

GETTING STARTED WITH MIOTY™

Tutorial for setting up and operating a
MIOTY™ Network

GETTING STARTED WITH MIOTY™

Tutorial for setting up and operating a MIOTY™ Network

Fraunhofer Institute for Integrated Circuits IIS, Erlangen

Johanna Gros

© Fraunhofer IIS
Erlangen, October 2022

All images: © Fraunhofer IIS

Contents

Contents	3
1 General Information	5
1.1 Symbols within the document.....	5
1.2 Components and supplies.....	5
2 Overview of the MIOTY™ System	8
2.1 Introduction to MIOTY™.....	8
2.1.1 What makes MIOTY™ so unique?	8
2.1.2 Application of MIOTY™	9
2.2 How do the MIOTY™ System and its underlying technology work? ..	9
2.2.1 Data Chain and Key Components	10
2.2.2 Telegram splitting as unique technology and advantageous feature of MIOTY™.....	11
3 End Point setup: Programming and commissioning of the sensor nodes	13
3.1 LZE MIOTY™ M3B magnoling MAKERBOARD.....	13
3.1.1 Getting Started: Preparatory steps for programming and flashing the board	13
3.1.2 Writing the program	15
3.1.2.1 Writing Sketches with the Arduino IDE.....	15
3.1.2.2 The Program code	15
3.1.3 Flashing and placement of the board	19
3.2 Arduino Pro Mini equipped with RFM69W transmitter module and Bme weather data sensor.....	20
3.2.1 Getting Started: Hardware setup.....	21
3.2.2 Preparations for programming the board	21
3.2.1 The Program Code	22
3.2.2 Flashing and placement of the board	25
4 Base Station setup and Connecting Endpoints: AVA Gateway start-up...	28
4.1 Context with the data transmission path.....	28
4.2 Base station setup and installation	29
4.3 Register new Endpoints.....	29
4.4 Data reception	31
4.5 Troubleshooting: Control of the received data	32
5 Correct representation of data on the Base station with the help of blueprints	34
5.1 Introduction to Blueprints and the Json data format.....	34
5.1.1 Why do we use Blueprints?	34
5.1.2 What is a Blueprint?	34
5.1.3 What is a TypeEUI and what is it needed for?.....	35
5.1.4 Summary Example Blueprint.....	35
5.1.5 How does the Json Data Format work?	35
5.2 Creating a Blueprint	36
5.2.1 Architecture.....	37
5.2.1.1 Component Datatypes	39
5.2.1.1 Function Literals	39
5.2.2 Create your own blueprint: Example Blueprint from our project	40

5.3	Adding and Using Blueprints in the Base Station Application Center	41
5.3.1	Adding a Blueprint to the Base Station	42
5.3.2	Assigning a Blueprint to an End Node	43
5.3.3	Troubleshooting	44
6	Data transfer via MQTT and display on the IoT platform	46
6.1	Data transfer to ThingsBoard	46
6.2	The IoT Platform ThingsBoard	47
6.2.1	Using the Platform	47
6.2.2	Adding new devices	47
6.3	Introduction to MQTT	48
6.3.1	Why do we use MQTT?	48
6.3.2	What Is MQTT?	48
6.3.2.1	Publish/ Subscribe Architecture	49
6.3.2.2	Client/ Broker Model	49
6.3.2.3	Topics	49
6.3.2.4	Summary Example	50
6.3.2.5	MQTT and our MIOTY™ Project	51
6.4	Implementation of the MQTT Clients	51
6.4.1	... using the Paho Python Client	52
6.4.1.1	Preparatory Steps	52
6.4.1.2	The Program Code	53
6.4.1.3	Executing the Program	56
6.4.1.4	Troubleshooting	57
6.4.2	... using the Paho Python Client in connection with Eclipse Mosquitto	57
6.5	Create your own IoT-Dashboard: Data Visualization	58
7	Appendix	60
7.1	Blueprint for the MIOTY™ M3B magnoling MAKERBOARD	60
7.2	Blueprint for the Arduino Pro Mini sensor node	60
7.3	Code for the MIOTY™ M3B magnoling MAKERBOARD sensor node	61
7.4	Overview of the most relevant functions of the Mioty_at_client_c and m3b_helper library	63
7.5	Code for the Arduino Pro Mini sensor node	63
7.6	Overview of the most relevant functions of the ts_unb_node library	65
7.7	Code for the Paho-based MQTT Client to receive Data	66
7.8	Code for the Paho-based MQTT Client to receive and publish Data	67
7.9	Code for the Mosquitto-based MQTT Client to receive and publish Data	68
7.10	Overview of the most relevant Functions of the Paho Client Class	69

General Information

This document is intended for both commercially active system architects and all other (private) persons who wish to develop and use a MIOTY™ system.

It is intended to help in setting up and operating an own MIOTY™ based network. Illustrated and detailed instructions based on an exemplary setup give an overview of which hardware as well as software components are necessary and how they can be successfully configured, programmed and interconnected. Additional explanations and definitions provide further background information. Furthermore, additional video instructions are available for some practical set-up steps.

The "MIOTY™ project" described in this tutorial uses the data of a weather sensor as an example to show the system structure. The manual covers the following topics:

- Overview of the MIOTY™ System: Basic structure and technical background of MIOTY™
- End Point Setup: Configuration, programming and commissioning of the nodes
- Base Station Setup: Configuration and commissioning of the Base Station
- Creation and use of Blueprints
- Data transfer via MQTT and display on the IoT platform

The general structure of each MIOTY™ network is essentially the same, however, the explicit setup presented in the "MIOTY™ Project" can also be transferred to other sensors/components and use cases. The exact components needed for this tutorial are listed in section [1.2](#).

1.1

Symbols within the document

Text sections and chapters are labeled according to their content and the meaning to the setup of the project:



Setup step



Theoretical aspects relevant for the setup



Additional information and background knowledge



Warning/ Caution/ Attention



additional setup video available

1.2

Components and supplies

The following overview lists all required hardware components of the MIOTY™ project (either the variant with the Arduino Pro Mini or the M3B magnoling Makerboard should be chosen; see sections [3.1](#) and [3.2](#) for more information).

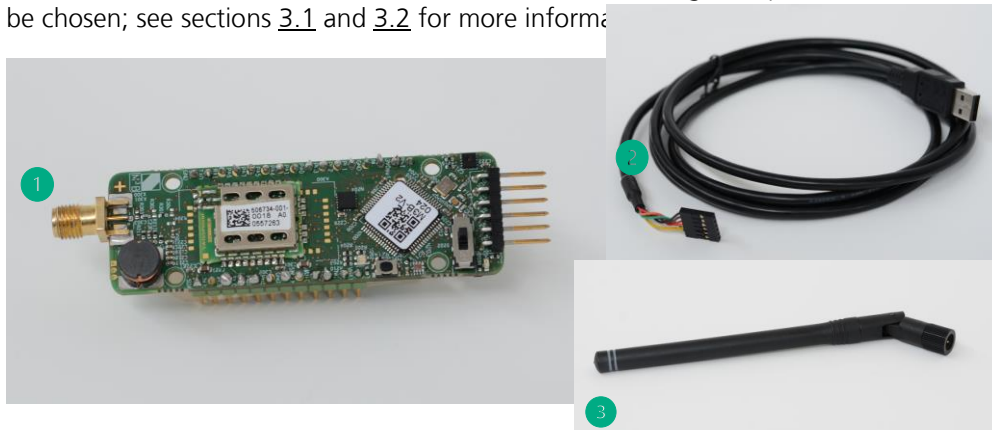


Figure 1: Components for the M3B magnoling Makerboard-Node-Variant

- ① [Mioty™ M3B magnoling Makerboard](#) (left with antenna socket and right with TTL interface)
- ② USB 2.0 to Serial TTL Converter
- ③ SMA Antenna

Arduino Pro Mini-Node-Variant:

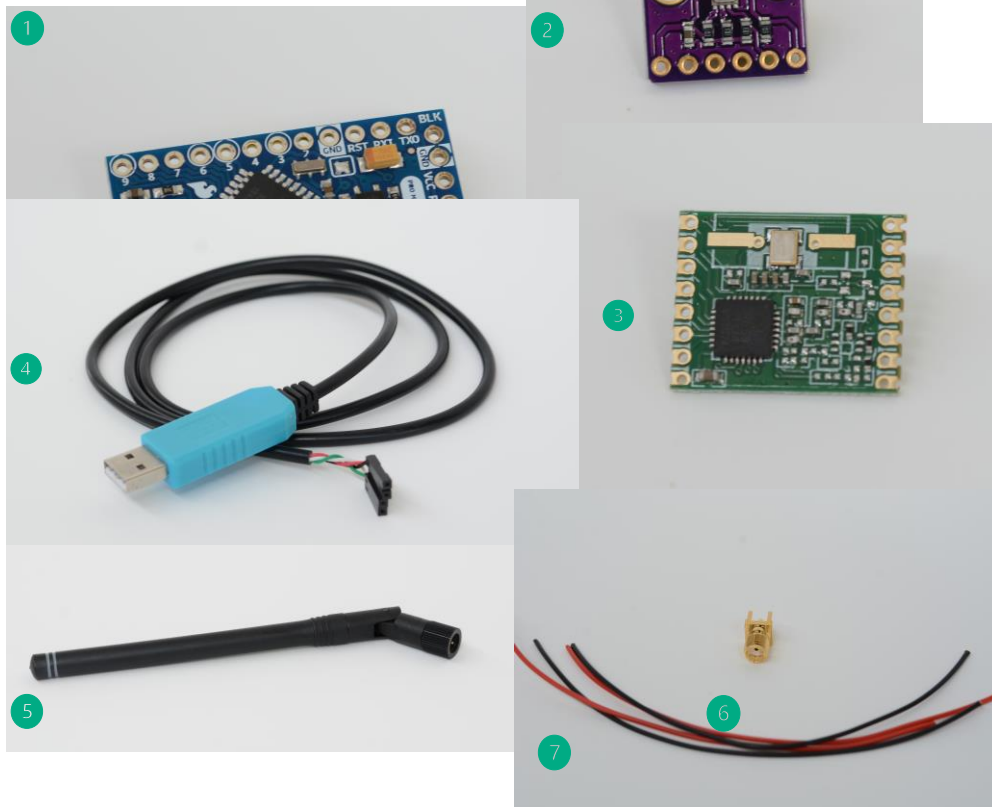


Figure 2: Components for the Arduino Pro Mini-Node-Variant

- ① Arduino Pro Mini
- ② BME280 Sensor
- ③ HOPERF RFM69W Transceiver Module
- ④ USB to UART TTL Cable Module
- ⑤ SMA Antenna
- ⑥ PCB RP-SMA male female plug RF Adapter Connector
- ⑦ Cable

!
The resonator of the Arduino Pro Mini may not be good enough to transmit Mioty. 8 MHz Crystal oscillator with 10 or 20 ppm recommended

For the assembly the additional supplies are necessary as well:

- Tin solder
- Soldering iron

For a complete overview of the finished module, as well as instructions on how to assemble it, see section [3.2.1](#).

Both Variants:

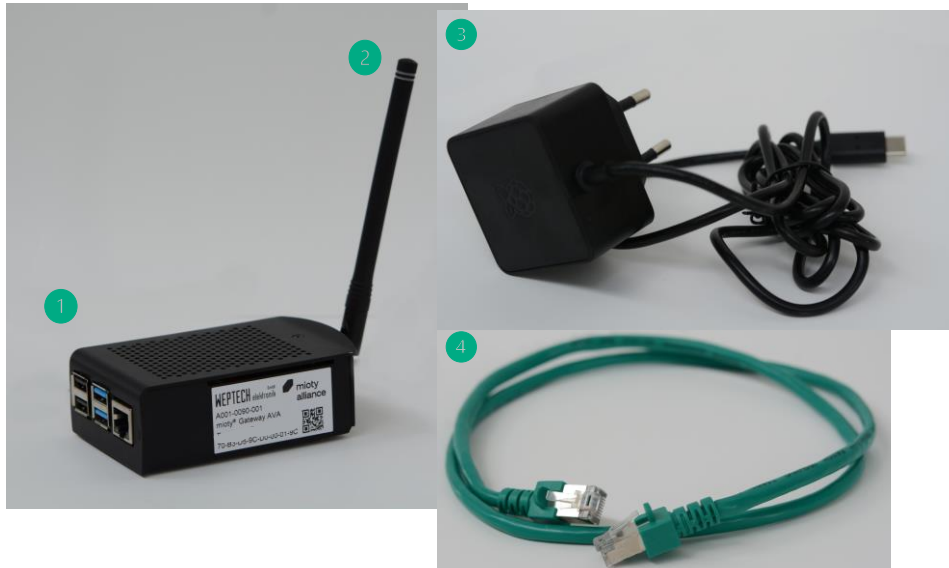


Figure 3: Components and supplies for the Base Station

- ① [miotyTM Gateway AVA](#)
- ② SMA Antenna
- ③ Plugin USB C Raspberry PI Power supply
- ④ LAN cable with dhcp support
- ⑤ Computer with internet connection

Any software that is required is addressed in the corresponding part of the tutorial. There you will also find links and installation/registration guides and instructions.

Overview of the MIOTY™ System

In this chapter, the MIOTY™ system and its technology are presented in their respective context. Section 2.1 presents the motivation, special features and advantages of the MIOTY™ technique compared to conventional techniques and shows first concrete application possibilities. Section 2.2 goes into more detail about the functioning and the underlying technical structure of the system and clarifies first basic backgrounds with regard to an own successful setup.

2.1

Introduction to MIOTY™



In the digital age, not only the number of Internet-capable devices is increasing, but also the possibility of wirelessly networking people, processes, data, devices, or entire plants. The latter, often as part of a smart factory, are increasingly sending, communicating, or exchanging time-critical measured values and sensor data in connection with the growing importance of Industry 4.0. The basis for this functionality is formed by wireless networks, which are characterized above all by flexible, robust, and cost-efficient solutions and infrastructures. LPWANs (Low Power Wide Area Network) are particularly advantageous for this, due to their beneficial properties such as high network coverage or low energy consumption. MIOTY™ is an LPWAN technology for the Internet of Things. The technology can be used to network a large number of sensors over long distances in an energy-efficient, robust and reliable manner. It combines the strengths of conventional LPWAN systems with advanced technology to further optimize its functionality and features. This results in a wide range of application domains and fields.

2.1.1

What makes MIOTY™ so unique?

MIOTY™ (= My IoT) is an LPWAN technology in the category of wireless network solutions. These are particularly well suited for private IoT (Internet of Things) applications. Due to its advanced and innovative protocol design, it not only ideally meets the demands and challenges of intelligent networking people and things in the Internet of Things, but also goes beyond conventional LPWAN solutions with its functions. In addition to a high range and scalability, the MIOTY™ system also offers much better transmission security and robustness against interference or transmission errors, thanks to its specially developed telegram splitting technology. This makes it possible to meet even demanding (industrial) IoT requirements. For a complete overview of MIOTY™ 's features see *Table 1*. Vendor-neutral software as well as commercial hardware designed directly for MIOTY™ use, enable the design of a cost-efficient IoT architecture by means of maximum flexibility and minimum complexity.

Overview of MIOTY™'s technical key qualities:

Huge network capacity	Current networks reach up to 3.3 mio messages/day with only one base station, future systems will be expanded
Extensive transmission range	up to 15 km in flat terrain up to 5 km in urban centers

Minimal power consumption	up to 20 years battery lifetime
Unique mobile communication	nodes operate at up to 120 km/h velocity
Quality-of-service	high interference immunity in a crowded spectrum, deep indoor penetration and multi-layer security
Worldwide operation	global license-free sub-GHz band
New standard	ETSI standard TS 103357 published in June 2018

Overview of the MIOTY™ System

Table 1: Overview of MIOTY™ 's technical key qualities

More information regarding the benefits of MIOTY™ and an overview can be found starting on p. 6 in the official document [“MIOTY™ - Physical Layer Technology”](#) or [online](#).

2.1.2

Application of MIOTY™

Due to its versatility, MIOTY™ can be used in a wide variety of application areas. In addition to traditional applications in the industry (4.0), such as cost-efficient and reliable monitoring of production plants, the MIOTY™ system is also used in mining and underground construction due to its robustness, as it can provide the high level of safety required there. Its wide coverage and efficiency also allow it to be used in oil and gas production and in agriculture. Here, the technology can be used to better monitor fields while helping to conserve resources and protect biodiversity. However, MIOTY™ is also used in the context of building management and urban planning: it not only enables the monitoring, control, and regulation of buildings, but can also contribute to the implementation of a smart city concept. A more detailed overview and additional information can be found on Fraunhofer's [official website](#).

MIOTY™'s flexibility and multiple benefits therefore allow it to be adapted to many circumstances and use cases. In addition to such large-scale projects, the technology thus also offers the possibility for own, individual ideas and projects in the private or educational area with simple and inexpensive means. Be it to realize smaller projects such as an irrigation system for the home garden or a simple temperature monitoring, as described as an example in this tutorial.

2.2

How do the MIOTY™ System and its underlying technology work?

The MIOTY™ system and its functionality are largely determined by its architecture. It describes the various hardware and software components within the system. These can, for example, be connected to each other in different ways, communicate or exchange data and thus fulfill various functions and tasks. In addition to forming the data chain, they also define the individual communication layers that represent the various functional levels of such a networked system. A basic understanding of the interrelationships and backgrounds of the individual components and the overall system can thus make it much easier to set up an own project.

2.2.1

Data Chain and Key Components

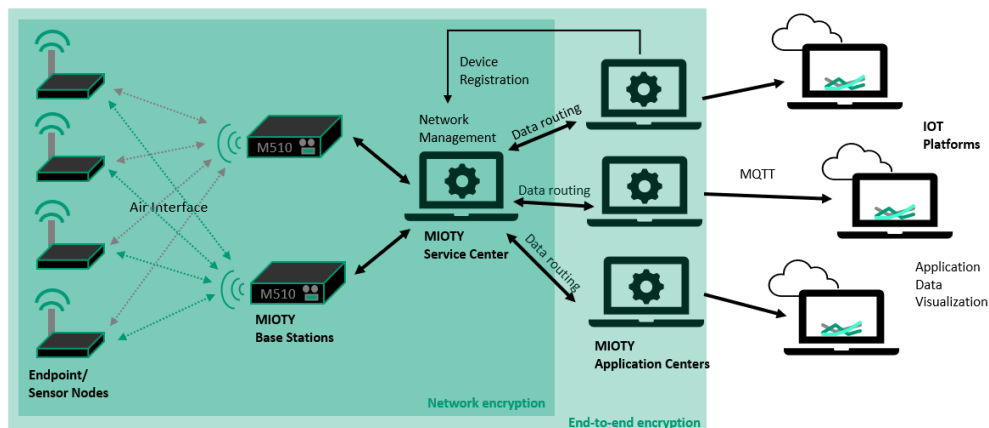


Figure 4: Data Chain of the MIOTY™ System

Figure 4 provides an overview of the entire data chain in a typical MIOTY™ deployment and as it is found in our project. The key components and their functions are explained in more detail below, also in relation to the MIOTY™ project:

- Remote End-points/ End-nodes with sensor
 - Each device is characterized and distinguished by a unique EUI (Extended Unique Identifier; typically printed on the label of your end node device or has to be acquired additionally in case of own designed modules; see the [official guidelines](#) for more information).
 - The built-in or connected sensors capture and measure field data such as environmental parameters or machine KPIs (key performance indicators).
 - After that, the data is processed by the node's CPU and sent from the transmitter module to the base station at regular intervals in analog form over the air using MIOTY™ Telegram Splitting technology.
 - A wide variety of node configurations can be supported for MIOTY™ projects. In this tutorial, the use of an Arduino Pro mini in connection with a Bme-sensor and the RFM69W Transceiver Module and the M3B magnoling Makerboard, which was co-designed especially for MIOTY™, is presented.
- Base station
 - The base station receives and collects all messages from the nodes registered with it. In general, it can be connected to any number of nodes.
 - It processes and decrypts the network control data contained in the messages and forwards the user data to the backend, which consists of the service and application center.
 - It is also responsible for generating and sending messages to bidirectional nodes that can receive messages as well.
 - A MIOTY™ network can consist of multiple base stations, for example to cover a larger area. In this case, the same message from a node can be received by multiple base stations.
 - For our project we use the MIOTY™ Gateway AVA. In addition to the pure functionalities of the base station, it comes with a local backend for stand-alone usage (an optional reconfiguration to external backends is possible if required) and thus facilitates usability and application.
 - The station also features a web interface that can be accessed via the local network and used to gain insight into data traffic and make settings and configurations.

- **Service Center**
 - The Service Center is responsible for device and network management as well as security and key management of the network-level cryptography.
 - It coordinates the messages within the system, for example by forwarding relevant messages to the Application Centers, deleting repeating messages or, in the case of bidirectional connections and several base stations, by selecting the Gateway that responds to the node and scheduling its downlink data.
 - There is always only one service center per MIOTY™ system where all devices are registered. It connects one or multiple Gateways to one or multiple Application Centers. From here they are also managed and controlled.
 - The Service Center functions required for our project are already pre-implemented on the AVA Gateway and run automatically after its installation. There is a possibility to configure the base station and use an external service center, e.g. when using multiple AVA gateways with pre-implemented service centers. The data can then be processed collectively by one gateway.
- **Application Center**
 - The Application Center serves as a point of contact to each end user or application operator. Accordingly, it can exist several times within a MIOTY™ system.
 - From here, new nodes are registered for the entire system with the help of the device EUIs for service and application center.
 - This enables the service center to distinguish which data from which end-node the individual application centers or their end users are interested in and to forward only these corresponding ones as a multi-tenant system.
 - When receiving messages from the service center, the application center encrypts and decodes the user data (by means of data format description/blueprints), which is then forwarded to the application platform (e.g. via MQTT protocol).
 - Similar to the Service Center, the required functions are already pre-implemented on our base station and can be accessed via the web interface. Again, there is the possibility of configuration to an external component.
- **Applications or Application platforms**
 - Applications or application platforms represent any server or cloud computing system whose core functions are data storage and analysis.
 - Received data can be examined here for patterns or visualized more vividly to make predictions and execute timely responses.
 - In our project, we use the open-source IoT platform ThingsBoard for this purpose.

Further information and background on this can also be found in the [official document](#) from page 7 onwards. For descriptions and explanations of specific functions and the use or commissioning of concrete components, see the corresponding chapters of this manual.

2.2.2

Telegram splitting as unique technology and advantageous feature of MIOTY™



MIOTY™ distinguishes itself from other comparable solutions through its system design, which is particularly advantageous for IoT applications. Traditional LPWAN systems constantly have to deal with network-related quality losses of the data. On the one hand,

the low available bandwidth and the desire for higher coverage increase transmission times. On the other hand, the data traffic takes place in the license-free spectrum, which is why there are often many users. This all increases the probability of interference and interruption of the signal when the channel is used more than once.

The Telegram splitting solution allows these challenges to be overcome. Instead of transmitting the entire message in one piece, each packet (a telegram) is split into numerous sub-packets and these are distributed over time and frequency. This reduces the transmission time of each individual packet and thus the probability that collisions occur. If they do occur nevertheless, only a few parts of the message are affected compared to before (this advantage is illustrated by *Figure 5*).

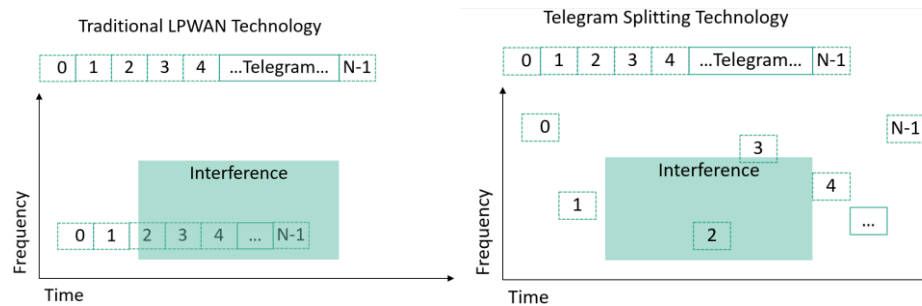


Figure 5: Interferences with a classic LPWAN and a system with Telegram splitting

Additional forward error correction allows for complete network reliability - even if only half of the data packets arrive at the base station. As a result, this exceptional system design provides unrivaled quality of service for critical industrial IoT applications where network quality is a top priority.

In our system, this is rendered by the technical realization and special programming of the node and base station hardware. Further information regarding telegram splitting can also be found in the official document [“MIOTY™ - Physical Layer Technology”](#).

3

End Point setup: Programming and commissioning of the sensor nodes

End Point setup: Programming
and commissioning of the sensor
nodes

End nodes represent the starting point of the data transmission chain. Usually in combination with sensors, they measure and record critical data for analysis or monitoring, which can then in a later step be further processed and visualized (see [2.2.1](#) for more detailed information). Depending on the objectives and structure of the project, a wide variety of hardware components can be used for this purpose. In this manual, two of these alternatives are presented: the specially for MIOTY™ developed M3B magnoling MAKERBOARD, which facilitates the entry into the MIOTY™ project by already existing functions and peripheral components or interfaces (see section [3.1](#)) and an open source approach by means of an Arduino board in connection with a Bme sensor and corresponding transmitting components (see section [3.2](#)).

Depending on the project, other prefabricated MIOTY™-related end node options and sensors can also be found in the [MIOTY™ -Alliance portfolio](#).

3.1

LZE MIOTY™ M3B magnoling MAKERBOARD

The specially developed LZE MIOTY™ M3B magnoling Makerboard, M3 Board for short, makes it easier to get started using MIOTY™ and then implementing projects thanks to its structure and existing components. The embedded, self-programmable microcontroller not only allows the implementation of a variety of individual projects, but the development and programming of the board is easily possible with the help of the widely used and well-documented open-source Arduino IDE. In addition, peripheral components or sensors for standard functions are already integrated. The board can nevertheless also be expanded as desired through external interfaces and therefore individually configured for a wide range of applications. More details and information about the board or its technical features can be found in the [MIOTY™ -Alliance Portfolio](#) or on the [LZE Product Page](#). For an alternative open-source approach, which requires a partial self-assembly of the board, see also chapter [3.2](#).

3.1.1

Getting Started: Preparatory steps for programming and flashing the board



Since the M3 board already has some built-in sensors like the SHT31 temperature and humidity sensor or the MS5637 barometric pressure sensor, the hardware for our MIOTY™ project is complete and no further peripheral components have to be connected. However, in order to be able to effectively use the board as a sensor node as the next step, a suitable program code must first be written and then flashed into the board's memory. This requires some preparatory steps:

- **Install the Arduino IDE** (tested with v1.8.16):
 - with the help of a suitable integrated development environment (IDE) the software for our board can be written and uploaded quite easily via the USB port
 - the latest version can be downloaded from the [download page](#) and installed in a few simple steps with the help of the detailed installation instructions for each operating system (see [Windows](#), [Linux](#) or [macOS](#))
- **Install the STM32CubeProgrammer:**
 - Not all boards are supported by the Arduino IDE from the start. Since the core of the M3 board belongs to the STM32 microcontroller family, extra software is needed to program it.
 - the latest version for different operating systems can be found [here](#)

- **Add stm32duino to Board Managers:**

- After installing the latest version of the STM32CubeProgrammer, the board itself has to be installed in the Arduino IDE to make it accessible via its board manager and compatible with the Arduino software.
- This requires several steps, which are explained in the paragraph "Install STM32 Cores" on the following [Github page](#). The board can then be selected and the previously installed programming software is automatically used during flashing.
- general information about adding new boards to the Arduino IDE can be found [here](#) (the stm32duino counts as third party core).

- **Add to library manager:**

- In programming, libraries are generally used to provide additional program resources and functionalities such as functions, subroutines, configuration data or documentation as a collection, in form of prewritten code. In order to easily address and use the various sensors of the M3 board, it is necessary to include their corresponding libraries. To make this possible, they have to be added to the Library Manager of the Arduino IDE.

- **Manually add Libraries as zip file:** Many libraries can be added very easily directly via the library manager. However, this can lead to compatibility issues between different library versions and prebuilt sample source code, which therefore this is not the best option. It is hence recommended to import the required libraries manually as a zip file. This should be done as follows:

- Download the zip file from the corresponding Github page via the **Code**① and then **Download ZIP**② buttons as seen in Figure 6:

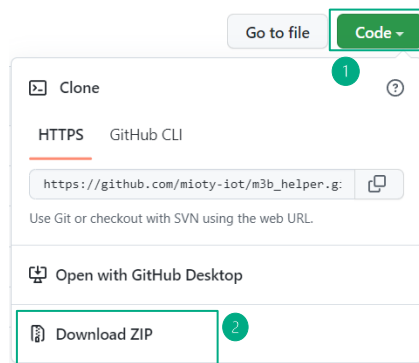


Figure 6: Manually adding Libraries in the Arduino IDE

- Via the menu bar navigate through *Sketch > Include Library > Add .ZIP Library...*:
 - Navigate to the library's .zip file location and open it.
- general information about adding libraries to the Arduino IDE can also be found [here](#)
 - The following libraries are needed for the MIOTY™ project:
 - [SparkFun MS5637 Barometric Pressure Sensor Library](#) (v1.0.1)
 - [SHT31 temperature and humidity sensor library](#) (v0.3.4)
 - ([ADXL362 Micropower 3-axis accelerometer library](#); not used for our MIOTY™ project but can add further functionality) (v1.5.0)
 - [MIOTY AT-Client](#): Implemented functions enable MIOTY™ -capable end nodes to send data to and communicate with the base station using MIOTY™ technology.
 - [MIOTY m3b helper](#): contain functions to facilitate the use and addressing of the M3B board and its hardware

- ➔ The above links also contain further information, documentation, sample code and other relevant/needed resources if required.

End Point setup: Programming
and commissioning of the sensor
nodes

- **Upload Settings**

- as last step the default settings of the IDE must be adapted to the board
- for this the following items are to be selected under the *Tools* path:
 - *Board > Generic STM32L0 Series*
 - *Board Part Number > Generic L072RBTx*
 - *USART Support > Enabled (no generic, serial)*
 - *Upload Method > Stm32CubeProgrammer (Serial)*

Corresponding information can also be found on the [m3b_helper Github page](#).

3.1.2

Writing the program

After all preparations for the successful development of the board as described in section [3.1.1](#) have been made, we can start with the actual coding. For this we use the Arduino IDE.

3.1.2.1

Writing Sketches with the Arduino IDE



The Arduino IDE is especially comfortable to use because it hides many complex operations or even takes them over completely. Thus, a board-independent programming is possible without the corresponding hardware knowledge. To get yourself familiar with writing programs and the basic elements of the Arduino IDE, please check out the section "Writing Sketches" on the [following Arduino page](#). Here all relevant aspects and buttons concerning the interface are explained, allowing to write our own first programs/ sketches afterwards. Additionally, the page offers further comprehensive introductory documentation to the IDE if required.

3.1.2.2

The Program code



This section introduces the M3 board sensor node program for the successful transmission of weather data using MIOTY™ to a base station. The programming language is C/C++, but with some Arduino specific features, which will be shown later. On the Internet there are numerous introductions and courses for writing C programs. If there should be a demand, see for example [W3Schools](#) or [educative](#). The code can generally be divided into individual segments: head area, setup-function and loop-function. All relevant functions and program aspects are described hereafter. In the appendix the [complete code](#) can be found as well. The code can be transferred depending on the application and your wishes, all sections to be adjusted are marked accordingly.

3.1.2.2.1

Head Area

This preparatory code segment is used to include all the required libraries into the sketch, to create some important objects and to generate assignments.

To be able to address the sensors integrated on the board as well as to use MIOTY™ and M3 board specific functionalities, some libraries have to be integrated into the sketch first:

```
// Libraries for Sensor use, MIOTY and M3 Board
#include <miotyAtClient.h>
#include "m3bDemoHelper.h"
#include "m3b_sensors/si1141.h"
#include <SHT31.h>
#include <SparkFun_MS5637_Arduino_Library.h>
```

End Point setup: Programming
and commissioning of the sensor
nodes

Next, we enable serial communication and data output directly in the IDE via serial monitor, using the respective library (see the [following page](#) for more detailed information). Afterwards, we can define two instances of SoftwareSerial objects, one for debugging purposes and the other for bidirectional MIOTY™:

```
//enable Serial Monitor
#include "SoftwareSerial.h"

// Debug Serial
SoftwareSerial SerialM3B(PA10, PA9);
// Mioty Bidi Stamp Serial
SoftwareSerial SerialMioty(PC11, PC10);
```

As the sensors use I2C/TWI communication, it must also be enabled (see [Wire](#) for more information). To address the sensors in the code, they are declared next. The same applies to the M3 board. The instance Wire2 furthermore facilitates the later port assignment.

```
//enable I2C
#include <Wire.h>

//Sensor Declaration
TwoWire Wire2(PB9, PB8);

MS5637          ms5637;
SHT31           sht31;
SI1141          si1141;

M3BDemoHelper m3bDemo;
```

Next, we declare an input variable for EUI and Network Key to be able to adjust them comfortably later (below are sample addresses).

```
// input new EUI
uint8_t eui64[8] = {0x70, 0xb3, 0xd5, 0x67, 0x70, 0x11, 0x14, 0x47};
// input new Network Key
uint8_t nwKey[16]={0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
0x01, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
```

Finally, we define a symbolic constant. This allows us later to restrict and conditionally compile certain parts of the source code. This is necessary because constantly resetting the network key and the required attach and detach operations can cause problems in the base station. Each time the module is reset, the setup() function (and the setting of the key) is executed again. To avoid this, **the define function should be commented after the first flash of the board and the code should be reloaded onto the device.** This way the corresponding code elements are not executed again.

```
//Conditional compilation to set the network key once at the beginning,
    should be executed only ONCE and commented afterwards
#define SET_NETWORKKEY
```



3.1.2.2.2

Setup-Function

The Setup() function is a special feature of the Arduino IDE. It is called only once at the beginning of the code execution (e.g. after reset of the board) and used to initialize variables, to set pins, etc.

In the setup() function we first start with some function calls to set the data rate of the serial monitors and some already defined configurations of the M3 board.

```
m3bDemo.begin();  
SerialM3B.begin(9600);  
SerialMioty.begin(9600);
```

Afterwards we initialize the sensors with their corresponding data and pins (for a corresponding overview see the [code beginning](#)). This function can also be used to detect malfunctions or connection interruptions.

```
//initialize sensors  
Wire2.begin();  
if (ms5637.begin(Wire2) == false)  
{  
    SerialM3B.println("MS5637 sensor did not respond. Please check  
    wiring.");  
    while(1);  
}  
sht31.begin(0x44, &Wire2);  
si1141.begin(&Wire2);
```

To be able to receive the data in the base station later, a unique network key must be defined for each node. If changes are necessary or the node is to be initialized at the beginning, this is done using the following section. To reassign the EUI or the network key, the node must first be disconnected and then reconnected. By using appropriate functions from the [miotyAtClient library](#) this can be implemented easily (the execution of the code is bound to the previous definition of the SET_NETWORKKEY constant and should be run only once):

```
//Code for initial one-time setting of the network key, will only be  
//executed if the constant SET_NETWORKKEY was defined previously  
#ifdef SET_NETWORKKEY  
  
// assign new EUI and Network Key  
uint8_t MSTA; // status of mac state machine  
  
// Local Dettach  
SerialM3B.print("Local Dettach:");  
miotyAtClient_macDetachLocal(&MSTA);  
  
// Set-EUI  
// SerialM3B.print("Set EUI");  
// miotyAtClient_getOrSetEui(eui64, true);  
// SerialM3B.println("");  
  
// Set-Network Key  
SerialM3B.print("Set Network Key");  
miotyAtClient_setNetworkKey(nwKey);  
SerialM3B.println("");
```

```
// Local Attach - required only once
SerialM3B.print("Local Attach:");
miotyAtClient_macAttachLocal(&MSTA);
SerialM3B.print("New Mac State:");
SerialM3B.println(MSTA);
SerialM3B.println("");
#endif
```

If the EUI is not known, it can be read out using the following code section (any 32 hexadecimal digits string can be specified as the network key):

```
// get Device EUI
uint8_t eui64[8];
miotyAtClient_getOrSetEui( eui64, false);
SerialM3B.print("Device EuI ");
for (int i = 0; i < 8;i++) {
    SerialM3B.print(eui64[i], HEX);
    SerialM3B.print("-");
}
}
```

3.1.2.2.3

Loop function

After creating the setup() function, the actual program is executed here continuously in a loop and thus gives the board its functionality.

In our code it consists of only one function call sendWeatherData(), with the help of which the weather data is read out from the sensors and sent via MIOTY™ Technology. The blue LED is used to visualize this.

```
digitalWrite(BLUE_LED, LOW);
float temperature = sendWeatherData();
digitalWrite(BLUE_LED, HIGH);

delay(3000);
```

The called function sendWeatherData() first reads the weather data using suitable functions, stores them in variables and can then send them via the predefined function transmitWeather(). Using the print functions the data can also be read out via the serial interface.

```
float sendWeatherData() {
// readout of weather data; humidity in %, temperature in °C, pressure in hPa
float pres=0., temp=0., hum=0.;
temp = ms5637.getTemperature();
pres = ms5637.getPressure();
sht31.read();
hum = sht31.getHumidity();
uint16_t lux;
si1141.readLuminosity(&lux);

// serial Monitor
SerialM3B.println(" ");
SerialM3B.print("Pressure [hPa] ");
SerialM3B.println(pres);
SerialM3B.println(pres * 10);
```

```

SerialM3B.print("Temperature [°C] ");
SerialM3B.println(temp);
SerialM3B.println((temp + 273.15)* 10);
SerialM3B.print("Humidity ");
SerialM3B.println(hum);
SerialM3B.print("Luminosity (rawData) ");
SerialM3B.print(lux);

m3bDemo.transmitWeather(pres, temp, hum, lux);

delay(4000);
SerialM3B.println("");
return temp;
}

```

 End Point setup: Programming
 and commissioning of the sensor
 nodes

The [complete code](#) can be found in the appendix. Here all used [functions](#) are explained again. For further information and support please refer to the [official Arduino website](#) or the corresponding documentation of [MIOTY AtClient](#) or [the m3b_helper](#) on Github. Several example sketches can also be found there.

3.1.3

Flashing and placement of the board



Once the code is ready, it can be flashed onto the board. There are a few things to be considered, also in the subsequent placement of the node:

- **Flashing the Board**

- To enter the bootloader mode of the board, the ① bootloader switch should be turned inwards, and the ② reset button should be kept pressed when connecting the board to the PC according to *Figure 7*:

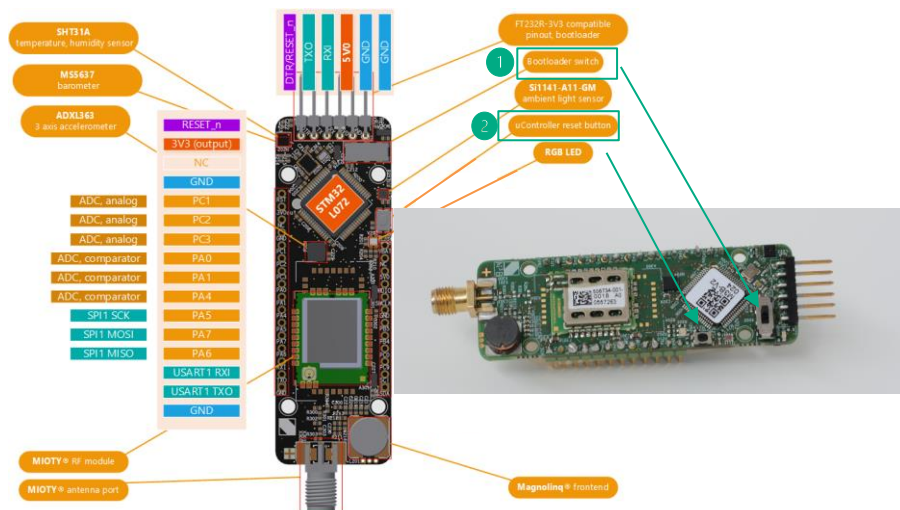
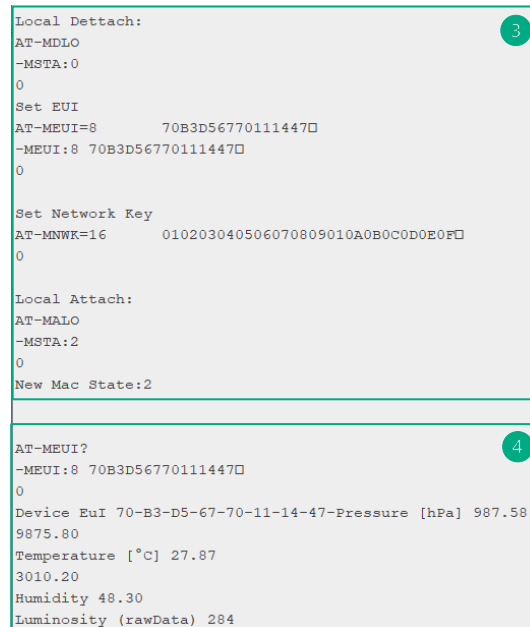


Figure 7: M3 magnolinq Makerboard Pinout and Interfaces: Bootloader switch and Reset Button

- Connect your Device via UART (e.g. FTDI cable; make sure you connect the appropriate pins and select the correct port in the IDE via *Tools > Port*)
- Upload the sketch
- Release the bootloader switch
- Press the reset button to start your sketch

- ➔ Now the program should run on the board and we can already read out the first data via the serial monitor (if the board is reconnected to the PC after programming, neither the switch has to be turned nor the button be pressed again). The output should look similar to *Figure 8* and provide us with information about the assignment of EUI and Network Key^③ as well as the sensor data^④ itself:



```
Local Detatch:
AT+MDLO
-MSTA:0
0
Set EUI
AT+MEUI=8      70B3D56770111447
-MEUI:8 70B3D56770111447
0

Set Network Key
AT+MNWK=16     010203040506070809010A0B0C0D0E0F
0

Local Attach:
AT+MALO
-MSTA:2
0
New Mac State:2

AT+MEUI?
-MEUI:8 70B3D56770111447
0
Device Eui 70-B3-D5-67-70-11-14-47-Pressure [hPa] 987.58
9875.80
Temperature [°C] 27.87
3010.20
Humidity 48.30
Luminosity (rawData) 284
```

Figure 8: Serial Monitor Output of the M3 Board after flashing

- **Correct placement of the device**

In order to optimize your system's operating range some additional aspects should be considered as a last step:

- The antenna should be placed upright (vertically)
- the transmitting antenna should be kept at least 6cm, if possible 70cm away from other objects, especially electrically conductive objects, walls and electronic devices (e.g. PC, monitor, LED lighting, etc.)
- the antenna should not be placed near transmitting equipment (e.g. cell phones, wireless headphones, etc.) as this may cause interference.

3.2

Arduino Pro Mini equipped with RFM69W transmitter module and Bme weather data sensor

In comparison to the specially developed M3 board (see chapter [3.1](#)), the MIOTY™ project can also be realized as an open-source solution. With the help of suitable hardware components and software provided by Fraunhofer¹, the full functionality can be achieved, and the design additionally creates an individual, independent, arbitrarily expandable, and customizable module. The well documented microcontroller board can

¹ The Fraunhofer software supports Arduino systems with ATmega328p processor (8MHz and 16MHz) and the Raspberry Pi Pico as well as the HopeRF RFM69hw transceiver.

easily development and programmed with the help of the widely used and also well-documented open-source Arduino IDE. This tutorial introduces the setup and use of an example project with an Arduino Pro mini in combination with an RFM69W transmitter module and Bme weather data sensor. However, many other configurations are possible as well. Basic information as well as technical features and background information on the individual components used in our example can be found on the corresponding pages: [Arduino pro mini](#), [RFM69HW transmitter module](#) and [bme sensor](#).

End Point setup: Programming
and commissioning of the sensor
nodes

3.2.1

Getting Started: Hardware setup



In order to use the Arduino as a sensor node, it must first be equipped with a sensor, transmitter module and a corresponding antenna with socket. In our project, the [RFM69HW transmitter module](#) and [bme sensor](#) are used. A possible connection configuration is shown in *Figure 9* (the I2C interface is used to drive the sensor):

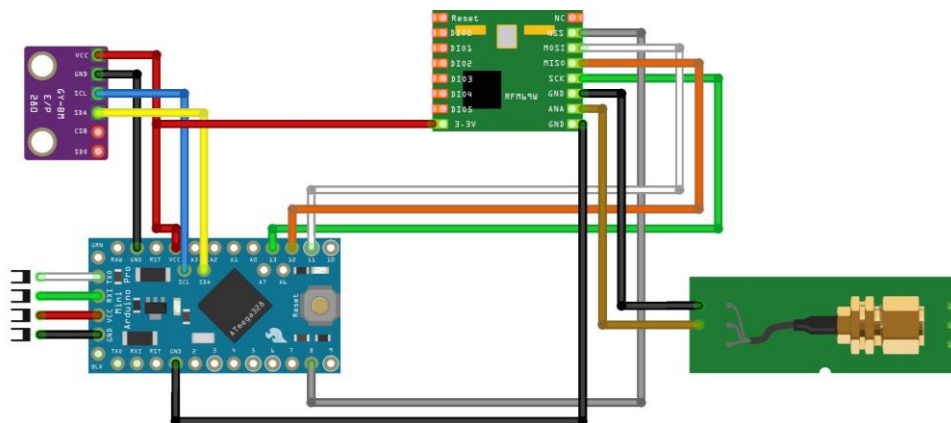


Figure 9: Wiring of the Arduino Pro Mini module

fritzing

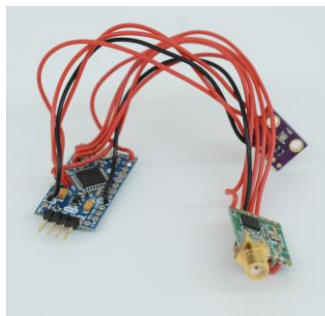


Figure 10 shows what the completed module might look like:

Figure 10: Example of a completely soldered module

!
The resonator of the Arduino Pro Mini may not be good enough to transmit Mioty. 8 MHz Crystal oscillator with 10 or 20 ppm recommended

3.2.2

Preparations for programming the board

After the hardware has been properly prepared (see section [3.2.1](#)), the next step is the programming. For basic information when using the board for the first time, you can also find hints and help in the [Getting started Guide](#) from Arduino. For this, first a few preparatory steps and then the coding itself are necessary (depending on the components, the libraries to be installed and the code must be adapted):



- **Install the Arduino IDE:**

the latest version can be downloaded from the [download page](#) and installed in a few simple steps with the help of the detailed installation instructions for each operating system (see [Windows](#), [Linux](#) or [macOS](#))

- **Install the required libraries:**

- [TS-UNB-lib](#):
 - The Fraunhofer Telegram Splitting - Ultra Narrowband Library ("TS-UNB-Lib") is software that enables the MIOTY™ standard for wireless data transmission in IoT.
 - the RFM69W transmitter module functionality and software are already integrated in it
 - The library is provided as a .zip file and can therefore be easily included via the path *Sketch > Include Library < ADD .ZIP Library...* (also see **Manually add Libraries as zip file**)
- [BME280 Library](#):
 - library for our weather data sensor
 - can either be added externally as a .zip file as well, or via the search box using the path *Tools < Manage Libraries....* (see *Figure 11*)

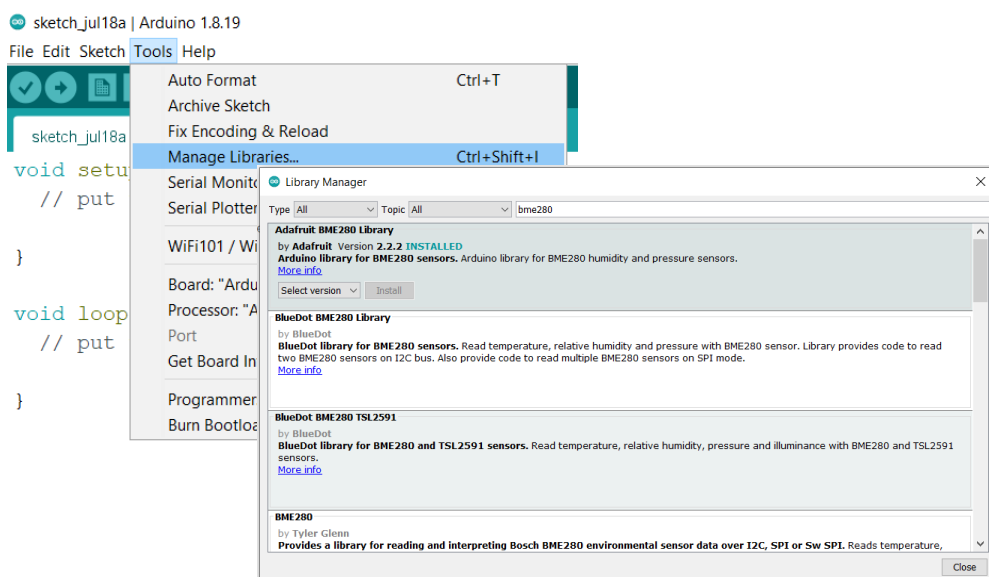



Figure 11: Adding Libraries in the Arduino IDE via search box

More information about adding libraries in the Arduino IDE can be found [here](#).

3.2.1

The Program Code

The code with its most important functionalities and elements is presented below. An overview of the most relevant [functions](#) of the TS-UNB library as well as all the related [code](#) can be found in the appendix (if the code is to be adopted, the elements to be adjusted accordingly are marked there). For more information and help on programming with the Arduino IDE see the [official guide](#). 

```
//Libraries for Sensor use and MIOTY
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>
//Import MIOTY TS-UNB-Node Library
#include <ArduinoTsUnb.h>

//enable Serial Monitor
#include "SoftwareSerial.h"

//enable I2C
#include <Wire.h>
```

```
//Sensor Declaration
Adafruit_BME280 bme;
```

End Point setup: Programming
and commissioning of the sensor
nodes

After all necessary libraries have been integrated and all required functionalities have been enabled accordingly, the next code section can be used to define the individual node configuration (only example values are shown here). In addition, the use of different RFM69 chip variants is very easy with the help of the namespace operator:

```
//Node specific configurations
//input new EUI
#define MAC_EUI64          0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
                          0x07, 0x08
//input new Network Key
#define MAC_NETWORK_KEY    0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
                          0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10

//input Transmit power in dBm, please keep in mind local regulations
#define TRANSMIT_PWR      10

using namespace TsUnbLib::Arduino;
// Select preset depending on TX chip, use Arduino PIN 8 for Chip Select
TsUnb_EU1_Rfm69hw_t<8> TsUnb_Node;
```

The setup function largely consists of function calls to initialize individual components of the code: First, the serial bus and the sensor are initialized to be able to use the serial monitor and sensor later and to detect possible malfunctions. This is indicated by the flashing of the onboard led:

```
void setup() {
  //setup serial Monitor
  Serial.begin(9600);

  //Sensor init
  unsigned status;
  status = bme.begin();
  if (!status) {
    Serial.println("Could not find a valid bme280 sensor, check wiring,
address, sensor ID!");
    while(1);
  }

  delay(100);

  // Blink the LED
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, HIGH);
  delay(100);
  digitalWrite(LED_BUILTIN, LOW);
}
```

Subsequently, already defined configurations of the node are transmitted and the packet counter, which is to prevent duplicate reception of identical messages, is initialized. This is symbolized by a second flashing of the LED:

```

// Init the node and its parameters
TsUnb_Node.init();
TsUnb_Node.Tx.setTxPower(TRANSMIT_PWR);
TsUnb_Node.Mac.setNetworkKey(MAC_NETWORK_KEY);
TsUnb_Node.Mac.setAddress(MAC_EUI64);

// TS-Unb ignores packets with an PkgCnt already received
// We use this function to configure the PkgCnt from the
// EEPROM
TsUnb_Node.Mac.extPkgCnt = initExtPkgCnt();

// Blink LED
pinMode(LED_BUILTIN, OUTPUT);
digitalWrite(LED_BUILTIN, HIGH);
delay(1000);
digitalWrite(LED_BUILTIN, LOW);
}

```

End Point setup: Programming
and commissioning of the sensor
nodes

In the main function, the sensor data for temperature and humidity are read out first and stored each in a float variable. Since the send function `TsUnb.Node.send()` can only process integers, the data is next converted.

```

void loop() {

    // read sensor data and convert to integer
    //Temperatur
    float payload_sensor_t = bme.readTemperature();
    Serial.println(payload_sensor_t);
    payload_sensor_t *= 100;
    int16_t payload_transmitter_t = (int16_t)payload_sensor_t;

    // Humidity
    float payload_sensor_h = bme.readHumidity();
    Serial.println(payload_sensor_h);
    payload_sensor_h *= 100;
    int16_t payload_transmitter_h = (int16_t)payload_sensor_h;
}

```

In order to define a unique order and arrangement of the sensor data bits within the payload array, they are assigned explicitly in the next step¹. Thus, we later know exactly how the data must be interpreted at the receiver, see also [Troubleshooting: Control of the received data](#). The `TsUnb_Node.send()` function then receives the data array and sends the payload using MIOTY™ technology:

```

//Prepare payload for correct format and save to array
uint8_t txdata[4];
txdata[0]= (payload_transmitter_t>>8)& 0xFF; //shift bits 8 to the
right and limit to 8 bit -> upper 8bit are stored in field,
remaining zeros are cut off in front
txdata[1]= (payload_transmitter_t)& 0xFF; // rear 8bit saved and
front cut off
txdata[2]= (payload_transmitter_h >>8)& 0xFF;
txdata[3]= (payload_transmitter_h)& 0xFF;

```

¹ The function `TsUnb_Node.send()` splits the message into 8 bit blocks, which are then sent sequentially. Since the sensor data are 16bit in size and depending on the compiler the LSB or MSB are interpreted first, errors can occur during transmission or later data interpretation. To avoid this, the data is therefore assigned with the help of this code section exactly to the 8bit large blocks in the correct arrangement.


```
//data transmission via mioty
TsUnb_Node.send(txdata, sizeof(txdata));

Serial.println(payload_transmitter_t);
Serial.println(payload_transmitter_h);
```

End Point setup: Programming
and commissioning of the sensor
nodes

As a last step, the packet counter is updated and these operations are signaled by the LED flashing twice. Since the node only sends new data every few seconds, it can be temporarily deactivated with the sleep() function:

```
// We store the current PkgCnt to the EEPROM to
// avoid the repetition of packets with the same count.
// This value is only written every 256 packets to
// save energy.
updateExtPkgCnt(TsUnb_Node.Mac.extPkgCnt);

// Blink the LED twice to indicate end of transmission
pinMode(LED_BUILTIN, OUTPUT);
digitalWrite(LED_BUILTIN, HIGH);
delay(100);
digitalWrite(LED_BUILTIN, LOW);
delay(100);
digitalWrite(LED_BUILTIN, HIGH);
delay(100);
digitalWrite(LED_BUILTIN, LOW);

// Sleep the device for 5 seconds using the watchdog timer
delay(5000);
}}
```

In order to make these steps visible in the serial monitor and to facilitate any necessary debugging, appropriate print commands have also been added. The whole code and further explanations can be found in the [appendix](#).

3.2.2

Flashing and placement of the board

- **Upload Settings:**

The appropriate board with the corresponding processor must be selected in the board manager via the paths shown in *Figure 12* (for the MIOTY™ project the 3,3V model of the Arduino Pro Mini with 8Mhz clock rate is used):

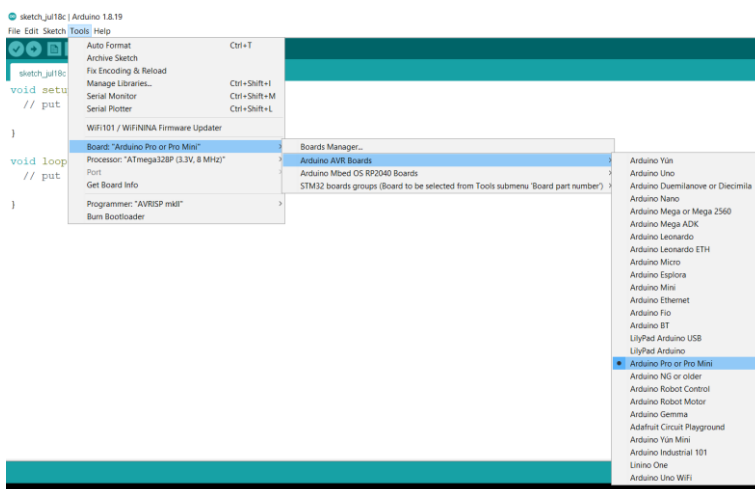


Figure 12: Configuring the Upload Settings of the Arduino Pro Mini in the Arduino IDE

- *Tools > Processor > ATmega238P (3.3V, 8 Mhz)*
- **Connecting the device to the PC:**
 - the board has to be connected to the PC via a TTL adapter according to *Figure 13* (see color coding of the wires as per product description, here: red VCC, black GND, white RXD, green TXD)

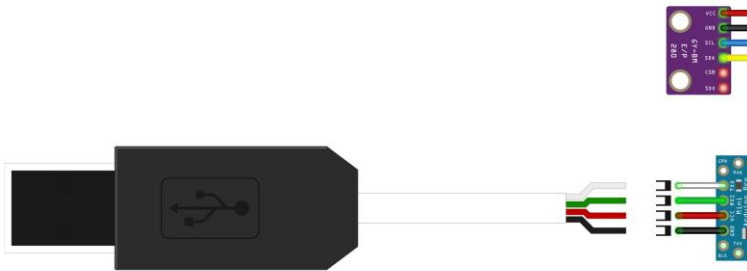


Figure 13: Connecting the Module with the TTL

Adapter

- the correct port has to be selected in the IDE under *Tools > Port*.

• Flashing the Board:

Depending on the adapter, the sketch upload is slightly different. The Arduino must receive a reset signal before each upload. This can be done by the GRN/ DTR connector. If the adapter has such an extra connector, then the code can be uploaded to the board as usual. However, if it does not (as in our case here), in order to program the Arduino Pro Mini, the reset button must be pressed during the upload until the compilation process is complete (and then released). This is the case when the white text appears as shown in *Figure 14*:

Uploading
Sketch uses 16192 bytes (52%) of program storage space. Maximum is 30720 bytes.
Global variables use 650 bytes (31%) of dynamic memory, leaving 1398 bytes for local variables. Maximum is 2048 bytes.

Figure 14: Time to press the reset button when the white text appears

➔ Now the program should run on the board and we can already read out the first data via the serial monitor. An output could look like shown in *Figure 15*:

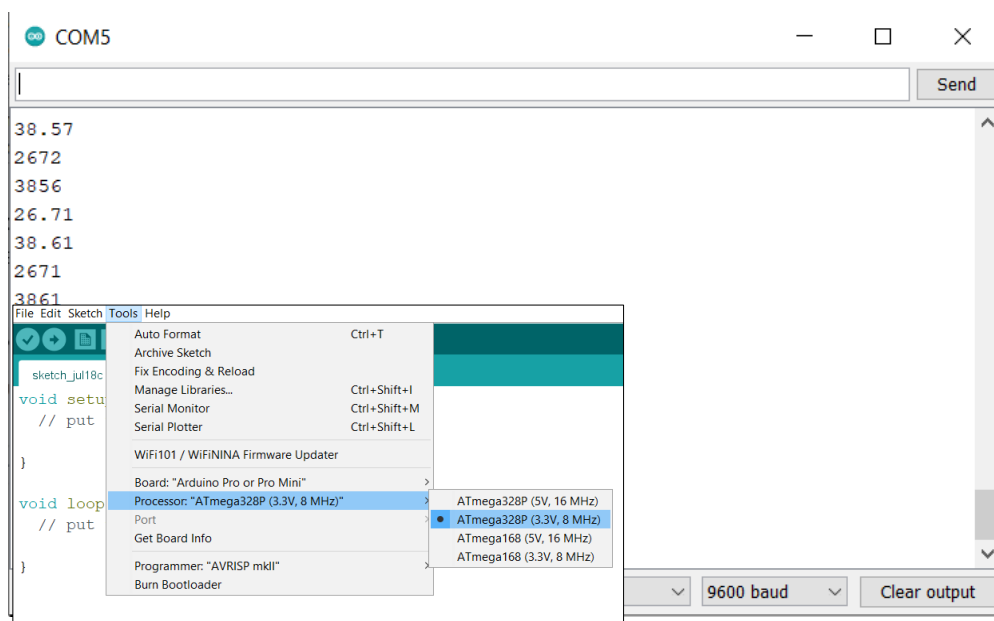


Figure 15: Serial Monitor Output of the Arduino Module after flashing

- **Correct placement of the device:**

In order to optimize your system's operating range some additional aspects should be considered as a last step:

- The antenna should be placed upright (vertically)
- the transmitting antenna should be kept at least 6cm, if possible 70cm away from other objects, especially electrically conductive objects, walls and electronic devices (e.g. PC, monitor, LED lighting, etc.)
- the antenna should not be placed near transmitting equipment (e.g. cell phones, wireless headphones, etc.) as this may cause interference.

The following section will explain how to install and use the MIOTY™ developed Ava Gateway as our base station. It comes with some pre-implemented features and functions which facilitate the setup of our project (for more detailed information see the [MIOTY™ -Alliance Portfolio](#) or [Webtech's Product Page](#)) and is therefore well suited for our purposes. In this regard, the theoretical background of the data transmission and the gateway's context within is explained first in section 4.1. Section 4.2 describes how to set up and commission the base station. Section 4.3 deals with registering new end nodes and section 4.4 with receiving their data. Finally, section 4.5 explains how to check the still raw data as a first troubleshooter.

4.1

Context with the data transmission path

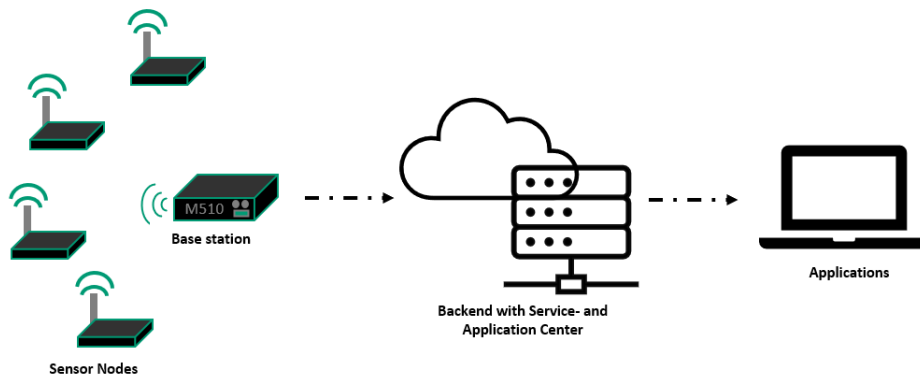


Figure 16: The base station within the data transmission path

As Figure 16 illustrates, our communication and data transmission chain consist of several components, each with a different set of tasks. The base station (the Ava Gateway in our MIOTY™ project) collects the messages from all nodes registered to it and forwards them to the backend. The backend consists of the service and application center and is responsible for network management, encryption, and communication with the end user or applications. It is located either on a third-party cloud platform or on an internal user server. For a general overview and introduction to MIOTY™ data transmission or the special functionalities of its components, see section [2.2](#).

In our case, important functions of the backend are already pre-implemented on the base station. The AVA Gateway thus goes beyond the pure functionality of a base station and describes a combined component with the backend, which thereby already enables rudimentary basic functions of Service and Application Center such as the configuration of sensors, the decoding of their data traffic and the analysis of their data via MQTT without further steps, simply via the web interface. If desired, the gateway can also be configured in such a way that the backend components are located externally, and the service and application centers are possibly even managed separately (see section 4.2 of the [official document](#)).

4.2

Base station setup and installation

Base Station setup and
Connecting Endpoints: AVA
Gateway start-up

Before the gateway can be used, the following steps should be carried out during installation:

- Connecting the gateway with the Lan socket, including DHCP support;
Note: for the use an Internet connection is required, if necessary consultation with the IT department must be kept
- Screwing the included antenna to the gateway
- Connecting the USB-C Raspberry power adapter

Afterwards, the base station is ready for use and its web interface can be accessed via the browser using `ava/` or the corresponding IP address. More information about this can be found in the manual or Quick Start Guide included with your device.

In addition, there are some aspects to consider when placing the Gateway itself in order to optimize its operating range:

- the AVA Gateway is designed for indoor use only.
- its antenna should be placed upright (vertically)
- the receiving antenna should be kept at least 6cm, if possible 70cm away from other objects, especially electrically conductive objects, walls and electronic devices (e.g. PC, monitor, LED lighting, etc.)
- the receiving antenna should not be placed near transmitting equipment (e.g. cell phones, wireless headphones, etc.) as this may cause interference.

In our project, both transmitters (nodes) and receivers (base station) are located indoors, where the most interference and other attenuation effects are to be expected. The reception quality is determined not only by the devices themselves, but also by their placement and distance. The more walls and obstacles there are between them, the worse the connection can become. For this scenario, operating distances of between 32m and 200m can be achieved, depending on the situation and the setup (compare indoor-outdoor: 130m- 1800m, outdoor-outdoor: 1.8km-30 km). This should be taken into account when setting up the system. More detailed information, including different scenarios with indoor and outdoor placement, can be found in the [official document](#). For the correct setup of the end nodes, see section [3.1.3](#) or [3.2.2](#).

4.3

Register new Endpoints



Before registering new end nodes to the base station, make sure that the AVA Gateway has already been correctly commissioned as described in Section [4.2](#) and that all corresponding end nodes have been set up ready for operation, as described in sections [3.1](#) or [3.2](#).



To enable communication between devices and the gateway and to receive messages, the end nodes must be registered in the application center of the base station. Through our web browser, we can access the web interface of the base station via `ava/` or with the corresponding local IP-address (e.g. `ava/` or just `192.168.178.168`), which can be found in the router settings as soon as the base station is connected (for the correct setup see section [4.2](#)). Afterwards the *dashboard* should be visible and it gives us an

overview and insights into our project traffic (here not visible yet, since no nodes are connected):

Base Station setup and
Connecting Endpoints: AVA
Gateway start-up

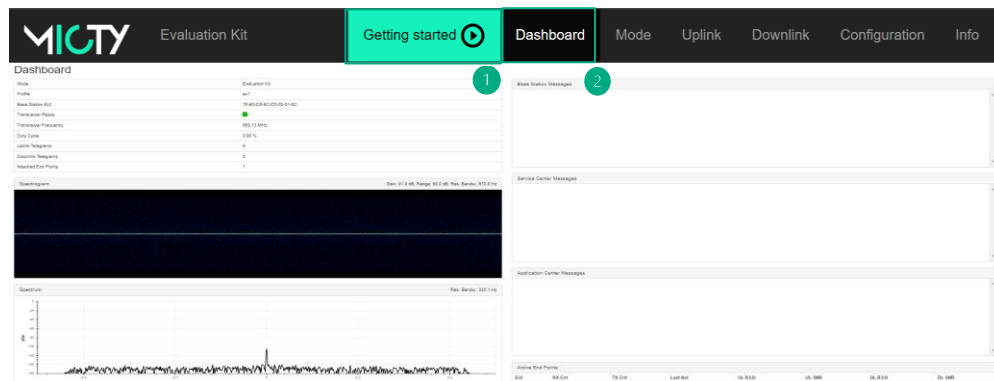


Figure 17: View of the MIOTY™ Dashboard

If more information on the web interface is required, the *Getting-started guide*^① can also provide a good introduction and initial overview (see *Figure 17*).

The next step is to navigate to the Configuration Page^② and fill in the Endpoint Registration form with the following elements:

- **EUI** (Extended Unique Identifier): Typically printed on the label of your end node device. It should be formatted as 8 hexadecimal numbers, separated by hyphens, e.g. 70-b3-d5-9c-d0-a0-33-01. See section 5.1.3 for more information about the identifier and section 3.1.2.2.2 to read out and redefine it for the Arduino.
- **ShAddr** (short address, reduces the required bandwidth): Defaults to the 5th and 6th byte of the EUI, formatted as a single hexadecimal number. It is entered automatically and therefore should not be filled in, if possible, as it can lead to problems in case of errors. !
- **BiDi** (bidirectionality): should be set for bidirectional nodes that can both send and receive messages
- **PreAtt** (pre-attachment): All unidirectional devices must be pre-attached. Bidi devices can also do an over-the-air attach.
- **CarrOff** (Carrier offset): Depending on the crystal tolerance of the end-point carrier offset 1 or 5 is used.
- **Network Key**: The sensor's network key, formatted as a single string of 32 hexadecimal digits. Is defined via the node codes, compare sections 3.1.2.2.2 or 3.2.1.
- **Application Key**: The sensors application key, formatted as a single string of hexadecimal digits. Optional.
- **Type EUI**: The EUI of the sensors type description, which Specifies the format of the application data. Optional.

A fully completed Endpoint Registration form (based on our MIOTY™ project) might look like in *Figure 18*:

Figure 18: End Point Registration form in the basestation web interface

Base Station setup and
Connecting Endpoints: AVA
Gateway start-up

No short address is required, as this is automatically assigned during registration. Since we only want to send sensor data with our end nodes, but do not want to/cannot receive messages, we do not check the box for a bidirectional connection. Therefore, we also need a pre-registration. Since we do not want to additionally encrypt our data at the application layer (our data is already encrypted end-to-end), no key is needed here.

The type EUI can be used to assign data description formats to the individual devices and to interpret the data. It is recommended to first check the correct reception of the raw data (see section 4.5) before adding the type EUI in a further step (see section 5.3.2).

Afterwards the entries can be confirmed with Register^③ (see Figure 18) and the node is added to the base station. If it was registered correctly, it should now be visible under Registered Endpoints and its messages can be received.

4.4 Data reception



If the sensor nodes were connected correctly according to section 4.3, their received messages should be visible in the Dashboard^① or the most recent message and its data in the Uplink Page^② as seen in Figure 19 and Figure 20.

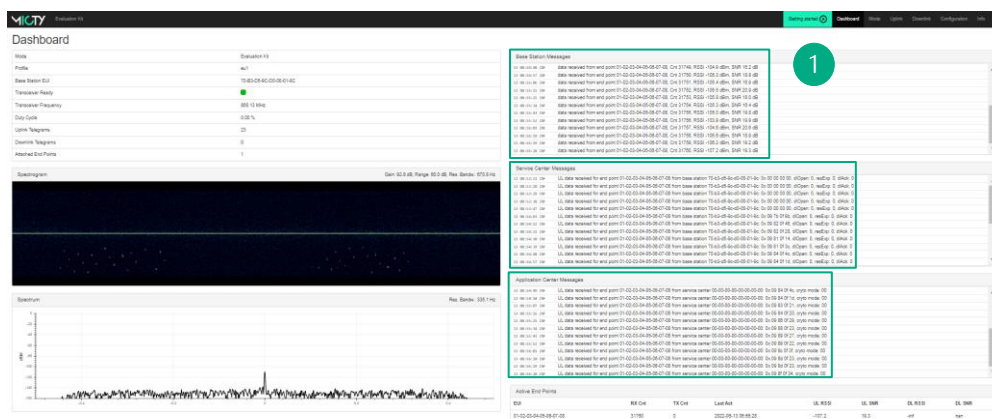


Figure 19: Dashboard Overview when receiving first messages

Uplink ^②

Received Uplink Data

EUI	Cnt	Time	RSSI	SNR	Type	Data
01-02-03-04-05-06-07-08	35858	2022-06-13 13:27:49	-108,9	18,2		0x 0a 67 0e fa

© 2021 Fraunhofer-Gesellschaft | Imprint | 4.6.1

Figure 20: Uplink Page Overview

Additional information on adding nodes can also be obtained from the Ava Gateway manual included with your device.

4.5

Troubleshooting: Control of the received data

Base Station setup and
Connecting Endpoints: AVA
Gateway start-up



After the base station has been successfully commissioned (see section 4.2), all end nodes have been registered (see section 4.3) and the first data has been received (see section 4.4), a next relevant step is to check the correctness of the messages, which are still available as raw data, before they are interpreted with the help of blueprints (see section 5). This step is optional, but it can facilitate the troubleshooting considerably.

For this purpose, the sent data must be compared with the received data. This is shown hereafter using the example of our MIOTY™ project with the temperature sensors:

Thanks to our code for the nodes, we can read the sent data quite comfortably with the help of the serial monitor in the Arduino IDE. To start it, you can either follow the path *Tool -> Serial Monitor*, use the shortcut *Ctrl + shift + m* or click the icon in the upper right corner ① as shown in Figure 21 (for more detailed information regarding the usage and functions of the Arduino IDE see the [official Getting Started Guide](#)).

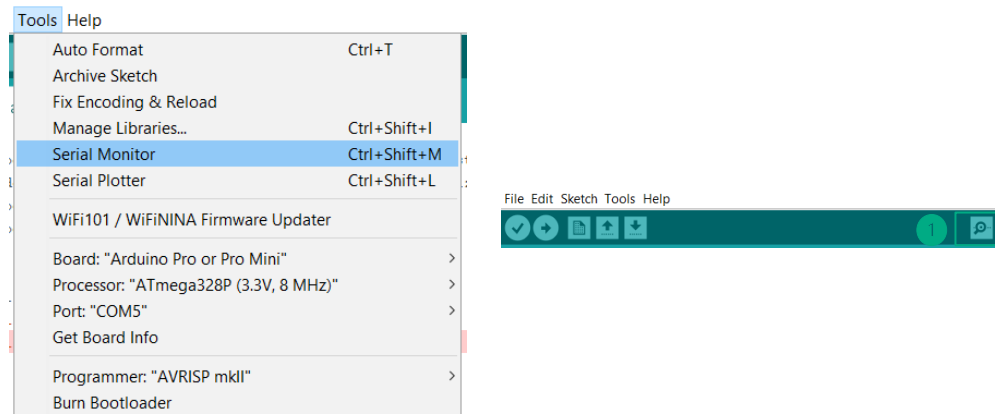
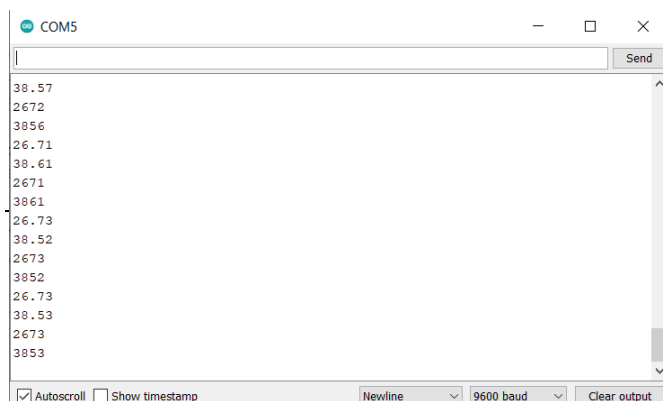


Figure 21: Possibilities to open the Serial Monitor in the Arduino IDE

Afterwards the data (first sensor data and then the data converted for transmission) should be displayed in the serial monitor if the board is connected to the computer ② (see Figure 22 and Figure 23). These can then be compared with the current message in the uplink page of the Application Center of our base station ③. To compare the data, the corresponding data format due to the node code must be taken into account, see sections 3.1.2.2.3 or 3.2.1 for the code with notes, or for the complete code [appendix arduino](#) or [appendix m3b](#).¹

If the four/ seven hexadecimal digits (corresponding to 16/ 28 bits as defined in the code) are converted, the values received are the same and the transmission has worked correctly (shown exemplarily for the Arduino node in Figure 22 and for the M3 board in Figure 23):



2

d quite easily from the code, for the
send function `sendWeatherData()` is

Base Station setup and
Connecting Endpoints: AVA
Gateway start-up

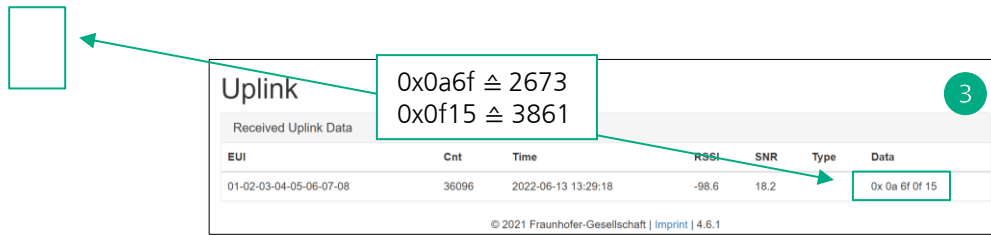


Figure 22: Verification of the received data by converting the send message at the Arduino node

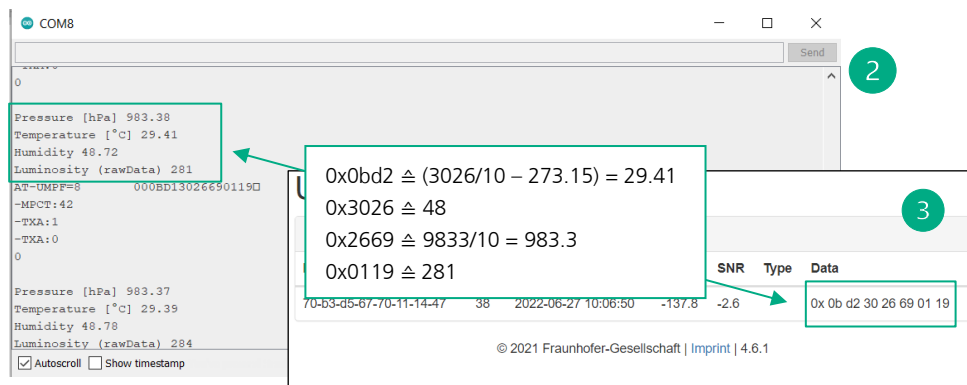


Figure 23: Verification of the received data by converting the send message at the M3 Board

If the values do not match, the code for the sensors should be checked again for possible error sources.

Blueprints enable the received data to be interpreted at the base station and displayed in a readable form. They thus facilitate subsequent further processing or transport. Chapter [5.1](#) provides a general overview and introduction to the Json data format and blueprints, chapter [5.2](#) describes and explains their architecture and how blueprints can be created, and chapter [5.3](#) concludes by showing how finished blueprints can be added to the base station or addressed to end nodes.

5.1

Introduction to Blueprints and the Json data format



A Blueprint is a payload format description that enables the received raw data to be interpreted. Json, on contrast, is a compact data format that can be easily understood by both humans and machines due to its easy-to-read text form. A data format generally specifies (still independent of the actual application) how the data is structured and represented. The payload format description on the other hand is often based on such a data format and does not describe the data itself, but how to interpret it. Both aspects are necessary for a functioning data processing and data exchange between applications. They also occur together in our MIOTY™ project, where the Blueprint is described in Json format. The application of these two concepts is relevant, both for the correct representation of the raw data at the base station and later when processing and sending it further (e.g. to our IoT platform). It can therefore be crucial to be able to use the received data in a meaningful way.

The following section will therefore provide a brief introduction to the functionality and meaning of the Blueprint and with respect to that explain how the Json format is constructed and subsequently interpreted.

5.1.1

Why do we use Blueprints?

The concept of blueprints not only facilitates data exchange and transport between different applications, but also offers some advantages for systems with limited bandwidth and battery-powered endpoints, as is the case with MIOTY™. The use of format descriptions creates a very flexible and extensible approach. Individual devices and applications can be developed according to individual requirements rather than strict protocol constraints that dictate, for example, the type of payload data. Thus, any type of payload data is supported and can be sent as long as there is a corresponding blueprint that explains how the data is to be interpreted. Furthermore, only dynamic information has to be transmitted, since all static information, e.g., what kind of data type it is or what unit the information has, is already known due to the blueprint. It is this bandwidth saving that benefits LPWANs such as MIOTY™.

5.1.2

What is a Blueprint?

The Blueprint provides all the information needed to interpret the raw data of an end-point device. In our MIOTY™ project, for example, it describes exactly how the payload of the with MIOTY™ technology transmitted data (which arrives as a hexadecimal

sequence of numbers at the base station) is assigned byte by byte to (understandable) information and interpreted.

The Blueprint for MIOTY™ related data is represented in Json format. Each Blueprint must be assigned a unique ID ("typeEUI") from the provider's IEEE EUI64 range.

Correct representation of data on
the Base station with the help of
blueprints

5.1.3

What is a TypeEUI and what is it needed for?

An EUI describes a standardized MAC address format for identifying network devices, formatted as 8 hexadecimal numbers. In addition to devices, it can also be used to uniquely distinguish and define data format descriptions or blueprints. This is then referred to as a TypeEUI. These are stored in a database of the Application Center with the associated Blueprint. As soon as the Application Center receives a message containing a TypeEUI, it knows what type of data it is and how this raw data must be interpreted or decoded.

5.1.4

Summary Example Blueprint

Let us now assume that the following raw data are available. Without a corresponding explanation, however, not much can be said about them yet. Through the Blueprint, however, we know the following:

Byte	0	1
Payload Data	00	F3

- Bytes 0 and 1 belong to temperature data of a sensor
- the data size is 2 bytes
- the unit of the data is °C
- the data type is integer, i.e., the data must be converted into decimal numbers and to get the sensor data, this decimal value must be divided by 10.

If we convert the hexadecimal number 0x00F3 and divide the result 243 by 10 we get 24.3. If we now interpret the raw data, we see that our end-node has measured and sent 24.3°C.

5.1.5

How does the Json Data Format work?

The JSON data exchange format is defined by its lightweight, text-based, language-independent syntax. It is a data format for storing, processing, and transporting data, easy to read and has a small data size. Different data types are supported, but the data itself is described as a text file, a so-called "string".

Understanding the syntax of Json is necessary to be able to comprehend the structure of blueprints and later create our own. The easiest way to explain it is by using a concrete example, so in the following we will build a Json object step by step and explain the corresponding elements:

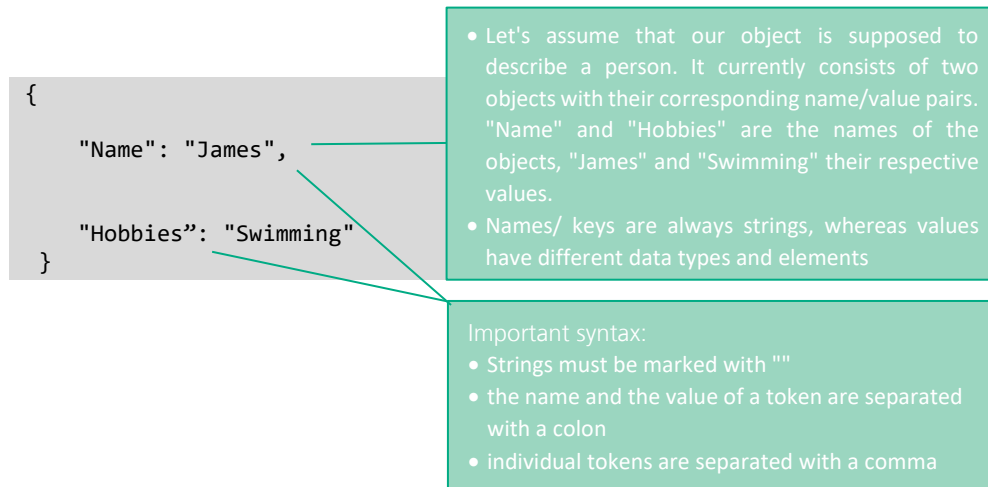
- Generally, JSON describes data in the form of objects. An object is defined by two curly braces {}:

```
{  
    
}
```

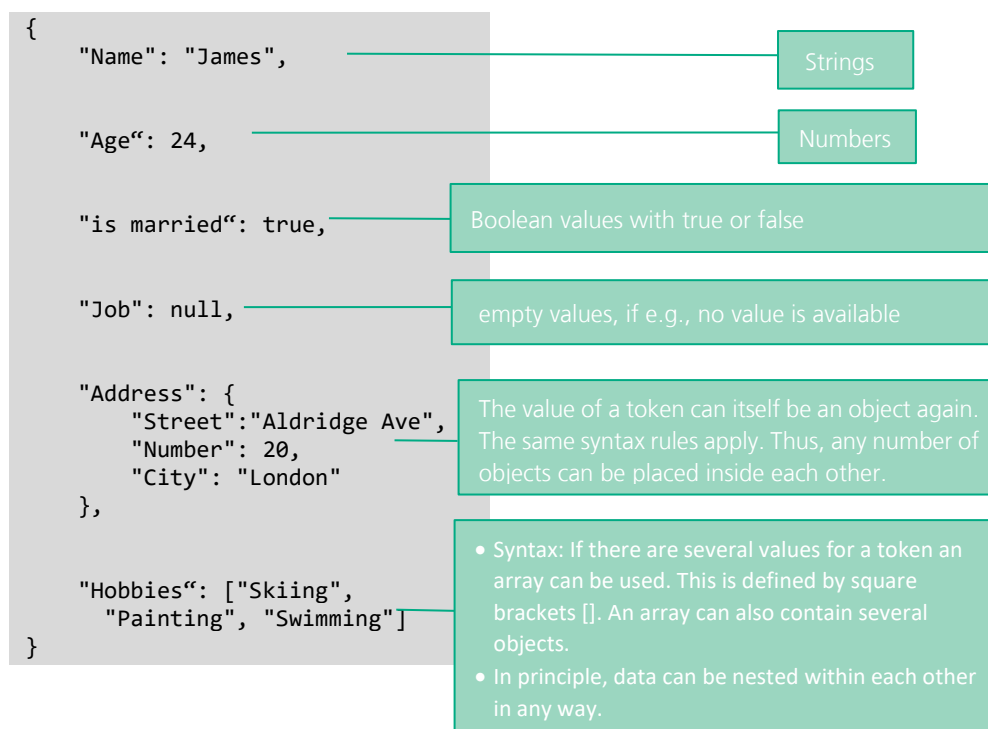
By the two brackets a new object is defined. This one is still empty.

- The actual object description is then located within these brackets. Objects consist of key pairs, which are defined by name (also called key) and value. The name/ key describes the object's name, whereas the value includes the object's value:

Correct representation of data on the Base station with the help of blueprints



- Furthermore, different value elements, which define the kind and content of the data and thus ultimately the object type can be used:



All Json files follow these basic syntax rules. More information can be found in the [official document](#) or this [website](#) about the Json standard.

5.2 Creating a Blueprint

The Blueprint allows to interpret and transform the raw data of the end nodes into understandable values. This takes place in the application Center of the base station by providing all necessary instructions to assign the payload bits to the corresponding information. The payload format description itself is described in Json format. More

detailed information and background about Blueprints, as well as the basic functionality and explanation of the Json format can be found in the previous sections.

Correct representation of data on
the Base station with the help of
blueprints

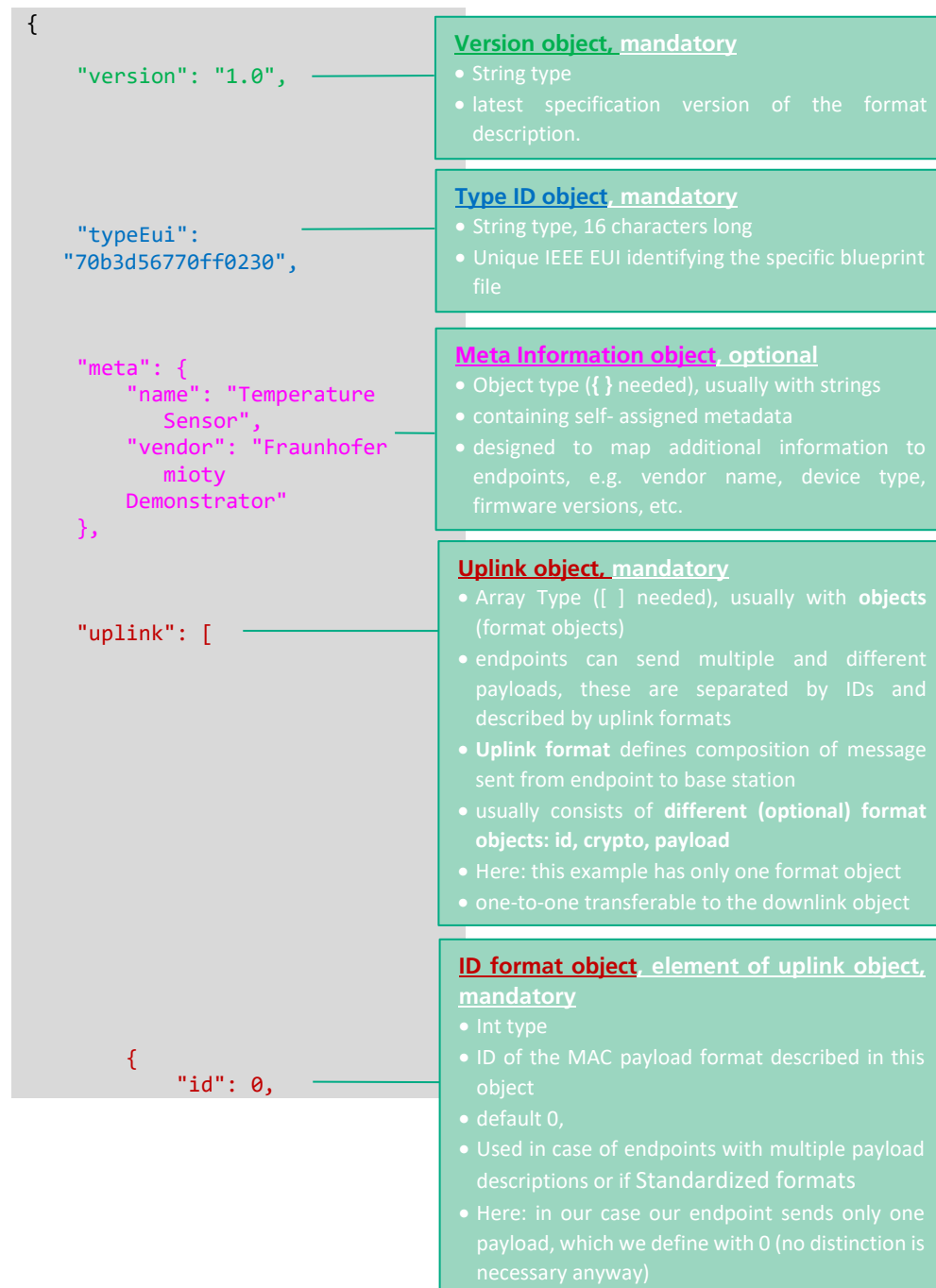
The following section, on the contrary, is intended to show the basic structure of the Blueprint architecture and to explain its contents based on the example from our project. For this purpose, all contents relevant for our project are described. In addition, in section [5.2.2](#) we will use an example to show exactly how the template code must be adapted and filled in. If further information is required, it can be found in the Application Layer Specification.

In addition, the sensor-specific blueprint of the [M3 Board](#) as well as [Arduino Module](#) can be found in the Appendix. These can be adopted or modified for own future designs.

5.2.1 Architeture



Blueprints should always be structured according to the following scheme. In the sequel, the Blueprint from the MIOTY™ project is presented and all its relevant objects including their value elements/ types are described:



Correct representation of data on the Base station with the help of blueprints

```
"crypto": 0,
```

Crypto format object, element of uplink object, optional

- Int type
- defines if and with which cryptography mode the data is end-to-end encrypted in the AL
- default value is 0 (our example does not use encryption)
- if used, interoperability between node and AC must be ensured
- more information can be found in the specification

```
"payload": [  
  {  
    "name":  
    "temperature",  
    "component":  
    "16bitTemperature"  
  },  
  {  
    "name":  
    "humidity",  
    "component":  
    "16bitHumidity"  
  }  
],
```

payload format object, element of uplink object, mandatory

- Array type with component objects
- Contains component description for every component in payload
- **The Order of the components in the payload array defines the order of the interpretation of our payload message**
- Two methods:
 - Inline description (instead of Reference to component, see the [M3 Board blueprint](#) in the appendix as example)
 - Reference to component (see example [here](#))
- Here: the two components "temperature" and "humidity" are created
- Their component values (e.g. "16bitTemperature") can be used to reference to their **component description** that defines the component

```
"component": {  
  "16bitTemperature": {
```

Component object/ component description, element of uplink object, mandatory

- Object type with component attributes
- Components referenced in the uplink section must be described (with component attributes/objects)
- Component keys (here e.g. "16bitTemperature") link component descriptions to their references and make them reusable

```
"size": 16,  
"type": "int",  
"func": "$/100",  
"unit": "°C",
```

Component attributes, element of component object, mandatory

- Different types, describe the component
- The following attributes are possible:

Key	Type	Description
size	int	Size of component in bit
type	string	Datatype of component, for a list with all types see table Component Datatype
unit	string	Unit of component

```

        "littleEndian": false
    },

    "16bitHumidity": {
        "size": 16,
        "type": "int",
        "func": "$/100",
        "unit": "%",
        "littleEndian": false
    }
}

```

Correct representation of data on the Base station with the help of blueprints

func	string	Mathematical function to calculate components output value, for a list with all function literals see table Function Literals
Little Endian	boolean	Defines bit order for data interpretation: if false, first bit in row (most significant bit) gets interpreted first

Type	Description	
int	Signed integer (positive and negative)	
uint	Unsigned integer (only positive values), <= 64 bit	23; 5,698
float	Floating point number, size must be 16, 32 or 64 bit	12.345, -55.3
bool	Boolean values	True; false
string	String of UTF-8 characters , must be a multiple of 8	"Multiple"; "absolute"
binary	Binary data	100101; 1111 ¹

Table 2: Component Datatypes

5.2.1.1 Component Datatypes

5.2.1.1 Function Literals

Variables	Input value of the enclosing component: \$ Input value of other components \$<component name>
constants	Euler's constant: e= 2.71828..., Archimede's constant: π = 3.14159...
operators	arithmetic +, -, *, /, % relational ==, !=, <, >, <=, >= logical &&, , !

¹ The binary data represents the unconverted user data in base64. The most significant bits of the most significant byte are padded with 0 if the size is not a multiple of 8.

bitwise	&, , ^, ~, <<, >>	Correct representation of data on the Base station with the help of blueprints
functions	abs, acos, asin, atan, atan2, ceil, cos, cosh, exp, floor, ln, log, max, min, pow, round, sin, sinh, sqrt, tan, tanh	
parenthesis	()	

Table 3: Funktion Literals

5.2.2

Create your own blueprint: Example Blueprint from our project

In the following is briefly shown how the provided template code can best be adapted to our own projects. For this purpose, all relevant aspects have been marked and described how they were filled based on an Arduino based end-node (the schematic can be transferred to the blueprint of the M3 board in the same way). As we can see, it is often sufficient to exchange the values, since the keys can be retained. If different or additional information is needed for individual projects, it can be found either in the previous overview of the Blueprint architecture or in more detail in the official document. Furthermore, the blueprints of the [M3Board](#) and the [Arduino](#) can be found in the appendix.



```

{
  "version": "1.0",
  "typeEui":
  "70b3d567700ff00230",

  "meta": {
    "name": "Temperature
    Sensor",
    "vendor": "Fraunhofer
    Mioty Demonstrator"
  },
  "uplink": [
    {
      "id": 0,
      "crypto": 0,

      "payload": [

        {
          "name": "temperature",
          "component":
            "16bitTemperature"
        },
        {
          "name": "humidity",
          "component":
            "16bitHumidity"
        }
      ]
    }
  ],
}
```

Insert the TypeEui:

- Here the Blueprint must be assigned a unique ID (TypeEui) from the vendors IEEE EUI64 range, allowing the data to be interpreted later in the base station using it.
- the TypeEui is always 16 characters long

Insert Metadata if required:

- Metadata allows to send even more project-specific and individual information
- They are optional, which means that the meta block can be omitted completely.
- Metadata are for example useful to better distinguish many different end nodes with different functions and payloads or to transmit further information which is not defined by default.
- They cannot be read out directly in the base station, but e.g. later in an MQTT client.

Define data components:

- When defining and describing the components, it is very important to exactly pay attention how and in which order the data are arriving in the base station
- It is also particularly important that the order of the data is maintained, otherwise the data will be misinterpreted. If we send the temperature data first, these must also be defined as the first component!
- The form of the data is substantially defined by the programming of our end nodes
- in our case we want to send temperature and humidity data of a sensor
- since these data are processed and interpreted differently, we create a separate component for each of them
- the exact definition of the component based on its attributes is located as a reference using the name in the component description

Correct representation of data on the Base station with the help of blueprints

```
"component": {
  "16bitTemperature": {
    "size": 16,
    "type": "int",
    "func": "$/100",
    "unit": "°C",
    "littleEndian": false
  },
  "16bitHumidity": {
    "size": 16,
    "type": "int",
    "func": "$/100",
    "unit": "%",
    "littleEndian": false
  }
}
```

Specify the data interpretation using the component attributes:

- With the help of the component description, the exact interpretation of the data can be determined
- The interpretation is based on how the data is processed in our end node and subsequently sent
- This in turn is determined by our node programming
- The attributes are therefore defined in our example as follows:

- Size: the size of our temperature and humidity data is 16 bits each.
- Type: the provided function to send the data in the code for the nodes always needs integer, which is why we specify this here
- Func: since our node can only send integers, the sensor data had to be converted beforehand. To get the original values and undo the preprocessing, we divide the raw data by 100 to get floating point numbers again.
- Unit: here we only have to specify the respective unit as string, in our case °C for temperature and % for humidity
- littleEndian: because we want to interpret our data in the order they are sent, we set this value to false
 - ➔ in sections [3.1.2.2](#) and [3.2.1](#) you can find again the whole code of the end nodes, from which the above attribute values can be derived
 - ➔ for an illustration of a possible data interpretation with the help of a blueprint see section [5.1.4](#)

5.3 Adding and Using Blueprints in the Base Station Application Center



The final step before the functionality of the Blueprints can be fully utilized is to add them to the application center of our base station and then assign them to our end nodes. However, before this is tackled, a few points should be considered first:

- In order to access the web interface of the base station, our AVA gateway must first be put into operation (see section [4.2](#))
- The correct transmission of the received data should be checked before adding a Blueprint (for more information on how to check the data see section [4.5](#)).



- An individual TypeEUI is required for each Blueprint
- Blueprints should already been adapted or created in advance to the corresponding end nodes and their data (for more information on how to create a blueprint see section 5.2).
- TypeEUIs of nodes that have already been added may have to be updated (for more information about TypeEUIs see section 5.1.3)

Correct representation of data on the Base station with the help of blueprints

If basic information and background on Blueprints is required, this can be found in section 5.1. In the following, it will be explained step by step how to add or edit already completed Blueprints to our AVA Gateway, how to use them and finally how to assign them to a Blueprint and check their correct functionality.

5.3.1

Adding a Blueprint to the Base Station

Before we can add a Blueprint, we need to open our web browser and access the web interface of the base station through `ava/` or with the corresponding local IP-address (e.g. `ava/or` just `192.168.178.168`), which can be found in the router settings as soon as the base station is connected.

We should then see the *Dashboard Overview* as in Figure 24:

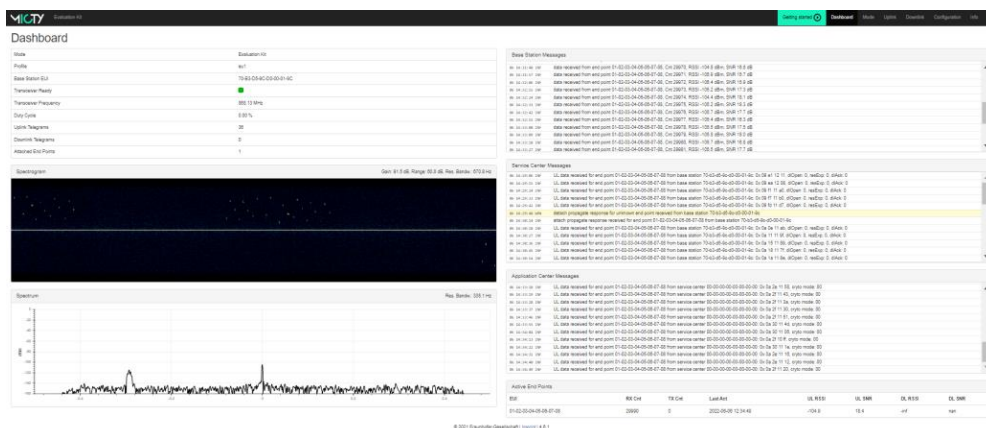


Figure 24: Dashboard Overview in the basestation web interface

To add a new Blueprint, we next navigate to the *Configuration* tab ① as in Figure 25:

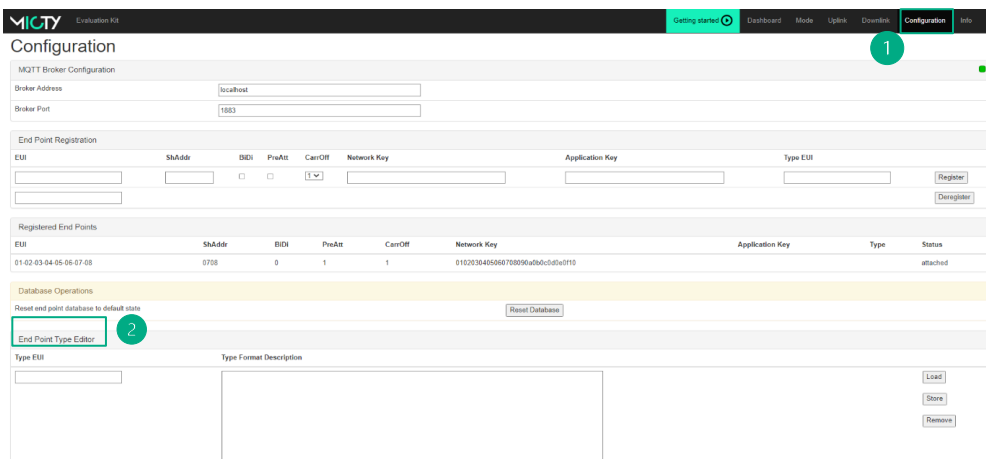


Figure 25: Configuration Tab in the basestation web interface

Under the entry "End Point Type Editor" ② shown in *Figure 26* all necessary components can be filled in:

- ③ A unique TypeEUI for our Blueprint, which we can later assign to our end nodes and thus interpret their data
- ④ and the actual description.

Afterwards, everything is confirmed with the *Store* button ⑤. If necessary, old Blueprints can also be viewed here with *Load* and easily edited ⑥ or removed with *Remove* ⑦.

Correct representation of data on the Base station with the help of blueprints

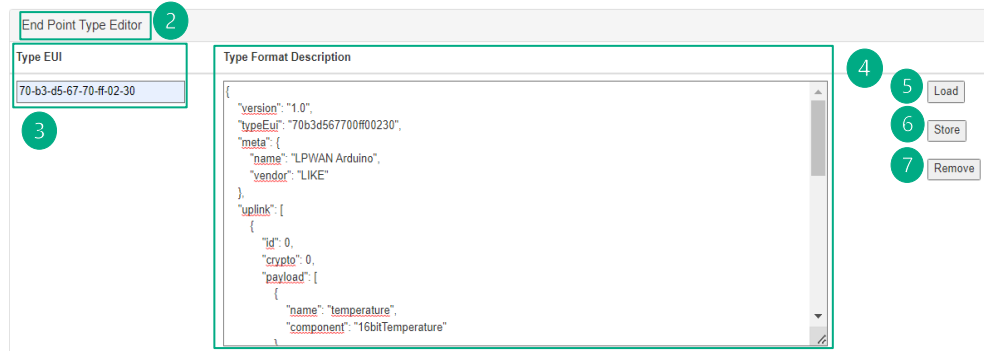


Figure 26: End Point Type Editor in the base station web interface

If the Blueprint was saved correctly, the message "Type format stored" should appear below. The Blueprint is then available in the database and can be used.

5.3.2

Assigning a Blueprint to an End Node

Before assigning a Blueprint to an end node, it is recommended to first check the correct transmission of the data in order to simplify any troubleshooting that may be necessary in the event of incorrect data representation. If the data is displayed incorrectly after adding the Blueprint, this is therefore due to the specification of our data format description.



Looking again at our *Dashboard*, it should appear as in *Figure 27*:

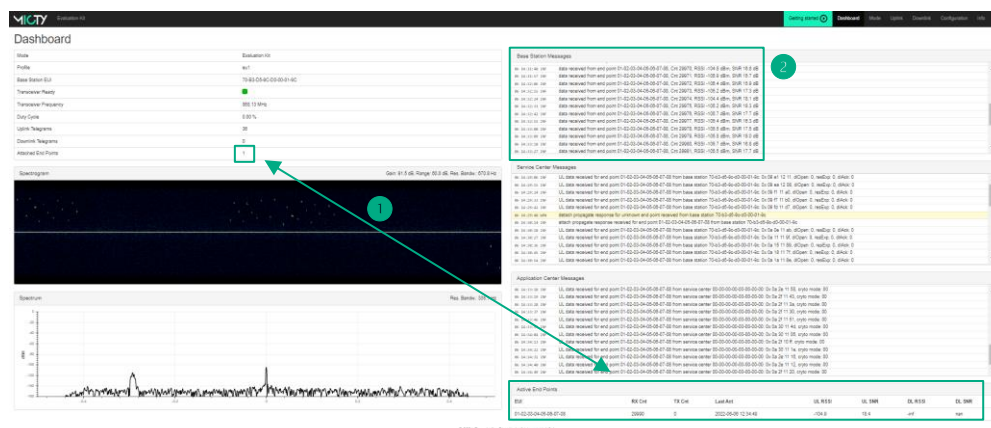


Figure 27: Dashboard Overview with data reception

As can be seen here, one node is currently registered at the base station and is sending data①. They are received by the base station②.

Correct representation of data on the Base station with the help of blueprints

As Figure 28 illustrates, the data is currently still in its raw form and not interpreted③ because the corresponding end node has not yet been assigned to a TypeEUI, respectively a Blueprint④ (the most recent record can be read in the Uplink tab⑤):

MICTY

Evaluation Kit

Getting started

Dashboard

Mode

Uplink

Download

Configuration

Info

Uplink

Received Uplink Data

EUI	Cnt	Time	RSSI	SNR	Type	Data
01-02-03-04-05-06-07-08	30464	2022-06-06 13:25:15	-101.4	13.6		0x 0a 13 0f c7

© 2021 Fraunhofer-Gesellschaft | Imprint | 4.6.1

Figure 28: Data reception in the uplink page without Blueprint

To change this, it is first necessary to switch back to the *Configuration* tab. In order to assign a Blueprint to an already existing end node, the data can simply be entered again and the TypeEUI of the desired blueprint needs to be added to the corresponding field (as if a new node was added, see section 4.3 for more information⑥) and saved using the Register Button⑦. The node will then be updated automatically⑧. See Figure 29 for an overview of the respective interfaces.

End Point Registration

EUI	ShAddr	BIDI	PreAtt	CanOff	Network Key	Application Key	Type EUI	
01-02-03-04-05-06-07-08		<input type="checkbox"/>	<input checked="" type="checkbox"/>	1	0102030405060708090a0b0c0d0e0f10		70-b3-d5-67-70-ff-02-30	Register

☒ End point registered

Registered End Points

EUI	ShAddr	BIDI	PreAtt	CanOff	Network Key	Type	Status
01-02-03-04-05-06-07-08	0708	0	1	1	0102030405060708090a0b0c0d0e0f10	70-b3-d5-67-70-ff-02-30	attached

Database Operations
Reset end point database to default state

Figure 29: Interface for assigning a new blueprint to a node

Now the node should be assigned a type⑨ and the correctly interpreted data⑩ should be displayed in the uplink tab as seen in Figure 30:

MICTY

Evaluation Kit

Getting started

Dashboard

Mode

Uplink

Downlink

Configuration

Info

Uplink

Received Uplink Data						
EUI	Cnt	Time	RSSI	SNR	Type	Data
01-02-03-04-05-06-07-08	30976	2022-06-06 16:07:55	-114.6	13.5	70-b3-d5-67-70-ff-02-30	humidity: 49.56 %, temperature: 25.47 °C

© 2021 Fraunhofer-Gesellschaft | Imprint | 4.6.1

Figure 30: Data reception in the uplink page with Blueprint

5.3.3

Troubleshooting

- if the data is still displayed as hexadecimal numbers: it should be checked if the TypeEUI has been assigned correctly and e.g., no typing errors have slipped in
- if the data is displayed incorrectly (but converted): either the code for the blueprint should be checked again for coding errors or the blueprint/ TypeEUI the node is assigned to might be wrong and therefore its data is not interpreted correctly.

Since even small spelling mistakes or mix-ups lead to problems, both the Blueprint code and the TypeEUIs should be created and described with care.

Correct representation of data on the Base station with the help of blueprints

Using the AVA Gateway web interface, first sensor data of the nodes can already be read out (see section 4.4). For many applications and projects, however, storage, further processing, visualization, or intelligent coordination of individual data points is required or desirable. For this purpose, IoT platforms are mostly used. They represent an important tool within the data control of the IoT network (for a complete overview of the data chain see section 2.2.1), for example, by capturing and storing all data and then processing it appropriately using versatile features. They allow not only to connect physical with virtual objects, but also to create an interface to possible application programs. In our MIOTY™ project, the open-source IoT platform [ThingsBoard](#) is used for this purpose (as an alternative open-source platform check out [Grafana](#) for example). As with most IoT platforms, the data exchange takes place via the so-called MQTT protocol.

A general overview of the data transfer is given in section 6.1. How ThingsBoard can be used as an IoT platform, and all the necessary steps are explained in sections 6.2 and 6.5. Since a basic understanding of the MQTT protocol is necessary for the implementation, it is introduced in section 6.3. This is followed by a description of the actual implementation of the MQTT functions in section 6.4.

6.1

Data transfer to ThingsBoard

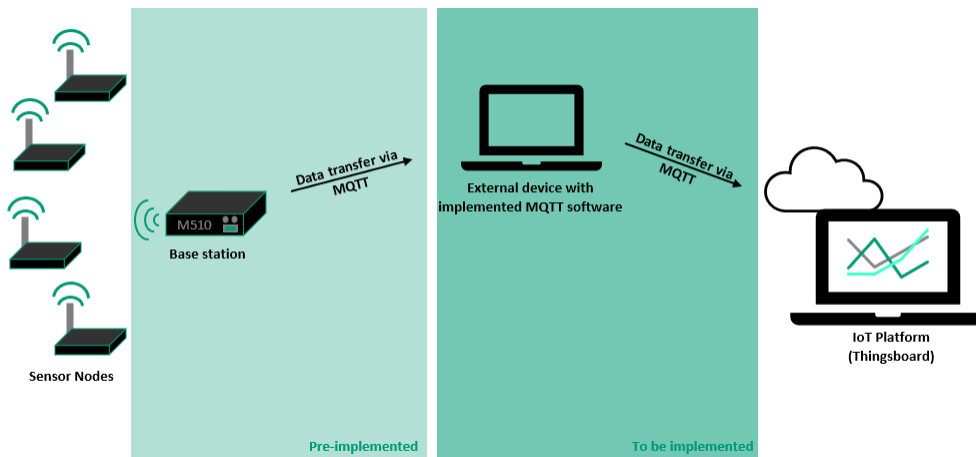


Figure 31: Data transfer structure to ThingsBoard

Several steps are necessary to transfer the data from the base station to the IoT platform ThingsBoard using the MQTT protocol. As *Figure 31* shows, this requires several MQTT-capable devices that are adapted to each other. Some of these elements are already pre-implemented by the base station, whilst others still need to be realized. For our project, a further external device has to be integrated into the data chain and equipped with appropriate software to use MQTT functions. The reason for this specific structure is illustrated in section 6.3.2.5.

6.2

The IoT Platform ThingsBoard

Data transfer via MQTT and display on the IoT platform



IoT platforms often represent an important tool within the IoT network as a data management and control tool as well as an interface to applications (compare the introduction section 6). The open-source IoT platform ThingsBoard used for our MIOTY™ project allows an easy integration and use in the own project due to its diverse features, well-documented tools and application examples. As a no-code platform, it also enables uncomplicated and intuitive use and configuration based on its graphical user interface and thus offers successfully usable application software even without programming skills. Furthermore, with the help of clear tutorials and instructions, even inexperienced users can exploit the possibilities of the platform. In addition to the reliable collection and storage of data, ThingsBoard allows for customizable processing and a vivid presentation of data using predefined visualization tools. Flexible and customizable configurations of settings, functions and tools allow the platform to be used in a variety of projects. A first overview of the platform can be found [here](#).

6.2.1

Using the Platform

To use the platform and view your first data, a ThingsBoard Cloud Maker account is required. This is free for the first month (see [here](#)). It is also possible to do a free on-premise installation, a corresponding tutorial can be found [here](#). Under the different menu tabs of this page further help like documentations or guides to different topics are available as well.

6.2.2

Adding new devices

In order to be able to send data to ThingsBoard, each node requires a provision via which it authenticates itself to the platform. It is obtained when registering a new device:

Under the *Device Groups* tab^① and the *All* submenu^②, new nodes can be added via the *+ icon*^③. In the window that opens, the name and other data can be entered^④ and added using the *add button*^⑤ (see *Figure 32*).

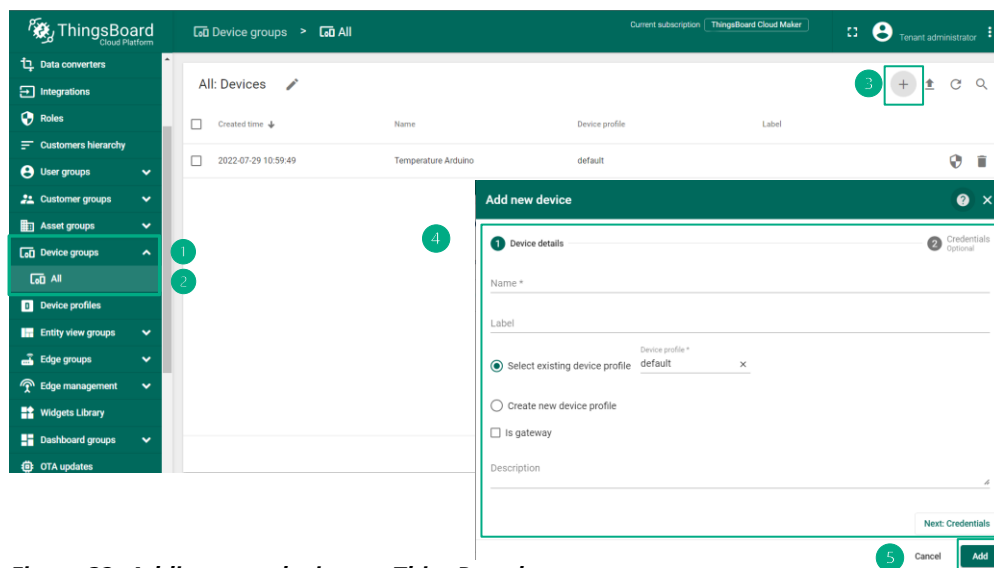


Figure 32: Adding new devices to ThingBoard

After adding the new device, the token can be retrieved as seen in *Figure 33*:

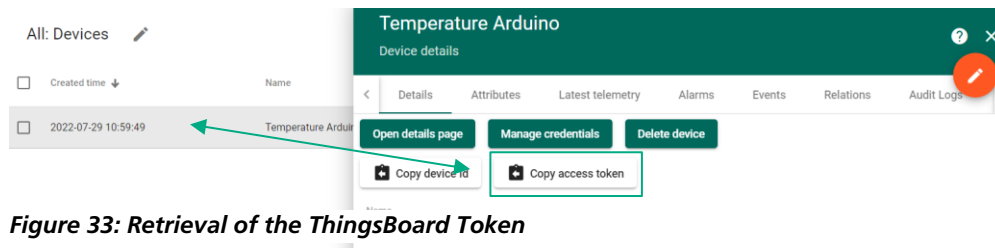


Figure 33: Retrieval of the ThingsBoard Token

This token is very important, since no data can be sent without it (it should therefore be well saved). It will be part of the MQTT message later, together with the payload. See section

6.4.1.2. For further detailed instructions, see also the [following page](#).

6.3 Introduction to MQTT



The MQTT (Message Queue Telemetry Transport) protocol defines how messages and data are exchanged on the Internet between IoT network participants such as embedded devices, sensors, industrial PLCs or IoT platforms, so that communication can take place. In terms of our MIOTY™ project, it is used to transport our (sensor) data from the application center of the base station to the server of the IoT platform. Here, in a further step if needed, the data can then be collected, processed, and visualized in a more graphically appealing way.

The following section is therefore intended to provide a brief introduction to the messaging protocol and to provide information on the main functions and concepts, relevant for the setup of the MIOTY™ project. More comprehensive information, tutorials, deeper explanations, and links can be found on the official [MQTT website](#).

6.3.1

Why do we use MQTT?

MQTT was originally developed in 1999 for monitoring an oil pipeline. Since the circumstances in mobile communication were very challenging during that time, the protocol had to be adapted in a correspondingly robust way. And it is precisely these principles, on which the protocol is still based today, that make it so suitable for communication on the Internet of Things. It features simple implementation and resource-efficient use of bandwidth and energy, which makes it particularly well suited for small, distributed devices with limited bandwidth and energy. It also provides reliable communication across unstable networks and constrained environments, allowing many devices to be networked together. This makes it very well suited for the Internet of Things and is part of the reason why it is so popular and the most widely used messaging protocol for IoT.

6.3.2

What is MQTT?

MQTT is a client/server model and designed as an extremely lightweight publish/subscribe messaging transport. The communication is based on a publish and subscribe system. Devices can publish messages to a specific topic and all devices subscribed to that topic receive the message. The message management itself is handled by an intermediary server, the so-called broker. The main applications include sending messages to control outputs and reading and publishing data from sensor nodes. Beside this basic functionality, MQTT offers further features, such as [Quality of Service Levels](#), [Persistent Session and Queuing Messages](#), [Retained Messages](#), [Last Will & Testament](#) and [Keep Alive & Client Take-Over](#).

Data transfer via MQTT and
display on the IoT platform

6.3.2.1

Publish/ Subscribe Architecture

In a publish and subscribe system, IoT devices can either send a message on a specific topic (publish) or receive messages on a specific topic (subscribe). There are devices that are designed to do both, but only one function can be performed at a time. It is possible for several clients to publish messages to one topic and for multiple clients to be subscribed to the same topic. This model offers the great advantage that participating devices do not have to communicate directly with each other or even know of each other's existence. Only the respective topic and, if necessary, an access token for authentication at the broker must be known. The connection and all other functions and tasks are handled by the server/broker. For more information refer to [here](#)

6.3.2.2

Client/ Broker Model

Client: A client can be any device that runs a client implementation of the MQTT protocol and connects to an MQTT broker via the Internet. Both publisher and subscriber fall under this category. The designation is simply a result of whether the client is currently publishing a message or receiving a subscribed message.

Broker: The broker/server is the heart of the MQTT protocol. It is the central node that connects all clients. It receives all messages and then has the task of filtering and forwarding them so that each client only receives the messages to which it has subscribed. It also authorizes and authenticates all clients that want to publish or receive messages to it. For more information about the Client/ Broker concept refer to [here](#).

6.3.2.3

Topics

Topics can be used to identify individual messages. They can be understood as a kind of subject of the message. A client can either publish a message for a specific topic or subscribe to messages from individual topics. These can then be filtered and forwarded by the broker accordingly. Topics are hierarchically structured similar to URLs and consist of several levels, which are separated from each other with a forward slash. By specifying the exact levels of the respective topics, the broker knows exactly which message a client wants to receive. Here is a short example:

Temperature/Living Room/Sensor 1
Temperature /Living Room/Sensor2
Temperature /Bedroom/Sensor1

The example above shows how the Topics of 3 different sensors could look like, with which one would like to measure for example the temperature at different places in the house. The topic defines exactly which data is involved and you can, for example, only

read out the data from the sensor in the bedroom if you only subscribe to the lowest topic.

Data transfer via MQTT and
display on the IoT platform

If you want to subscribe to more than one topic at the same time, you can use the so-called wildcards. These can be divided into two types, namely single- and multi-level. A single-level wildcard replaces one topic level and is represented with a + symbol. A multi-level wildcard can cover many topics, is represented with a # symbol and is always at the end of a topic. If a client is subscribed to a topic with a multi-level wildcard, he will receive all topics that match the levels up to the wildcard. This example illustrates these functions once again

Temperature /+/Sensor1
Temperature /Living Room/#
Temperature / #

The first topic uses a single-level wildcard. Here the client wants to receive all messages from Sensor1 clients, regardless of the room. With the second topic, only the data from the living room should be subscribed. With the third example, the client would receive messages from all topics, since neither the sensor nor the room matter. Further information can be found [here](#).

6.3.2.4

Summary Example

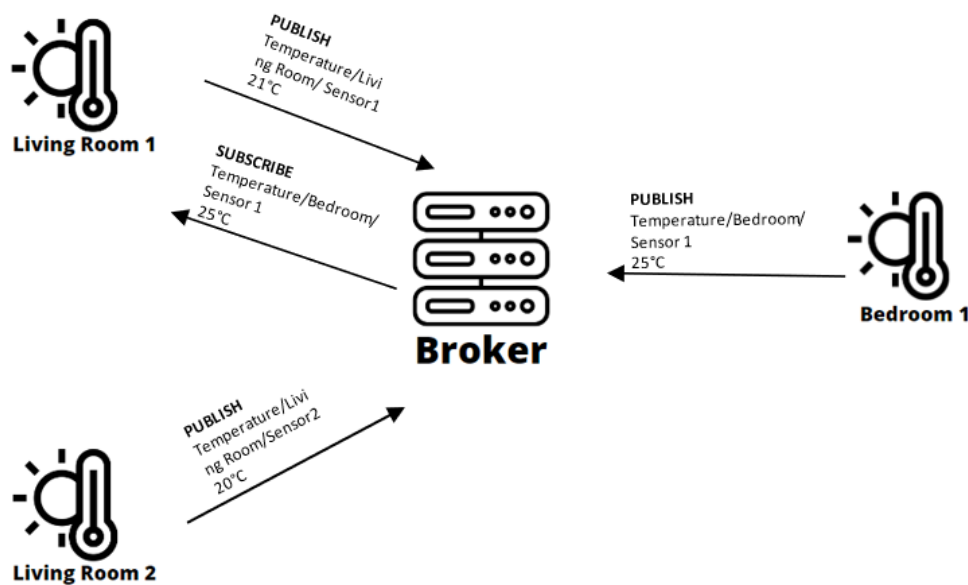


Figure 34: Typical MQTT Architecture

The example in *Figure 34* shows how a typical MQTT data exchange could look like. There are three clients/sensors, each of which sends the temperature of their location to the broker with the corresponding topic. From there, they can be subscribed to, which is what client Living Room 1, does. It publishes its own data on the one hand, but also receives all messages from the sensor in the bedroom.

6.3.2.5

MQTT and our MIOTY™ Project

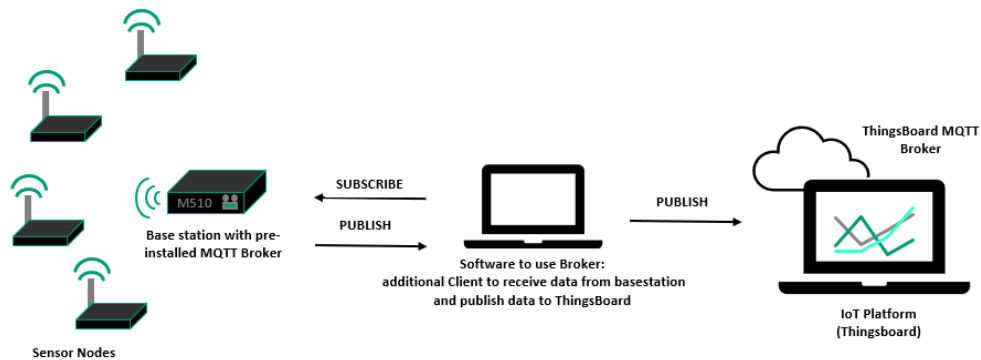


Figure 35: MQTT Architecture in the MIOTY™ Project

As shown in *Figure 35*, our project requires two brokers to transport the data from the Base Station (originally from our sensors) to our IoT platform ThingsBoard. The first broker runs on the Base Station and is already pre-implemented. It receives the sensor data from our end nodes using MIOTY™ technology and stores it temporarily. It then publishes the data on a given topic. But since the pre-defined topics have a different format than our IoT platform requires, a direct subscription from ThingsBoard is not possible and the topics must be converted correctly beforehand. In addition, the data format of the Base Station and ThingsBoard also do not match and must be adjusted (otherwise the data would be misinterpreted).

This is done with a software that includes two separate clients, one which first subscribes to the base station data and then converts it to the correct format. The other client then publishes the data to ThingsBoard via the second (ThingsBoard) broker with the adjusted topic. The software for this implementation can either run on a separate PC, on a microcontroller such as a Raspberry Pi or on the MIOTY™ Base Station itself, if the login information is available. For a more detailed overview of MQTT in our project, see the corresponding implementation chapters [6.4.1](#) and [6.4.2](#).

6.4

Implementation of the MQTT Clients...

As illustrated in sections [6.1](#) and [6.3.2.5](#), the data from the base station cannot be transmitted directly to our IoT platform via MQTT due to format and topic incompatibilities. This requires an additional client that receives the data from the base station and an additional client/ server that then sends it correctly converted to ThingsBoard (for a more detailed overview of the MQTT Protocol and how it works, see chapter [6.3](#)).

Several open-source programs and brokers such as [Eclipse Mosquitto](#), [MQTT Explorer](#) or the [Paho Python client](#) are already available for the implementation of this software. In this tutorial, the realization of the MQTT clients functions using the Paho Python Client

and a combination of the Paho Client and Eclipse Mosquitto in Linux is presented. The appendix furthermore contains the [complete codes](#) and a short overview of its [most important functions](#).

Data transfer via MQTT and display on the IoT platform

6.4.1

... using the Paho Python Client

The following steps describe how to use the Paho Python client for the MIOTY™ project requirements.

6.4.1.1

Preparatory Steps



In order to execute and develop the software for the MQTT applications successfully, a few preparatory steps and considerations are necessary.

- **Install a suitable IDE**

It is advisable to use a suitable IDE (development environment). Among the open-source solutions, there are several possibilities such as [KDevelop](#), [Spider](#), [PyCharm](#) or Microsoft's [VisualStudioCode](#), which is used in this project not only due to very helpful intext or extension functions and code navigation, but also because of its Github integration. Respective user and installation guides for the IDEs can be found on the corresponding pages. To use **Python in Visual Studio Code**, see the following steps (more detailed Information can be found [here](#)):

- Install [Visual Studio Code](#)
- Install the [Python Extension](#)
- Install a [Python Interpreter](#) (should not be necessary for Linux based operating systems due to the pre-installation of Python 3)

For a linux based operating system a simple text editor like vim is also sufficient (see section [6.4.2](#)).

- **Install the required Client**

For the implementation of the MQTT Client/ Server, our MIOTY™ project uses the [Paho Python client](#) from the [Eclipse Paho Project](#). This freely available open-source implementation, with detailed documentation that includes tutorials, examples, and other support, makes it easy to implement MQTT functions for various programming languages. In this project, Python is used for this purpose (a good introduction to this coding language can be found, for example, on the interactive websites [learnpython.org](#) or [W3School](#)). To be able to use all functionalities, the library must be integrated (the instruction refers to VSC, for more information see [here](#)):

- Access the VSC Terminal via *Terminal < New Terminal* (a new Power Shell based Window should open). Linux-based operating systems can also use the console as usual.
- Make sure pip¹ is installed with the following command:

```
$ pip --version
```

(if pip is not found then see [this page](#) for the installation)
- Run the following command to install the paho client:


¹ Pip is the standard package manager for Python. It allows to install and manage additional packages that are not part of the standard library. For more information see [here](#).

```
$ pip install paho-mqtt
```

Data transfer via MQTT and
display on the IoT platform

6.4.1.2

The Program Code

For the [complete code](#) together with all [relevant functions](#) see the appendix. Here you will also find a program for just receiving and reading out the data from the base station ([MQTT Client](#)). Basic information and help for the Paho Python client can be found in the corresponding [documentation](#). 

First some necessary modules must be imported for the correct functionality of the software:

```
# allows to use features from possibly higher Python versions by
# backporting them into the current interpreter
from __future__ import print_function

# import client class, necessary for important paho functions and
# features
import paho.mqtt.client as mqtt

# import server class, necessary for important paho functions and
# features
import paho.mqtt.publish as publish

# if required, further security functions such as encryption of the
# connection or identity verification can be added
import ssl

# Allows among other things to decode Jason coded strings
import json
```

This is followed by some variables for easy customization of node-specific data: the devices EUI and the ThingsBoard token (is obtained when adding the nodes to ThingsBoard, see section [6.2.2](#)), which can later be used to verify the nodes at the brokers (the code contains sample data):

```
# Put in your Mioty-EUI and ThingsBoard Token
# Format: [{"EUI1","Token1"}, {"EUI2","Token2"}, {"EUI3","Token3"}]
euiTokenPair = [
    ["01-02-03-04-05-06-07-08", "n77wTHZcffg1jFZrPvqH"]
]
```

In addition, we define variables for the configuration of the MQTT transport for better customization. With the help of this data, the messages of the base station can be subscribed to and received:

- the IP address of the base station (under which the web interface can also be accessed)
- The Broker port (port 1883 is typically reserved for MQTT; this should also be visible in the configurations of the base station, see *Figure 36*):

Configuration

MQTT Broker Configuration	
Broker Address	localhost
Broker Port	1883

Data transfer via MQTT and
display on the IoT platform

Figure 36: Retrieving the broker port for the MQTT software from the configuration tab of the base station web interface

- the topic under which the broker of the base station publishes the messages (see also section 5.5.1 on page 9 in the Ava Gateway manual):

```
# mqtt configuration
miotyServer = "ava.fritz.box"      # Broker's IP address
miotyPort = 1883                  # port reserved to MQTT
path = 'mioty/+/+/uplink'         # topic, client should be
                                  # subscribed to (this topic
                                  # subscribes to all uplink
                                  # messages from the base station)
```

The Paho Python client works by means of a loop function that is executed permanently. It reads the receive and send buffer and processes all found messages. Depending on the message type, a corresponding callback function is triggered afterwards. The callbacks are predefined in their structure, but their functionality can be adapted to the corresponding tasks and applications of the program. This creates the overall functionality of the software.

For our MIOTY™ project two callback functions are relevant, which have to be defined:

- **on_message** (also see [the appendix](#))
In general, this function allows to receive messages. In our case, it has additionally been configured to convert these messages directly into the correct format and then forward them to another broker from where ThingsBoard can subscribe to them. For this, the message is first decoded from Json, the individual entries are stored and then assembled as a new string¹. After the node has been verified via the EUI, the new data string can be published using the `publish.single` function (for the description of the function parameters and their values, see the [this section](#) in the appendix):

```
# callback function to receive messages from broker and forward it
# correctly converted to ThingsBoard;
# Called when a message has been received on a topic that the client
# subscribed to without specific topic filter

def on_message(client, userdata, message):
    # new mioty telegram received
    print("received topic:", message.topic)
    #print message as Json string
    print(" ", message.payload)
    #decode Json string (see manual for uplink message format)
```

¹ To better understand how the data is (must be) converted, see the formats of the messages that the base station publishes (see section 5.5.2 on page 10 in the Ava Gateway Manual) or that Thingsboard receives (see the [following page](#)) .

```

fields = json.loads(message.payload.decode("utf-8"))

# search for correct EUI Token Pair
for sens in euiTokenPair:
    sensorEui = sens[0]
    sensorToken = sens[1]
    if(message.topic.find(sensorEui)>=0):
        print("Publish to thingserver:")

        # Assemble Data String: Convert Data into correct Format for
        # ThingsBoard
        first = True
        valueDict = fields['components']
        msg = ""

        #successively run through all data entries and rewrite them
        # in the correct format into a new variable msg
        for x in valueDict:
            if(first == False):
                msg += ","
            msg += "\"%s\":"%s"%(x, str(valueDict[x]["value"]))
            first=False
        msg = "{" + msg + "}"
        print(msg)

        # Publish converted message to ThingsBoard
        authDict = {'username': sensorToken }
        publish.single(topic="v1/devices/me/telemetry", payload= msg,
            qos=1, retain=True, hostname="mqtt.thingsboard.cloud",
            port=1883, keepalive=10, auth=authDict)
        print('sucessfully published')

```

Data transfer via MQTT and
display on the IoT platform

- **on_connect** (also see [the appendix](#))

This callback function is triggered as soon as the broker sends a confirmation after the client's connection request. It thus verifies the successful connection of client and broker (in our case with the base station) and enables subscribing to the topics:

```

# callback function, called when broker responds to connection request
def on_connect(client,userdata,flag,rc):
    print("connect")
# when connected, subscribe to topic(s) client wants to receive
miotyClient.subscribe(path)

```

Afterwards a new client object must be created to use the functions of this class, the corresponding descriptions must be assigned to the functions and a connection configuration must be defined:

```

# creates new client object
miotyClient = mqtt.Client("mioty")

# functions get assigned to the actual callbacks
miotyClient.on_connect = on_connect
miotyClient.on_message = on_message

# connects client to broker
miotyClient.connect(miotyServer, miotyPort, 60)

```

Finally, the loop function is started, and possible interrupts are defined, so that it can be exited if necessary:

```
try:
    #necessary to maintain network traffic flow and trigger the
    appropriate callback function
    miotyClient.loop_forever()

except KeyboardInterrupt:
    #interrupt by user stops loop and disconnects client
    miotyClient.loop_stop()
    miotyClient.disconnect()
```

Data transfer via MQTT and
display on the IoT platform

6.4.1.3

Executing the Program



As soon as the code is complete, the program can be started using the run button. This should produce the following output in the terminal, as shown in *Figure 37*:

```
received topic: mioty/00-00-00-00-00-00-00-00/01-02-03-04-05-06-07-08/uplink
b'{"baseStations":[{"bsEui":8121069422560412060,"rssi":-126.26846313476563,"rxTime":16596
04961957438267,"snr":6.289093017578125}], "cnt":41218,"components":{"humidity":{"unit":"%", "v
alue":46.49},"temperature":{"unit":"\xc2\xbcC", "value":26.21}}, "data": [10,61,18,41], "format"
:0, "meta":{"name":"LPWAN Arduino", "vendor":"LIKE"}, "typeEui":812106919333244464}'
Publish to thingsserver:
{"humidity":46.49,"temperature":26.21}
successfully published
```

Figure 37: Terminal Output by correct execution of the MQTT software

With the help of print commands in the code, individual data such as node EUIs, original and converted sensor data as well as generally the achievement of certain code segments can be displayed. This facilitates necessary configurations or debugging.

If a MIOTY™ device sends a message every few minutes, it's perfectly fine to establish a new connection every time. If the device is very active, it might be worth considering rewriting the code to maintain a stable connection to ThingsBoard.

Under the *Latest Telemetry* tab^① of the corresponding node, the current values should also be available in ThingsBoard (see *Figure 38*):

Temperature Arduino			
Device details			
Details	Attributes	Latest telemetry	Alarms Events Relations Audit Logs
Latest telemetry			
<input type="checkbox"/>	Last update time	Key ↑	Value
<input type="checkbox"/>	2022-08-04 11:23:09	humidity	46.19
<input type="checkbox"/>	2022-08-04 11:23:09	temperature	26.3

Figure 38: Overview of the latest data under the "Latest Telemetry" tab



6.4.1.4

Troubleshooting

If error messages occur (most of the time they already give a concrete hint where the problem is) or the data cannot be transmitted correctly to ThingsBoard, see the following troubleshooting tips:

- With the help of the `on_log()` function of the Paho Python client, data and information about the MQTT connection and communication can be output. See [this section](#) in the appendix or the [following page](#) for more information.
- Console prints between sections allow output to determine where errors may be occurring in the code and the extent to which the program is executing.
- If the connection between broker and client fails, authentication (EUI, token) and connection data (IP addresses, ports) and topics should be checked.
- If the connection is established, but the messages are not or not correctly displayed in ThingsBoard, the format of the sent data should be checked again.

6.4.2

... using the Paho Python Client in connection with Eclipse Mosquitto



This approach shows another simple method for the client implementation of the MQTT software in Linux. Therefore, the software differs only in the function that publishes the converted data. In contrast to the previous chapter, the MQTT broker Mosquitto is used for this. Since this solution approach requires a Python script (that calls an external process within the Python code), it is presented with the help of a Linux operating system, since it is easier to implement. The following steps are necessary:

- **Installation of the required software:**
 - A program to open and edit the Python script, a simple text editor like vim is sufficient (see [here](#) for more information). However, corresponding IDEs can also be used as described in section [6.4.1.1](#).
 - the Paho Python Client, which is used to receive the data from the base station within the Python script (see [here](#) for more information).
 - the Eclipse Mosquitto Broker to implement the publish functionality and send the messages to ThingsBoard (see [here](#) for more information):

```
$ Sudo apt update
$ Sudo apt install vim
$ pip install paho-mqtt
$ Sudo apt install mosquitto
```

- **The Script Code**

For the [complete code](#) see the appendix. The following aspects differ from the implementation using the Paho Python Client (for the corresponding explanations of the overlapping code sections see section [6.4.1.2](#)):

First the `Os` module must be imported to get access to functions of the operating system to be able to execute external commands (e.g. running Mosquitto) later. In addition to importing the important modules for the software, a shebang line must also be inserted at the beginning of the script to define the interpreter location:

```
#!/usr/bin/python3
Import os
```

Besides that, only the code for publishing differs. After successfully receiving and converting the data into the correct format for ThingsBoard (as in [6.4.1.2](#)), the Mosquitto program is called and publishes the data under the correct topic. For this, the Mosquitto command is stored as a string in a variable and executed via the `Os` module in a subshell. For the exact configuration of the `mosquitto_pub` client see [this page](#):

Data transfer via MQTT and
display on the IoT platform

```
#Publish Data to ThingsBoard via mosquitto
mosCmd = "\
mosquitto_pub -d -q 1 \ -h \"mqtt.thingsboard.cloud\" -p
  \"1883\" \ -t \"v1/devices/me/telemetry\" -u \"%s\" -m
  {%s}\"%(sensorToken,msg)
print(mosCmd)
os.system(mosCmd)
print("\n\n")
```

- **Executing the script**

After the code is saved, it can be executed by its name using the following command:

```
python3 mqtt_server.py
```

The output should look similar to *Figure 39*:

```
received topic: mioty/00-00-00-00-00-00-00-00/01-02-03-04-05-06-07-08/uplink
b'{"baseStations":[{"bsEui":8121069422560412060,"rssi":-110.05772399902344,"rxTime":1659953774047433741,"snr":16.139236450195313}],{"cnt":45194,"components":{"humidity":{"unit":"%", "value":0.0},"temperature":{"unit":"\xc2\xbaC", "value":0.0}},"data":[0,0,0,0],"format":0,"meta":{"name":"LPWAN Arduino","vendor":"LIKE"},"typeEui":8121069193333244464}'
Publish to thingserver
mosquitto_pub -d -q 1 -h "mqtt.thingsboard.cloud" -p "1883" -t "v1/devices/me/telemetry" -u "n76wTHZcfftgljFZrPvqH" -m {"humidity":0.0,"temperature":0.0}
Client (null) sending CONNECT
Client (null) received CONNACK (0)
Client (null) sending PUBLISH (d0, q1, r0, m1, 'v1/devices/me/telemetry', ... (30 bytes))
Client (null) received PUBACK (Mid: 1, RC:0)
Client (null) sending DISCONNECT
```

Figure 39: Terminal Output using the Eclipse Mosquitto Broker

After receiving a MQTT message from the base station the script assembles the data into the correct format for ThingsBoard. Afterwards the program `mosquitto` executes a single publish. This includes a `CONNECT` operation followed by a `CONNACK` from ThingsBoard. Then the message gets `PUBLISHED` and ThingsBoard acknowledges it with `PUBACK`. Finally the connection is closed.

Now your Data should be visible on ThingsBoard (see [6.4.1.3](#) also). Under [6.4.1.4](#) further debugging help and tips can be found.

6.5

Create your own IoT-Dashboard: Data Visualization



ThingsBoard offers a variety of different and customizable widgets for visualizing and monitoring data. [This guide](#) shows how to create a dashboard and how to use and configure individual widgets (e.g. table, chart or alarm). Afterwards the dashboard of our MIOTY™ project could look like *Figure 40*):

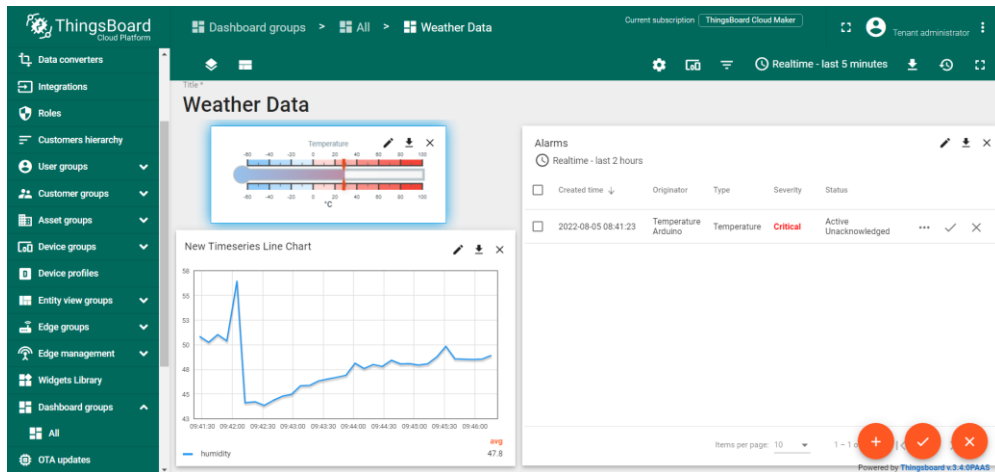


Figure 40: Example display of weather data on ThingsBoard Dashboard

Data transfer via MQTT and display on the IoT platform

Appendix

Here once again all complete codes, including their most important functions as well as blueprints of the MIOTY™ project can be found. All elements to be adjusted for an individual project are marked in red (for the MIOTY™ project only the EUIs, network keys and MQTT configurations have to be modified).

7.1 Blueprint for the MIOTY™ M3B magnoling MAKERBOARD

```
{
  "version": "1.0",
  "typeEui": "70B3D567700F0100",
  "meta": {
    "vendor": "Fraunhofer Mioty Demonstrator",
    "name": "M3B Sensor"
  },
  "uplink": {
    "id": 0,
    "payload": [
      { "name": "temperature", "type": "uint", "size": 16, "func": "$ / 10.0 - 273.15", "unit": "°C" },
      { "name": "humidity", "type": "uint", "size": 8, "unit": "%rel" },
      { "name": "pressure", "type": "uint", "size": 16, "func": "$ / 10.0", "unit": "hPa" },
      { "name": "luminosity", "type": "uint", "size": 16, "func": "$", "unit": "" }
    ]
  }
}
```

7.2 Blueprint for the Arduino Pro Mini sensor node

```
{
  "version": "1.0",
  "typeEui": "70B3D59CD0000095",
  "meta": {
    "name": "Temperature Sensor",
    "vendor": "Fraunhofer Mioty Demonstrator"
  },
  "uplink": [
    {
      "id": 0,
      "crypto": 0,
      "payload": [
        {
          "name": "temperature",
          "component": "16bitTemperature"
        },
        {
          "name": "humidity",
          "component": "16bitHumidity"
        }
      ]
    }
  ],
  "component": {
    "16bitTemperature": {
      "size": 16,
      "type": "int",
      "func": "$/10",
      "unit": "°C",
      "littleEndian": false
    },
    "16bitHumidity": {
      "size": 16,
      "type": "int",
      "func": "$/10",
      "unit": "%",
      "littleEndian": false
    }
  }
}
```

```
}
```

7.3 Code for the MIOTY™ M3B magnoling MAKERBOARD sensor node

```
/**
 * Hardware components on M3Bv2:
 *   SerialM3B   - PinHeader           - PA9, PA10, 9600baud 8N1
 *   SerialMioty - mioty module         - PC10, PC11, 9600baud 8N1
 *   adxl362     - 3 axis accelerometer - SPI, (CS-PA8, MISO-PA6, MOSI-PA7, SCLK-
 *   PA5, INT1-PA11, INT2-PA12)
 *   ms5637      - pressure sensor      - I2C Wire2(PB9, PB8)
 *   sht31       - temperature & humidity sensor - I2C Wire2(PB9, PB8)
 *   si1141      - light sensor         - I2C Wire2(PB9, PB8)
 *   RGB LED     - low active           - BLUE-PC6, LED-PC7, RED-PC8
 *   Status LED  - high active          - PC13
 */

// Libraries for Sensor use, MIOTY and M3 Board
#include <miotyAtClient.h>
#include "m3bDemoHelper.h"
#include "m3b_sensors/si1141.h"
#include <SHT31.h>
#include <SparkFun_MS5637_Arduino_Library.h>

//enable Serial Monitor
#include "SoftwareSerial.h"

// Debug Serial
SoftwareSerial SerialM3B(PA10, PA9);
// Mioty Bidi Stamp Serial
SoftwareSerial SerialMioty(PC11, PC10);

//enable I2C
#include <Wire.h>

// Declarations
TwoWire Wire2(PB9, PB8);

MS5637      ms5637;
SHT31       sht31;
SI1141      si1141;

M3BDemoHelper m3bDemo;

//Node specific configurations
// input new EUI
uint8_t eui64[8] = {0x70, 0xb3, 0xd5, 0x67, 0x70, 0x11, 0x14, 0x47};
// input new Network Key
uint8_t nwKey[16]={0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x01, 0x0a, 0x0b,
0x0c, 0x0d, 0x0e, 0x0f};

//Conditional compilation to set the network key once at the beginning, should be executed
only ONCE and commented afterwards
#define SET_NETWORKKEY

void setup() {
  // put your setup code here, to run once:

  m3bDemo.begin();
  SerialM3B.begin(9600);
  SerialMioty.begin(9600);

  //initialize sensors
  Wire2.begin();
  if (ms5637.begin(Wire2) == false)
  {
    SerialM3B.println("MS5637 sensor did not respond. Please check wiring.");
    while(1);
  }
  sht31.begin(0x44, &Wire2);
  si1141.begin(&Wire2);

#ifdef SET_NETWORKKEY
  // assign (new EUI and) Network Key
  uint8_t MSTA; // status of mac state machine

```

```

// Local Dettach
SerialM3B.print("Local Dettach:");
miotyAtClient_macDetachLocal(&MSTA);

// Set-EUI
// SerialM3B.print("Set EUI");
// miotyAtClient_getOrSetEui(eui64, true);
// SerialM3B.println("");

// Set-Network Key
SerialM3B.print("Set Network Key");
miotyAtClient_setNetworkKey(nwKey);
SerialM3B.println("");

// Local Attach - required only once
SerialM3B.print("Local Attach:");
miotyAtClient_macAttachLocal(&MSTA);
SerialM3B.print("New Mac State:");
SerialM3B.println(MSTA);
SerialM3B.println("");
#endif

// get Device EUI
uint8_t eui64[8];
miotyAtClient_getOrSetEui( eui64, false);
SerialM3B.print("Device EuI ");
for (int i = 0; i < 8;i++) {
    SerialM3B.print(eui64[i], HEX);
    SerialM3B.print("-");
}
}

void loop() {
    // put your main code here, to run repeatedly:

    //send Weather data and visualize with blue LED
    digitalWrite(BLUE_LED, LOW);
    float temperature = sendWeatherData();
    digitalWrite(BLUE_LED, HIGH);

    delay(3000);
}

float sendWeatherData() {
    // readout of weather data; humidity in %, temperature in °C, pressure in hPa
    float pres=0., temp=0., hum=0.;
    temp = ms5637.getTemperature();
    pres = ms5637.getPressure();
    sht31.read();
    hum = sht31.getHumidity();
    uint16_t lux;
    si1141.readLuminosity(&lux);

    // Serial Monitor
    SerialM3B.println(" ");
    SerialM3B.print("Pressure [hPa] ");
    SerialM3B.println(pres);
    SerialM3B.println(pres * 10);
    SerialM3B.print("Temperature [°C] ");
    SerialM3B.println(temp);
    SerialM3B.println((temp + 273.15)* 10);
    SerialM3B.print("Humidity ");
    SerialM3B.println(hum);
    SerialM3B.print("Luminosity (rawData) ");
    SerialM3B.print(lux);

    // transmit Data
    m3bDemo.transmitWeather(pres, temp, hum, lux);

    delay(4000);
    SerialM3B.println("");
    return temp;
}

```

7.4 Overview of the most relevant functions of the Mioty_at_client_c and m3b_helper library

Below, only the functions relevant for the MIOTY™ project are presented. For a complete overview, see the corresponding Github pages for [Mioty_at_client_c](#) and [m3b_helper](#).

In general all functions of the Mioty_at_Client_c library return an element of the enumeration type `miotyAtClient_returnCode`, which is supposed to show current status messages of the system and also assist in debugging. For example, the return code 0 stands for the correct functioning of the node, whereas many others represent various error codes (for the complete table see the following [Github page](#)):

- `miotyAtClient_macDetachLocal(uint8_t * MSTA)`
Detaches the MAC locally and stores the return code in the function argument MSTA.
- `miotyAtClient_macAttachLocal(uint8_t * MSTA);`
Attaches the MAC locally and stores the return code in the function argument MSTA.
- `miotyAtClient_getOrSetEui(uint8_t * eui64, bool set);`
By bool either the EUI can be reset or read out. With a positive bool value, the eui64 is set, with negative the value is written to the address.
- `miotyAtClient_setNetworkKey(uint8_t * nwKey)`
Sets a new network key. Requires `miotyAtClient_macDetachLocal()` and subsequent `miotyAtClient_macAttachLocal()`.

With the help of the `m3b_helper` functions the onboard sensors can be accessed, and the data can be sent:

- `.begin()`
Initializes the onboard LEDs.
- `.transmitWeather(float pres, float temp, float hum, uint16_t lum)`
Sends weather sensor data by means of MIOTY™ technology. Attention should be paid to the order of the individual sensor data. Returns a `miotyAtClient_returnCode` as well.

7.5 Code for the Arduino Pro Mini sensor node

```
//Libraries for Sensor use and MIOTY
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>
//Import MIOTY TS-UNB-Node Library
#include <ArduinoTsUnb.h>

//enable Serial Monitor
#include "SoftwareSerial.h"

//enable I2C
#include <Wire.h>

//Sensor Declaration
Adafruit_BME280 bme;

//Node specific configurations
//input new EUI
#define MAC_EUI64 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08
//input new Network Key
```

```

#define MAC_NETWORK_KEY      0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
                             0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10
//input Transmit power in dBm, please keep in mind local regulations
#define TRANSMIT_PWR        10

using namespace TsUnbLib::Arduino;
// Select preset depending on TX chip, use Arduino PIN 8 for Chip Select
TsUnb_EU1_Rfm69hw_t<8> TsUnb_Node;

void setup() {
    //setup serial Monitor
    Serial.begin(9600);

    //Sensor init
    unsigned status;
    status = bme.begin();
    if (!status) {
        Serial.println("Could not find a valid bme280 sensor, check wiring, address, sensor
ID!");
        //while(1);
    }

    delay(100);

    // Blink the LED
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, HIGH);
    delay(100);
    digitalWrite(LED_BUILTIN, LOW);

    // Init the node and its parameters
    TsUnb_Node.init();
    TsUnb_Node.Tx.setTxPower(TRANSMIT_PWR);
    TsUnb_Node.Mac.setNetworkKey(MAC_NETWORK_KEY);
    TsUnb_Node.Mac.setAddress(MAC_EUI64);

    // TS-Unb ignores packets with an PkgCnt already received
    // We use this function to configure the PkgCnt from the
    // EEPROM
    TsUnb_Node.Mac.extPkgCnt = initExtPkgCnt();

    // Blink LED
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, HIGH);
    delay(1000);
    digitalWrite(LED_BUILTIN, LOW);
}

void loop() {

    // read sensor data and convert to integer
    //Temperatur
    float payload_sensor_t = bme.readTemperature();
    Serial.println(payload_sensor_t);
    payload_sensor_t *= 100;
    int16_t payload_transmitter_t = (int16_t)payload_sensor_t;

    // Humidity
    float payload_sensor_h = bme.readHumidity();
    Serial.println(payload_sensor_h);
    payload_sensor_h *= 100;
    int16_t payload_transmitter_h = (int16_t)payload_sensor_h;

    //Prepare payload for correct format and save to array
    uint8_t txdata[4];
    txdata[0]= (payload_transmitter_t>>8)& 0xFF; //shift bits 8 to the right and limit to 8
bit -> upper 8bit are stored in field, remaining zeros are cut off in front
    txdata[1]= (payload_transmitter_t)& 0xFF; // rear 8bit saved and front cut off
    txdata[2]= (payload_transmitter_h >>8)& 0xFF;
    txdata[3]= (payload_transmitter_h)& 0xFF;

    //data transmission via mity
    TsUnb_Node.send(txdata, sizeof(txdata));

    Serial.println(payload_transmitter_t);
    Serial.println(payload_transmitter_h);

    // We store the current PkgCnt to the EEPROM to avoid the repetition of packets with the
    same count. This value is only written every 256 packets to save energy.
}

```



```

updateExtPkgCnt(TsUnb_Node.Mac.extPkgCnt);

// Blink the LED twice to indicate end of transmission
pinMode(LED_BUILTIN, OUTPUT);
digitalWrite(LED_BUILTIN, HIGH);
delay(100);
digitalWrite(LED_BUILTIN, LOW);
delay(100);
digitalWrite(LED_BUILTIN, HIGH);
delay(100);
digitalWrite(LED_BUILTIN, LOW);

// Sleep the device for 5 seconds using the watchdog timer
delay(5000);
}

```

7.6 Overview of the most relevant functions of the ts_unb_node library

Below are all the functions of the ts_unb_node library relevant to the MIOTY™ project. A complete overview can be found [here](#).

Configuration

class TsUnb_Node to configure the Node:

- **.init()**
Initializes the node and the underlying Tx and Mac module. It should be called very early after the start-up of the program code to bring the transmitter into a defined state.
- **.Tx.setTxPower(TRANSMIT_PWR)**
Sets the transmit power in dBm. Should be coordinated with local regulations.
- **.Mac.setNetworkKey(const uint8_t k0, const uint8_t k1, const uint8_t k2, const uint8_t k3, const uint8_t k4, const uint8_t k5, const uint8_t k6, const uint8_t k7, const uint8_t k8, const uint8_t k9, const uint8_t k10, const uint8_t k11, const uint8_t k12, const uint8_t k13, const uint8_t k14, const uint8_t k15)**
Sets a new network key consisting of 16 (two-digit hexadecimal) numbers. Can be defined separately for convenience.
- **.Mac.setAddress(const uint8_t e0, const uint8_t e1, const uint8_t e2, const uint8_t e3, const uint8_t e4, const uint8_t e5, const uint8_t e6, const uint8_t e7)**
Sets a new EUI and automatically the ShortAddress of the node. Consists of 8 (two-digit hexadecimal) numbers and can be defined separately for convenience.

Packet counter related functions:

- **initExtPkgCnt();**
Initializes the extended Packet Counter using the EEPROM data and returns the current package number extPkgCnt. Used to avoid repetitive packets.
- **updateExtPkgCnt(uint32_t extPkgCnt, bool forceWrite = false)**
Updates the Extended Packet Counter in the EEPROM in regular intervals. Returns a boolean value if the update was successful.

Transmission

- **.send(const uint8_t* const payload, const uint16_t payloadLength, const uint8_t MPF_value = 0, const bool priority= false)**

This method transmits the requested payload data. It does the MAC and PHY encoding as well as the transmission of the data using the transmitter. Both the data and its length in bytes are passed to the function. In addition, the message can be prioritized by the bool variable.

7.7 Code for the Paho-based MQTT Client to receive Data

```
# allows to use features from possibly higher Python versions by backporting them into the
current interpreter
from __future__ import print_function

# imports client class, necessary for important paho functions and features
import paho.mqtt.client as mqtt

# if required, further security functions such as encryption of the connection or identity
verification can be added
import ssl

# allows among other things to decode Jason coded strings
import json

# mqtt configuration
server = "ava.fritz.box"                # Broker's IP adress
port = 1883
path = 'mioty/+/+/uplink'              # topic, client should be subscribed to

#callback function to receive messages from broker; Called when a message has been received
on a topic that the client subscribed to without specific topic filter
def on_message(client, userdata, message):
    print("received topic:", message.topic)          #print subscribed topic
    print(" ", message.payload)                     #print message as Json String
    fields = json.loads(message.payload.decode("utf-8")) #decode Json String

    # print information about base stations in corresponding representation or conversion
    print("  baseStations:")
    for i in fields['baseStations']:
        print("    bsEui:", hex(i["bsEui"]), ", rssi: ", round(i["rssi"],1), " dBm, snr:",
            round(i["snr"],1), " dB")

    if(fields['typeEui'] != 0):                      #check if Endpoint Type data available
        #print information about Endpoint
        print("  Endpoint Type:")
        print("    typeEui:", hex(fields['typeEui']))
    if(isinstance(fields['meta'], dict)):
        for x in fields['meta']:
            print("    ", x, ":", fields['meta'][x])

    #print Endpoint Data
    print("  Endpoint Data:")
    print("    raw:", fields['data'])
    valueDict = fields['components']
    if(isinstance(valueDict, dict)):
        for x in valueDict:
            print("    ", x, ":", valueDict[x]["value"], valueDict[x]["unit"])

    print("\n\n")

#callback function, called when broker responds to connection request
def on_connect(client,userdata,flag,rc):
    print("connect")
    client.subscribe(path)                      #when connected, subscribe to topic(s)
    client wants to receive

client = mqtt.Client()                          #creates new client object
# functions get assigned to the actual callbacks
client.on_message = on_message
client.on_connect = on_connect
# connect client to Broker
client.connect(server, port, 60)

try:
    client.loop_forever()                      #necessary to maintain network traffic flow and
                                              trigger the appropriate callback function
except KeyboardInterrupt:                    #interrupt by user stops loop and disconnects client
```

```
client.loop_stop()
client.disconnect()
```

Appendix

7.8 Code for the Paho-based MQTT Client to receive and publish Data

```
# allows to use features from possibly higher Python versions by backporting them into the
current interpreter
from __future__ import print_function

# imports client class, necessary for important paho functions and features
import paho.mqtt.client as mqtt

# imports server class, necessary for important paho functions and features
import paho.mqtt.publish as publish

# if required, further security functions such as encryption of the connection or identity
verification can be added
import ssl

# allows among other things to decode Jason coded strings
import json

# Put in your Mioty-EUI and ThingsBoard Token
# Format: [{"EUI1","Token1"}, {"EUI2","Token2"}, {"EUI3","Token3"}]
euiTokenPair = [
    ["01-02-03-04-05-06-07-08", "n76wTHZcfftjFZrPvqH"]
]

# mqtt configuration
miotyServer = "ava.fritz.box"          # Broker's IP address
miotyPort = 1883                      # port reserved to MQTT
path = 'mioty/+/+/uplink'             # topic, client should be subscribed to (this topic
                                     # subscribes to all uplink messages from the base
                                     # station)

# callback function to receive messages from broker and forward it correctly converted to
ThingsBoard;
# Called when a message has been received on a topic that the client subscribed to without
specific topic filter
def on_message(client, userdata, message):
    # new mioty telegram received
    print("received topic:", message.topic)          #print subscribed topic
    print(" ", message.payload)                     #print message as Json string
    fields = json.loads(message.payload.decode("utf-8")) #decode Json string (see manual
                                                         # for uplink message format)

    # search for correct EUI Token Pair (EUI of the sending node matches settings)
    for sens in euiTokenPair:
        sensorEui = sens[0]
        sensorToken = sens[1]
        if(message.topic.find(sensorEui)>=0):
            print("Publish to thingsserver:")

            # Assemble Data String: Convert Data into correct Format for ThingsBoard
            first = True
            valueDict = fields['components']
            msg = ""
            #successively run through all data entries and rewrite them in the correct format
            into a new variable msg
            for x in valueDict:
                if(first == False):
                    msg += ","
                msg += "\"%s\":"%(x, str(valueDict[x]["value"]))
                first=False
            msg = "{" + msg + "}"
            print(msg)

            # Publish converted message to ThingsBoard
            authDict = {'username': sensorToken }
            publish.single(topic="v1/devices/me/telemetry", payload= msg, qos=1, retain=True,
hostname="mqtt.thingsboard.cloud", port=1883, keepalive=10, auth=authDict)
            print('successfully published')
```

```
# callback function, called when broker responds to connection request
def on_connect(client,userdata,flag,rc):
    print("connect")
    miotyClient.subscribe(path)          # when connected, subscribe to topic(s) client wants
                                         # to receive

# debugging function
#def on_log(client, userdata, level, buf):
#    #print("log: ",buf)

# creates new client object
miotyClient = mqtt.Client("mioty")

# functions get assigned to the actual callbacks
miotyClient.on_connect = on_connect
# miotyClient.on_log = on_log
miotyClient.on_message = on_message

# connects client to broker
miotyClient.connect(miotyServer, miotyPort, 60)

try:
    miotyClient.loop_forever()            #necessary to maintain network traffic flow and trigger
                                         #the appropriate callback function

except KeyboardInterrupt:                #interrupt by user stops loop and disconnects client
    miotyClient.loop_stop()
    miotyClient.disconnect()
```

7.9 Code for the Mosquitto-based MQTT Client to receive and publish Data

```
# shebang line to define interpreter location
#!/usr/bin/python3

# allows to use features from possibly higher Python versions by backporting them into the
# current interpreter
from __future__ import print_function

# imports client class, necessary for important paho functions and features
import paho.mqtt.client as mqtt

# if required, further security functions such as encryption of the connection or identity
# verification can be added
import ssl

# allows among other things to decode Json coded strings
import json

# Access to operating system functions
import os

# Put in your Mioty-EUI and ThingsBoard Token
# Format: [{"EUI1","Token1"}, {"EUI2","Token2"}, {"EUI3","Token3"}]
euiTokenPair = [
    ["01-02-03-04-05-06-07-08", "n76wTHZcFfg1jFZrPvqH"]
]

# mqtt configuration
miotyServer = "ava.fritz.box"          # Broker's IP address
miotyPort = 1883                       # port reserved to MQTT
path = 'mioty/+/+/uplink'              # topic, client should be subscribed to (this topic
                                         # subscribes to all uplink messages from the base
                                         # station)

# callback function to receive messages from broker and forward it correctly converted to
# ThingsBoard;
# Called when a message has been received on a topic that the client subscribed to without
# specific topic filter
def on_message(client, userdata, message):
    # new mioty telegram received
    print("received topic:", message.topic)          #print subscribed topic
    print(" ", message.payload)                     #print message as Json string
    fields = json.loads(message.payload.decode("utf-8")) #decode Json string (see manual)
```

```

                                for uplink message format)

# search for correct EUI Token Pair (EUI of the sending node matches settings)
for sens in euiTokenPair:
    sensorEui = sens[0]
    sensorToken = sens[1]
    if(message.topic.find(sensorEui)>=0):
        print("Publish to thingserver:")

    # Assemble Data String: Convert Data into correct Format for ThingsBoard
    first = True
    valueDict = fields['components']
    msg = ""
    #successively run through all data entries and rewrite them in the correct format
    #into a new variable msg
    for x in valueDict:
        if(first == False):
            msg += ","
            msg += "\"%s\":"%(x, str(valueDict[x]["value"]))
            first=False
        msg = "{" + msg + "}"
    print(msg)

    #Publish Data to ThingsBoard via mosquitto
    mosCmd = "\
        mosquitto_pub -d -q 1 \ -h \"mqtt.thingsboard.cloud\" -p \"1883\" \-t
        \"v1/devices/me/telemetry\" -u \"%s\" -m {%s}\"%(sensorToken,msg)
    print(mosCmd)
    # execute command string in subshell
    os.system(mosCmd)
    print("\n\n")

# callback function, called when broker responds to connection request
def on_connect(client,userdata,flag,rc):
    print("connect")
    miotyClient.subscribe(path)          # when connected, subscribe to topic(s) client wants
                                         to receive

# creates new client object
miotyClient = mqtt.Client("mioty")

# functions get assigned to the actual callbacks
miotyClient.on_connect = on_connect
miotyClient.on_message = on_message

# connects client to broker
miotyClient.connect(miotyServer, miotyPort, 60)

try:
    miotyClient.loop_forever()           #necessary to maintain network traffic flow and trigger
                                         the appropriate callback function

except KeyboardInterrupt:              #interrupt by user stops loop and disconnects client
    miotyClient.loop_stop()
    miotyClient.disconnect()

```

7.10 Overview of the most relevant Functions of the Paho Client Class

Below are all the functions of the Paho Python Client relevant to the MIOTY™ projects implementation of the MQTT Clients. A complete overview and documentation can be found [here](#).

Loop Functions

The loop functions are the driving force behind the client. They detect the events that trigger the callback functions which represent the actual functionality of the software. Network data and message processing is not possible without them. For more information about the loop functions see the [following page](#).

- **.loop_start()/ .loop_stop()**
Starts or ends the background thread of the loop-function (this is necessary to execute asynchronous callback functions).
- **.loop_forever()**
Is executed permanently and processes network events (e.g., calls the corresponding callbacks when triggered). Can only be terminated with disconnect().

Connect and disconnect Functions

See the following documentation for [more information](#).

- **Connect(host, port, keepalive)**
By specifying the host name or its IP address, the corresponding port of the server host and the maximum communication time, the Paho client can be connected to a broker (in our case to the basestation broker). The corresponding callback function on_connect is triggered when the broker confirms the connection.
- **Disconnect()**
Permanently disconnects the client from the broker.

Callback Functions

Callback functions are called in response to an event. They are therefore not part of the actual code and can occur asynchronously at any time. To use a callback, the corresponding function that is to be triggered when the event occurs must first be defined and this function must then be assigned to the callback. Function parameters are passed automatically. For more information about callbacks in the Paho Python client, see [here](#).

- **on_connect(client, userdata, flags, rc)**
Called when the broker confirms the client's connection request. For debugging purposes the connection result can be retrieved using the rc parameters.
- **on_message(client, userdata, message)**
Triggered by the receipt of a message that the client has subscribed to. With the help of the class members topic, payload, qos and retain of the message function variable, information and send data of the received message can be read and processed within the callback function.
- **on_log(client, userdata, level, buf)**
Triggered when the client has log information and typically used for debugging. The level variable gives the severity of the message, the buf variable contains the message itself.

Publish Function

- **single(topic, payload=None, qos=0, retain=False, hostname="localhost", port=1883, keepalive=60, auth=None)**
This function allows publishing a single message to the broker and then disconnects. The following function variables are/ can be relevant for the MIOTY™ project (for [more information](#) on publish functions see the following page):
 - **topic**
The topic on which the payload is published. ThingsBoard receives all messages with the following topic: v1/devices/me/telemetry (see the [following page](#) under 'Telemetry Upload API'). For general information about topics see section 6.3.2.3.
 - **payload**

The payload to be published. It should be structured as follows:
`{"key1": "value1", "key2": "value2"}` or `[{"key1": "value1"}, {"key2": "value2"}]`
 (see [here](#)).

- **qos**
 The Quality of Service (QoS) to use when publishing. 1 guarantees that the message will arrive at least once. See [this page](#) for more information on the QoS in MQTT.
- **retain**
 Sets a message flag if the Broker should store the last retained message and QoS (retained = True). See [this page](#) for more information.
- **hostname**
 A string containing the address of the broker to connect to. ThingsBoards own Broker can be accessed via 'mqtt.thingsboard.cloud'.
- **port**
 The port to connect to the broker on, the default port for MQTT is 1883.
- **Keepalive**
 The time period in s during which the connection between client and server is maintained.
- **Auth**
 Authentication parameters for the client (in our case ThingsBoard). In order for ThingsBoard to receive the data, the corresponding node must first be authenticated via a token. This token is obtained when registering the node (see section [6.2.2](#)) The format is as follows:
`auth = {'username': "<username>", 'password': "<password>"}`.
 (Only the username is needed).