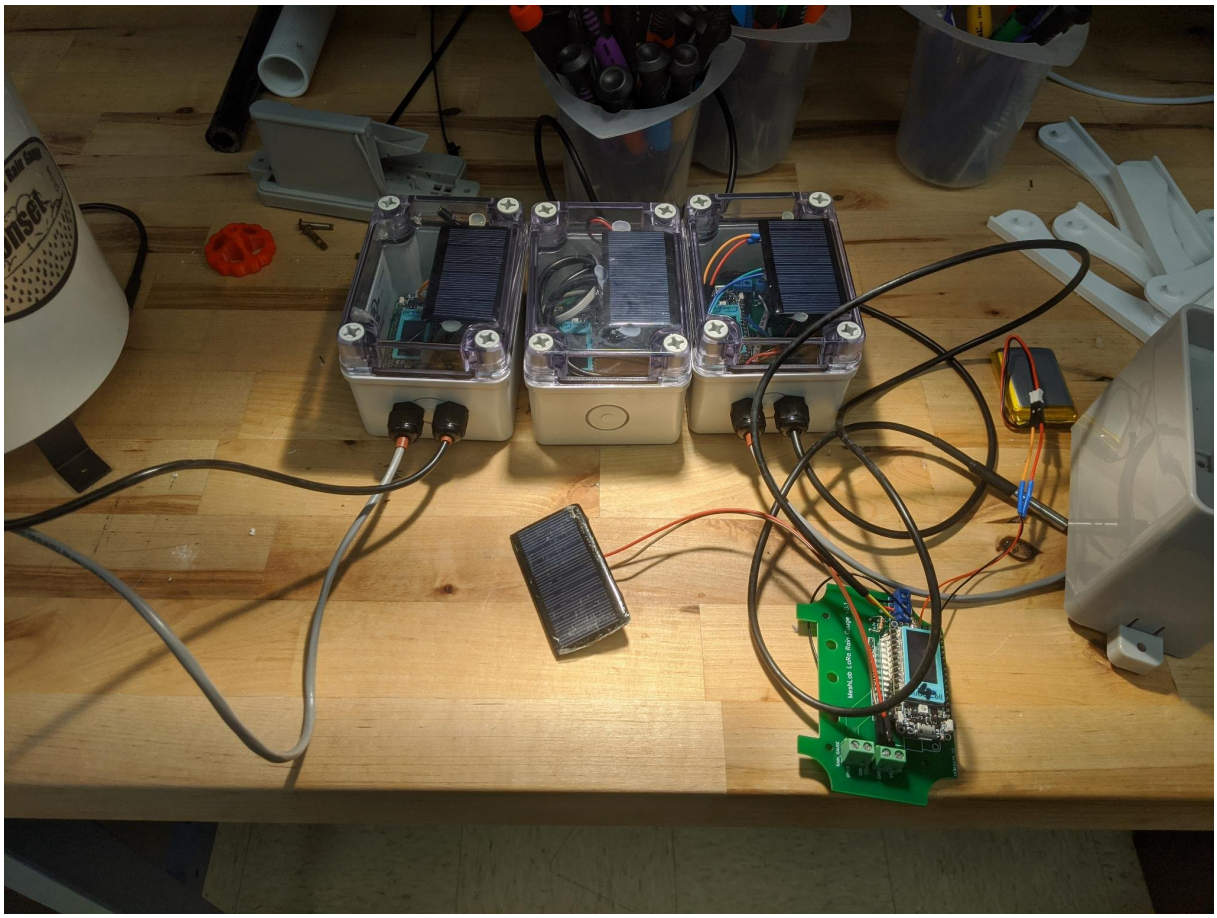

Figure 10: Rain Gauge Setup

2.3.2 PCB Editor

To design PCBs a variety of tools is available for free. However the number of tools that are available on the three main platforms Windows, MacOSX and Linux is limited. Within this project EasyEda was used for it's simple usage and integration with part libraries, which would simplify the ordering process.

EasyEda comes with a tight integration of the PCB vendor *JLCPCB* which offers both creation of PCBs and also soldering of selected components.

An extensive official documentation is available in English on the vendors website include video tutorials.

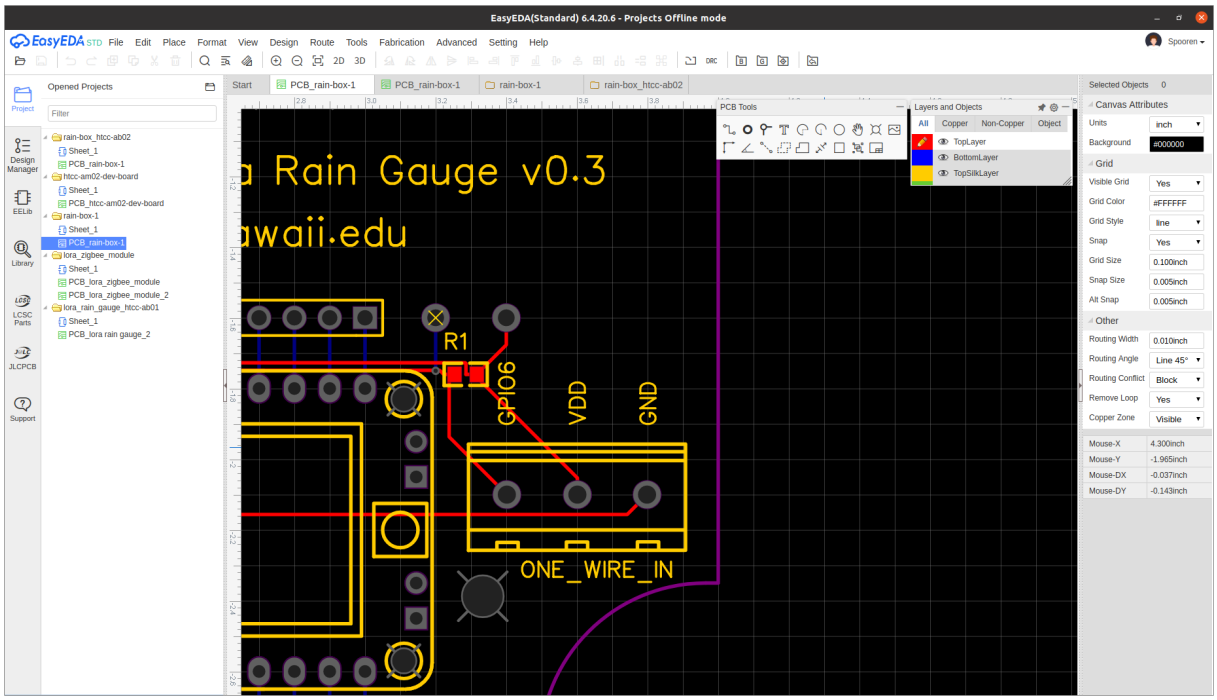

Figure 11: EasyEda PCB Designer

If EasyEda is not an option the open source tools LibrePCB or KiCad could be used instead, however they are not covered here.

3 Software

This section describes all code and tooling used and created within this project. It is recommended to read the Hardware section first to have the required sensor nodes.

3.1 TheThingsNetwork

This section gives a brief introduction to the free *TheThingsNetwork* service which provides a *Network server* to handle LoRaWAN authentication, forwarding and data access. More information on the LoRaWAN stack *Rizzi et al*[2] give a great introduction how communication from sensor node to user facing *Application server* works.

In short, *TheThingsNetwork* is a free, partly commercial, partly community driven service which handles the management of nodes, keys, base stations and APIs to access collected data. The picture below show a simplified overview of the stack.

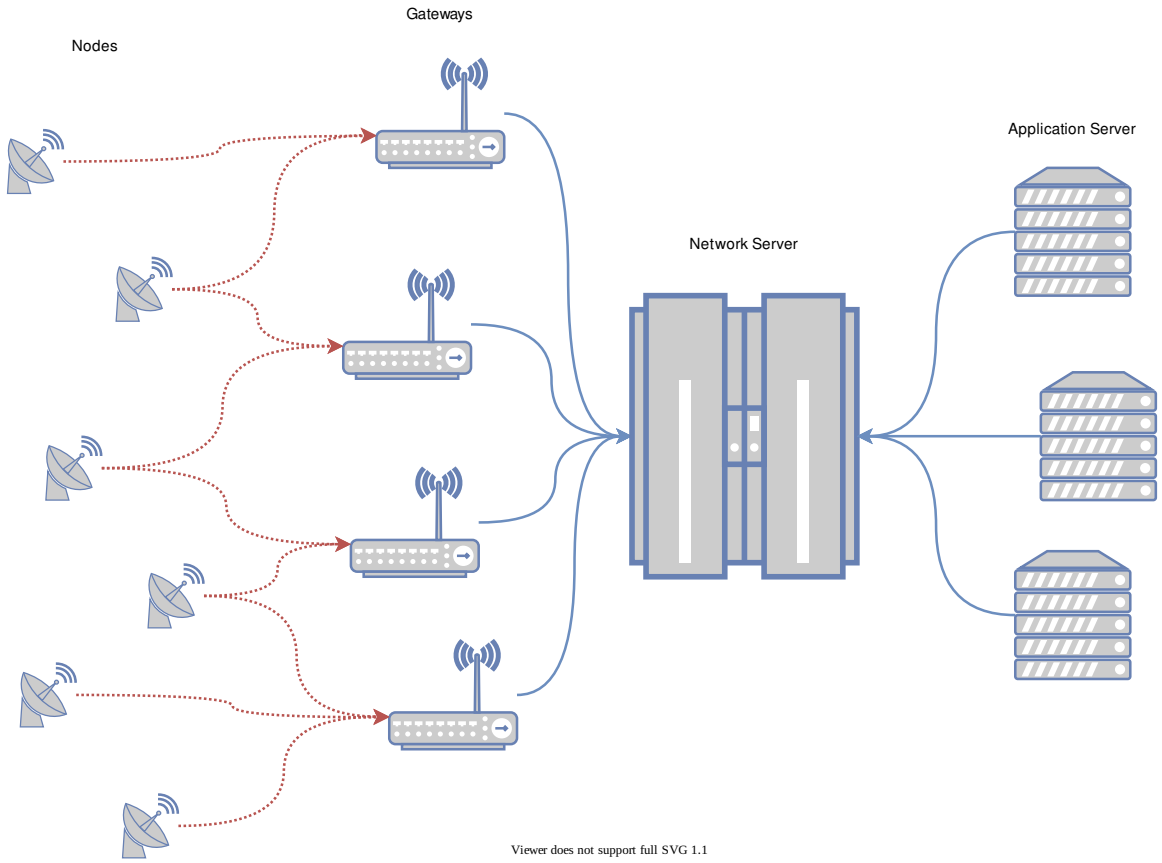

Figure 12: The LoRaWAN stack

Sensor **Nodes** collect data and send it via LoRa radio frequency modulation (red lines) to **Gateways**. A gateway has a LoRa compatible radio module and antenna attach and listens for incoming messages. Once received, messages are forwarded to a broker or **Network Server**. The network server, in this setups case thethingsnetwork.org, manages user accounts and **applications**. Each application has multiple **Nodes** assigned with individual login credentials. Forwarded messages from gateways are offered via MQTT to user of specified application. User controlled **Application Servers** can listen to MQTT streams and store sensor data for further processing.

No user can access applications of other users and all traffic, starting from sensor nodes until the application server is encrypted. This allows a federation of infrastructure since multiple entities can share access to a single gateway without being able to manipulate each others data.

As an outcome, both private hobbyists and institutions like universities can offer gateway access and thereby work together to cover large areas. Different projects don’t have to manage their infrastructure independently but can share gateways. Since all traffic is encrypted and authenticated as defined by the LoRaWAN standard, no code has to be written by users to allow secure connections.

Lastly multiple integrations allow even easier setups than described within this document. Instead of running a self managed database, it is also possible to rely on cloud services storing MQTT messages and offering convenient interfaces.

3.1.1 Web interface (Console)

The cloud service requires a free online account and the setup of an **Application**. Each application then contains multiple **End devices** which which provide sensor data. Received data is offered via an MQTT API can should be consumed by a user controlled database, in this setup InfluxDB or one of the integrations.

Within this project the *North American Cloud* (nam1) was used, other locations should use *Europe 1* (eu1) or *Australia 1* (au1). While the outdated v2 stack is still offered, it **should not be used** due to it's upcoming end of life (December 2021). Many resources online still document the *v2* which should be avoided.

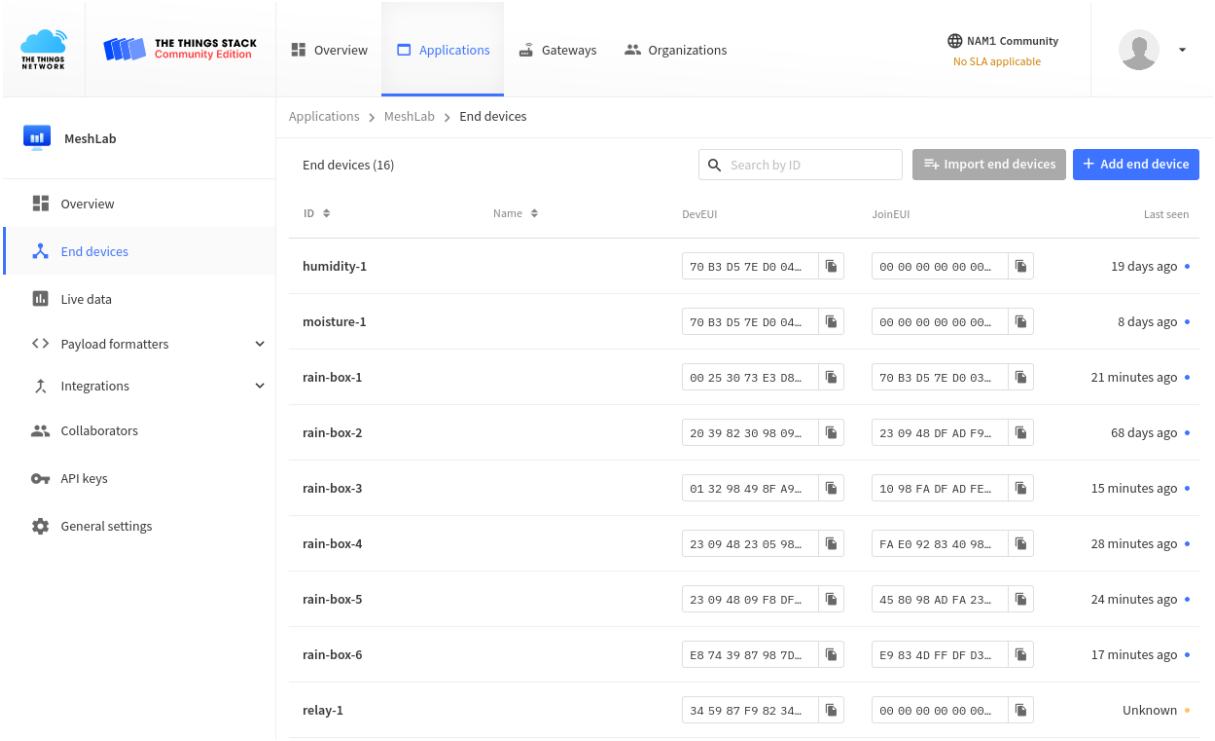

Figure 13: TheThingsNetwork device overview

The above image shows currently active end devices, their identifiers as well as the time of the last received message.

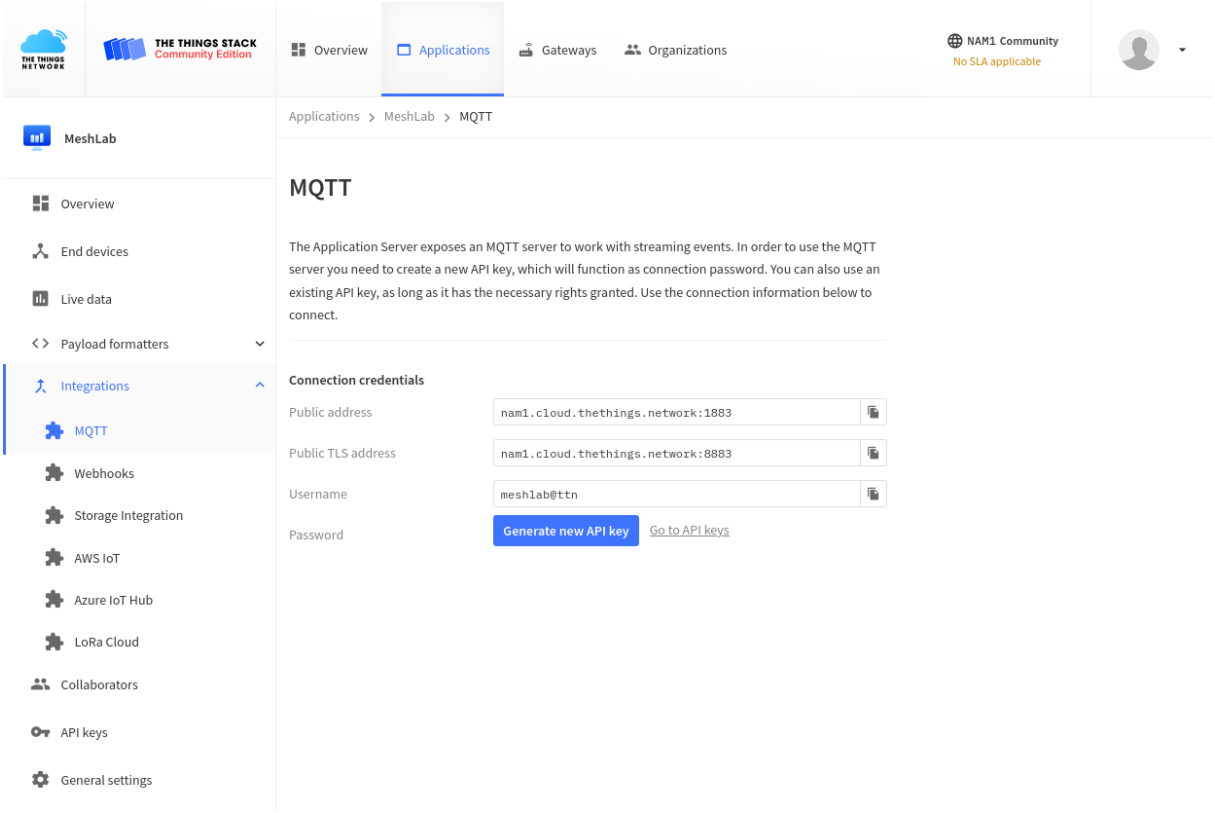

Figure 14: TheThingsNetwork MQTT access

Aboves picture shows the MQTT login credentials which can be either used within the InfluxDB setup to have *Telegraf* listen to MQTT and store data or for other applications.

In the code appendix is a short Python script called `mqtt2json.py` which listens to the MQTT stream and outputs lines of JSON which can be consumed by other applications. A generic script like this allows custom downstream handling of incoming sensor data. As proof of concept the script below is parsed by another tool adding measurements to a SQL database and visualize them via a custom web pagegrogdata. Nodes are identified via the `from` field and all payload is stored in the `p` field, containing data fields based on CayenneLPP.

Below is an example output of the script which could be read by future tooling using the *JSON Lines* standard

```
{"from": "rain-box-5", "p": {"temperature_1": 23.1}}
{"from": "rain-box-1", "p": {"temperature_1": 23.5}}
{"from": "sonic-2", "p": {"distance_1": 4.991}}
{"from": "rain-box-6", "p": {"temperature_1": 21.4}}
{"from": "rain-box-4", "p": {"digital_out_1": 2, "voltage_1": 4.15}}
{"from": "sonic-2", "p": {"distance_1": 5}}
{"from": "rain-box-5", "p": {"digital_out_1": 2, "voltage_1": 4.22}}
{"from": "sonic-2", "p": {"distance_1": 5}}
{"from": "rain-box-4", "p": {"temperature_1": 24.7}}
{"from": "rain-box-5", "p": {"temperature_1": 23.6}}
{"from": "rain-box-1", "p": {"temperature_1": 23.6}}
{"from": "sonic-2", "p": {"distance_1": 5}}
{"from": "rain-box-6", "p": {"temperature_1": 21.5}}
```

```
{"from": "rain-box-3", "p": {"temperature_1": 24.1}}
{"from": "sonic-2", "p": {"distance_1": 5}}
{"from": "sonic-2", "p": {"distance_1": 5}}
{"from": "rain-box-4", "p": {"temperature_1": 24.6}}
{"from": "rain-box-5", "p": {"temperature_1": 23}}
{"from": "rain-box-1", "p": {"temperature_1": 23.7}}
...
```

3.2 Platformio

This section describes the usage of the *Platformio* framework for embedded development. Similar to the *Arduino IDE* it allows the management of dependencies and compilation for a wide variety of devices.

Due to its simple installation and full support for the used hardware it was preferred over other the *Arduino IDE*, however other tooling should be possible as well.

3.2.1 Installation

A full installation guide is available in the upstream documentation however the basic installation boils down to a running Python 3 installation combined with the Python Packet Manager or `curl`:

```
pip install -U platformio

# or

python3 -c "$(curl -fsSL https://raw.githubusercontent.com/platformio/
    platformio/master/scripts/get-platformio.py)"
```

After the installation the shortcut `pio` executes the binary.

3.2.2 Creating a project

Adding projects is as easy as creating a folder and running `pio` to initialize the project in the current folder.

```
mkdir project/ && cd project/
pio project init
```

After the initialization the following files are created:

```
include/       # header files
lib/           # libraries
platformio.ini # platformio configuration file
src/           # source code (e.g. main.cpp)
test/          # automatic tests
```

See the Git repository folder `node/` to see the main project structure.

3.2.3 `platformio.ini`

The configuration file contains one or multiple environments per used device type. Within this projects environments for the devices `cube_cell_board`, `cubecell_board_plus` and `cubecell_module_plus` are used. The file below shows one of the three environments and defines multiple aspects or the project which simplify development:

- `board_build.ardiono.*` defines values passed during compile time to enable and configure specific features.
- `upload_port` defines the local connection, which allows to have multiple devices connected at once and test them in parallel
- `lib_deps` defines all required dependencies which can be hosted within the Platformio registry or Git repositories directly.

```
[env:cubecell_board_plus]
board = cubecell_board_plus
platform = asrmicro650x
framework = arduino
board_build.arduino.lorawan.region = US915
board_build.arduino.lorawan.netmode = OTAA
board_build.arduino.lorawan.adr = ON
monitor_speed = 115200
upload_port = /dev/ttyUSB0
lib_deps =
    https://github.com/ElectronicCats/CayenneLPP.git#master
    bblanchon/ArduinoJson @ ^6.17.2
    practicalarduino/SHT1x@0.0.0-alpha+sha.be7042c3e3
```

3.2.4 Flashing nodes

Once a sensor node is connected it's can be flashed using the following command:

```
pio run --target upload
```

For convenience it is also possible to upload and monitor the serial output at once:

```
pio run --target upload --target monitor
```

3.3 Sensor Node

This section describes the software used on sensor nodes. All code is deployed on devices as described in the Platformio section. After flashing nodes are provisioned using the `provision_node.py` script in combination with YAML files containing the per device specific information.

The detail section below contain the following content: * config.h Contains node specific configuration options, like the sensors to enable on which PINs. This should be modified per use case, for instance when creating a new series of rain gauges measuring moisture as well, the sensor should be enabled here. Per node configuration happens via the `provision_node.py` script after flashing. * main.cpp Contains the main loop which connects

to sensors and sends LoRaWAN packets. * VH400.cpp Minimal library to communicate with *VH400* moisture sensor. * MB7389.cpp Minimal library to communicate with *MB7389* ultrasonic sensor.

3.3.1 Using `provision_node.py` for `AT_Commands`

The *Heltec Dev Boards* allow the configuration of certain on-device variables via `AT_Commands`. Those commands can be communicated via a serial connection (e.g. over USB) and modify per-device specific LoRaWAN credentials or user defined values.

A list of the most important `AT_Commands` is shown below:

AT Command	Value
+LORAWAN=1	LoRaWAN 1, LoRa 0
+OTAA=1	OTAA -1, ABP-0
+Class=A	Class A or C
+ADR=1	1 on 0 for off
+IsTxConfirmed=1	LoRaWAN ACK Message 1 on, 0 off.
+AppPort=2	The Application Port 2 for general APPs and 10 for TTN MAPPER.
+DutyCycle=60000	The time between transmission in mS. Typically, 15000 to 3600000
+DevEui=???	Unique (OTAA Mode)
+AppEui=???	Unique (OTAA Mode)
+AppKey=???	Unique (OTAA Mode)
+ChipID=?	get ChipID
+JOIN=1	start join

To allow managing multiple devices YAML configuration files are used containing credentials of nodes. After flashing a node via Platformio a node can be configured by running a command like the folloing:

```
python provision_node.py rain-box-1
```

Below is an illustrative content of `./nodes/rain-box-1.yml`. All contents of the `at_commands` dictionary are executed via the provision script. All other values like **`case`** and `sensors` can be used to keep an inventory in a machine-readable format.

The special value `mmPerTip` is a *user* `AT_Commands` which is defined within `main.cpp` and determines how many millimeters a rain gauge tip adds to the total value.

```
case:
  type: PTK-18420-C
  label: rain-box-1
```

```
4
5  sensors:
6    - HOBO RG-3
7
8  at_commands:
9    DevEui: 0025307XXXXXXXXX
10   AppEui: 70B3D57ED003A1D1
11   AppEui: 70B3D57XXXXXXXXX
12   AppKey: 1F41DD0XXXXXXXXXXXXXXXXXXXXXXXX
13   mmPerTip: 0.3851
```

3.4 InfluxDB Database

This sections covers briefly the setup of InfluxDB, a time series database which allows storing and querying metrics. Just like all other tools used within this document it's open source and all code is available online.

Using InfluxDB comes with the two advantages of offering integrations for both Grafana as well consuming MQTT events, which is the way the LoRaWAN brooker reports metrics.

3.4.1 Installation of InfluxDB

> While InfluxDB 2.0 was released this year, this document describes the setup and usage of InfluxDB 1.8. All tooling should be compatible with both versions and a migration guide might be added later on.

The upstream documentation explains the installation process for various setups, for the RaspberryPi setup the Ubuntu & Debian steps should be followed. Within this documentation both database and user are called `telegraf`, however other use cases may use other values.

3.4.2 Installation of Telegraf

> Telegraf is used to consume MQTT events and store them inside InfluxDB.

InfluxDB and Telegraf are both developed by *InfluxData Inc.*, so a similar installation process is offered. Again the official installation guide for Ubuntu & Debian should be used.

Once installed a custom configuration is required to consume data from TheThingsNetwork and store them locally. All values in `<...>` should be changed accordingly to the actual setup.

```
1  # /etc/telegraf/telegraf.conf
2  [global_tags]
3  [agent]
4    interval = "10s"
5    round_interval = true
6    metric_batch_size = 1000
7    metric_buffer_limit = 10000
8    collection_jitter = "0s"
9    flush_interval = "10s"
10   flush_jitter = "0s"
11   precision = ""
```

```
  hostname = ""
  omit_hostname = false

[[outputs.influxdb]]
  database = "telegraf"
  urls = [ "http://localhost:8086" ]
  username = "<influxdb_username>"
  password = "<influxdb_password>"

[[inputs.mqtt_consumer]]
  servers = ["tcp://nam1.cloud.thethings.network:1883"]
  topics = [ "#" ]
  username = "<thethingsnetwork_username>"
  password = "NNSXS.<thethingsnetwork_api_token>"
  data_format = "json"
  tag_keys = [
    "end_device_ids_device_id"
  ]
```

Once both services are running one should proceed setting up Grafana to create a dashboard presenting collected metrics.

3.5 Grafana Dashboard

Presenting collected data is important to make them accessible and give an overview of events before starting in-depths analytics. While multiple approaches to visualize data exists, this section describes the setup of Grafana.

The advantages of Grafana compared to other solutions is the fact that without any coding skills appealing graphs and gauges can be created, offering a live and interactive user interface.

Within this setup the combination with the InfluxDB database is described, however it is possible to visualize other data sources via Grafana.

3.5.1 Installation

To allow a cheap and reproducible setup these steps are done on a RaspberryPi running Raspbian, which allows low cost data hosing without requiring any server setups. However it is equally possible to run this on a virtual machine or even a laptop for testing.

The Raspbian ports of Grafana are outdated and therefore the official Grafana repository is used instead:

```
echo "deb https://packages.grafana.com/oss/deb stable main" | \docs
    sudo tee -a /etc/apt/sources.list.d/grafana.list
wget -q -O - https://packages.grafana.com/gpg.key | sudo apt-key add -
sudo apt update
sudo apt install -y grafana-rpi
```

More details on the Grafana installation steps are available in the official documentation.

3.5.2 Adding InfluxDB as data source

This setup run both Grafana as well as InfluxDB on a single RaspberryPi. Therefore it's possible to access the data locally. Other setups may use remote database on distributed systems, where data is stored on different machines than visualized.

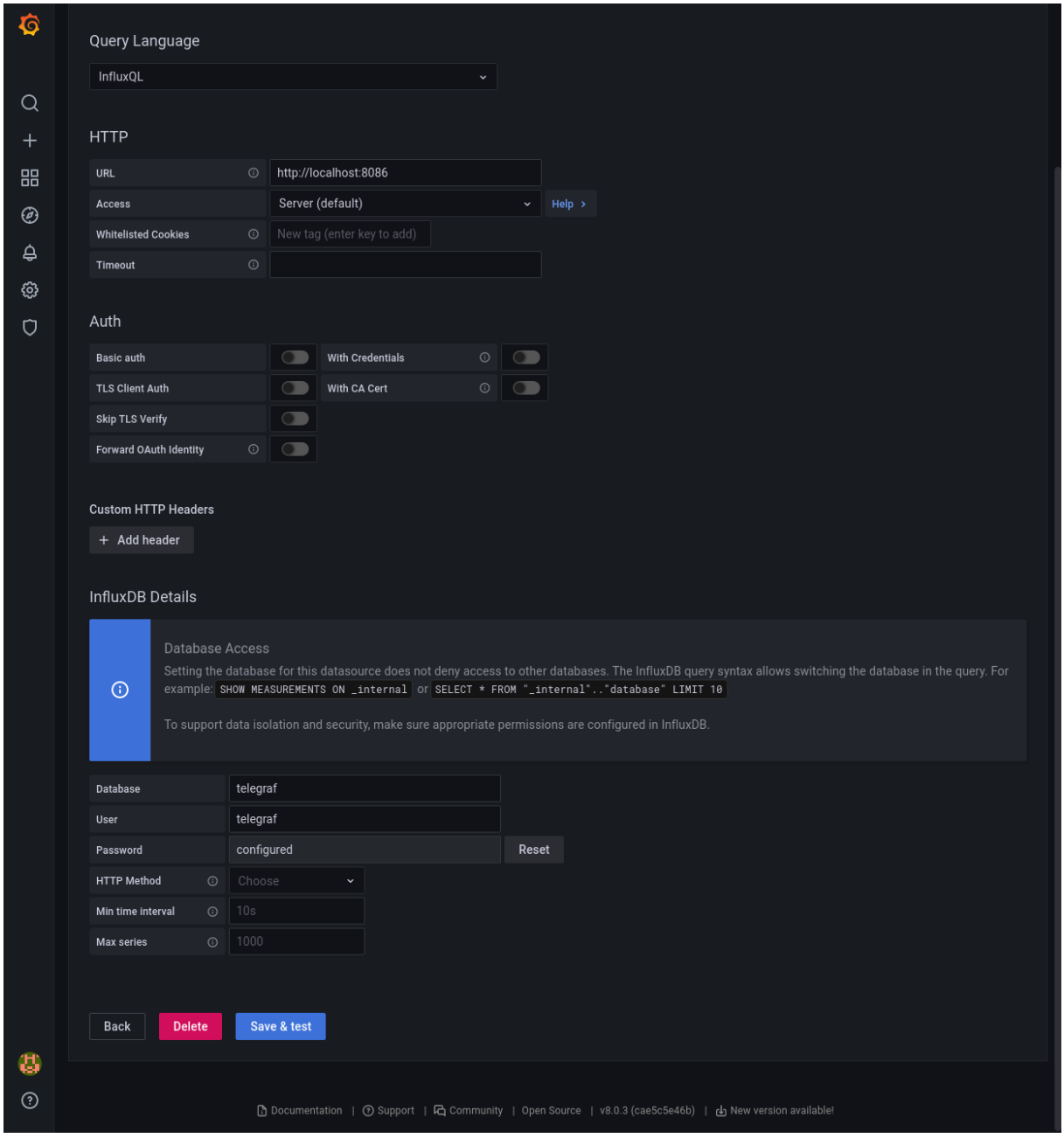

Figure 15: Grafana add datasource

InfluxDB runs per default on port `8086` and if running, is available on `localhost`. None of the extra options

need to be activated since a local setup doesn't require extra authentication settings except the login credentials set in the InfluxDB database setup.

3.5.3 Configure a graphs

A generic manual to add graphs or *panels* is provided by the Grafana Project itself within their documentation.

This example describes how to get the battery voltage of all available nodes. A running InfluxDB receives data via *Telegraf*, feeding a MQTT stream into the time series database. If using the same options as in the InfluxDB section, Grafana will present a `mqtt_consumer` table from which data can be selected.

Since the CayenneLPP protocol is used to encode senor data for LoRaWAN transmission, all fields in the table follow a specific schema:

```
1 uplink_message_decoded_payload_<cayennelpp_type>_<cayennelpp_id>
```

All metrics start with `uplink_message_decoded_payload` since that contains transmissions from the node. Other fields may contain IDs of used gateways or transmission quality (`uplink_message_rx_metadata_0_rssi`), however these are more relevant to monitor the nodes *health*.

The field of our interest is voltage with the ID `1`, the field is therefore called as follows:

```
1 uplink_message_decoded_payload_voltage_1
```

Once the field is selected measurements should be *grouped by* the device IDs. Grouping by means that a graph is created per device instead of showing all data in a single graph. The *tag* for grouping is called `end_device_ids_device_id`.

Lastly to improve readability it is possible to assign an *alias* to each graph. Instead of having the lengthy full description of a measurement (see below) only the device identifier can be shown. To do so the special value `$tag_end_device_ids_device_id` is added to the *alias* field.

```
1 mqtt_consumer.uplink_message_decoded_payload_voltage_1 {
      end_device_ids_device_id: rain-box-6}
```

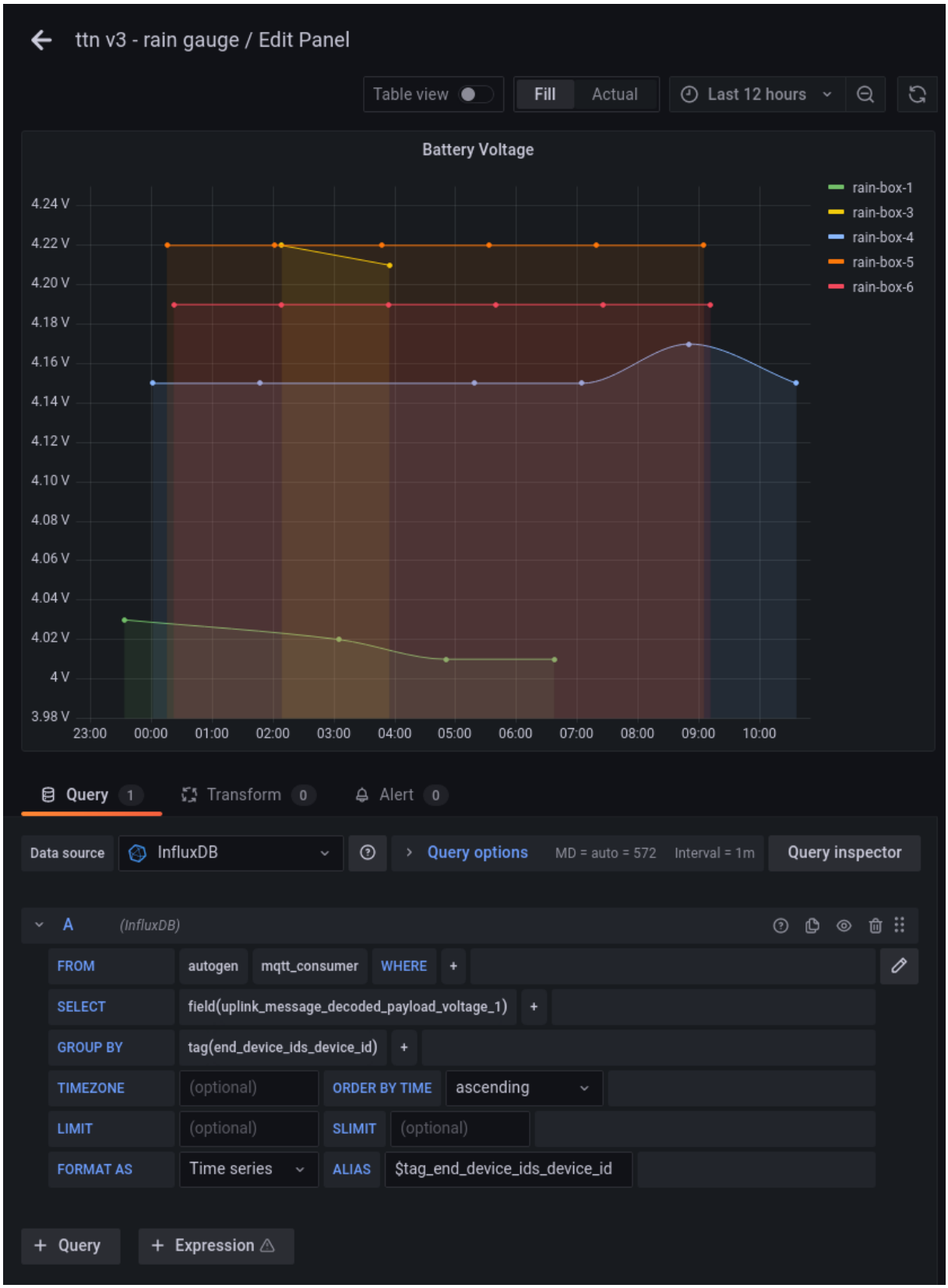

Figure 16: Grafana modifying graph battery voltage

3.5.4 Example Dashboard

Below is a screenshot of a demo setup which includes two graphs of measured metrics (rain fall and temperature) and below an overview of the nodes over the last week. It is possible to interactively select smaller and bigger time frames and see precise values for specific times and dates.

Figure 17: Grafana graph showing rain and temperature

Aboves two graphs show rain events that happened at different location in Honolulu, Hawaii. It's possible to zoom into specific time ranges to see the total amount of rain for that day, hour or minute.

The temperature graph below the *rain fall* shows a repeating pattern of temperature changes. Two special cases are handled in this graph showing where `rain-box-1` shows the *inner enclosing temperature* and `rain-box-6` is stationed in a cooled lab, therefore the constantly low temperature.

Additional values can be measured to track the *health* of sensors, most importantly the battery voltages and connection quality (*RSSI*).

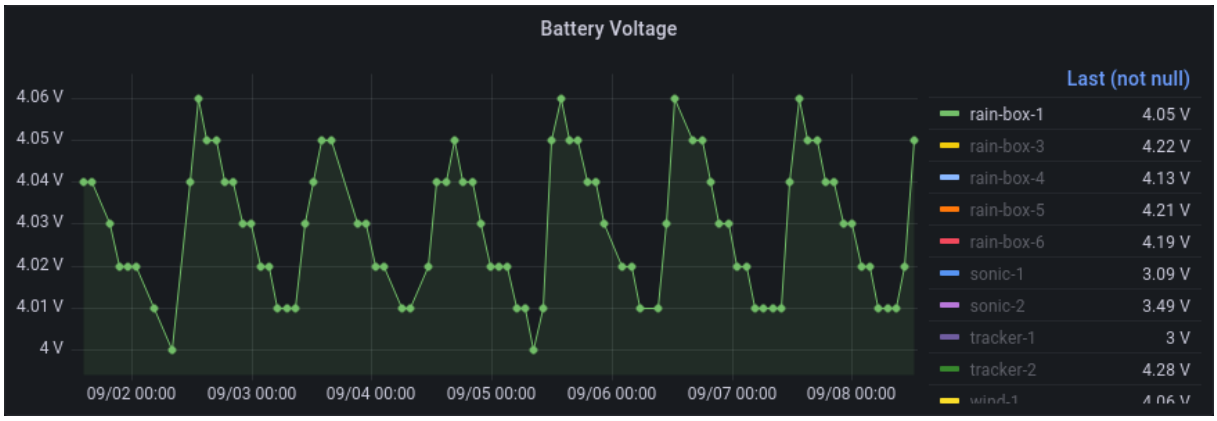

Figure 18: Grafana graph showing battery voltage

The graph above shows how the battery voltage is dropping every night but recharged on sunrise.

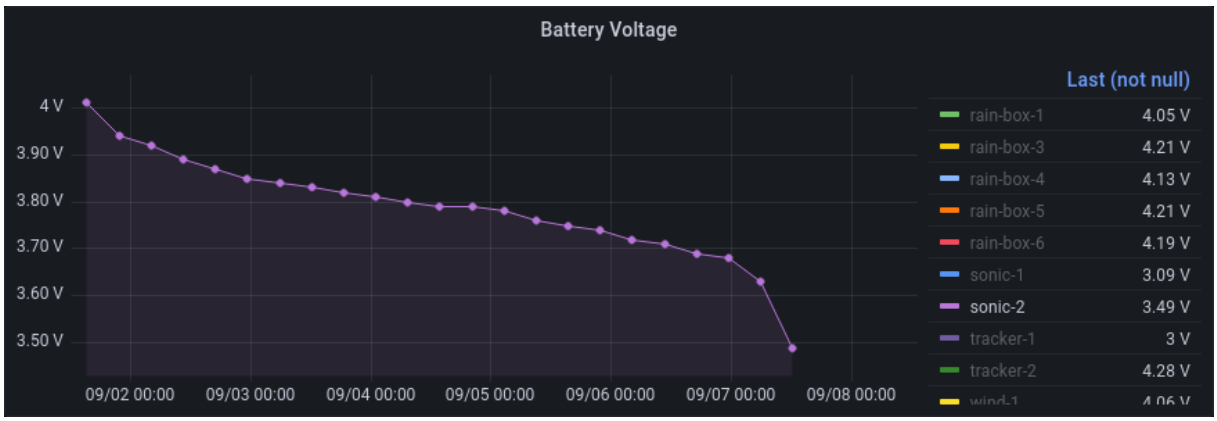

Figure 19: Grafana graph showing low battery voltage

On the contrary, above graphs show that the node `sonic-2` doesn't recharge via it's attached solar panel and therefore shuts off once the battery voltage is to low for further LoRa transmissions. The node need manual inspection and probably a solar panel replacement.

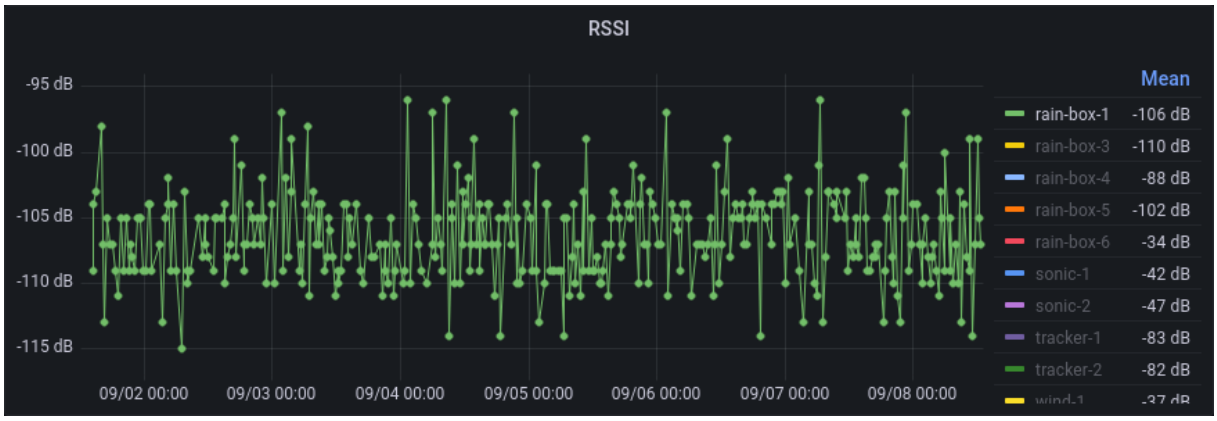

Figure 20: Grafana graph showing RSSI values

The graph above shows shows the varying values of RSSI, which describes the connection quality. For LoRa values down to **-120dB** are fine for a stable connection, so the node `rain-box-1` should have a stable connection.

4 Resources

4.1 CayenneLPP

This section explain CayenneLPP (Low Power Protocol) which is used to encode data on sensor nodes into a size efficient format and decode it in the backend back for further processing.

Cayenne is a platform developed by *myDevices Inc.* offering custom dashboards for device monitoring, alerts and more. They developed CayenneLPP and multiple open source implementation (C, Python of that protocol exists. The protocol allows a compromise between self describing data packets and size efficiency, which will be explained in the next section.

This sections below show three ways to encode node data to a binary format for sending over LoRa, a custom *raw* format, *JSON* and CayenneLPP. The example data is a **temperature** value of 18.6 degree Celsius, a **battery voltage** of 3.9V and a **distance** value of 43mm.

4.1.1 Encoding using raw Bits

The smaller a LoRaWAN packet is the less power is required to send it and the more airtime is available for other sensors. A simple solution would be just store all sensor data as binary fields one after another. The packet could have the following format:

```
1  <temperature><voltage><distance>
```

Representing floats in binary isn't trivial to instead a prevision can be defined. For **temperature** two float digits are enough, for **battery voltage** a single float digit is fine and **dinstance** is always a integer.

This means values can be multiple by `100` respectively `10`. The temperature value becomes `1860` and the voltage becomes `39`. Both considering that measured temperature will never be above 99.99 degree Celsius and batter voltage should exceed 9.9V (4.2V really). For distance a maximum of 5000mm should be used.

To calculate the maximum bits needed data, the maximum values are converted to binary format the bit counted, easily possible with some Python:

```
>>> format(9999, "b") # max value
'10011100001111'
>>> len(format(9999, "b"))
14
>>> format(99, "b") # max value
'1100011'
>>> len(format(99, "b"))
7
>>> format(5000, "b") # max value
'1001110001000'
>>> len(format(5000, "b"))
13
>>>
```

Above calculation shows that **14** Bits are needed for temperature, **7** for battery voltage and **13** for distance. Now the custom data format becomes more specific as shown below:

```
| temperature 14 | volt 7  | distance 13   |
| xxxxxxxxxxxxxx | xxxxxxx | xxxxxxxxxxxxx |
```

Below we calculate the specific values and pad them with zeros.

```
>>> format(1860, "b")
'11101000100'
>>> format(39, "b")
'100111'
>>> format(43, "b")
'101011'
```

The resulting packet would look like this with a total size of **34 Bits**.

```
| temperature 14 | volt 7  | distance 13   |
| 00011101000100 | 0101011 | 0000000101011 |
```

The receiving backend could decode this data by reading the first 14 Bits as integer and dividing the value by 100 to calculate the temperature, followed by reading Bit 15 until 21 as integer and dividing it by 10 to calculate the battery voltage. The remaining 13 Bit can be directly read as distance.

While this approach works and data is efficiently exchanged between sensor node and backend, the calculation is not trivial to understand. More problematic is the extendability of this approach: Newly added sensors may use a different set of sensors and therefore require data fields. As the exchanged data is just Bits, it's not possible to know what the data contains. In other words, the format is extremely static and ideally the data would be more self explaining so a backend would know what value it is decoding.

4.1.2 Encoding using JSON

The JSON format is extremely popular in web application to exchange all kinds of data. It is human readable, supports all our data types directly (float and integer) and allows to verbosely describe data fields. The example data could be decoded as below:

```
{
    "temperature_1": 18.6,
    "voltage_1": 3.9,
    "distance_1": 43
}
```

As printed above, the format would take a total of 73 Bytes (584 Bits) which is nearly 20 times bigger than using the raw format. Even a slightly optimized version as shown below would still require 28 Bytes (224 Bits) meaning 7 times bigger.

```
{"t1":18.6,"v1":3.9,"d1":43}
```

4.1.3 Encoding using CayenneLPP

CayenneLPP uses a compromise of both approaches, the data is very compact while being self descriptive. To archive that common measurement types are available with a 1 Byte data type descriptor and a 1 Byte data type channel (or ID). Each data type, be it temperature, GPS or relative humidity is described by a number between 0 and 255, a official reference implementation is available in the Cayenne Docs. Since all data types are based on IPSO data types other implementations support additional values, like voltage, altitude and distance.

Adding those values on device is done by a creating a `CayenneLPP` frame and adding values. More details are available in `main.cpp` implementation, however below is an simplified example.

```
CayenneLPP lpp(LORAWAN_APP_DATA_MAX_SIZE);      // create frame
lpp.addVoltage(1, getBatteryVoltage());         // add voltage
lpp.addDistance(1, distance);                   // add distance in mm
lpp.addTemperature(1, temperature);             // add temperature in Celsius
appDataSize = lpp.getSize();                    // calculate size
memcpy(appData, lpp.getBuffer(), appDataSize); // copy Bytes to LoRaWAN packet
```

The same is possible to do via Python using the `pycayennelpp` package:

```
from cayennelpp import LppFrame

frame = LppFrame()
frame.add_temperature(0, 18.6)
frame.add_distance(1, 43)
frame.add_voltage(1, 3.9)
buffer = bytes(frame)
print(len(buffer))
```

The result are **14 Bytes** and thereby half the size of using JSON. For measurements which are not directly supported by the implementations, like *number of satelites* for GPS measurements, it is possible to use the commands `addDigitalInput` or `addDigitalOutput` (unsigned 32 Bit Integer) and `addAnalogInput` or `addAnalogOutput` (float with 3 decimals).

4.1.4 Decoding in Backend

The receiving backend allows to decode incoming data before offering it via MQTT. This is very useful so different applications listening to the MQTT stream can directly process the decoded payload rather than implementing that per client, allowing as well to upgrade the communication used for nodes without modifying clients.

TheThingsNetwork offers to automatically decode CayenneLPP frames using the *Payload Formatter* menu entry.

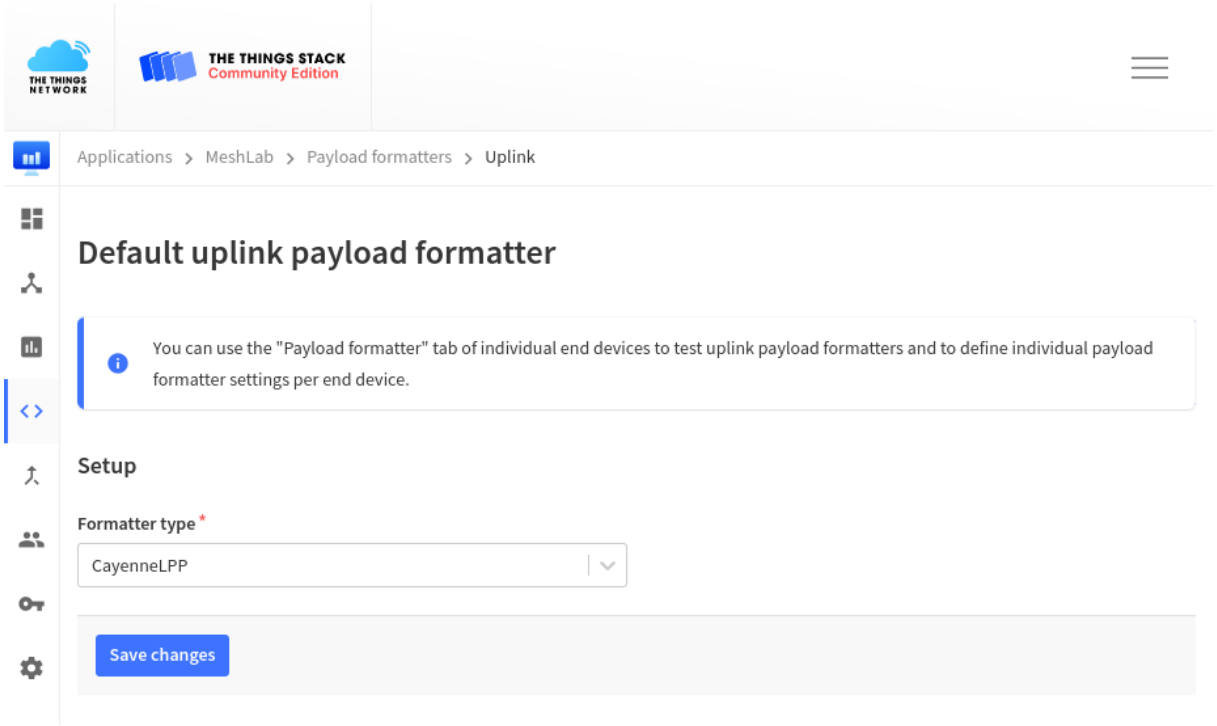

Figure 21: TheThingsNetwork CayenneLPP deocder

However, since TheThingsNetwork uses the reference implementation of *MyDevices* they don't support decoding some additional data types, like distance. If these measurements are used it is possible to use a custom JavaScript decoder as offered by *ElectronicCats* to be used directly in TheThingsNetwork. The `decoder.min.js` can be pasted into the text field of the *Payload Formatter* menu entry:

> The `min` decoder version is required as TheThingsNetwork limits the total size of decoding scripts and the formated version exceeds that limit.

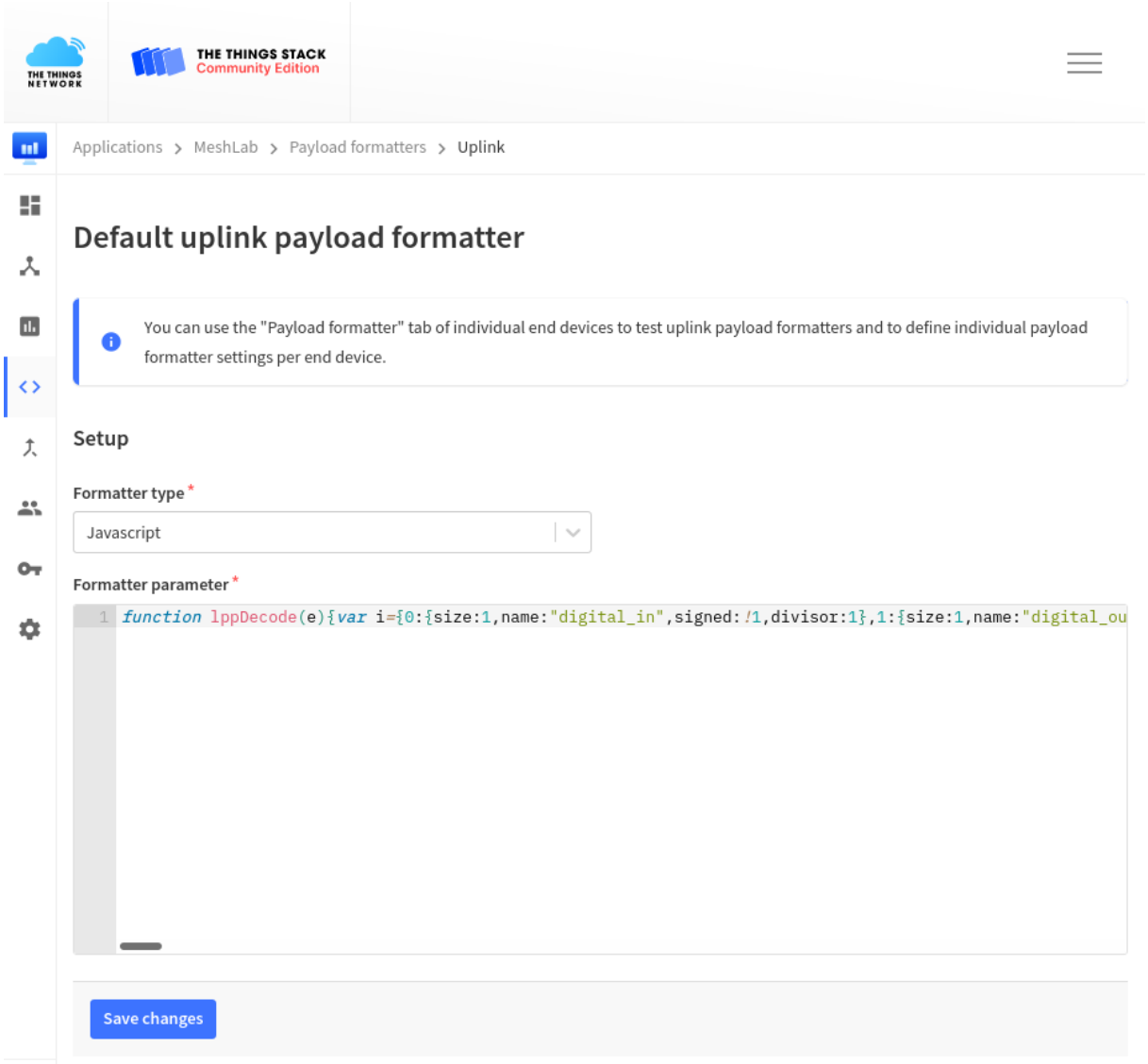

Figure 22: TheThingsNetwork set JavaScript decoder

4.1.5 Used payload types

Within this project the following payload types and IDs are used.

JSON value	description
`analog_in_1`	rain mm/m
`analog_in_2`	soil moisture
`digital_out_1`	running software version (integer)
`temperature_1`	temperature

JSON value	description
`voltage_1`	battery voltage
`distance_1`	distance to sea level

4.2 Sensors

This section describes briefly sensors used within this project. All of them are supported by the *Arduino* framework an can be connected in any variation to the described microcontrollers.

4.2.1 Distance

To measure the distance between a fixed point and sea level the ultrasonic **MB7389** sensor from *MaxBotix Inc.* was used. It offers a distance range of `300mm` to `5000mm` (`5m`). Measurements happen 6 times per second and are readable via an analog, pulse width or serial output. The easiest implementation is done by using Arduinos `softwareserial` and read the distance values as chars. A full data sheet is available on the vendors website.

4.2.2 Rain fall

For rain fall two different rain gauges are used. While one is the de-facto standard for scientific publications, the other allows measurements at a much lower price. Both are usable by using a simple tip counter via an `GPIO` interrupt and are simply connected to a `GND` PIN and whichever PIN was configured with the interrupt.

4.2.2.1 ONSET HOBO RG-3 The **HOBO Rain Gauge** by *Onset Computer Corporation* offers a high quality metal casing and is described as the default rain gauge for scientific publications. With it's precision comes the price of around $400 which may exceed the budget for some use cases.

> One tip means `0.254mm` or `0.1"` of rainfall.

4.2.2.2 Misol WH-SP-RG An alternative to is the **WH-SP-RG** by *Misol*. At the time of writing (2021-09-21) the vendors website is not reachable, however other websites sell the rain gauge for around $20.

> One tip means `0.3851mm` of rainfall.

4.2.3 Soil Moisture

For soil moisture the **VH400** by *Vegetronix, Inc* is supported. The sensor is connected to an `ADC` (*Analog Digital Converter*) PIN and the measured resistance describes the *volumetric water content* by using a piecewise curve provided by the vendor. The curve is implemented in the provided mini library. The sensor costs around $40.

4.2.4 Temperature

For waterproof temperature measuring a **DS18B20** sold by *Adafruit Industries, LLC* is used. Adafruit ships their own libraries so minimal integration needs to be coded. The sensor uses three wires which can be directly connected to the custom PCB. The sensor costs ardoung $10 on the Adafruit website.

5 Conclusion

This project started initially with the evaluation of pure LoRa frequency modulation using the Arduino library `arduino-LoRa`. While the tests worked fine encryption, authentication as well as link optimization needed manual solutions. Instead of pursuing the pure LoRa approach the switch to LoRaWAN solved all these and even allowed easily redundant base station setups plus roaming (e.g. for buoy setups).

Sensor nodes with the custom PCB board and low-cost *Heltec Dev Boards* were tested for multiple month in the wild and delivered stable measurements. Due to the modular code different combinations of sensors were easy to setup and new sensors added within hours. While writing code for the Davis Anemometer (wind direction and speed) is tested and may be fully supported in later versions of the code.

During the time working on this projects multiple researcher at the *University of Hawaii, Manoa* (UH) mentioned their interest in low-cost remote setups, using all kinds of sensors which all would be compatible with the Arduino framework. At the same time the Information Technology Services of *UH* collaborated on improving the coverage of *Oahu* by deploying multiple high-performance base stations around Honolulu.

While this project could be considered successful as in creating a *drop-in* replacement for an existing setup and being at the extendible by other, no *production* deployment is currently running. Due to the COVID-19 pandemic the in-person collaboration with other scientists was lower than hoped for, resulting in a lower variety of real setups around *Oahu*. Future work and efforts are needed to establish LoRaWAN as the *goto* solution for wireless metric transportation.

6 Outlook

Parts of the code should be refactored to allow PIN settings of sensors via the `provision_node.py` script rather than the `config.h` file. This would allow a single image for all sensors and thereby requiring only Python by the scientists deploying sensors (in contrast to the current need of Platformio.

Also the assembling of a LoRaWAN tracker should be fully documented. The tracker allows interval or manual tracking of LoRaWAN coverage and can thereby be used by scientists for evaluation of sensor node sites. Before a deployment the tracking device would be taken in-field to verify sufficient coverage of LoRaWAN.

Lastly a custom development board was designed to use the *CubeCell Module Plus* instead of the development boards, which contain unnecessary parts (like LEDs and buttons) which would not be used in production setups. The custom board is in early development and not yet fully functional.

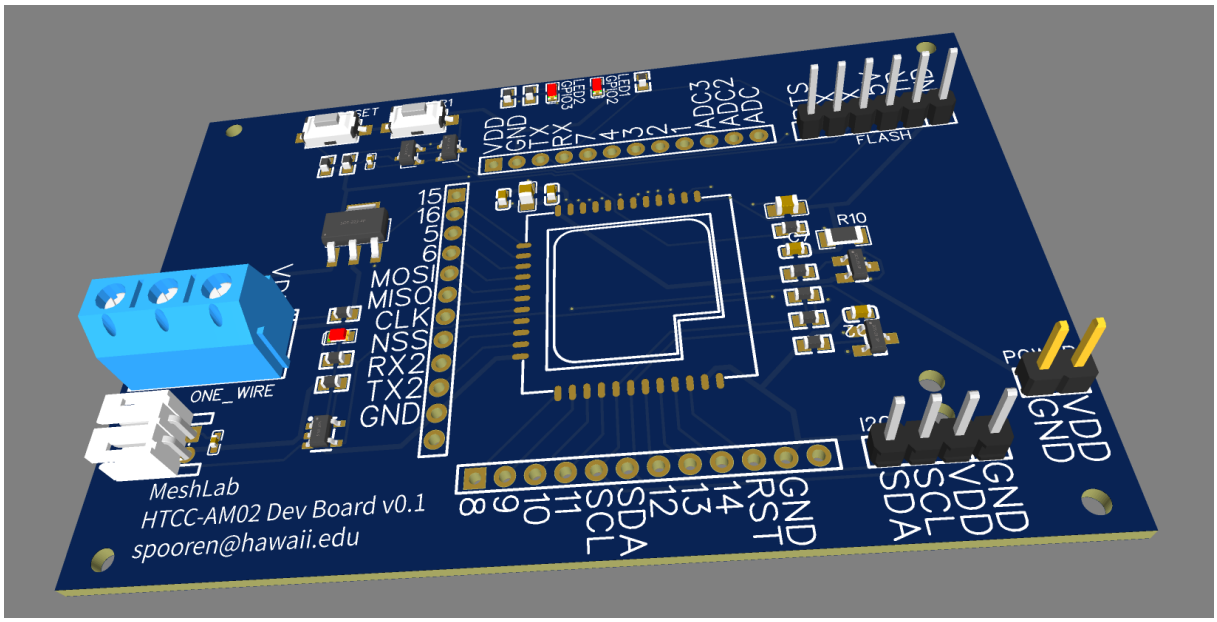

Figure 23: Custom PCB Module

7 Code

Code used within this project. It could be used as a base and inspiration for similar projects. The section consists of a *utility* section containing Python code for monitoring and deployment and *sensor node*, C code which runs on nodes.

7.1 Utility

Monitoring and deployment code written in Python.

7.1.1 mqtt2json.py

```
1  # Copyright Paul Spooren <mail@aparcar.org>
2  #
3  # SPDX-License-Identifier: GPL-2.0-or-later
4
5  import json
6  from os import getenv
7  from sys import stderr
8
9  import paho.mqtt.client as mqtt
10
11 broker = getenv("MQTT_URL", "nam1.cloud.thethings.network")
12 username = getenv("MQTT_USER", "meshlab@ttn")
13 token = getenv("MQTT_TOKEN")
```

```
14
15 assert token is not None, "Please set MQTT_TOKEN env variable"
16
17
18 def on_connect(mqttc, obj, flags, rc):
19     print("rc: " + str(rc), file=stderr)
20
21
22 def on_message(mqttc, obj, msg):
23     decoded_message = json.loads(msg.payload.decode("utf-8"))
24
25     # use this as node_id to have the unique dev_eui
26     #node_id = decoded_message["end_device_ids"]["dev_eui"]
27
28     # use this as node_id to have the human readable identifier
29     node_id = msg.topic.split("/")[-2]
30
31     if "decoded_payload" in decoded_message.get("uplink_message", {}):
32         print(
33             json.dumps(
34                 {
35                     "from": node_id,
36                     "p": decoded_message["uplink_message"]["decoded_payload"],
37                 }
38             )
39         )
40
41
42 def on_publish(mqttc, obj, mid):
43     print("mid: " + str(mid), file=stderr)
44
45
46 def on_subscribe(mqttc, obj, mid, granted_qos):
47     print("Subscribed: " + str(mid) + " " + str(granted_qos), file=stderr)
48
49
50 def on_log(mqttc, obj, level, string):
51     print(string)
52
53
54 mqttc = mqtt.Client()
55 mqttc.username_pw_set(username, token)
56 mqttc.on_message = on_message
57 mqttc.on_connect = on_connect
58 mqttc.on_publish = on_publish
59 mqttc.on_subscribe = on_subscribe
60 mqttc.tls_set_context()
61
62 # Uncomment to enable debug messages
63 # mqttc.on_log = on_log
64
65 mqttc.connect(broker, 8883, 60)
66 mqttc.subscribe("#", 0)
67
68 mqttc.loop_forever()
```

7.1.2 provision_node.py

```python
# Copyright Paul Spooren <mail@aparcar.org>
#
# SPDX-License-Identifier: GPL-2.0-or-later

import serial
import sys
import yaml
from pathlib import Path

print(f"{sys.argv[0]} [node_name] [serial_port] [bitrage]")
if len(sys.argv) == 2:
    node_name = sys.argv[1]
else:
    node_name = "develop"

if len(sys.argv) == 3:
    node_serial = sys.argv[2]
else:
    node_serial = "/dev/ttyUSB0"

if len(sys.argv) == 4:
    node_bitrate = int(sys.argv[3])
else:
    node_bitrate = 115200

print(f"Provisioning {node_name} at {node_serial}")

if not Path(node_serial).is_char_device():
    print(f"Missing serial connection for node at {node_serial}")
    quit(1)

config_path = Path(f"./nodes/{node_name}.yml")

if not config_path.is_file():
    print(f"Missing node config at nodes/{node_name}.yml")
    quit(1)

config = yaml.safe_load(config_path.read_text())


def AT_command(key, value):
    serial_connection.write(bytes("\n", "ascii"))
    serial_connection.readline()

    at_command = f"AT+{key}={value}"
    serial_connection.write(bytes(at_command, "ascii"))
    print(at_command)
    status = serial_connection.readline().decode("utf-8")
    print(f"STATUS: {status}")
    if not status.startswith("+OK"):
        print(status)
        sys.exit(1)

    return serial_connection.readline().decode("utf-8")
```

```

serial_connection = serial.Serial(node_serial, node_bitrate, timeout=1)

for key, value in config["at_commands"].items():
    if not value:
        value = "0" * 16
    AT_command(key, str(value))

# restart node
AT_command("RESET", 1)

serial_connection.close()
```

7.2 Sensor Node

Code running on sensor node written in C.

7.2.1 `main.cpp`

```
// Copyright Paul Spooren <mail@aparcar.org>
//
// SPDX-License-Identifier: GPL-2.0-or-later

#include "Arduino.h".
#include "LoRaWan_APP.h"
#include "MB7389.h"
#include "VH400.h"
#include "config.h"
#include <CayenneLPP.h>
#include <DallasTemperature.h>
#include <OneWire.h>
#include <SPI.h>
#include <Wire.h>
#include <softSerial.h>

#define LoraWan_RGB 0

// This section allows to store data on the device. Specifically this allows to
// store the rain gauge *mm per tip* value.
#define ROW 0
#define ROW_OFFSET 100
// CY_FLASH_SIZEOF_ROW is 256 , CY_SFLASH_USERBASE is 0x0ffff400
#define addr_rain_gauge
    \
  CY_SFLASH_USERBASE + CY_FLASH_SIZEOF_ROW *ROW + ROW_OFFSET

CayenneLPP lpp(LORAWAN_APP_DATA_MAX_SIZE);

// Set to BATTERY_SEND_INTERVAL to send on start
uint32_t battery_send_counter = BATTERY_SEND_INTERVAL;

// Run cycle every minute
```

```
33 uint32_t appTxDutyCycle = 60000;
34
35 // Send messages on specific port. For this application all data is transmitted
36 // on port 2. The port is used to distinguish between packet types, however
37 // since this implementation uses CayenneLPP which is parsed by the backend,
       all
38 // data can be sent over the same port.
39 uint8_t appPort = 2;
40
41 // Initialize variables required for rain gauge use
42 #ifdef RAIN_GAUGE_PIN
43 // union to store directly on EEPROM
44 union {
45   float number;
46   uint8_t bytes[4];
47 } mm_per_tip;
48
49 // The tip counter and mm, must reset after each cycle
50 static volatile uint8_t tips = 0;
51 static unsigned long last_switch = 0;
52 #endif
53
54 #ifdef DAVIS_SPEED_PIN
55 static uint32_t rotations = 0;
56 static unsigned long davis_speed_debounce = 0;
57 uint32_t davis_speed_send_counter = 0;
58 #endif
59
60 #ifdef DAVIS_DIRECTION_PIN
61 uint32_t davis_direction_send_counter = 0;
62 uint32_t davis_direction_total = 0;
63 #endif
64
65 // Initialize variables required for OneWire temperature
66 //
67 // The current setup only supports a single connected sensor while it is easily
68 // possible to monitor multiple OneWire sensors at the same time.
69 #ifdef ONEWIRE_PIN
70 OneWire onewire(ONEWIRE_PIN);
71 DallasTemperature onewire_sensor(&onewire);
72 uint32_t onewire_send_counter = 0;
73 float onewire_total = 0.0;
74 #endif
75
76 // Initialize variables required for VH400 moisture sensor
77 #ifdef VH400_PIN
78 uint32_t vh400_send_counter = 0;
79 float vh400_total = 0.0;
80 #endif
81
82 // Initialize variables for MB7389 ultrasonic distance sensor
83 #ifdef SONIC_RX_PIN
84 uint32_t sonic_send_counter = 0;
85 uint32_t sonic_total = 0;
86 softSerial sonicSerial(0, SONIC_RX_PIN);
87 #endif
88
```

```
89  // Prepare the outgoing packet stored in CayenneLPP format
90  //
91  // The packet might be empty and sending is omitted for that cycle.
92  static void prepareTxFrame() {
93    lpp.reset();
94
95    // Send battery status only every "BATTERY_SEND_INTERVAL" minutes
96    if (battery_send_counter >= BATTERY_SEND_INTERVAL) {
97      Serial.printf("[%i/%i] Add battery volt\n", battery_send_counter,
98                    BATTERY_SEND_INTERVAL);
99
100     // Battery voltage is stored as voltage with ID 1
101     lpp.addVoltage(1, getBatteryVoltage() / 1000.0);
102
103     // Running version is stored as digital output with ID 1
104     lpp.addDigitalOutput(1, VERSION);
105
106     // Reset counter
107     battery_send_counter = 0;
108   } else {
109     Serial.printf("[%i/%i] Skip battery volt\n", battery_send_counter,
110                   BATTERY_SEND_INTERVAL);
111     battery_send_counter++;
112   }
113
114 #ifdef SONIC_RX_PIN
115   sonic_total += get_sonic_distance(sonicSerial, 60);
116   sonic_send_counter++;
117   if (sonic_send_counter == SONIC_SEND_INTERVAL) {
118     Serial.printf("[%i/%i] Add sonic\n", sonic_send_counter,
119                   SONIC_SEND_INTERVAL);
120
121     // Sea level is stored as distance with ID 1
122     lpp.addDistance(1, ((float)sonic_total / sonic_send_counter) / 100.0);
123
124     // Reset counters
125     sonic_send_counter = 0;
126     sonic_total = 0;
127   } else {
128     Serial.printf("[%i/%i] Skip sonic\n", sonic_send_counter,
129                   SONIC_SEND_INTERVAL);
130   }
131 #endif
132
133 #ifdef RAIN_GAUGE_PIN
134   if (tips > 0) {
135     Serial.printf("Tips = %i\n", tips);
136
137     // Rain fall is stored as analog with ID 1
138     // The CayenneLPP specification do not support rain fall directly
139     lpp.addAnalogInput(1, mm_per_tip.number * tips);
140
141     // Reset counter
142     tips = 0;
143   } else {
144     Serial.printf("Skip rain gauge\n");
145   }
```

```
146 #endif
147
148 #ifdef DAVIS_SPEED_PIN
149   davis_speed_send_counter++;
150   if (davis_speed_send_counter >= DAVIS_SPEED_SEND_INTERVAL) {
151     Serial.printf("[%i/%i] Add Davis speed\n", davis_speed_send_counter,
152                   DAVIS_SPEED_SEND_INTERVAL);
153
154     // Moisture is stored as analog with ID 2
155     // The CayenneLPP specification do not support moisture directly
156     lpp.addFrequency(1, rotations / (60 * davis_speed_send_counter));
157
158     // Reset counters
159     davis_speed_send_counter = 0;
160     rotations = 0;
161   } else {
162     Serial.printf("[%i/%i] Skip Davis speed\n", davis_speed_send_counter,
163                   DAVIS_SPEED_SEND_INTERVAL);
164   }
165 #endif
166
167 #ifdef DAVIS_DIRECTION_PIN
168   davis_direction_total += analogReadmV(DAVIS_DIRECTION_PIN);
169   davis_direction_send_counter++;
170   if (davis_direction_send_counter >= DAVIS_DIRECTION_SEND_INTERVAL) {
171     Serial.printf("[%i/%i] Add Davis direction\n", davis_direction_send_counter
          ,
172                   DAVIS_DIRECTION_SEND_INTERVAL);
173
174     // Moisture is stored as analog with ID 2
175     // The CayenneLPP specification do not support moisture directly
176     int direction =
177         ((davis_direction_total / davis_direction_send_counter) / 2400) * 360;
178     lpp.addDirection(1, direction);
179
180     // Reset counters
181     davis_direction_send_counter = 0;
182     davis_direction_total = 0;
183   } else {
184     Serial.printf("[%i/%i] Skip davis_direction\n",
185                   davis_direction_send_counter, DAVIS_DIRECTION_SEND_INTERVAL);
186   }
187 #endif
188
189 #ifdef VH400_PIN
190   vh400_total += read_VH400(VH400_PIN);
191   vh400_send_counter++;
192   if (vh400_send_counter >= VH400_SEND_INTERVAL) {
193     Serial.printf("[%i/%i] Add VH400\n", vh400_send_counter,
194                   VH400_SEND_INTERVAL);
195
196     // Moisture is stored as analog with ID 2
197     // The CayenneLPP specification do not support moisture directly
198     lpp.addAnalogInput(2, vh400_total / vh400_send_counter);
199
200     // Reset counters
201     vh400_send_counter = 0;
```

```
202       vh400_total = 0.0;
203     } else {
204       Serial.printf("[%i/%i] Skip VH400\n", vh400_send_counter,
205                     VH400_SEND_INTERVAL);
206     }
207   #endif
208
209   #ifdef ONEWIRE_PIN
210     onewire_sensor.requestTemperatures();
211     onewire_total += onewire_sensor.getTempCByIndex(0);
212     onewire_send_counter++;
213     if (onewire_send_counter >= ONEWIRE_SEND_INTERVAL) {
214       Serial.printf("[%i/%i] Add OneWire\n", onewire_send_counter,
215                     ONEWIRE_SEND_INTERVAL);
216
217       // Temperature is stored as temperature with ID 1
218       lpp.addTemperature(1, onewire_total / onewire_send_counter);
219
220       // Reset counters
221       onewire_send_counter = 0;
222       onewire_total = 0.0;
223     } else {
224       Serial.printf("[%i/%i] Skip OneWire\n", onewire_send_counter,
225                     ONEWIRE_SEND_INTERVAL);
226     }
227   #endif
228
229     // Copy CayenneLPP frame to appData, which will be send
230     lpp.getBuffer(), appDataSize = lpp.getSize();
231     memcpy(appData, lpp.getBuffer(), appDataSize);
232   }
233
234   // Custom commands to provision device
235   //
236   // At this point only the rain gauge command `mmPerTip` is supported, however
        it
237   // is easily possible to extend this to e.g. support height setting of the tide
238   // gauges or other user specific commands.
239   bool checkUserAt(char *cmd, char *content) {
240   #ifdef RAIN_GAUGE_PIN
241     // This command is used to set the tipping bucket size
242     // - HOBO RG3: 0.254mm/t
243     // - Misol WH-SP-RG: 0.3851mm/t
244     if (strcmp(cmd, "mmPerTip") == 0) {
245       if (atof(content) != mm_per_tip.number) {
246         mm_per_tip.number = atof(content);
247         Serial.print("+OK Set mm per tip to ");
248         Serial.print(mm_per_tip.number, 4);
249         Serial.println("mm");
250         Serial.println();
251         FLASH_update(addr_rain_gauge, mm_per_tip.bytes, sizeof(mm_per_tip.bytes))
              ;
252       } else {
253         Serial.println("+OK Same mm per tip as before");
254       }
255       return true;
256     }
```

```
257 #endif
258   return false;
259 }
260 
261 #ifdef RAIN_GAUGE_PIN
262 // Run on each interrupt aka tip of the tipping bucket
263 void handler_rain_gauge() {
264   if ((millis() - last_switch) > 15) {
265     tips++;
266     last_switch = millis();
267   }
268 }
269 #endif
270 
271 #ifdef DAVIS_SPEED_PIN
272 // Run on each interrupt aka tip of the tipping bucket
273 void handler_davis_speed() {
274   if ((millis() - davis_speed_debounce) > 15) {
275     rotations++;
276     davis_speed_debounce = millis();
277   }
278 }
279 #endif
280 
281 int get_direction(int davis_voltage) { return (davis_voltage / 2400) * 360; }
282 
283 // The Arduino setup() function run on every boot
284 void setup() {
285   // Set baudrate to 115200
286   Serial.begin(115200);
287   Serial.println("So it begins...");
288 
289 #ifdef RAIN_GAUGE_PIN
290   // Read mm/t from EEPROM
291   Serial.println("Enable rain gauge sensor");
292   FLASH_read_at(addr_rain_gauge, mm_per_tip.bytes, sizeof(mm_per_tip.bytes));
293   if (mm_per_tip.number == 0.0) {
294     mm_per_tip.number = DEFAULT_MM_PER_COUNT;
295   }
296   Serial.print("Current mm per tip is ");
297   Serial.print(mm_per_tip.number, 4);
298   Serial.println("mm");
299 
300   // Enable interrupt pin for tips
301   PINMODE_INPUT_PULLUP(RAIN_GAUGE_PIN);
302   attachInterrupt(RAIN_GAUGE_PIN, handler_rain_gauge, FALLING);
303 #endif
304 
305 #ifdef DAVIS_SPEED_PIN
306   // Read mm/t from EEPROM
307   Serial.println("Enable Davis speed sensor");
308 
309   // Enable interrupt pin for tips
310   PINMODE_INPUT_PULLUP(DAVIS_SPEED_PIN);
311   attachInterrupt(DAVIS_SPEED_PIN, handler_davis_speed, FALLING);
312 #endif
313 
```

```
#ifdef ONEWIRE_PIN
  // Setup OneWire and print single test measurement
  Serial.println("Enable OneWire sensor");
  onewire_sensor.begin();
  onewire_sensor.requestTemperatures();
  Serial.print("temperature = ");
  Serial.print(onewire_sensor.getTempCByIndex(0), 2);
  Serial.println();
#endif

  // Enable user input via Serial for AT commands
  enableAt();

#ifdef DAVIS_DIRECTION_PIN
  // Setup Davis direction sensor and print single test measurement
  Serial.println("Enable Davis direction sensor");
  pinMode(DAVIS_DIRECTION_PIN, INPUT);
  Serial.print("direction = ");
  Serial.print(get_direction(analogReadmV(DAVIS_DIRECTION_PIN)));
  Serial.println();
#endif

#ifdef VH400_PIN
  // Setup VH400 sensor and print single test measurement
  Serial.println("Enable VH400 sensor");
  pinMode(VH400_PIN, INPUT);
  Serial.print("moisture = ");
  Serial.print(analogReadmV(VH400_PIN));
  // Serial.print(read_VH400(VH400_PIN), 2);
  Serial.println();
#endif

#ifdef DAVIS_SPEED_PIN

#endif

#ifdef DAVIS_DIRECTION_PIN

#endif

#ifdef SONIC_RX_PIN
  // Print single test measurement
  Serial.println("Enable sonic sensor");
  Serial.print("distance = ");
  Serial.print(get_sonic_distance(sonicSerial, 1));
  Serial.println();
#endif

  // Set Arduino into the initialization state
  deviceState = DEVICE_STATE_INIT;

  LoRaWAN.ifskipjoin();
}

void loop() {
  switch (deviceState) {
  case DEVICE_STATE_INIT: {
```

```
371       getDevParam();
372       printDevParam();
373       LoRaWAN.init(loraWanClass, loraWanRegion);
374       deviceState = DEVICE_STATE_JOIN;
375       break;
376     }
377     case DEVICE_STATE_JOIN: {
378       LoRaWAN.join();
379       break;
380     }
381     case DEVICE_STATE_SEND: {
382       prepareTxFrame();
383
384       // Do not send empty packages
385       if (appDataSize > 0) {
386         LoRaWAN.send();
387       } else {
388         Serial.println("Package is empty, don't send anything");
389       }
390       deviceState = DEVICE_STATE_CYCLE;
391       break;
392     }
393     case DEVICE_STATE_CYCLE: {
394   // If sonic sensor is enabled it will measure 60 times a minute and return the
395   // average. This way waves don't influence tide estimations as much. Since a
396   // measurement is required every second the duty cycle is skipped and instead
397   // the get_sonic_distance function runs, which blocks for 60 seconds.
398   // If no sonic sensor is attached, use the
399   #ifdef SONIC_RX_PIN
400       LoRaWAN.sleep();
401       // Add random delay so sensors don't repetitively send at the same time.
402       // This is done do avoid a situation where many sensors start at the same
403       // time and collectively overload the gateway every 60 seconds.
404       delay(randr(0, APP_TX_DUTYCYCLE_RND));
405       deviceState = DEVICE_STATE_SEND;
406   #else
407       // Same random delay is applied here.
408       txDutyCycleTime = appTxDutyCycle + randr(0, APP_TX_DUTYCYCLE_RND);
409       LoRaWAN.cycle(txDutyCycleTime);
410       deviceState = DEVICE_STATE_SLEEP;
411   #endif
412       break;
413     }
414     case DEVICE_STATE_SLEEP: {
415       LoRaWAN.sleep();
416       break;
417     }
418     default: {
419       deviceState = DEVICE_STATE_INIT;
420       break;
421     }
422     }
423   }
```

7.2.2 config.h

```
/*
 * Describes the currently running version. This value is transmitted next to
 * the battery voltage and helps to keep track what software version is running
 * on nodes. It should be increased whenever a significant change to logic
 * happens which may change the behaviour of a running device.
 */
#define VERSION 4

// How often to send battery voltage (and version) in minutes
#define BATTERY_SEND_INTERVAL 360 // 6 hours
// How often to send OneWire temperature in minutes
#define ONEWIRE_SEND_INTERVAL 30
// How often to send MB7389 sonic distance in minutes
#define SONIC_SEND_INTERVAL 10
// How often to send VH400 moisture in minutes
#define VH400_SEND_INTERVAL 3
// How often to send Davis wind speed in minutes
#define DAVIS_SPEED_SEND_INTERVAL 2
// How often to send Davis wind direction in minutes
#define DAVIS_DIRECTION_SEND_INTERVAL 2

/*
 * Setting PINs to devices will enable them within the code. The default values
 * presented here work fine with Heltec CubeCell boards, allowing to attach all
 * sensor at once. However it is also possible to remove any PIN definition and
 * thereby disable the sensor completely.
 */
// #define ONEWIRE_PIN GPIO4
//#define SONIC_RX_PIN GPIO1
//#define RAIN_GAUGE_PIN GPIO5
//#define VH400_PIN ADC2
#define DAVIS_SPEED_PIN GPIO5
#define DAVIS_DIRECTION_PIN ADC2

// This defines the fallback value of mm/t if not provided by the node
// configuration file. The default value corresponds to a HOBO RG3.
#define DEFAULT_MM_PER_COUNT 0.254 // 0.01"

/*
 * All options below are advanced and specifically for the LoRaWAN library used
    .
 * None of these values should require manual changes except when using outside
 * the US915 zone or using ABP rather than OTAA.
 */

// OTAA parameters should be set via AT commands in the configuration file.
uint8_t devEui[] = {0xC0, 0xFF, 0xEE, 0xC0, 0xFF, 0xEE, 0xCA, 0xFE};
uint8_t appEui[] = {0xC0, 0xFF, 0xEE, 0xC0, 0xFF, 0xEE, 0xCA, 0xFE};
uint8_t appKey[] = {0xC0, 0xFF, 0xEE, 0xC0, 0xFF, 0xEE, 0xC0, 0xFF,
                    0xEE, 0xC0, 0xFF, 0xEE, 0xC0, 0xFF, 0xEE, 0x42};

/*
 * While it is possible to use ABP it is not recommended, please use OTAA!
 * OTAA is more secure and allows the node and backend to negotiate an ideal
```

```
 * transmission rate to safe power and air-time.
 */
uint8_t nwkSKey[] = {};
uint8_t appSKey[] = {};
uint32_t devAddr = (uint32_t)0x0;

/*LoraWan channelsmask, default channels 0-7 sub 2*/
uint16_t userChannelsMask[6] = {0xFF00, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
    };

/*LoraWan region, select in arduino IDE tools*/
LoRaMacRegion_t loraWanRegion = ACTIVE_REGION;

/*LoraWan Class, Class A and Class C are supported*/
DeviceClass_t loraWanClass = LORAWAN_CLASS;

/*OTAA or ABP*/
bool overTheAirActivation = LORAWAN_NETMODE;

/*ADR enable*/
bool loraWanAdr = LORAWAN_ADR;

/* set LORAWAN_Net_Reserve ON, the node could save the network info to flash,
 * when node reset not need to join again */
// bool keepNet = LORAWAN_NET_RESERVE;
bool keepNet = false;

/* Indicates if the node is sending confirmed or unconfirmed messages */
bool isTxConfirmed = false;

/*!
 * Number of trials to transmit the frame, if the LoRaMAC layer did not
 * receive an acknowledgment. The MAC performs a datarate adaptation,
 * according to the LoRaWAN Specification V1.0.2, chapter 18.4, according
 * to the following table:
 *
 * Transmission nb | Data Rate
 * ----------------|-----------
 * 1 (first)       | DR
 * 2               | DR
 * 3               | max(DR-1,0)
 * 4               | max(DR-1,0)
 * 5               | max(DR-2,0)
 * 6               | max(DR-2,0)
 * 7               | max(DR-3,0)
 * 8               | max(DR-3,0)
 *
 * Note, that if NbTrials is set to 1 or 2, the MAC will not decrease
 * the datarate, in case the LoRaMAC layer did not receive an acknowledgment
 */
uint8_t confirmedNbTrials = 4;
```

7.2.3 VH400.cpp

```
#include "Arduino.h"
```

```
2
3  float read_VH400(uint8_t VH400_PIN) {
4
5    float vh400_voltage = analogReadmV(VH400_PIN) / 1000.0;
6    float VWC;
7
8    // Calculate VWC based on voltages provided by vendor:
9    // https://vegetronix.com/Products/VH400/VH400-Piecewise-Curve.phtml
10   if (vh400_voltage <= 1.1) {
11     VWC = 10 * vh400_voltage - 1;
12   } else if (vh400_voltage <= 1.3) {
13     VWC = 25 * vh400_voltage - 17.5;
14   } else if (vh400_voltage <= 1.82) {
15     VWC = 48.08 * vh400_voltage - 47.5;
16   } else if (vh400_voltage <= 2.2) {
17     VWC = 26.32 * vh400_voltage - 7.89;
18   } else if (vh400_voltage <= 3.0) {
19     VWC = 62.5 * vh400_voltage - 87.5;
20   } else {
21     VWC = 0.0;
22   }
23   return (VWC);
24 }
```

7.2.4 MB7389.cpp

```
1  #include <softSerial.h>
2
3  uint32_t get_sonic_distance(softSerial softwareSerial, uint32_t count) {
4    boolean foundValue;
5    boolean foundStart;
6    String inString;
7    uint32_t measurements = 0;
8    uint32_t total = 0;
9    uint32_t start;
10
11   softwareSerial.begin(9600);
12   while (!softwareSerial.available()) {
13   }
14
15   // Run loop until enough measurements were read
16   while (measurements < count) {
17     // Since measurements may fail resulting in a retry after 1/6 seconds,
         store
18     // the start time so one a measurement is successfully read, start the next
19     // measurement exactly a second later.
20     start = millis();
21
22     // Reset variables used to find measurement
23     foundStart = false;
24     foundValue = false;
25     inString = "";
26
27     // Flush software serial input
28     softwareSerial.flush();
```

```
29
30      // Loop until a single measurement is found
31      while (!foundValue) {
32        if (softwareSerial.available()) {
33          char inChar = softwareSerial.read();
34          if (foundStart == false) {
35            // New measurements begin with the character `R`
36            if (inChar == 'R') {
37              foundStart = true;
38            } else {
39              // If not, skip the letter and repeat
40              softwareSerial.read();
41            }
42          } else {
43            // Measurements end with character `\r`. If any other value is found
44            // add it to our measurement.
45            if (inChar != '\r') {
46              inString += inChar;
47            } else {
48              // If found character is `\r` and therefore end of a measurement,
49              // check if the length is actually four character. If not, repeat
50              // the process.
51              if (inString.length() != 4) {
52                Serial.println("incomplete measurement");
53                foundStart = false;
54                inString = "";
55              } else {
56                // Found available measurement, increase counter and begin again
57                foundValue = true;
58                total += inString.toInt();
59                measurements += 1;
60                // Wait 1 second minus the time it took to read the last
61                // measurement
62                delay(1000 - (millis() - start));
63              }
64            }
65          }
66        } else {
67          // Give the software serial more time to read characters
68          delay(50);
69        }
70      }
71    }
72    // Return average of measurements and convert it to cm (sensor returns mm)
73    return (total / (count * 10)); // to cm
74  }
```

8 References

[1] Khutsoane, Oratile, Bassey Isong, and Adnan M. Abu-Mahfouz. “IoT devices and applications based on LoRa/LoRaWAN.” IECON 2017-43rd Annual Conference of the IEEE Industrial Electronics Society. IEEE, 2017.

[2] Rizzi, Mattia, et al. “Evaluation of the IoT LoRaWAN solution for distributed measurement applications.” IEEE

Transactions on Instrumentation and Measurement 66.12 (2017): 3340-3349.

[3] Davcev, Danco, et al. “IoT agriculture system based on LoRaWAN.” 2018 14th IEEE International Workshop on Factory Communication Systems (WFCS). IEEE, 2018.