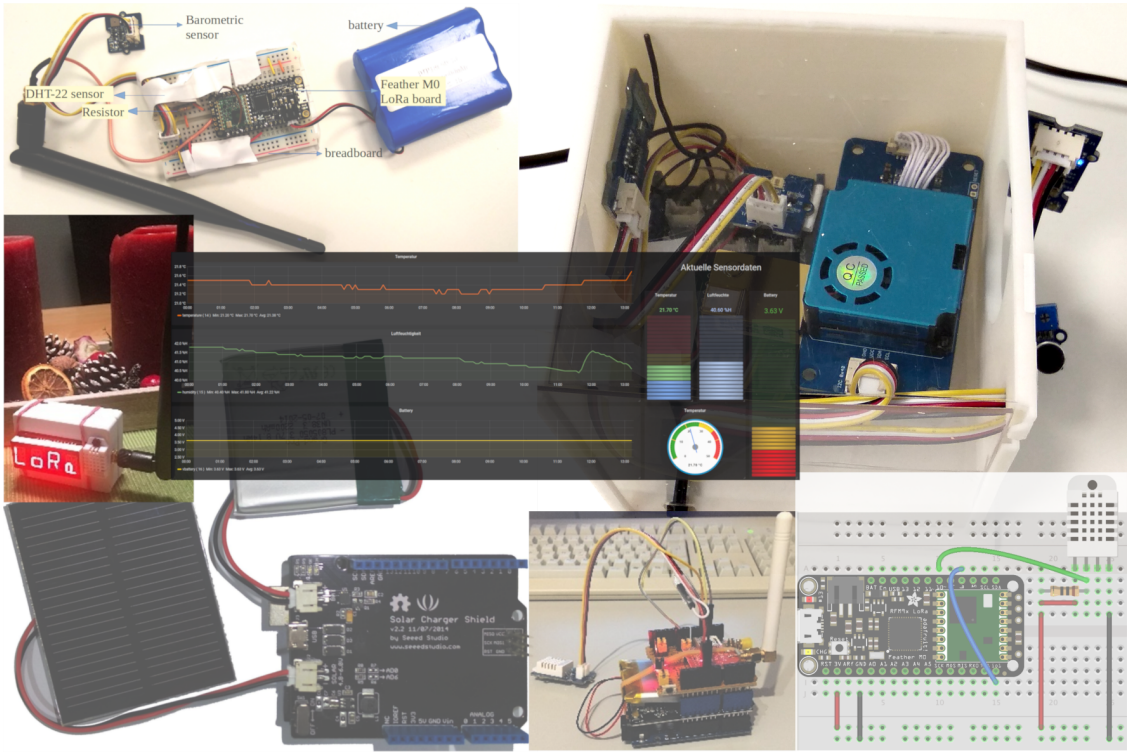
TUM-GIS Sensor Nodes

Release v0.0.1

Jan 08, 2020

Contents

This repo contains documentation, Arduino sketches, and images of our sensor nodes and the sensor services we used.



CHAPTER 1

Contact and contribution

We are happy for any kind of comments, questions, corrections, and own contributions. Please visit the Github Repo of this documentation to report a correction, bug, or question or contribute with a pull request.

CHAPTER 2

Contents

2.1 Solar powered Seeeduino

This sensor node is made to showcase a use-case of LoRaWAN sensor node powered using a solar panel. For achieving this a Seeeduino LoRaWAN microcontroller was used along with a solar panel connected using a solar shield. To show a generic use-case we have used a temperature and humidity sensor in this case, but it can be easily replaced with some other sensor as well. The entire setup was carefully placed in the ABS Waterproof case which is an easy to install water-proof and dust-proof case for an indoor or outdoor sensor installations. However, this case has no provision for the ventilation unlike the TFA case and so the readings obtained by the sensor may not accurately represent the outdoor weather conditions. In this example, we measure parameters such as temperature, humidity, and battery voltage.

2.1.1 Hardware

To build this sensor node we have used following hardware components:

- Seeeduino LoRaWAN board V4.2
- Grove - DHT-22 Temperature & Humidity Sensor
- Solar charger shield
- 1.5 W Solar panel
- 0 ohm resistor
- ABS Waterproof case
- 2000 mAH Battery

2.1.2 Wiring setup

First of all, the solar panel is connected with the SOLAR pin and a battery is connected with a BAT pin on the solar charger shield as shown in the figure below. A DHT-22 Sensor is connected to A2 pin on the Seeeduino board using a connector cable and then the solar charger shield prepared in the previous step is mounted on the board.

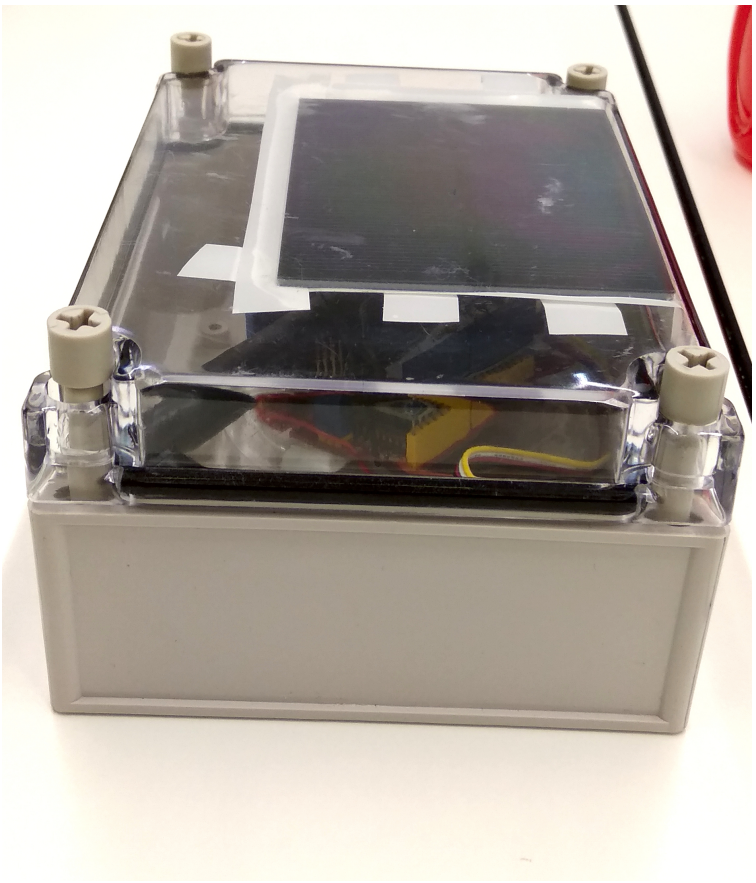


Fig. 1: Sensor node in ABS Waterproof case.

Fig. 2: Solar shield connections with the solar panel and a battery.

Apart from this, to measure the voltage of Lipo Battery we need to connect the VBAT pin to Analog pin A0, so that we can read the data from A0 pin. To achieve this, we need to Short R7 using a 0ohm resistor as shown in the figure here.

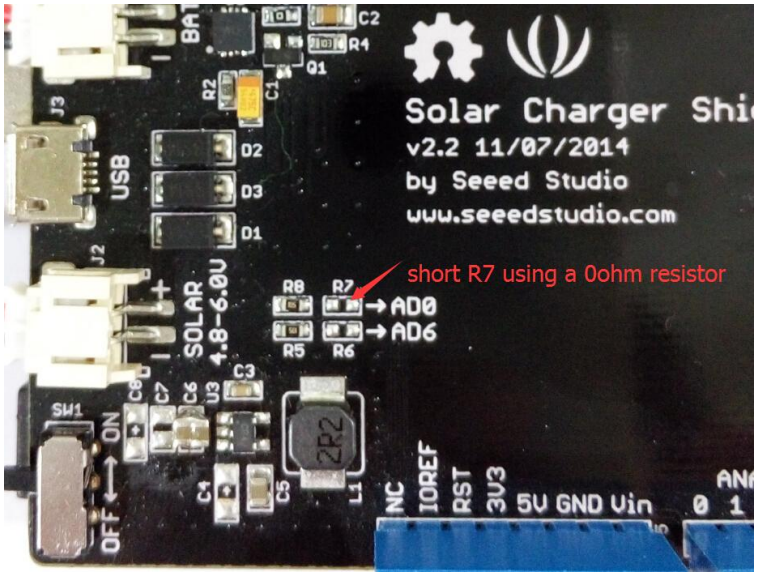


Fig. 3: Short R7 using a 0 ohm resistor for battery voltage measurement.

Final hardware setup looked as following:

Once all these connection were made, the board is connected with a computer using a USB cable. Further, steps of software part needs to be followed.

2.1.3 Software

To create this node, we use Arduino IDE for setting up the Seeeduino LoRaWAN device. First, install the Seeeduino LoRaWAN board to your Arduino IDE and select the correct port. Then following libraries needs to be installed before compiling the code:

- Wire.h to communicate with I2C devices.
- DHT.h for reading DHT-22 sensor.
- RTCZero.h for controlling internal clock for time.
- CayenneLPP.h for Cayenne Protocol.

Apart from this LoRaWan.h library is also used but it is bundled within Seeeduino Board and is not required to be separately installed.

Now download and run the *Arduino Sketch for Solar powered Seeeduino sensor node* file in the Arduino IDE. This code was created by merging the example code of both the sensors and the ttn-otaa example from the lmic library. Some required changes were made while merging the example codes. The user should change the network session key, app session key and device address in the code before compiling. These keys can be obtained from the TTN, SWM or other service providers.

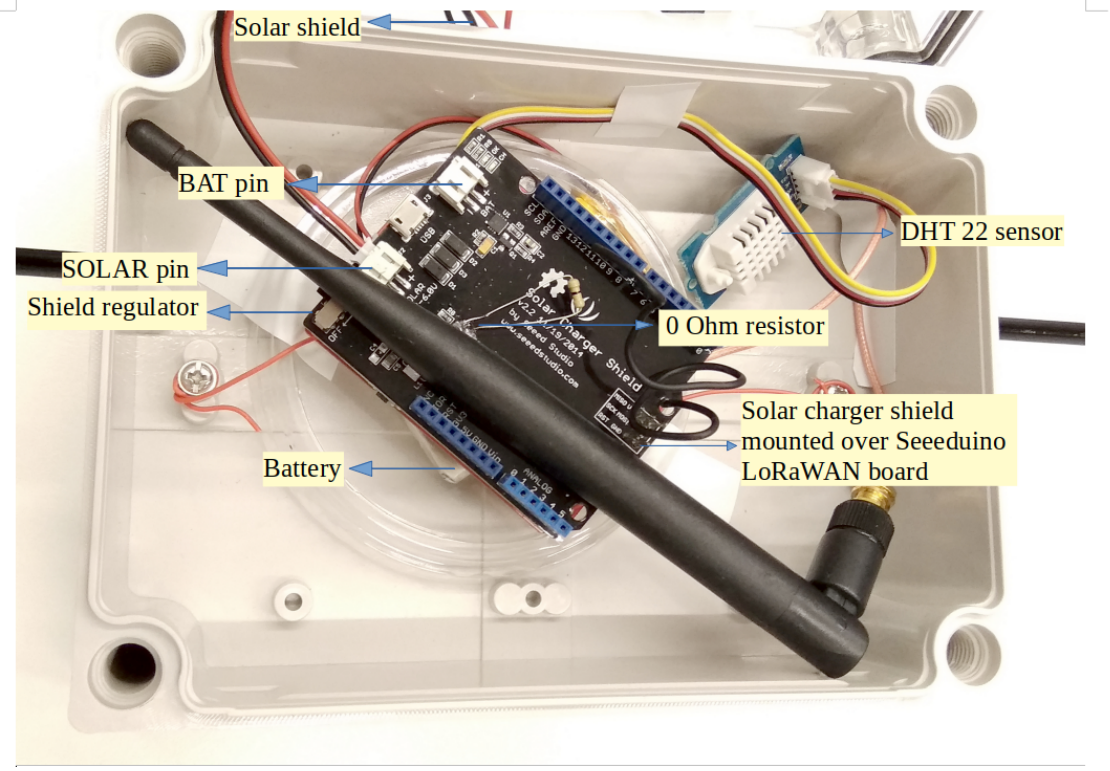


Fig. 4: Final hardware wiring setup.

Listing 1: Modify the keys in highlighted lines.

```
    // The EUIs and the AppKey must be given in big-endian format, i.e. the
    // most-significant-byte comes first (as displayed in the TTN console).
    // For TTN issued AppEUIs the first bytes should be 0x70, 0xB3, 0xD5.

    // void setId(char *DevAddr, char *DevEUI, char *AppEUI);
    lora.setId(NULL, "00942FBXXXXXXXXX", "70B3D57XXXXXXXXX");

    // setKey(char *NwkSKey, char *AppSKey, char *AppKey);
    lora.setKey(NULL, NULL, "CB89A0AA43F6C5XXXXXXXXXXXXXXXXXX");
```

Following is the example code that can be used to measure the battery voltage of the Seeed solar charger shield:

Listing 2: Code for measuring the battery voltage

```
    BatteryValue = analogRead(analogInPin);
    // Calculate the battery voltage value
    outputValue = (float(BatteryValue)*5)/1023*2;
    // print the results to the serial monitor:
    SerialUSB.print("Analog value = " );
    SerialUSB.print(BatteryValue);
    SerialUSB.print("\t voltage = ");
    SerialUSB.println(outputValue);
    SerialUSB.println("V \n");
```

2.1.4 Services

This node is connected using the TheThingsNetwork service. Further, a node-red work bench is used to forward this collected data from the TTN platform to the OGC Sensor Things API configured on the FROST Server. The node-red workbench that was used for forwarding the data is available at *Node red flow for Solar powered Seeeduino sensor node* for Solar powered Seeeduino. To use this node-red-workbench go to the node-red platform https://iot.gis.bgu.tum.de:1885/, login with the credentials, go to the options and select Import>Clipboard. Select the downloaded .json file with the given option and click on import. Make necessary changes and deploy the flow.

Datastreams setup for this sensor node on the FROST server can be seen at: http://iot.gis.bgu.tum.de:8081/FROST-Server-gi3/v1.0/Things(19)/Datastreams

The node-red workbench for this sensor node could be found at: https://iot.gis.bgu.tum.de:1885/#flow/58838bc1.4ce6a4

The GRAFANA dash-board for visualizing the collected data is available at: https://iot.gis.bgu.tum.de:3050/d/TfCVFRNWz/solar-powered-seeeduino-with-dht22?orgId=1&refresh=10s

2.1.5 Code files

Listing 3: Arduino Sketch for Solar powered Seeeduino sensor node

```
#include <DHT.h>
#include <RTCZero.h>
#include <LoRaWan.h>
#include <Wire.h>
#include <CayenneLPP.h>

// Keep the following line, if the board is a Seeeduino LoRaWAN with GPS,
// otherwise comment the line out

// #define HAS_GPS 1

const int analogInPin = A0;
#define DHTPIN A2
#define DHTTYPE DHT22

DHT dht(DHTPIN, DHTTYPE);

int BatteryValue = 0;
float outputValue = 0;

```

(continues on next page)

(continued from previous page)

```
RTCZero rtc;
char buffer[256];                       // buffer for text messages received from the LoRaWAN module for display

CayenneLPP lpp(51);

void setup(void)
{
    digitalWrite(38, HIGH);             // Provide power to the 4 Grove connectors of the board

    for(int i = 0; i < 26; i ++)        // Set all pins to HIGH to save power (reduces the
    {                                   // current drawn during deep sleep by around 0.7mA).
        if (i!=13) {                    // Don't switch on the onboard user LED (pin 13).
          pinMode(i, OUTPUT);
          digitalWrite(i, HIGH);
        }
    }

    delay(5000);                        // Wait 5 secs after reset/booting to give time for potential upload
    dht.begin();                                    // of a new sketch (sketches cannot be uploaded when in sleep mode)
    SerialUSB.begin(115200);            // Initialize USB/serial connection
    delay(500);
    // while(!SerialUSB);
    SerialUSB.println("Seeeduino LoRaWAN board started!");

    // nrgSave.begin(WAKE_RTC_ALARM);
    // rtc.begin(TIME_H24);

#ifdef HAS_GPS
    Serial.begin(9600);                 // Initialize serial connection to the GPS module
    delay(500);
    Serial.write("$PMTK161,0*28\r\n");  // Switch GPS module to standby mode as we don't use it in this sketch
#endif

    lora.init();                        // Initialize the LoRaWAN module

    memset(buffer, 0, 256);             // clear text buffer
    lora.getVersion(buffer, 256, 1);
    memset(buffer, 0, 256);             // We call getVersion() two times, because after a reset the LoRaWAN module can be
    lora.getVersion(buffer, 256, 1);    // in sleep mode and then the first call only wakes it up and will not be performed.
    SerialUSB.print(buffer);

    memset(buffer, 0, 256);
    lora.getId(buffer, 256, 1);
    SerialUSB.print(buffer);

    // The following three constants (AppEUI, DevEUI, AppKey) must be changed
```

(continues on next page)

(continued from previous page)

```
    // for every new sensor node. We are using the LoRaWAN OTAA mode (over the
    // air activation). Each sensor node must be manually registered in the
    // TTN console at https://console.thethingsnetwork.org before it can be
    // started. In the TTN console create a new device with the DevEUI also
    // being automatically generated. After the registration of the device the
    // three values can be copied from the TTN console. A detailed explanation
    // of these steps is given in
    // https://learn.adafruit.com/the-things-network-for-feather?view=all

    // The EUIs and the AppKey must be given in big-endian format, i.e. the
    // most-significant-byte comes first (as displayed in the TTN console).
    // For TTN issued AppEUIs the first bytes should be 0x70, 0xB3, 0xD5.

    // void setId(char *DevAddr, char *DevEUI, char *AppEUI);
    lora.setId(NULL, "00942FBXXXXXXXXX", "70B3D57XXXXXXXXX");

    // setKey(char *NwkSKey, char *AppSKey, char *AppKey);
    lora.setKey(NULL, NULL, "CB89A0AA43F6C5XXXXXXXXXXXXXXXXXX");

    lora.setDeciveMode(LWOTAA);           // select OTAA join mode (note that␣
↪setDeciveMode is not a typo; it is misspelled in the library)
    // lora.setDataRate(DR5, EU868);         // SF7, 125 kbps (highest data rate)
    lora.setDataRate(DR3, EU868);         // SF9, 125 kbps (medium data rate and␣
↪range)
    // lora.setDataRate(DR0, EU868);         // SF12, 125 kbps (lowest data rate,␣
↪highest max. distance)

    // lora.setAdaptiveDataRate(false);
    lora.setAdaptiveDataRate(true);       // automatically adapt the data rate

    lora.setChannel(0, 868.1);
    lora.setChannel(1, 868.3);
    lora.setChannel(2, 868.5);
    lora.setChannel(3, 867.1);
    lora.setChannel(4, 867.3);
    lora.setChannel(5, 867.5);
    lora.setChannel(6, 867.7);
    lora.setChannel(7, 867.9);

    // The following two commands can be left commented out;
    // TTN works with the default values. (It also works when
    // uncommenting the commands, though.)
    // lora.setReceiceWindowFirst(0, 868.1);
    // lora.setReceiceWindowSecond(869.525, DR0);

    lora.setDutyCycle(false);             // for debugging purposes only - should␣
↪normally be activated
    lora.setJoinDutyCycle(false);         // for debugging purposes only - should␣
↪normally be activated

    lora.setPower(14);                    // LoRa transceiver power (14 is the␣
↪maximum for the 868 MHz band)

    // while(!lora.setOTAAJoin(JOIN));
    while(!lora.setOTAAJoin(JOIN,20));    // wait until the node has successfully␣
↪joined TTN

```

(continues on next page)

(continued from previous page)

```
    lora.setPort(33);                       // all data packets are sent to LoRaWAN port 33
}

void loop(void)
{
    bool result = false;
    float temp_hum_val[2] = {0};
    float temperature, humidity;
    // Reading temperature or humidity takes about 250 milliseconds!
    // Sensor readings may also be up to 2 seconds 'old' (its a very slow sensor)


    if(!dht.readTempAndHumidity(temp_hum_val)){
        SerialUSB.print("Humidity: ");
        SerialUSB.print(humidity = temp_hum_val[0]);
        SerialUSB.print(" %\t");
        SerialUSB.print("Temperature: ");
        SerialUSB.print(temperature = temp_hum_val[1]);
        SerialUSB.println(" *C");
    }
    else{
       SerialUSB.println("Failed to get temprature and humidity value.");
    }

    BatteryValue = analogRead(analogInPin);
    // Calculate the battery voltage value
    outputValue = (float(BatteryValue)*5)/1023*2;
    // print the results to the serial monitor:
    SerialUSB.print("Analog value = " );
    SerialUSB.print(BatteryValue);
    SerialUSB.print("\t voltage = ");
    SerialUSB.println(outputValue);
    SerialUSB.println("V \n");

    SerialUSB.println("-- LOOP");
    lpp.reset();
    lpp.addTemperature(1, temperature);
    lpp.addRelativeHumidity(2, humidity);
    lpp.addAnalogInput(3, outputValue);

    result = lora.transferPacket(lpp.getBuffer(), lpp.getSize(), 5);   // send the data packet (n byts) with a default timeout of 5 secs

    if(result)
    {
        short length;
        short rssi;

        memset(buffer, 0, 256);
        length = lora.receivePacket(buffer, 256, &rssi);

        if(length)
        {
            SerialUSB.print("Length is: ");
            SerialUSB.println(length);
            SerialUSB.print("RSSI is: ");
```

(continues on next page)

(continued from previous page)

```
            SerialUSB.println(rssi);
            SerialUSB.print("Data is: ");
            for(unsigned char i = 0; i < length; i ++)
            {
                SerialUSB.print("0x");
                SerialUSB.print(buffer[i], HEX);
                SerialUSB.print(" ");
            }
            SerialUSB.println();
        }
    }

    lora.setDeviceLowPower();      // bring the LoRaWAN module to sleep mode
    doSleep((5*60-8)*1000);        // deep sleep for 292 secs (+ 3 secs transmission time + 5 secs timeout = 300 secs period)
    lora.setPort(33);              // send some command to wake up the LoRaWAN module again
}

// The following function implements deep sleep waiting. When being called the
// CPU goes into deep sleep mode (for power saving). It is woken up again by
// the CPU-internal real time clock (RTC) after the configured time.
//
// A similar function would also be available in the standard "ArduinoLowPower" library.
// However, in order to be able to use that library with the Seeeduino LoRaWAN board,
// four files in the package "Seeed SAMD boards by Seeed Studio Version 1.3.0" that is
// installed using the Arduino IDE board manager need to be patched. The reason is that
// Seeed Studio have not updated their files to a recent Arduino SAMD version yet
// and the official "ArduinoLowPower" library provided by the Arduino foundation is
// referring to some missing functions. For further information see here:
// https://forum.arduino.cc/index.php?topic=603900.0 and here:
// https://github.com/arduino/ArduinoCore-samd/commit/b9ac48c782ca4b82ffd7e65bf2c956152386d82b

void doSleep(uint32_t millis) {
    if (!rtc.isConfigured()) {     // if called for the first time,
        rtc.begin(false);          // then initialize the real time clock (RTC)
    }

    uint32_t now = rtc.getEpoch();
    rtc.setAlarmEpoch(now + millis/1000);
    rtc.enableAlarm(rtc.MATCH_HHMMSS);

    rtc.standbyMode();             // bring CPU into deep sleep mode (until woken up by the RTC)
}
```

Listing 4: Node red flow for Solar powered Seeeduino sensor node

```
[
    {
        "id": "58838bc1.4ce6a4",
        "type": "tab",
        "label": "Device1",
```

(continues on next page)

(continued from previous page)

```
        "disabled": false,
        "info": ""
    },
    {
        "id": "daeb7602.698d18",
        "type": "switch",
        "z": "58838bc1.4ce6a4",
        "name": "Separate",
        "property": "key",
        "propertyType": "msg",
        "rules": [
            {
                "t": "cont",
                "v": "temperature",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "humidity",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "analog",
                "vt": "str"
            },
            {
                "t": "else"
            }
        ],
        "checkall": "true",
        "repair": false,
        "outputs": 4,
        "x": 220,
        "y": 180,
        "wires": [
            [
                "a3a522a5.a81a9"
            ],
            [
                "367717e8.191318"
            ],
            [
                "466fd2c5.586efc"
            ],
            []
        ]
    },
    {
        "id": "e2798231.c9314",
        "type": "split",
        "z": "58838bc1.4ce6a4",
        "name": "",
        "splt": "\\n",
        "spltType": "str",
        "arraySplt": 1,
        "arraySpltType": "len",
```

(continues on next page)

(continued from previous page)

```
        "stream": false,
        "addname": "key",
        "x": 90,
        "y": 180,
        "wires": [
            [
                "daeb7602.698d18"
            ]
        ]
    },
    {
        "id": "5c3e3ed9.0b4dd",
        "type": "debug",
        "z": "58838bc1.4ce6a4",
        "name": "",
        "active": false,
        "tosidebar": true,
        "console": false,
        "tostatus": false,
        "complete": "false",
        "x": 810,
        "y": 180,
        "wires": []
    },
    {
        "id": "367717e8.191318",
        "type": "function",
        "z": "58838bc1.4ce6a4",
        "name": "Humidity",
        "func": "var humValue = msg.payload.valueOf();\nvar newMessage =  { payload:
↪{  \"result\": humValue, \"Datastream\": {\"@iot.id\": 102}} };\nnewMessage.headers␣
↪= {\"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 440,
        "y": 200,
        "wires": [
            [
                "c777922b.84784"
            ]
        ]
    },
    {
        "id": "c777922b.84784",
        "type": "http request",
        "z": "58838bc1.4ce6a4",
        "name": "POST Observation",
        "method": "POST",
        "ret": "obj",
        "paytoqs": false,
        "url": "http://iot.gis.bgu.tum.de:8081/FROST-Server-gi3/v1.0/Observations",
        "tls": "",
        "proxy": "",
        "authType": "basic",
        "x": 630,
        "y": 180,
        "wires": [
```

(continues on next page)

(continued from previous page)

```
            [
                "5c3e3ed9.0b4dd"
            ]
        ]
    },
    {
        "id": "a3a522a5.a81a9",
        "type": "function",
        "z": "58838bc1.4ce6a4",
        "name": "Temperature",
        "func": "var tempValue = msg.payload.valueOf();\nvar newMessage =  { payload:
↪{  \"result\": tempValue, \"Datastream\": {\"@iot.id\": 101}} };\nnewMessage.
↪headers = {\"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 450,
        "y": 160,
        "wires": [
            [
                "c777922b.84784"
            ]
        ]
    },
    {
        "id": "41ae6239.73f9bc",
        "type": "ttn uplink",
        "z": "58838bc1.4ce6a4",
        "name": "TTN Input",
        "app": "58ceff1f.8576a",
        "dev_id": "tum-gis-device1",
        "field": "",
        "x": 80,
        "y": 60,
        "wires": [
            [
                "491bb4da.0eb58c"
            ]
        ]
    },
    {
        "id": "491bb4da.0eb58c",
        "type": "cayennelpp-decoder",
        "z": "58838bc1.4ce6a4",
        "name": "",
        "x": 280,
        "y": 60,
        "wires": [
            [
                "e2798231.c9314",
                "f2d3534b.0f44f"
            ]
        ]
    },
    {
        "id": "466fd2c5.586efc",
        "type": "function",
        "z": "58838bc1.4ce6a4",
```

(continues on next page)

(continued from previous page)

```
        "name": "Battery Voltage",
        "func": "var batteryvolt = msg.payload.valueOf();\nvar newMessage =  {␣
↪payload: {  \"result\": batteryvolt, \"Datastream\": {\"@iot.id\": 104}} };
↪\nnewMessage.headers = {\"Content-type\" : \"application/json\"}\nreturn newMessage;
↪",
        "outputs": 1,
        "noerr": 0,
        "x": 440,
        "y": 240,
        "wires": [
            [
                "c777922b.84784"
            ]
        ]
    },
    {
        "id": "f2d3534b.0f44f",
        "type": "debug",
        "z": "58838bc1.4ce6a4",
        "name": "",
        "active": true,
        "tosidebar": true,
        "console": false,
        "tostatus": false,
        "complete": "false",
        "x": 490,
        "y": 60,
        "wires": []
    },
    {
        "id": "58ceff1f.8576a",
        "type": "ttn app",
        "z": "",
        "appId": "gis-tum-sensors",
        "accessKey": "ttn-account-ACCESSKEY_HERE",
        "discovery": "discovery.thethingsnetwork.org:1900"
    }
]
```

2.1.6 References

- *Arduino Sketch for Solar powered Seeeduino sensor node*
- *Node red flow for Solar powered Seeeduino sensor node*
- Wiki guide for Seeeduino LoRaWAN board
- Adding Seeed boards to Arduino IDE
- Seeed Solar charger shield guide

2.2 Indoor Ambient Monitoring

This sensor node is made to showcase a use-case of LoRaWAN for indoor ambience monitoring. For achieving this a multitude of sensors were used which can monitor the quality of the ambience. In this example we measure parameters

such as temperature, humidity, air pressure, air quality, CO2, loudness, gas, PM2.5, and light.

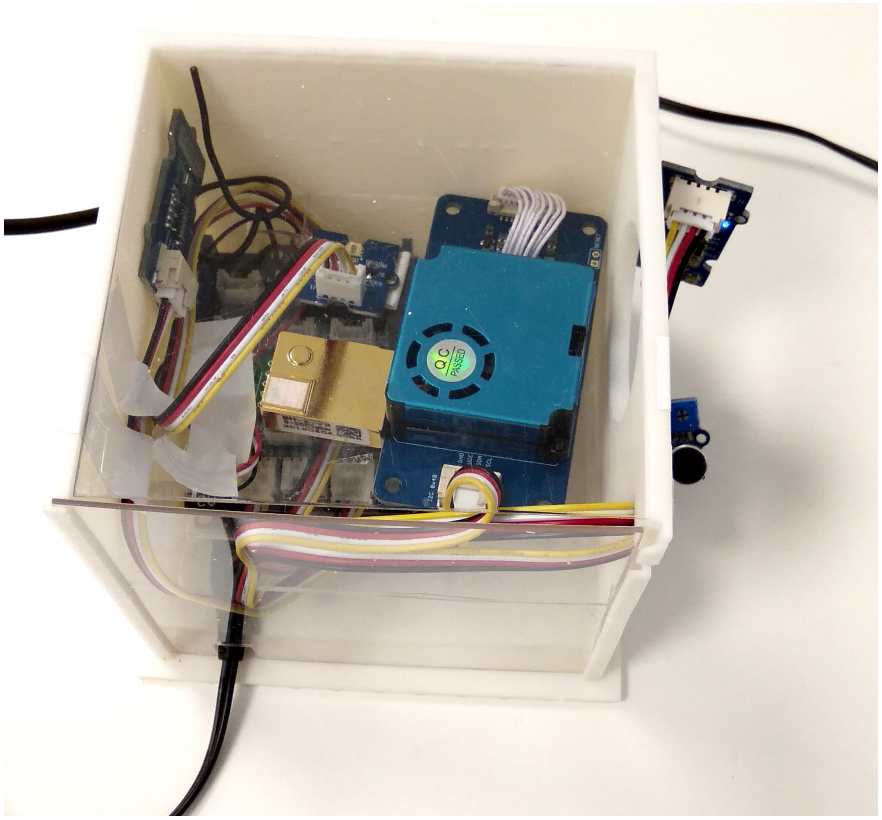


Fig. 5: Hardware setup.

2.2.1 Hardware

To realize the objective, following components were used:

- Seeeduino LoRaWAN board V4.2
- Grove Base Shield Arduino V2
- Grove - Air Quality Sensor
- Grove - Loudness Sensor
- Grove - Digital Light Sensor
- Grove - BME680 Sensor
- Grove - Laser PM2.5 Sensor (HM3301)
- MHZ19B CO2 Sensor
- Micro USB Charger

2.2.2 Wiring setup

First of all, the grove base shield was connected over the Seeeduino LoRaWAN board. The board was set at the 5V mode. Then, the sensor connections were made using the connector cables as following:

- Loudness Sensor – Analog Pin A0
- PM 2.5 Sensor – I2C pin
- Digital Light Sensor – I2C pin
- BME680 Sensor – I2C pin
- MHZ19B CO2 Sensor – Digital Pin D4
- Air Quality Sensor - A2

Apart from this, there is no need of any other wiring in this case.

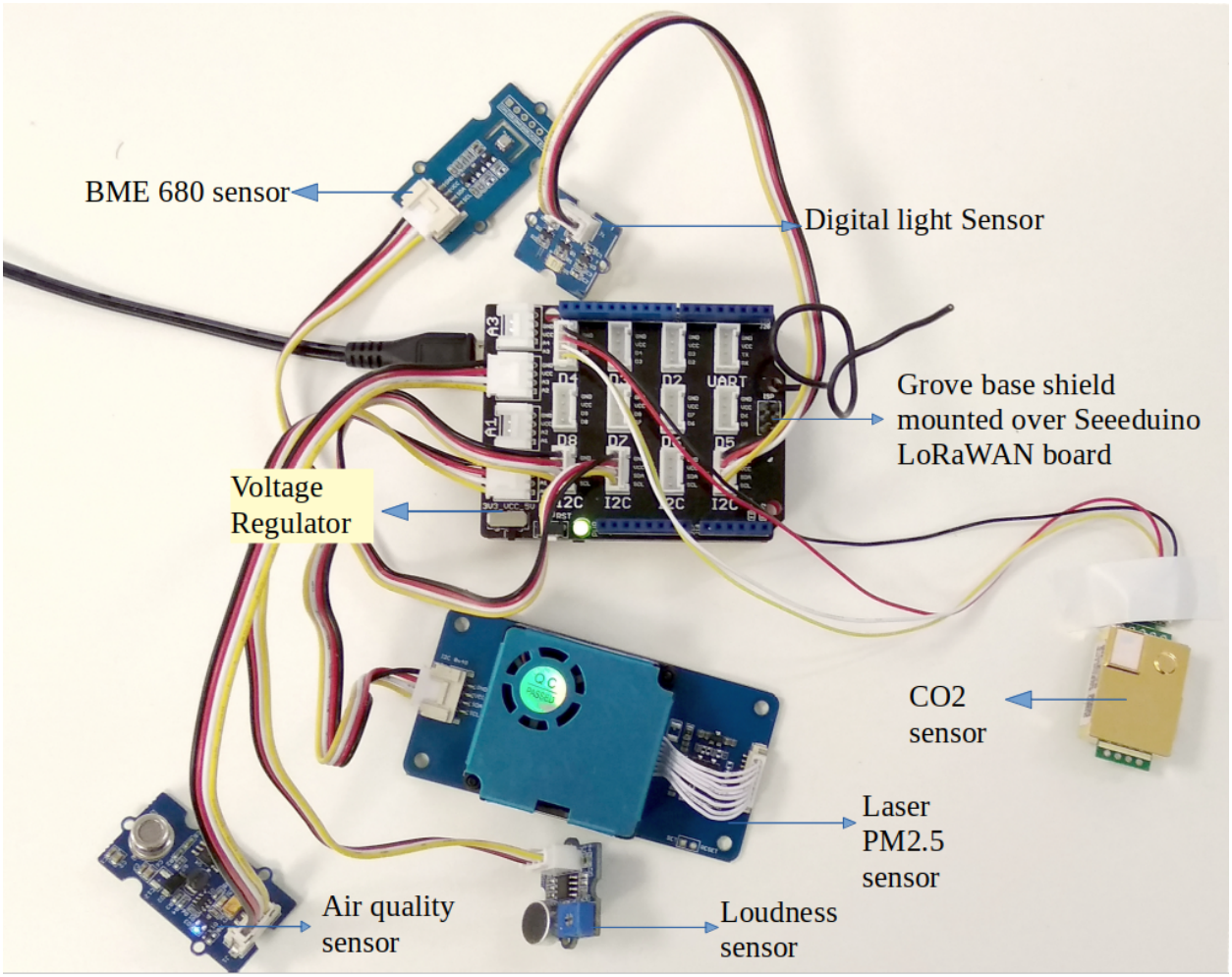


Fig. 6: Hardware connections.

Once all these connection were made, the board is connected with a computer using a USB cable. Further, steps of software part needs to be followed.

2.2.3 Software

To create this node, we use Arduino IDE for setting up the Seeeduino LoRaWAN device. First, install the Seeeduino LoRaWAN board board to your Arduino IDE and select the correct port. Then following libraries needs to be installed before compiling the code:

- Digital_Light_TSL2561.h for Digital Light Sensor
- Air_Quality_Sensor.h for Air Quality Sensor
- Seeed_bme680.h for BME680 Sensor
- Seeed_HM330X.h for Laser PM2.5 Sensor
- MHZ19.h for MHZ19B CO2 Sensor
- Wire.h to communicate with I2C devices
- SoftwareSerial.h for Serial Communication
- RTCZero.h for controlling internal clock for time
- CayenneLPP.h for Cayenne Protocol

Apart from this LoRaWan.h library is also used but it is bundled within Seeeduino Board and is not required to be separately installed.

Now download and run the *Arduino Sketch for Indoor Ambient Monitoring sensor node* file in the Arduino IDE. This code was created by merging the example code of each of these attached sensor and the ttn-otaa example from the Seeeduino board. Some required changes were made while merging the example codes. For example, as there are multiple sensors each needs to be defined with a unique name. So, here HM330X was named as sensor while AirqualitySensor as sensors.

Listing 5: Setup the sensors in *Arduino Sketch for Indoor Ambient Monitoring sensor node*

```
AirQualitySensor sensors(A2);

SoftwareSerial ss(4,5);
MHZ19 mhz(&ss);

#define BME_SCK 13
#define BME_MISO 12
#define BME_MOSI 11
#define BME_CS 10

#define IIC_ADDR  uint8_t(0x76)
Seeed_BME680 bme680(IIC_ADDR);

int loudness,a;

HM330X sensor;
u8 buf[30];
```

The user should change the network session key, app session key and device address in the code before compiling. These keys can be obtained from the TTN, SWM or any other service providers.

Listing 6: Modify the keys in highlighted lines

```
    // The EUIs and the AppKey must be given in big-endian format, i.e. the
    // most-significant-byte comes first (as displayed in the TTN console).
    // For TTN issued AppEUIs the first bytes should be 0x70, 0xB3, 0xD5.

    // void setId(char *DevAddr, char *DevEUI, char *AppEUI);
    lora.setId(NULL, "00942FBXXXXXXXXX", "70B3D57XXXXXXXXX");

```

(continues on next page)

(continued from previous page)

```
    // setKey(char *NwkSKey, char *AppSKey, char *AppKey);
    lora.setKey(NULL, NULL, "CB89A0AA43F6C5XXXXXXXXXXXXXXXXXX");
```

2.2.4 Services

This node is connected using the TheThingsNetwork service. Further, a node-red work bench is used to forward this collected data from the TTN platform to the OGC Sensor Things API configured on the FROST Server. The node-red workbench that was used for forwarding the data is available at *Node red flow for Indoor Ambient Monitoring sensor node*. To use this node-red-workbench go to the node-red platform https://iot.gis.bgu.tum.de:1885/, login with the credentials, go to the options and select Import>Clipboard. Select the downloaded .json file with the given option and click on import. Make necessary changes and deploy the flow.

Datastreams setup for this sensor node on the FROST server can be seen at: http://iot.gis.bgu.tum.de:8081/FROST-Server-gi3/v1.0/Things(21)/Datastreams

The node-red workbench for this sensor node could be found at: https://iot.gis.bgu.tum.de:1885/#flow/7d5c6b14.d2af94

The GRAFANA dash-board for visualizing the collected data is available at: https://iot.gis.bgu.tum.de:3050/d/jDJ1li1Wz/indoor-ambient-monitoring-with-seeeduino-lorawan-and-sensors?orgId=1

2.2.5 Code files

Listing 7: Arduino Sketch for Indoor Ambient Monitoring sensor node

```
#include <Wire.h>
#include <Digital_Light_TSL2561.h>
#include "Air_Quality_Sensor.h"
#include "seeed_bme680.h"
#include "Seeed_HM330X.h"
#include <SoftwareSerial.h>
#include <MHZ19.h>
#include <RTCZero.h>
#include <LoRaWan.h>
#include <CayenneLPP.h>

RTCZero rtc;
char buffer[256];                       // buffer for text messages received from the␣
↪LoRaWAN module for display

CayenneLPP lpp(51);

AirQualitySensor sensors(A2);

SoftwareSerial ss(4,5);
MHZ19 mhz(&ss);

#define BME_SCK 13
#define BME_MISO 12
#define BME_MOSI 11
#define BME_CS 10

#define IIC_ADDR  uint8_t(0x76)
Seeed_BME680 bme680(IIC_ADDR);
```

(continues on next page)

(continued from previous page)

```

int loudness,a;

HM330X sensor;
u8 buf[30];

const char *str[]={"sensor num: ","PM1.0 concentration(CF=1,Standard particulate matter,unit:ug/m3): ",
                   "PM2.5 concentration(CF=1,Standard particulate matter,unit:ug/m3): ",
                   "PM10 concentration(CF=1,Standard particulate matter,unit:ug/m3): ",
                   "PM1.0 concentration(Atmospheric environment,unit:ug/m3): ",
                   "PM2.5 concentration(Atmospheric environment,unit:ug/m3): ",
                   "PM10 concentration(Atmospheric environment,unit:ug/m3): ",
                   };

err_t print_result(const char* str,u16 value)
{
    if(NULL==str)
        return ERROR_PARAM;
    SerialUSB.print(str);
    SerialUSB.println(value);
    return NO_ERROR;
}

/*parse buf with 29 u8-data*/
err_t parse_result(u8 *data)
{
    u16 value=0;
    err_t NO_ERROR;
    if(NULL==data)
        return ERROR_PARAM;
    for(int i=1;i<8;i++)
    {
         value = (u16)data[i*2]<<8|data[i*2+1];
         print_result(str[i-1],value);
         if(i==6)
           {   a=value;
               SerialUSB.println(a);
           }
    }
}

err_t parse_result_value(u8 *data)
{
    if(NULL==data)
        return ERROR_PARAM;
    for(int i=0;i<28;i++)
    {
        SerialUSB.print(data[i],HEX);
        SerialUSB.print("  ");
        if((0==(i)%5)||(0==i))
        {
            SerialUSB.println(" ");
        }
    }
```

(continues on next page)

(continued from previous page)

```
    u8 sum=0;
    for(int i=0;i<28;i++)
    {
        sum+=data[i];
    }
    if(sum!=data[28])
    {
        SerialUSB.println("wrong checkSum!!!!");
    }
    SerialUSB.println(" ");
    SerialUSB.println(" ");
    return NO_ERROR;
}


void setup()
{
  Wire.begin();

  for(int i = 0; i < 26; i ++)         // Set all pins to HIGH to save power (reduces the
    {                                    // current drawn during deep sleep by around 0.7mA).
        if (i!=13) {                     // Don't switch on the onboard user LED (pin 13).
          pinMode(i, OUTPUT);
          digitalWrite(i, HIGH);
        }
    }

  delay(5000);

  SerialUSB.begin(115200);
      delay(100);
  SerialUSB.println("SerialUSB start");

  lora.init();                          // Initialize the LoRaWAN module

    memset(buffer, 0, 256);              // clear text buffer
    lora.getVersion(buffer, 256, 1);
    memset(buffer, 0, 256);              // We call getVersion() two times, because after a reset the LoRaWAN module can be
    lora.getVersion(buffer, 256, 1);     // in sleep mode and then the first call only wakes it up and will not be performed.
    SerialUSB.print(buffer);

    memset(buffer, 0, 256);
    lora.getId(buffer, 256, 1);
    SerialUSB.print(buffer);

    // The following three constants (AppEUI, DevEUI, AppKey) must be changed
    // for every new sensor node. We are using the LoRaWAN OTAA mode (over the
    // air activation). Each sensor node must be manually registered in the
    // TTN console at https://console.thethingsnetwork.org before it can be
    // started. In the TTN console create a new device with the DevEUI also
    // being automatically generated. After the registration of the device the
    // three values can be copied from the TTN console. A detailed explanation
```

(continues on next page)

(continued from previous page)

```
    // of these steps is given in
    // https://learn.adafruit.com/the-things-network-for-feather?view=all

    // The EUIs and the AppKey must be given in big-endian format, i.e. the
    // most-significant-byte comes first (as displayed in the TTN console).
    // For TTN issued AppEUIs the first bytes should be 0x70, 0xB3, 0xD5.

    // void setId(char *DevAddr, char *DevEUI, char *AppEUI);
    lora.setId(NULL, "00942FBXXXXXXXXX", "70B3D57XXXXXXXXX");

    // setKey(char *NwkSKey, char *AppSKey, char *AppKey);
    lora.setKey(NULL, NULL, "CB89A0AA43F6C5XXXXXXXXXXXXXXXXXX");

    lora.setDeciveMode(LWOTAA);           // select OTAA join mode (note that setDeciveMode is not a typo; it is misspelled in the library)
    // lora.setDataRate(DR5, EU868);         // SF7, 125 kbps (highest data rate)
    lora.setDataRate(DR3, EU868);         // SF9, 125 kbps (medium data rate and range)
    // lora.setDataRate(DR0, EU868);         // SF12, 125 kbps (lowest data rate, highest max. distance)

    // lora.setAdaptiveDataRate(false);
    lora.setAdaptiveDataRate(true);       // automatically adapt the data rate

    lora.setChannel(0, 868.1);
    lora.setChannel(1, 868.3);
    lora.setChannel(2, 868.5);
    lora.setChannel(3, 867.1);
    lora.setChannel(4, 867.3);
    lora.setChannel(5, 867.5);
    lora.setChannel(6, 867.7);
    lora.setChannel(7, 867.9);

    // The following two commands can be left commented out;
    // TTN works with the default values. (It also works when
    // uncommenting the commands, though.)
    // lora.setReceiceWindowFirst(0, 868.1);
    // lora.setReceiceWindowSecond(869.525, DR0);

    lora.setDutyCycle(false);             // for debugging purposes only - should normally be activated
    lora.setJoinDutyCycle(false);         // for debugging purposes only - should normally be activated

    lora.setPower(14);                    // LoRa transceiver power (14 is the maximum for the 868 MHz band)

    // while(!lora.setOTAAJoin(JOIN));
    while(!lora.setOTAAJoin(JOIN,20));    // wait until the node has successfully joined TTN

    lora.setPort(33);

    if(sensor.init())
    {
        SerialUSB.println("HM330X init failed!!!");
        while(1);
```

(continues on next page)

(continued from previous page)

```
    }

  if (sensors.init()) {
    SerialUSB.println("Sensor ready.");
  }
  else {
    SerialUSB.println("Sensor ERROR!");
  }

  TSL2561.init();

    while (!bme680.init())
  {
    SerialUSB.println("bme680 init failed ! can't find device!");
  delay(10000);
  }

  ss.begin(9600);

}

void loop()
{
  bool result = false;
  float temperature,humidity,pressure,airquality,light,gas,CO2;

  loudness = analogRead(0);
  SerialUSB.print("The Loudness Sensor value is: ");
  SerialUSB.println(loudness);
  SerialUSB.println();
  delay(3000);

  int quality = sensors.slope();

  SerialUSB.print("Air Quality Sensor value is: ");
  SerialUSB.println(airquality=sensors.getValue());

  if (quality == AirQualitySensor::FORCE_SIGNAL) {
    SerialUSB.println("High pollution! Force signal active.");
  }
  else if (quality == AirQualitySensor::HIGH_POLLUTION) {
    SerialUSB.println("High pollution!");
  }
  else if (quality == AirQualitySensor::LOW_POLLUTION) {
    SerialUSB.println("Low pollution!");
  }
  else if (quality == AirQualitySensor::FRESH_AIR) {
    SerialUSB.println("Fresh air.");
  }
  SerialUSB.println();
  delay(3000);

  SerialUSB.print("The Light Sensor value is: ");
  SerialUSB.println(light=TSL2561.readVisibleLux());
  SerialUSB.println();
  delay(3000);

```

(continues on next page)

(continued from previous page)

```
    if(sensor.read_sensor_value(buf,29))
    {
        SerialUSB.println("HM330X read result failed!!!");
    }
    parse_result_value(buf);
    parse_result(buf);
    SerialUSB.println(" ");
    delay(3000);

    if (bme680.read_sensor_data())
  {
    SerialUSB.println("Failed to perform reading :(");
    return;
  }
  SerialUSB.print("temperature ===>> ");
  SerialUSB.print(temperature = bme680.sensor_result_value.temperature);
  SerialUSB.println(" C");

  SerialUSB.print("pressure ===>> ");
  SerialUSB.print(pressure = bme680.sensor_result_value.pressure/ 1000.0);
  SerialUSB.println(" KPa");

  SerialUSB.print("humidity ===>> ");
  SerialUSB.print(humidity = bme680.sensor_result_value.humidity);
  SerialUSB.println(" %");

  SerialUSB.print("gas ===>> ");
  SerialUSB.print(gas = bme680.sensor_result_value.gas/ 1000.0);
  SerialUSB.println(" Kohms");

  SerialUSB.println();

  delay(3000);

  MHZ19_RESULT response = mhz.retrieveData();
  if (response == MHZ19_RESULT_OK)
  {
    SerialUSB.print(F("CO2: "));
    SerialUSB.println(CO2=mhz.getCO2());
    SerialUSB.print(F("Min CO2: "));
    SerialUSB.println(mhz.getMinCO2());
    SerialUSB.print(F("Temperature: "));
    SerialUSB.println(mhz.getTemperature());
    SerialUSB.print(F("Accuracy: "));
    SerialUSB.println(mhz.getAccuracy());
    SerialUSB.println();
  }
  else
  {
    SerialUSB.print(F("Error, code: "));
    SerialUSB.println(response);
  }

    lpp.reset();
    lpp.addTemperature(1, temperature);
    lpp.addRelativeHumidity(2, humidity);
    lpp.addAnalogInput(3, airquality);
```

(continues on next page)

(continued from previous page)

```
    lpp.addLuminosity(4, light);
    lpp.addBarometricPressure(5, pressure);
    lpp.addLuminosity(6, CO2);
    lpp.addAnalogInput(7, gas);
    lpp.addLuminosity(8, loudness);
    lpp.addLuminosity(9, a);
    result = lora.transferPacket(lpp.getBuffer(), lpp.getSize(), 5);   // send the data packet (n byts) with a default timeout of 5 secs

    if(result)
    {
        short length;
        short rssi;

        memset(buffer, 0, 256);
        length = lora.receivePacket(buffer, 256, &rssi);

        if(length)
        {
            SerialUSB.print("Length is: ");
            SerialUSB.println(length);
            SerialUSB.print("RSSI is: ");
            SerialUSB.println(rssi);
            SerialUSB.print("Data is: ");
            for(unsigned char i = 0; i < length; i ++)
            {
                SerialUSB.print("0x");
                SerialUSB.print(buffer[i], HEX);
                SerialUSB.print(" ");
            }
            SerialUSB.println();
        }
    }

    lora.setDeviceLowPower();     // bring the LoRaWAN module to sleep mode
    doSleep((5*60-8)*1000);       // deep sleep for 292 secs (+ 3 secs transmission time + 5 secs timeout = 300 secs period)
    lora.setPort(33);

}

void doSleep(uint32_t millis) {
    if (!rtc.isConfigured()) {    // if called for the first time,
        rtc.begin(false);         // then initialize the real time clock (RTC)
    }

    uint32_t now = rtc.getEpoch();
    rtc.setAlarmEpoch(now + millis/1000);
    rtc.enableAlarm(rtc.MATCH_HHMMSS);

    rtc.standbyMode();            // bring CPU into deep sleep mode (until woken up by the RTC)
}
```

Listing 8: Node red flow for Indoor Ambient Monitoring sensor node


```
[
    {
        "id": "7d5c6b14.d2af94",
        "type": "tab",
        "label": "Device3",
        "disabled": false,
        "info": ""
    },
    {
        "id": "4d581d8.f14c0e4",
        "type": "switch",
        "z": "7d5c6b14.d2af94",
        "name": "Separate",
        "property": "key",
        "propertyType": "msg",
        "rules": [
            {
                "t": "cont",
                "v": "temperature_1",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "humidity",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "analog_in_3",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "luminosity_4",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "barometric",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "luminosity_6",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "analog_in_7",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "luminosity_8",
                "vt": "str"
```


(continues on next page)

(continued from previous page)

```
            },
            {
                "t": "cont",
                "v": "luminosity_9",
                "vt": "str"
            }
        ],
        "checkall": "true",
        "repair": false,
        "outputs": 9,
        "x": 220,
        "y": 180,
        "wires": [
            [
                "92425d9f.bca98"
            ],
            [
                "9b919750.d6fff8"
            ],
            [
                "620b1ab2.a69224"
            ],
            [
                "eee677bf.fe16d8"
            ],
            [
                "4b424590.139b4c"
            ],
            [
                "bc3f8433.7b89f8"
            ],
            [
                "13968bca.c2cd34"
            ],
            [
                "c7fcb372.4b2df"
            ],
            [
                "b52fa683.7c5fb8"
            ]
        ]
    },
    {
        "id": "9010a80d.dfcdb8",
        "type": "split",
        "z": "7d5c6b14.d2af94",
        "name": "",
        "splt": "\\n",
        "spltType": "str",
        "arraySplt": 1,
        "arraySpltType": "len",
        "stream": false,
        "addname": "key",
        "x": 90,
        "y": 180,
        "wires": [
            [
```

(continues on next page)

(continued from previous page)

```
                "4d581d8.f14c0e4"
            ]
        ]
    },
    {
        "id": "a3b86e6f.56637",
        "type": "debug",
        "z": "7d5c6b14.d2af94",
        "name": "",
        "active": false,
        "tosidebar": true,
        "console": false,
        "tostatus": false,
        "complete": "false",
        "x": 870,
        "y": 240,
        "wires": []
    },
    {
        "id": "9b919750.d6fff8",
        "type": "function",
        "z": "7d5c6b14.d2af94",
        "name": "Humidity",
        "func": "var humValue = msg.payload.valueOf();\nvar newMessage =  { payload:
↪{  \"result\": humValue, \"Datastream\": {\"@iot.id\": 113}} };\nnewMessage.headers␣
↪= {\"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 440,
        "y": 200,
        "wires": [
            [
                "c67491fc.f4755"
            ]
        ]
    },
    {
        "id": "c67491fc.f4755",
        "type": "http request",
        "z": "7d5c6b14.d2af94",
        "name": "POST Observation",
        "method": "POST",
        "ret": "obj",
        "paytoqs": false,
        "url": "http://iot.gis.bgu.tum.de:8081/FROST-Server-gi3/v1.0/Observations",
        "tls": "",
        "proxy": "",
        "authType": "basic",
        "x": 690,
        "y": 240,
        "wires": [
            [
                "a3b86e6f.56637"
            ]
        ]
    },
    {
```

(continues on next page)

(continued from previous page)

```
        "id": "92425d9f.bca98",
        "type": "function",
        "z": "7d5c6b14.d2af94",
        "name": "Temperature",
        "func": "var tempValue = msg.payload.valueOf();\nvar newMessage =  { payload:
↪{  \"result\": tempValue, \"Datastream\": {\"@iot.id\": 112}} };\nnewMessage.
↪headers = {\"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 450,
        "y": 160,
        "wires": [
            [
                "c67491fc.f4755"
            ]
        ]
    },
    {
        "id": "a16b3beb.1a5028",
        "type": "debug",
        "z": "7d5c6b14.d2af94",
        "name": "",
        "active": true,
        "tosidebar": true,
        "console": false,
        "tostatus": false,
        "complete": "payload",
        "targetType": "msg",
        "x": 490,
        "y": 60,
        "wires": []
    },
    {
        "id": "681442b7.1266ac",
        "type": "ttn uplink",
        "z": "7d5c6b14.d2af94",
        "name": "TTN Input",
        "app": "58ceff1f.8576a",
        "dev_id": "tum-gis-device3",
        "field": "",
        "x": 80,
        "y": 60,
        "wires": [
            [
                "531b0b35.751284"
            ]
        ]
    },
    {
        "id": "531b0b35.751284",
        "type": "cayennelpp-decoder",
        "z": "7d5c6b14.d2af94",
        "name": "",
        "x": 260,
        "y": 60,
        "wires": [
            [
```

(continues on next page)

(continued from previous page)

```
                "9010a80d.dfcdb8",
                "a16b3beb.1a5028"
            ]
        ]
    },
    {
        "id": "620b1ab2.a69224",
        "type": "function",
        "z": "7d5c6b14.d2af94",
        "name": "Air Quality",
        "func": "var quality = msg.payload.valueOf();\nvar newMessage =  { payload: {␣
↪ \"result\": quality, \"Datastream\": {\"@iot.id\": 119}} };\nnewMessage.headers =
↪{\"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 450,
        "y": 240,
        "wires": [
            [
                "c67491fc.f4755"
            ]
        ]
    },
    {
        "id": "eee677bf.fe16d8",
        "type": "function",
        "z": "7d5c6b14.d2af94",
        "name": "Light",
        "func": "var light = msg.payload.valueOf();\nvar newMessage =  { payload: {  \
↪"result\": light, \"Datastream\": {\"@iot.id\": 117}} };\nnewMessage.headers = {\
↪"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 430,
        "y": 280,
        "wires": [
            [
                "c67491fc.f4755"
            ]
        ]
    },
    {
        "id": "4b424590.139b4c",
        "type": "function",
        "z": "7d5c6b14.d2af94",
        "name": "Barometric Pressure",
        "func": "var pressure = msg.payload.valueOf();\nvar newMessage =  { payload:
↪{  \"result\": pressure, \"Datastream\": {\"@iot.id\": 114}} };\nnewMessage.headers␣
↪= {\"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 480,
        "y": 320,
        "wires": [
            [
                "c67491fc.f4755"
            ]
```

(continues on next page)

(continued from previous page)

```
        ]
    },
    {
        "id": "bc3f8433.7b89f8",
        "type": "function",
        "z": "7d5c6b14.d2af94",
        "name": "co2",
        "func": "var co2 = msg.payload.valueOf();\nvar newMessage =  { payload: {  \
↪"result\": co2, \"Datastream\": {\"@iot.id\": 118}} };\nnewMessage.headers = {\
↪"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 430,
        "y": 360,
        "wires": [
            [
                "c67491fc.f4755"
            ]
        ]
    },
    {
        "id": "13968bca.c2cd34",
        "type": "function",
        "z": "7d5c6b14.d2af94",
        "name": "Gas",
        "func": "var gas = msg.payload.valueOf();\nvar newMessage =  { payload: {  \
↪"result\": gas, \"Datastream\": {\"@iot.id\": 121}} };\nnewMessage.headers = {\
↪"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 430,
        "y": 400,
        "wires": [
            [
                "c67491fc.f4755"
            ]
        ]
    },
    {
        "id": "c7fcb372.4b2df",
        "type": "function",
        "z": "7d5c6b14.d2af94",
        "name": "Loudness",
        "func": "var loudness = msg.payload.valueOf();\nvar newMessage =  { payload:
↪{  \"result\": loudness, \"Datastream\": {\"@iot.id\": 120}} };\nnewMessage.headers␣
↪= {\"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 440,
        "y": 440,
        "wires": [
            [
                "c67491fc.f4755"
            ]
        ]
    },
    {
```

(continues on next page)

(continued from previous page)

```
        "id": "b52fa683.7c5fb8",
        "type": "function",
        "z": "7d5c6b14.d2af94",
        "name": "Dust-PM2.5",
        "func": "var Dust = msg.payload.valueOf();\nvar newMessage =  { payload: {  \
↪"result\": Dust, \"Datastream\": {\"@iot.id\": 128}} };\nnewMessage.headers = {\
↪"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 450,
        "y": 480,
        "wires": [
            [
                "c67491fc.f4755"
            ]
        ]
    },
    {
        "id": "58ceff1f.8576a",
        "type": "ttn app",
        "z": "",
        "appId": "gis-tum-sensors",
        "accessKey": "ttn-account-ACCESSKEY_HERE",
        "discovery": "discovery.thethingsnetwork.org:1900"
    }
]
```

2.2.6 References

- *Arduino Sketch for Indoor Ambient Monitoring sensor node*
- *Node red flow for Indoor Ambient Monitoring sensor node*
- Wiki guide for Seeeduino LoRaWAN board
- Adding Seeed boards to Arduino IDE

2.3 Feather M0 LoRa in TFA Housing

This sensor node is made to showcase a use-case of LoRaWAN technology for outdoor weather monitoring. For achieving this a Feather M0 LoRa module was used with temperature and pressure sensor. The entire setup was carefully placed in the TFA Housing which is an all-weather protective cover for outdoor transmitters. In this example we measure parameters such as temperature, humidity, altitude, and air pressure.

2.3.1 Hardware

To build this sensor node we have used following hardware components:

- Adafruit Feather M0 LoRA board
- Grove - DHT-22 Temperature & Humidity Sensor
- Grove - Barometric Pressure Sensor

Fig. 7: Sensor node in TFA Housing.

- Breadboard
- TFA Protective Cover
- 6600 mAH Battery

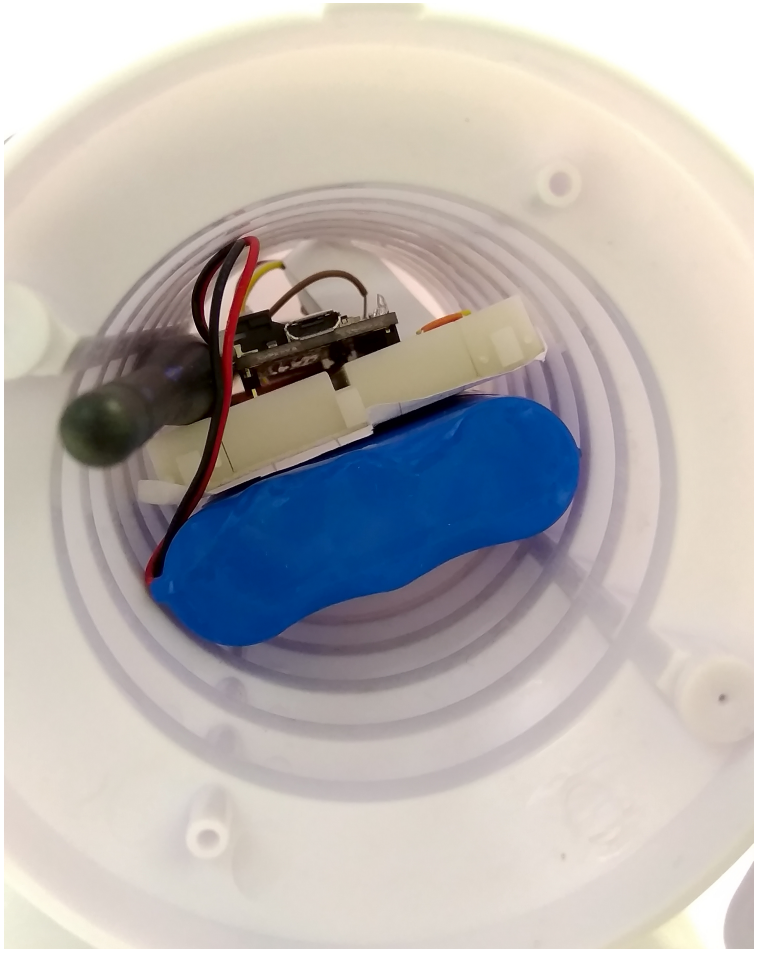


Fig. 8: Inside view of Sensor node in TFA Housing

Also, as the final hardware setup with antenna couldn't completely fit into the casing, a small hole was made at the bottom of the casing to allow the remaining portion of antenna to stay outside.

2.3.2 Wiring setup

First of all, the Feather M0 LoRa board was prepared by soldering the board with the provided grid of pins. Then the board is connected with the sensors using a breadboard. The sensor connections were made using the connector cables as following:

DHT-22 Sensor connections:

- Feather 3V to DHT22 pin 1
- Feather GND to DHT22 pin 4
- Feather pin 12 to DHT22 pin 2

Fig. 9: Bottom view of Sensor node in TFA Housing

- Resistor between DHT pin 1 and DHT pin 2

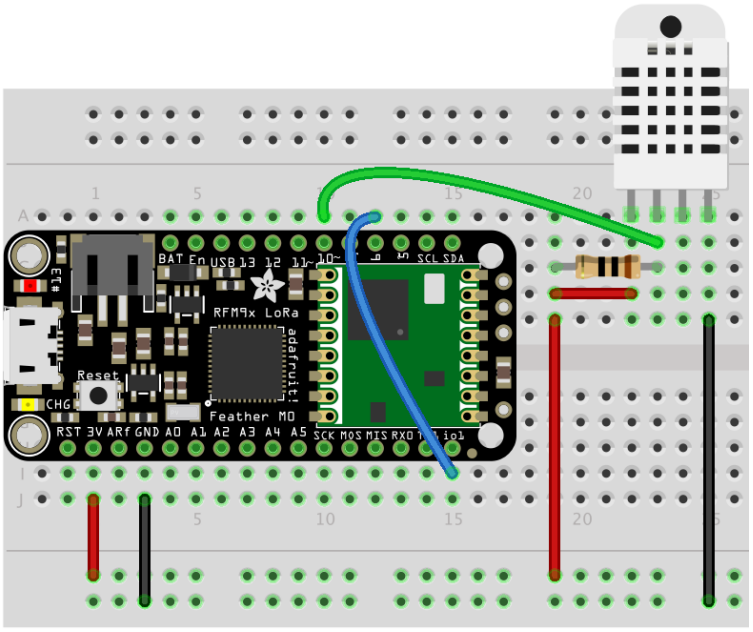


Fig. 10: Wiring with DHT-22 Sensor

Grove-Barometer Sensor connections:

- Feather SCL to Barometer Sensor pin 1 (yellow)
- Feather SDA to Barometer Sensor pin 2 (white)
- Feather 3V to Barometer Sensor pin 3 (red)
- Feather GND to Barometer Sensor pin 4 (black)

Apart from this, Feather pin 6 should be permanently wired with Feather pin io1 as shown in the figure above.

To ensure the durable connections, smaller jumper wires were used on the breadboard instead of longer connecting cables. Sensors and cables were also supported with an insulating duct tape.

Final hardware setup looked as following:

Once all these connection were made, the board is connected with a computer using a USB cable. Further, steps of software part needs to be followed.

2.3.3 Software

To create this node, we use Arduino IDE for setting up the Feather M0 LoRa module. First, install the Feather M0 LoRa board to your Arduino IDE and select the correct port. Then following libraries needs to be installed before compiling the code:

- lmic.h for implementing LoRaWAN on Arduino hardware.
- hal/hal.h bundled with lmic library.
- Adafruit_SleepyDog.h for controlling low power sleep mode.
- Wire.h to communicate with I2C devices.

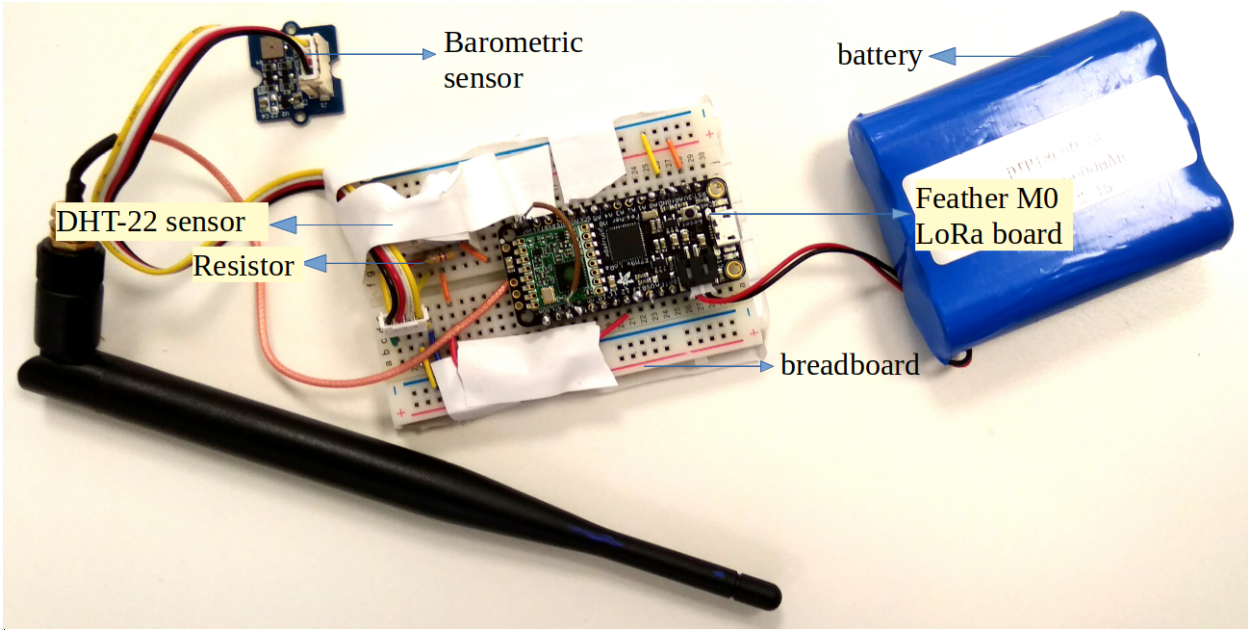


Fig. 11: Final hardware wiring

- BMP085.h for Barometer sensor.
- DHT.h for reading DHT-22 sensor.
- CayenneLPP.h for Cayenne Protocol.

Apart from this, SPI.h library is also used for communicating with serial peripheral interface but it is already inbuilt in Arduino IDE and is not required to be separately installed.

Now download and run the *Arduino Sketch for Outdoor Weather Monitoring sensor node* file in the Arduino IDE. This code was created by merging the example code of both the sensors and the ttn-otaa example from the lmic library. Some required changes were made while merging the example codes. The user should change the network session key, app session key and device address in the code before compiling. These keys can be obtained from the TTN, SWM or other service providers.

Listing 9: Modify the keys in highlighted lines.

```
// LoRaWAN NwkSKey, network session key
// This should be in big-endian (aka msb).
static const PROGMEM u1_t NWKSKEY[16] = {NETWORK_SESSION_KEY_HERE_IN_MSB_FORMAT};

// LoRaWAN AppSKey, application session key
// This should also be in big-endian (aka msb).
static const u1_t PROGMEM APPSKEY[16] = {APPLICATION_SESSION_KEY_HERE_IN_MSB_FORMAT};

// LoRaWAN end-device address (DevAddr)
// See http://thethingsnetwork.org/wiki/AddressSpace
// The library converts the address to network byte order as needed, so this should␣
↪be in big-endian (aka msb) too.
static const u4_t DEVADDR = 0x260XXXXX   ; // <-- Change this address for every node!
```

The pin mapping configured in the code should also be verified for the board that is being used. Current pin mapping is set as per the Feather M0 LoRa board.

Listing 10: Set the correct pin mapping for the board that is used.

```
// Pin mapping
const lmic_pinmap lmic_pins = {
    .nss = 8,
    .rxtx = LMIC_UNUSED_PIN,
    .rst = 4,
    .dio = {3, 6, LMIC_UNUSED_PIN},
```

Following is the example code that can be used to measure the battery voltage of the Feather M0 LoRa board:

Listing 11: Code for measuring the battery voltage

```
            measuredvbat = analogRead(VBATPIN);
            measuredvbat *= 2;    // we divided by 2, so multiply back
            measuredvbat *= 3.3;  // Multiply by 3.3V, our reference voltage
            measuredvbat /= 1024; // convert to voltage

            SERIALDEBUG_PRINT(" %\t");
            SERIALDEBUG_PRINT("Battery Voltage: ");
```

2.3.4 Services

This node is connected using the TheThingsNetwork service. Further, a node-red work bench is used to forward this collected data from the TTN platform to the OGC Sensor Things API configured on the FROST Server. The node-red workbench that was used for forwarding the data is available at *Node red flow for Outdoor Weather Monitoring sensor node*. To use this node-red-workbench go to the node-red platform https://iot.gis.bgu.tum.de:1885/, login with the credentials, go to the options and select Import>Clipboard. Select the downloaded .json file with the given option and click on import. Make necessary changes and deploy the flow.

Datastreams setup for this sensor node on the FROST server can be seen at: http://iot.gis.bgu.tum.de:8081/FROST-Server-gi3/v1.0/Things(20)/Datastreams

The node-red workbench for this sensor node could be found at: https://iot.gis.bgu.tum.de:1885/#flow/f6f7a740.c6b338

The GRAFANA dash-board for visualizing the collected data is available at: https://iot.gis.bgu.tum.de:3050/d/sMJ3jAAWz/featherm0lora-in-tfa-housing?orgId=1

2.3.5 Code files

Listing 12: Arduino Sketch for Outdoor Weather Monitoring sensor node

```
/*******************************************************************************
 * Copyright (c) 2015 Thomas Telkamp and Matthijs Kooijman
 *
 *
 *
 *
 * Permission is hereby granted, free of charge, to anyone
 * obtaining a copy of this document and accompanying files,
 * to do whatever they want with them without any restriction,
 * including, but not limited to, copying, modification and redistribution.
 * NO WARRANTY OF ANY KIND IS PROVIDED.
```

(continues on next page)

(continued from previous page)

```
 *
 * This example sends a valid LoRaWAN packet with payload "Hello,
 * world!", using frequency and encryption settings matching those of
 * the The Things Network.
 *
 * This uses ABP (Activation-by-personalisation), where a DevAddr and
 * Session keys are preconfigured (unlike OTAA, where a DevEUI and
 * application key is configured, while the DevAddr and session keys are
 * assigned/generated in the over-the-air-activation procedure).
 *
 * Note: LoRaWAN per sub-band duty-cycle limitation is enforced (1% in
 * g1, 0.1% in g2), but not the TTN fair usage policy (which is probably
 * violated by this sketch when left running for longer)!
 *
 * To use this sketch, first register your application and device with
 * the things network, to set or generate a DevAddr, NwkSKey and
 * AppSKey. Each device should have their own unique values for these
 * fields.
 *
 * Do not forget to define the radio type correctly in config.h.
 *
 *******************************************************************************/
// #define SERIALDEBUG

#ifdef SERIALDEBUG
  #define SERIALDEBUG_PRINT(...) Serial.print(__VA_ARGS__)
  #define SERIALDEBUG_PRINTLN(...) Serial.println(__VA_ARGS__)
#else
  #define SERIALDEBUG_PRINT(...)
  #define SERIALDEBUG_PRINTLN(...)
#endif


#include <lmic.h>
#include <hal/hal.h>
#include <SPI.h>
#include <Adafruit_SleepyDog.h>
#include <DHT.h>
#include <CayenneLPP.h>
#include "BMP085.h"
#include <Wire.h>

CayenneLPP lpp(51);

#define DHTPIN            12        // Pin which is connected to the DHT sensor.
#define DHTTYPE           DHT22     // DHT 22 (AM2302)

// DHT_Unified dht(DHTPIN, DHTTYPE);
DHT dht(DHTPIN, DHTTYPE);

#define VBATPIN A7

float temperature2;
float pressure;
float atm;
float altitude;
BMP085 myBarometer;
```

(continues on next page)

(continued from previous page)

```

// LoRaWAN NwkSKey, network session key
// This should be in big-endian (aka msb).
static const PROGMEM u1_t NWKSKEY[16] = {NETWORK_SESSION_KEY_HERE_IN_MSB_FORMAT};

// LoRaWAN AppSKey, application session key
// This should also be in big-endian (aka msb).
static const u1_t PROGMEM APPSKEY[16] = {APPLICATION_SESSION_KEY_HERE_IN_MSB_FORMAT};

// LoRaWAN end-device address (DevAddr)
// See http://thethingsnetwork.org/wiki/AddressSpace
// The library converts the address to network byte order as needed, so this should be in big-endian (aka msb) too.
static const u4_t DEVADDR = 0x260XXXXX   ; // <-- Change this address for every node!

// These callbacks are only used in over-the-air activation, so they are
// left empty here (we cannot leave them out completely unless
// DISABLE_JOIN is set in config.h, otherwise the linker will complain).
void os_getArtEui (u1_t* buf) { }
void os_getDevEui (u1_t* buf) { }
void os_getDevKey (u1_t* buf) { }

static osjob_t sendjob;

// Schedule TX every this many seconds (might become longer due to duty
// cycle limitations).
const unsigned TX_INTERVAL = 1;        // seconds transmit cycle plus ...
const unsigned SLEEP_TIME = 60*9+55;   // seconds sleep time plus ...
const unsigned MEASURE_TIME = 2;       // seconds measuring time should lead to ...
                                       // 5 minute(s) total cycle time

// Pin mapping
const lmic_pinmap lmic_pins = {
    .nss = 8,
    .rxtx = LMIC_UNUSED_PIN,
    .rst = 4,
    .dio = {3, 6, LMIC_UNUSED_PIN},
};


void onEvent (ev_t ev) {
//    Serial.print(os_getTime());
//    Serial.print(": ");
    SERIALDEBUG_PRINT(os_getTime());
    SERIALDEBUG_PRINT(": ");
    switch(ev) {
        case EV_SCAN_TIMEOUT:
            SERIALDEBUG_PRINTLN(F("EV_SCAN_TIMEOUT"));
            break;
        case EV_BEACON_FOUND:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_FOUND"));
            break;
        case EV_BEACON_MISSED:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_MISSED"));
            break;
        case EV_BEACON_TRACKED:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_TRACKED"));
```

(continues on next page)

(continued from previous page)

```
            break;
        case EV_JOINING:
            SERIALDEBUG_PRINTLN(F("EV_JOINING"));
            break;
        case EV_JOINED:
            SERIALDEBUG_PRINTLN(F("EV_JOINED"));
            break;
        case EV_RFU1:
            SERIALDEBUG_PRINTLN(F("EV_RFU1"));
            break;
        case EV_JOIN_FAILED:
            SERIALDEBUG_PRINTLN(F("EV_JOIN_FAILED"));
            break;
        case EV_REJOIN_FAILED:
            SERIALDEBUG_PRINTLN(F("EV_REJOIN_FAILED"));
            break;
        case EV_TXCOMPLETE:
            digitalWrite(LED_BUILTIN, LOW);     // turn the LED off by making the␣
↪voltage LOW
            SERIALDEBUG_PRINTLN(F("EV_TXCOMPLETE (includes waiting for RX windows)"));
            if (LMIC.txrxFlags & TXRX_ACK)
              SERIALDEBUG_PRINTLN(F("Received ack"));
            if (LMIC.dataLen) {
              SERIALDEBUG_PRINT(F("Received "));
              SERIALDEBUG_PRINT(LMIC.dataLen);
              SERIALDEBUG_PRINTLN(F(" bytes of payload"));
            }
            // Schedule next transmission
            os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(TX_INTERVAL), do_
↪send);

            SERIALDEBUG_PRINTLN("going to sleep now ... ");
            // lmic library sleeps automatically after transmission has been completed
            for(int i= 0; i < SLEEP_TIME / 16; i++) {
              Watchdog.sleep(16000); // maximum seems to be 16 seconds
              SERIALDEBUG_PRINT('.');
            }
            if (SLEEP_TIME % 16) {
              Watchdog.sleep((SLEEP_TIME % 16)*1000);
              SERIALDEBUG_PRINT('*');
            }
            SERIALDEBUG_PRINTLN("... woke up again");

            break;
        case EV_LOST_TSYNC:
            SERIALDEBUG_PRINTLN(F("EV_LOST_TSYNC"));
            break;
        case EV_RESET:
            SERIALDEBUG_PRINTLN(F("EV_RESET"));
            break;
        case EV_RXCOMPLETE:
            // data received in ping slot
            SERIALDEBUG_PRINTLN(F("EV_RXCOMPLETE"));
            break;
        case EV_LINK_DEAD:
            SERIALDEBUG_PRINTLN(F("EV_LINK_DEAD"));
            break;
```

(continues on next page)

(continued from previous page)

```
        case EV_LINK_ALIVE:
            SERIALDEBUG_PRINTLN(F("EV_LINK_ALIVE"));
            break;
         default:
            SERIALDEBUG_PRINTLN(F("Unknown event"));
            break;
    }
}

void do_send(osjob_t* j){
    // Check if there is not a current TX/RX job running
    if (LMIC.opmode & OP_TXRXPEND) {
        SERIALDEBUG_PRINTLN(F("OP_TXRXPEND, not sending"));
    } else {
        // Prepare upstream data transmission at the next possible time.

        float temperature, humidity, measuredvbat;
        int16_t int16_temperature, int16_humidity, int16_vbat;

        // Start a measurement to update the sensor's internal temperature & humidity␣
↪reading
        SERIALDEBUG_PRINTLN("Start measurement...");
        temperature = dht.readTemperature();
        // delay(2000);
        Watchdog.sleep(2000);
        // Now read the recently measured temperature (2 secs ago) as Celsius (the␣
↪default)
        temperature = dht.readTemperature();
        // Read the recently measured humidity (2 secs ago)
        humidity = dht.readHumidity();
        SERIALDEBUG_PRINTLN("... finished!");

        // Check if any reads failed and exit early (to try again).
        if (isnan(humidity) || isnan(temperature)) {
            SERIALDEBUG_PRINTLN("Failed to read from DHT sensor!");
            for (int i=0; i<5; i++) {
              digitalWrite(LED_BUILTIN, HIGH);    // turn the LED on by making the␣
↪voltage HIGH
              delay(150);
              digitalWrite(LED_BUILTIN, LOW);    // turn the LED on by making the␣
↪voltage HIGH
              delay(150);
            }
            // ok, then wait for another period and try it again
            os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(TX_INTERVAL), do_
↪send);
        } else {
            SERIALDEBUG_PRINT("Humidity: ");
            SERIALDEBUG_PRINT(humidity);
            SERIALDEBUG_PRINT(" %\t");
            SERIALDEBUG_PRINT("Temperature: ");
            SERIALDEBUG_PRINT(temperature);
            SERIALDEBUG_PRINT(" *C ");

            measuredvbat = analogRead(VBATPIN);
            measuredvbat *= 2;    // we divided by 2, so multiply back
            measuredvbat *= 3.3;  // Multiply by 3.3V, our reference voltage
```

(continues on next page)

(continued from previous page)

```
            measuredvbat /= 1024; // convert to voltage

            SERIALDEBUG_PRINT(" %\t");
            SERIALDEBUG_PRINT("Battery Voltage: ");
            SERIALDEBUG_PRINTLN(measuredvbat);

            temperature2 = myBarometer.bmp085GetTemperature(myBarometer.
↪bmp085ReadUT()); //Get the temperature, bmp085ReadUT MUST be called first
            pressure = myBarometer.bmp085GetPressure(myBarometer.bmp085ReadUP());//
↪Get the temperature

           /*
            To specify a more accurate altitude, enter the correct mean sea level
            pressure level.  For example, if the current pressure level is 1019.00 hPa
            enter 101900 since we include two decimal places in the integer value
           */
           altitude = myBarometer.calcAltitude(pressure);

           atm = pressure / 101325;

           lpp.reset();
           lpp.addTemperature(1, temperature);
           lpp.addRelativeHumidity(2, humidity);
           lpp.addAnalogInput(3, measuredvbat);
           lpp.addTemperature(4, temperature2);
           lpp.addBarometricPressure(5, pressure/100);
           lpp.addAnalogInput(6, atm);
           lpp.addAnalogInput(7, altitude);

//            LMIC_setTxData2(1, mydata, sizeof(mydata)-1, 0);

            // send the 6 bytes payload to LoRaWAN port 7
            LMIC_setTxData2(7, lpp.getBuffer(), lpp.getSize(), 0);
            SERIALDEBUG_PRINTLN(F("Packet queued"));
            digitalWrite(LED_BUILTIN, HIGH);    // turn the LED on by making the␣
↪voltage HIGH
        }

        // LMIC_setTxData2(1, mydata, sizeof(mydata)-1, 0);
        // Serial.println(F("Packet queued"));
    }
    // Next TX is scheduled after TX_COMPLETE event.
}

void setup() {
    delay(5000);

    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW

#ifdef SERIALDEBUG
    Serial.begin(9600);
    // while (!Serial);
#endif

    dht.begin();
    myBarometer.init();
```

(continues on next page)

(continued from previous page)

```
    SERIALDEBUG_PRINTLN(F("Starting"));

    #ifdef VCC_ENABLE
    // For Pinoccio Scout boards
    pinMode(VCC_ENABLE, OUTPUT);
    digitalWrite(VCC_ENABLE, HIGH);
    delay(1000);
    #endif

    // LMIC init
    os_init();
    // Reset the MAC state. Session and pending data transfers will be discarded.
    LMIC_reset();
    LMIC_setClockError(MAX_CLOCK_ERROR * 1 / 100);

    // Set static session parameters. Instead of dynamically establishing a session
    // by joining the network, precomputed session parameters are be provided.
    #ifdef PROGMEM
    // On AVR, these values are stored in flash and only copied to RAM
    // once. Copy them to a temporary buffer here, LMIC_setSession will
    // copy them into a buffer of its own again.
    uint8_t appskey[sizeof(APPSKEY)];
    uint8_t nwkskey[sizeof(NWKSKEY)];
    memcpy_P(appskey, APPSKEY, sizeof(APPSKEY));
    memcpy_P(nwkskey, NWKSKEY, sizeof(NWKSKEY));
    LMIC_setSession (0x1, DEVADDR, nwkskey, appskey);
    #else
    // If not running an AVR with PROGMEM, just use the arrays directly
    LMIC_setSession (0x1, DEVADDR, NWKSKEY, APPSKEY);
    #endif

    #if defined(CFG_eu868)
    // Set up the channels used by the Things Network, which corresponds
    // to the defaults of most gateways. Without this, only three base
    // channels from the LoRaWAN specification are used, which certainly
    // works, so it is good for debugging, but can overload those
    // frequencies, so be sure to configure the full frequency range of
    // your network here (unless your network autoconfigures them).
    // Setting up channels should happen after LMIC_setSession, as that
    // configures the minimal channel set.
    // NA-US channels 0-71 are configured automatically
    LMIC_setupChannel(0, 868100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(1, 868300000, DR_RANGE_MAP(DR_SF12, DR_SF7B), BAND_CENTI);
↪// g-band
    LMIC_setupChannel(2, 868500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(3, 867100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(4, 867300000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(5, 867500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(6, 867700000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(7, 867900000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
```

(continues on next page)

(continued from previous page)

```
    LMIC_setupChannel(8, 868800000, DR_RANGE_MAP(DR_FSK,  DR_FSK),  BAND_MILLI);
// g2-band
    // TTN defines an additional channel at 869.525Mhz using SF9 for class B
    // devices' ping slots. LMIC does not have an easy way to define set this
    // frequency and support for class B is spotty and untested, so this
    // frequency is not configured here.
    #elif defined(CFG_us915)
    // NA-US channels 0-71 are configured automatically
    // but only one group of 8 should (a subband) should be active
    // TTN recommends the second sub band, 1 in a zero based count.
    // https://github.com/TheThingsNetwork/gateway-conf/blob/master/US-global_conf.
json
    LMIC_selectSubBand(1);
    #endif

    // Disable link check validation
    LMIC_setLinkCheckMode(0);

    // TTN uses SF9 for its RX2 window.
    LMIC.dn2Dr = DR_SF9;

    // Set data rate and transmit power for uplink (note: txpow seems to be ignored
by the library)
    LMIC_setDrTxpow(DR_SF7,14);

    // Start job
    do_send(&sendjob);
}

void loop() {
    os_runloop_once();
}
```

Listing 13: Node red flow for Outdoor Weather Monitoring sensor node

```
[
    {
        "id": "f6f7a740.c6b338",
        "type": "tab",
        "label": "Device 2",
        "disabled": false,
        "info": ""
    },
    {
        "id": "fafe9ad3.9659e8",
        "type": "switch",
        "z": "f6f7a740.c6b338",
        "name": "Separate",
        "property": "key",
        "propertyType": "msg",
        "rules": [
            {
                "t": "cont",
                "v": "temperature_1",
                "vt": "str"
            },
```

(continues on next page)

(continued from previous page)

```
            {
                "t": "cont",
                "v": "humidity",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "analog_in_3",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "temperature_4",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "barometric",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "analog_in_6",
                "vt": "str"
            },
            {
                "t": "cont",
                "v": "analog_in_7",
                "vt": "str"
            }
        ],
        "checkall": "true",
        "repair": false,
        "outputs": 7,
        "x": 220,
        "y": 180,
        "wires": [
            [
                "492a1844.49a228"
            ],
            [
                "b5be1839.3121a8"
            ],
            [
                "d7e35050.187eb"
            ],
            [
                "c5363ad1.5d3418"
            ],
            [
                "ee2891fa.0dbbe"
            ],
            [
                "71354cb4.e6af04"
            ],
            [
                "d48c0c97.4eb08"
```

(continues on next page)

(continued from previous page)

```
            ]
        ]
    },
    {
        "id": "ccb2fb81.aacd58",
        "type": "split",
        "z": "f6f7a740.c6b338",
        "name": "",
        "splt": "\\n",
        "spltType": "str",
        "arraySplt": 1,
        "arraySpltType": "len",
        "stream": false,
        "addname": "key",
        "x": 90,
        "y": 180,
        "wires": [
            [
                "fafe9ad3.9659e8"
            ]
        ]
    },
    {
        "id": "657fd8a7.01c5e8",
        "type": "debug",
        "z": "f6f7a740.c6b338",
        "name": "",
        "active": false,
        "tosidebar": true,
        "console": false,
        "tostatus": false,
        "complete": "false",
        "x": 870,
        "y": 240,
        "wires": []
    },
    {
        "id": "b5be1839.3121a8",
        "type": "function",
        "z": "f6f7a740.c6b338",
        "name": "Humidity",
        "func": "var humValue = msg.payload.valueOf();\nvar newMessage =  { payload:
↪{  \"result\": humValue, \"Datastream\": {\"@iot.id\": 106}} };\nnewMessage.headers␣
↪= {\"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 440,
        "y": 200,
        "wires": [
            [
                "dd5d521b.5c984"
            ]
        ]
    },
    {
        "id": "dd5d521b.5c984",
        "type": "http request",
```

(continues on next page)

(continued from previous page)

```
        "z": "f6f7a740.c6b338",
        "name": "POST Observation",
        "method": "POST",
        "ret": "obj",
        "paytoqs": false,
        "url": "http://iot.gis.bgu.tum.de:8081/FROST-Server-gi3/v1.0/Observations",
        "tls": "",
        "proxy": "",
        "authType": "basic",
        "x": 690,
        "y": 240,
        "wires": [
            [
                "657fd8a7.01c5e8"
            ]
        ]
    },
    {
        "id": "492a1844.49a228",
        "type": "function",
        "z": "f6f7a740.c6b338",
        "name": "Temperature",
        "func": "var tempValue = msg.payload.valueOf();\nvar newMessage =  { payload:
↪{  \"result\": tempValue, \"Datastream\": {\"@iot.id\": 105}} };\nnewMessage.
↪headers = {\"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 450,
        "y": 160,
        "wires": [
            [
                "dd5d521b.5c984"
            ]
        ]
    },
    {
        "id": "739d03d0.606a6c",
        "type": "debug",
        "z": "f6f7a740.c6b338",
        "name": "",
        "active": true,
        "tosidebar": true,
        "console": false,
        "tostatus": false,
        "complete": "payload",
        "targetType": "msg",
        "x": 490,
        "y": 60,
        "wires": []
    },
    {
        "id": "cb8ef1e2.a85f6",
        "type": "ttn uplink",
        "z": "f6f7a740.c6b338",
        "name": "TTN Input",
        "app": "58ceff1f.8576a",
        "dev_id": "tum-gis-device2",
```

(continues on next page)

(continued from previous page)

```
        "field": "",
        "x": 80,
        "y": 60,
        "wires": [
            [
                "aae507e3.771c18"
            ]
        ]
    },
    {
        "id": "aae507e3.771c18",
        "type": "cayennelpp-decoder",
        "z": "f6f7a740.c6b338",
        "name": "",
        "x": 260,
        "y": 60,
        "wires": [
            [
                "ccb2fb81.aacd58",
                "739d03d0.606a6c"
            ]
        ]
    },
    {
        "id": "d7e35050.187eb",
        "type": "function",
        "z": "f6f7a740.c6b338",
        "name": "Battery Voltage",
        "func": "var Batteryvolt = msg.payload.valueOf();\nvar newMessage =  {␣
↪payload: {  \"result\": Batteryvolt, \"Datastream\": {\"@iot.id\": 107}} };
↪\nnewMessage.headers = {\"Content-type\" : \"application/json\"}\nreturn newMessage;
↪",
        "outputs": 1,
        "noerr": 0,
        "x": 460,
        "y": 240,
        "wires": [
            [
                "dd5d521b.5c984"
            ]
        ]
    },
    {
        "id": "c5363ad1.5d3418",
        "type": "function",
        "z": "f6f7a740.c6b338",
        "name": "Temperature2",
        "func": "var tempValue = msg.payload.valueOf();\nvar newMessage =  { payload:
↪{  \"result\": tempValue, \"Datastream\": {\"@iot.id\": 108}} };\nnewMessage.
↪headers = {\"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 460,
        "y": 280,
        "wires": [
            [
                "dd5d521b.5c984"
```

(continues on next page)

(continued from previous page)

```
            ]
        ]
    },
    {
        "id": "ee2891fa.0dbbe",
        "type": "function",
        "z": "f6f7a740.c6b338",
        "name": "Barometric Pressure",
        "func": "var pressure = msg.payload.valueOf();\nvar newMessage =  { payload:
↪{  \"result\": pressure, \"Datastream\": {\"@iot.id\": 109}} };\nnewMessage.headers␣
↪= {\"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 480,
        "y": 320,
        "wires": [
            [
                "dd5d521b.5c984"
            ]
        ]
    },
    {
        "id": "71354cb4.e6af04",
        "type": "function",
        "z": "f6f7a740.c6b338",
        "name": "Pressure atm",
        "func": "var atm = msg.payload.valueOf();\nvar newMessage =  { payload: {  \
↪"result\": atm, \"Datastream\": {\"@iot.id\": 110}} };\nnewMessage.headers = {\
↪"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 450,
        "y": 360,
        "wires": [
            [
                "dd5d521b.5c984"
            ]
        ]
    },
    {
        "id": "d48c0c97.4eb08",
        "type": "function",
        "z": "f6f7a740.c6b338",
        "name": "Altitude",
        "func": "var altitude = msg.payload.valueOf();\nvar newMessage =  { payload:
↪{  \"result\": altitude, \"Datastream\": {\"@iot.id\": 111}} };\nnewMessage.headers␣
↪= {\"Content-type\" : \"application/json\"}\nreturn newMessage;",
        "outputs": 1,
        "noerr": 0,
        "x": 440,
        "y": 400,
        "wires": [
            [
                "dd5d521b.5c984"
            ]
        ]
    },
```

(continues on next page)

(continued from previous page)

```
    {
        "id": "58ceff1f.8576a",
        "type": "ttn app",
        "z": "",
        "appId": "gis-tum-sensors",
        "accessKey": "ttn-account-ACCESSKEY_HERE",
        "discovery": "discovery.thethingsnetwork.org:1900"
    }
]
```

2.3.6 References

- *Arduino Sketch for Outdoor Weather Monitoring sensor node*
- *Node red flow for Outdoor Weather Monitoring sensor node*
- Feather M0 LoRa Arduino IDE Setup
- Sample Arduino codes of using an Adafruit feather M0 LoRa

2.4 Adafruit 32u4 LoRa

This tutorial is made to showcase the use of Adafruit 32u4 board to create a LoRaWAN enabled sensor node. In the following example, a temperature and humidity sensor was used with the Adafruit 32u4 board.

2.4.1 Hardware

To build this sensor node we have used following hardware components:

- Adafruit Feather 32u4 LoRa module
- Grove - DHT-22 Temperature & Humidity Sensor
- Breadboard
- Battery
- Resistor: 4.7k to 10k Ohm

Microcontroller

The Adafruit Feather 32u4 LoRa module is operated by the 8bit ATmega32u4 microcontroller running at 8MHz. It has 32 KB flash memory (to store the program code) and 2 KB of RAM (to store variables, status information, and buffers). The operating voltage of the board is 3.3V (this is important when attaching sensors and other peripherals; they also must operate on 3.3V). The board offers 20 general purpose digital input/output pins (20 GPIOs) with 10 analog input pins (with 12bit analog digital converters (ADC)), one serial port (programmable Universal Asynchronous Receiver and Transmitter, UART), one I2C port, one SPI port, one USB port. The board comes with an embedded Lithium polymer battery management chip and status indicator led, which allows to directly connect a 3.7V LiPo rechargeable battery that will be automatically recharged when the board is powered over its USB connector. The Adafruit Feather 32u4 LoRa board is available in German shops from around 37 € to 45 €.

The LoRa transmitter and receiver is encapsulated within an RFM95 module from the company HopeRF. This module uses the LoRa chip SX1276 from the company Semtech and is dedicated to the 868 MHz frequency band. The RFM95

module is connected via SPI interface to the microcontroller. Most of the required connections of the LoRa transceiver pins with the microcontroller are already built-in on the Adafruit Feather 32u4 LoRa board. However, Digital Pin 6 of the microcontroller must be connected to DIO1 of the LoRa transceiver module in addition using a simple wire. Since the module only implements the LoRa physical layer, the LoRaWAN protocol stack must be implemented in software on the microcontroller. We are using the Arduino library LMIC for that purpose (see below). The implemented LoRaWAN functionality is compatible with LoRaWAN Class A/C.

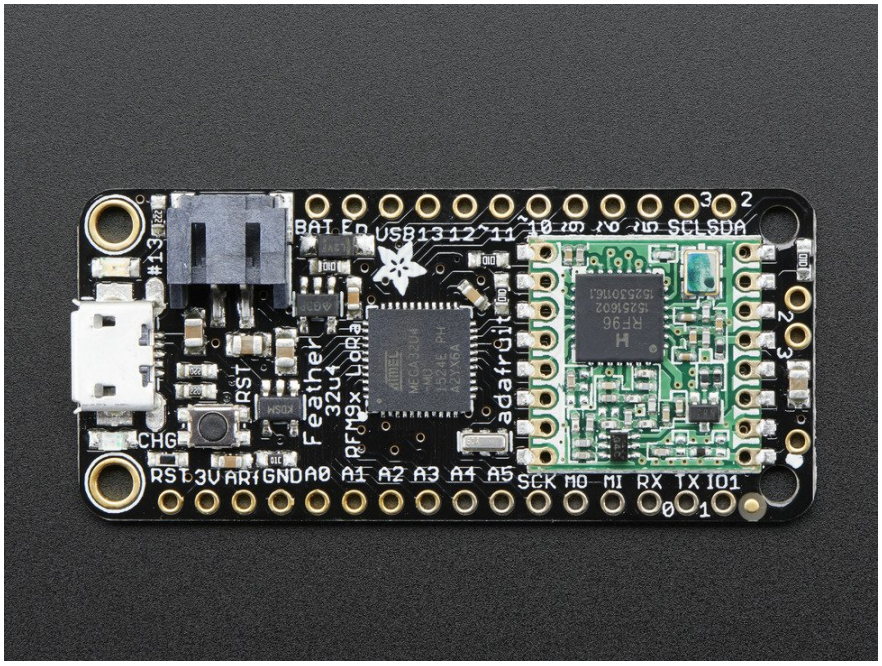


Fig. 12: Feather 32u4 with RFM95 LoRa Radio-868 MHz-RadioFruit from Adafruit. Feather 32u4 LoRa tutorial with explanations, datasheets, and examples

Sensor

We have attached a DHT22 sensor to the microcontroller board, which measures air temperature and humidity. The minimal time interval between two measurements is 2 seconds. All data transfers between the DHT22 and the microcontroller use a single digital line. The sensor data pin is attached to a GPIO pin (here: Digital Pin 5) of the microcontroller. In addition, a so-called pull-up resistor of 4.7k to 10k Ohm must be connected between the data line and VCC (+3.3V). The DHT22 datasheet provides more technical details about the DHT22 Sensor. A tutorial on how to use the DHT22 sensor with Arduino microcontrollers is provided here. The sensor is available in German shops for around 4 € to 10 €.

For more details on the wiring connections, follow this tutorial. Once all these connection are made, the board is connected with a computer using a USB cable. Further, steps of *software part* needs to be followed. But, before that we need to *register a new device on the service* that we are using.

2.4.2 Software

The sensor node has been programmed using the Arduino IDE. Please note, that in the Arduino framework a program is called a 'Sketch'.

After the sketch has successfully established a connection to The Things Network it reports the air temperature, humidity, and the voltage of a (possibly) attached LiPo battery every 5 minutes. All three values are being encoded in

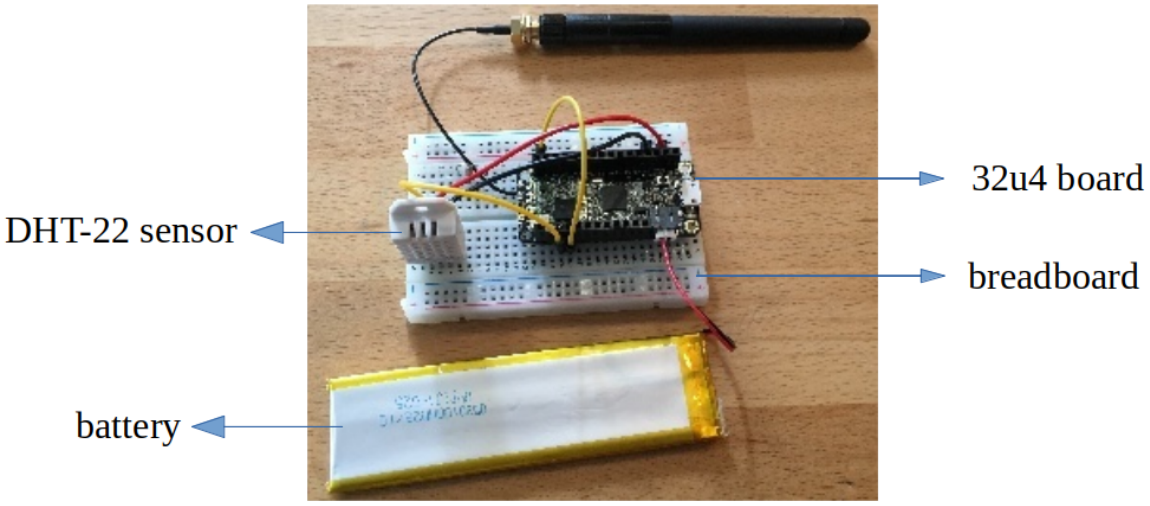


Fig. 13: The Adafruit Feather 32u4 RFM95 LoRa with attached antenna (top), a 1000 mAh lithium polymer (LiPo) battery (bottom), and an attached DHT22 temperature / humidity sensor (white box on the left)

two byte integer values each (in most significant byte order) and then sent as a 6 bytes data packet to the respective TTN application using LoRaWAN port 7. Please note, that LoRaWAN messages can be addressed to ports 1-255 (port 0 is reserved); these ports are similar to port numbers 0-65535 when using the Internet TCP/IP protocol. Voltage and humidity values are always greater or equal to 0, but the temperature value can also become negative. Negative values are represented as a two's complement; this must be considered in the Payload Decoding Function used in The Things Network (*see here*).

In between two sensor readings the microcontroller is going into deep sleep mode to save battery power. With a 1000 mAh LiPo battery and the current version of the sketch the system can run for at least 5 months. (Further optimizations would be possible, for example, not switching on the LED on the microcontroller board during LoRa data transmissions.)

The employed RFM95 LoRa module does not provide built-in support of the LoRaWAN protocol. Thus, it has to be implemented on the ATmega32u4 microcontroller. We use the IBM LMIC (LoraMAC-in-C) library for Arduino. Since the ATmega32u4 microcontroller only has 32 KB of flash memory and the LMIC library is taking most of it, there is only very limited code space left for the application dealing with the sensors (about 2 KB). Nevertheless, this is sufficient to query some sensors like in our example the DHT22.

Now download and run the *Arduino Sketch for Adafruit32u4 LoRa sensor node* file in the Arduino IDE. This code was created by merging the example code of both the sensors and the ttn-otaa example from the lmic library. Some required changes were made while merging the example codes. The user should change the network session key, app session key and device address in the code before compiling. These keys can be obtained from the TTN account as shown in the *services section*.

Listing 14: Modify the keys in highlighted lines.

```
1 // LoRaWAN NwkSKey, network session key
2 // This should be in big-endian (aka msb).
3 static const PROGMEM u1_t NWKSKEY[16] = {NETWORK_SESSION_KEY_HERE_IN_MSB_FORMAT};
4
```

(continues on next page)

(continued from previous page)

```
// LoRaWAN AppSKey, application session key
// This should also be in big-endian (aka msb).
static const u1_t PROGMEM APPSKEY[16] = {APPLICATION_SESSION_KEY_HERE_IN_MSB_FORMAT};

// LoRaWAN end-device address (DevAddr)
// See http://thethingsnetwork.org/wiki/AddressSpace
// The library converts the address to network byte order as needed, so this should be in big-endian (aka msb) too.
static const u4_t DEVADDR = 0x260XXXXX   ; // <-- Change this address for every node!
```

Following is the example code that can be used to measure the battery voltage of the sensor node:

Listing 15: Code for measuring the battery voltage

```
            measuredvbat = analogRead(VBATPIN);
            measuredvbat *= 2;    // we divided by 2, so multiply back
            measuredvbat *= 3.3;   // Multiply by 3.3V, our reference voltage
            measuredvbat /= 1023;  // convert to voltage
            int16_vbat = round(measuredvbat * 100);
            mydata[4] = (byte) (int16_vbat >> 8);
            mydata[5] = (byte) (int16_vbat & 0x00FF);
            SERIALDEBUG_PRINT(" \t");
            SERIALDEBUG_PRINT("Battery Voltage: ");
            SERIALDEBUG_PRINT(measuredvbat);
            SERIALDEBUG_PRINTLN(" V");
```

2.4.3 Services

The services used for this sensor-node are:

- *TheThingsNetwork* service for LoRaWAN network service.
- *TheThingsNetwork - OGC SensorWeb* integration for uploading LoRaWAN sensor data into OGC infrastructure.

Registration of the sensor node with The Things Network (TTN)

The LoRaWAN protocol makes use of a number of different identifiers, addresses, keys, etc. These are required to unambiguously identify devices, applications, as well as to encrypt and decrypt messages. The names and meanings are nicely explained on a dedicated TTN web page.

The sketch given above connects the sensor node with The Things Network (TTN) using the Activation-by-Personalisation (ABP) mode. In this mode, the required keys for data encryption and session management are created manually using the TTN console window and must be pasted into the source code of the sketch below. In order to get this running, you will need to create a new device in the TTN console window. This assumes that you already have a TTN user account (which needs to be created otherwise). In the settings menu of the newly created device the ABP mode must be selected and the settings must be saved. Then copy the DevAddr, the NwkSKey, and the AppSKey from the TTN console web page of the newly registered device and paste them into the proper places in the sketch above. Please make sure that you choose for each of the three keys the correct byte ordering (MSB for all three keys). A detailed explanation of these steps is given here. Then the sketch can be compiled and uploaded to the Adafruit Feather 32u4 LoRa microcontroller.

Important hint: everytime the sensor node is reset or being started again, make sure to reset the frame counter of the registered sensor in the TTN console web page of the registered device. The reason is that in LoRaWAN all transmitted data packets have a frame counter, which is incremented after each data frame being sent. This way a LoRaWAN application can avoid receiving and using the same packet again (replay attack). When TTN receives a

data packet, it checks if the frame number is higher than the last one received before. If not, the received packet is considered to be old or a replay attack and is discarded. When the sensor node is reset or being started again, its frame counter is also reset to 0, hence, the TTN application assumes that all new packages are old, because their frame counter is lower than the last frame received (before the reset). A manual frame counter reset is only necessary when registering the node using ABP mode. In OTAA mode the frame counter is automatically reset in the sensor node and the TTN network server.

TTN Payload Decoding

Everytime a data packet is received by a TTN application a dedicated Javascript function is being called (Payload Decoder Function). This function can be used to decode the received byte string and to create proper Javascript objects or values that can directly be read by humans when looking at the incoming data packet. This is also useful to format the data in a specific way that can then be forwarded to an external application (e.g. a sensor data platform like MyDevices or Thingspeak). Such a forwarding can be configured in the TTN console in the "Integrations" tab. *TTN payload decoder for Adafruit32u4 LoRa sensor node* given here checks if a packet was received on LoRaWAN port 7 and then assumes that it consists of the 6 bytes encoded as described above. It creates the three Javascript objects 'temperature', 'humidity', and 'vbattery'. Each object has two fields: 'value' holds the value and 'uom' gives the unit of measure. The source code can simply be copied and pasted into the 'decoder' tab in the TTN console after having selected the application. Choose the option 'Custom' in the 'Payload Format' field. Note that when you also want to handle other sensor nodes sending packets on different LoRaWAN ports, then the Payload Decoder Function can be extended after the end of the if (port==7) {...} statement by adding else if (port==8) {...} else if (port==9) {...} etc.

The Things Network - OGC SensorWeb Integration

The presented Payload Decoder Function works also with the TTN-OGC SWE Integration for the 52° North Sensor Observation Service (SOS). This software component can be downloaded from this repository. It connects a TTN application with a running transactional Sensor Observation Service 2.0.0 (SOS). Data packets received from TTN are imported into the SOS. The SOS persistently stores sensor data from an arbitrary number of sensor nodes and can be queried for the most recent as well as for historic sensor data readings. The 52° North SOS comes with its own REST API and a nice web client allowing to browse the stored sensor data in a convenient way.

We are running an instance of the 52° North SOS and the TTN-OGC SWE Integration. The web client for this LoRaWAN sensor node can be accessed on this page. Here is a screenshot showing the webclient:

Sending a message to the Sensor Node (Downlink)

Using the TTN console we can send a message (i.e. a byte string) to the sensor node. In the TTN console application page click on the respective application. Then click on the 'Devices' tab and choose the proper sensor node (here: adafruit-feather-32u4-lora3). On the overview page scroll down to the 'Downlink' section. In the 'Payload' field enter 1 to 4 bytes. In order to show digits or letters on the LED display these must be ASCII encoded and have to be entered as hexadecimal numbers. When you click on the 'Send' button the message will be queued and the next time when the node sends its data packet (uplink) it will receive the message. The first 4 bytes will be shown on the display and the beeper indicates the reception of a new downlink message. In order to blank the display just send a one byte message with the value '20' (hexadecimal for 32, which is the ASCII code for a space). When the node receives just a single blank character it will not produce a beeping sound. There is a nice web page offering online encoding of text to ASCII numbers in hexadecimal encoding. For example, in order to display the text 'LoRa', the four hexadecimal numbers 4C 6F 52 61 have to be entered in the Payload entry field.

2.4.4 Code files

Fig. 14: Web client for data visualization

Listing 16: Arduino Sketch for Adafruit32u4 LoRa sensor node

```
/*******************************************************************************
 * Arduino Sketch for a LoRaWAN sensor node that is registered with
 * 'The Things Network' (TTN) www.thethingsnetwork.org
 *
 * Author:  Thomas H. Kolbe, thomas.kolbe@tum.de
 * Version: 1.0
 * Last update: 2018-05-21
 *
 * The sensor node is based on the Adafruit Feather 32u4 LoRa microcontroller
 * board https://learn.adafruit.com/adafruit-feather-32u4-radio-with-lora-radio-
↪module/
 * The sensor node uses a DHT22 sensor measuring air temperature and humidity.
 * Also the voltage of an attached LiPo battery is monitored and sent as
 * an observation. All three values are encoded as 2 byte integer values each.
 * Hence, the total message payload is 6 bytes. Before the values are converted
 * to integers they are multiplied by 100 to preserve 2 digits after the decimal
 * point. Thus, the received values must be divided by 100 to obtain the measured
 * values. The payload is sent every 300s to LoRaWAN port 7. The following
 * Javascript function can be used as a payload decoding function in TTN:
 *
 * function Decoder(bytes, port) {
 *   // Decode an uplink message from a buffer
 *   // (array) of bytes to an object of fields.
 *   if (port==7) {
 *     var decoded = {
 *       "temperature": (bytes[0] << 8 | bytes[1]) / 100.0,
```

(continues on next page)

(continued from previous page)

```
 *       "humidity": (bytes[2] << 8 | bytes[3]) / 100.0,
 *       "vbattery": (bytes[4] << 8 | bytes[5]) / 100.0
 *     };
 *   } else {
 *     var decoded = null;
 *   }
 *   return decoded;
 * }
 *
 * In between two data transmissions the microcontroller board can go
 * into sleep mode to reduce energy consumption for extended operation
 * time when running on battery. Usage of the sleep mode must be
 * explicitly configured below.
 *
 * Important hint: everytime the sensor node is reset or being started again,
 * make sure to reset the frame counter of the registered sensor in the
 * TTN console at https://console.thethingsnetwork.org. The reason is that
 * in LoRaWAN all packets that are transmitted have a frame counter, which
 * is incremented after each data frame being sent. This way a LoRaWAN application
 * can avoid receiving and using the same packet again (replay attack). When
 * TTN receives a data packet, it checks if the frame number is higher than
 * the last one received before. If not, the received packet is considered
 * to be old or a replay attack and is discarded. When the sensor node is
 * reset or being started again, its frame counter is also reset to 0, hence,
 * the TTN application assumes that all new packages are old, because their
 * frame counter is lower than the last frame received (before the reset).
 *
 * Note, that the DHT22 data pin must be connected to Digital Pin 5 of the
 * microcontroller board. A resistor of 4.7k - 10k Ohm must be connected to
 * the data pin and VCC (+3.3V). Digital Pin 6 must be connected to IO1 of the
 * LoRa transceiver module using a simple wire.
 *
 * The code is based on the Open Source library LMIC implementing the LoRaWAN
 * protocol stack on top of a given LoRa transceiver module (here: RFM95 from
 * HopeRF, which uses the Semtech SX1276 LoRa chip). The library is originally
 * being developed by IBM and has been ported to the Arduino platform. See
 * notes below from the original developers.
 *
 *******************************************************************************
 * Copyright (c) 2015 Thomas Telkamp and Matthijs Kooijman
 *
 * Permission is hereby granted, free of charge, to anyone
 * obtaining a copy of this document and accompanying files,
 * to do whatever they want with them without any restriction,
 * including, but not limited to, copying, modification and redistribution.
 * NO WARRANTY OF ANY KIND IS PROVIDED.
 *
 * This uses ABP (Activation-by-personalisation), where a DevAddr and
 * Session keys are preconfigured (unlike OTAA, where a DevEUI and
 * application key is configured, while the DevAddr and session keys are
 * assigned/generated in the over-the-air-activation procedure).
 *
 * Note: LoRaWAN per sub-band duty-cycle limitation is enforced (1% in
 * g1, 0.1% in g2), but not the TTN fair usage policy (which is probably
 * violated by this sketch when left running for longer)!
 *
 * To use this sketch, first register your application and device with
```

(continues on next page)

(continued from previous page)

```
 * the things network, to set or generate a DevAddr, NwkSKey and
 * AppSKey. Each device should have their own unique values for these
 * fields.
 *
 * Do not forget to define the radio type correctly in config.h.
 *
 *******************************************************************************/

// If the following line is uncommented, messages are being printed out to the
// serial connection for debugging purposes. When using the Arduino Integrated
// Development Environment (Arduino IDE), these messages are displayed in the
// Serial Monitor selecting the proper port and a baudrate of 115200.

//#define SERIALDEBUG

#ifdef SERIALDEBUG
  #define SERIALDEBUG_PRINT(...) Serial.print(__VA_ARGS__)
  #define SERIALDEBUG_PRINTLN(...) Serial.println(__VA_ARGS__)
#else
  #define SERIALDEBUG_PRINT(...)
  #define SERIALDEBUG_PRINTLN(...)
#endif

// If the following line is uncommented, the sensor node goes into sleep mode
// in between two data transmissions. Also the 2secs time between the
// initialization of the DHT22 sensor and the reading of the observations
// is spent in sleep mode.
// Note, that on the Adafruit Feather 32u4 LoRa board the Serial connection
// gets lost as soon as the board goes into sleep mode, and it will not be
// established again. Thus, the definition of SERIALDEBUG should be commented
// out above when using sleep mode.

#define SLEEPMODE

#ifdef SLEEPMODE
#include <Adafruit_SleepyDog.h>
#endif

#include <lmic.h>
#include <hal/hal.h>
#include <SPI.h>

#include <DHT.h>

#define DHTPIN            5       // Arduino Digital Pin which is connected to the␣
↪DHT sensor.
#define DHTTYPE           DHT22   // DHT 22 (AM2302)

DHT dht(DHTPIN, DHTTYPE);         // create the sensor object

#define VBATPIN A9                // battery voltage is measured from Analog Input A9

// The following three constants (NwkSKey, AppSKey, DevAddr) must be changed
// for every new sensor node. We are using the LoRaWAN ABP mode (activation by
// personalisation) which means that each sensor node must be manually registered
// in the TTN console at https://console.thethingsnetwork.org before it can be
// started. In the TTN console create a new device and choose ABP mode in the
```

(continues on next page)

(continued from previous page)

```
// settings of the newly created device. Then, let TTN generate the NwkSKey and
// and the AppSKey and copy them (together with the device address) from the webpage
// and paste them below.

// LoRaWAN NwkSKey, network session key
// This should be in big-endian (aka msb).
static const PROGMEM u1_t NWKSKEY[16] = {NETWORK_SESSION_KEY_HERE_IN_MSB_FORMAT};

// LoRaWAN AppSKey, application session key
// This should also be in big-endian (aka msb).
static const u1_t PROGMEM APPSKEY[16] = {APPLICATION_SESSION_KEY_HERE_IN_MSB_FORMAT};

// LoRaWAN end-device address (DevAddr)
// See http://thethingsnetwork.org/wiki/AddressSpace
// The library converts the address to network byte order as needed, so this should be in big-endian (aka msb) too.
static const u4_t DEVADDR = 0x260XXXXX   ; // <-- Change this address for every node!

// These callbacks are only used in over-the-air activation, so they are
// left empty here (we cannot leave them out completely unless
// DISABLE_JOIN is set in config.h, otherwise the linker will complain).
void os_getArtEui (u1_t* buf) { }
void os_getDevEui (u1_t* buf) { }
void os_getDevKey (u1_t* buf) { }

// The following array of bytes is a placeholder to contain the message payload
// which is transmitted to the LoRaWAN gateway. We are currently only using 6 bytes.
// Please make sure to extend the size of the array, if more sensors should be
// attached to the sensor node and the message payload becomes larger than 10 bytes.
static uint8_t mydata[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0xA};

static osjob_t sendjob;

// Schedule transmission every TX_INTERVAL seconds (might become longer due to duty
// cycle limitations). The total interval time is 2secs for the measurement
// plus 3secs for the LoRaWAN packet transmission plus TX_INTERVAL_AFTER_SLEEP seconds
// plus SLEEP_TIME seconds (microcontroller in sleep mode)
const unsigned TX_INTERVAL = 300;       // overall cycle time (send one set of observations every 5mins)
const unsigned TX_WAIT_AFTER_SLEEP = 1; // seconds to wait after return from sleep mode before the next transmit is scheduled
const unsigned TX_TIME = 3;             // rough estimate of transmission time of a single packet
const unsigned MEASURE_TIME = 2;        // seconds measuring time
const unsigned SLEEP_TIME = TX_INTERVAL - TX_WAIT_AFTER_SLEEP - TX_TIME - MEASURE_TIME;
const unsigned WAIT_TIME = TX_INTERVAL - TX_TIME - MEASURE_TIME;

// Pin mapping
const lmic_pinmap lmic_pins = {
    .nss = 8,
    .rxtx = LMIC_UNUSED_PIN,
    .rst = 4,
    .dio = {7, 6, LMIC_UNUSED_PIN},
};

void onEvent (ev_t ev) {
```

(continues on next page)

(continued from previous page)

```
    SERIALDEBUG_PRINT(os_getTime());
    SERIALDEBUG_PRINT(": ");
    switch(ev) {
        case EV_SCAN_TIMEOUT:
            SERIALDEBUG_PRINTLN(F("EV_SCAN_TIMEOUT"));
            break;
        case EV_BEACON_FOUND:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_FOUND"));
            break;
        case EV_BEACON_MISSED:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_MISSED"));
            break;
        case EV_BEACON_TRACKED:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_TRACKED"));
            break;
        case EV_JOINING:
            SERIALDEBUG_PRINTLN(F("EV_JOINING"));
            break;
        case EV_JOINED:
            SERIALDEBUG_PRINTLN(F("EV_JOINED"));
            break;
        case EV_RFU1:
            SERIALDEBUG_PRINTLN(F("EV_RFU1"));
            break;
        case EV_JOIN_FAILED:
            SERIALDEBUG_PRINTLN(F("EV_JOIN_FAILED"));
            break;
        case EV_REJOIN_FAILED:
            SERIALDEBUG_PRINTLN(F("EV_REJOIN_FAILED"));
            break;
        case EV_TXCOMPLETE:
            digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the␣
↪voltage LOW
            SERIALDEBUG_PRINTLN(F("EV_TXCOMPLETE (includes waiting for RX windows)"));
            if (LMIC.txrxFlags & TXRX_ACK)
              SERIALDEBUG_PRINTLN(F("Received ack"));
            if (LMIC.dataLen) {
#ifdef SERIALDEBUG
              SERIALDEBUG_PRINT(F("Received "));
              SERIALDEBUG_PRINT(LMIC.dataLen);
              SERIALDEBUG_PRINT(F(" bytes of payload: 0x"));
              for (int i=0; i<LMIC.dataLen; i++) {
                if (LMIC.frame[LMIC.dataBeg + i] < 0x10) {
                  SERIALDEBUG_PRINT(F("0"));
                }
                SERIALDEBUG_PRINT(LMIC.frame[LMIC.dataBeg + i], HEX);
              }
              SERIALDEBUG_PRINTLN();
#endif
              // add your code to handle a received downlink data packet here
            }

#ifdef SLEEPMODE
            // Schedule next transmission in 1 second after the board returns from␣
↪sleep mode
            os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(TX_WAIT_AFTER_
↪SLEEP), do_send);
```

(continues on next page)

(continued from previous page)

```
245
246             SERIALDEBUG_PRINTLN("going to sleep now ... ");
247             // lmic library sleeps automatically after transmission has been completed
248             for(int i= 0; i < SLEEP_TIME / 8; i++) {
249               Watchdog.sleep(8000); // maximum seems to be 8 seconds
250               SERIALDEBUG_PRINT('.');
251             }
252             if (SLEEP_TIME % 8) {
253               Watchdog.sleep((SLEEP_TIME % 8)*1000);
254               SERIALDEBUG_PRINT('*');
255             }
256             SERIALDEBUG_PRINTLN("... woke up again");
257 #else
258             // Schedule next transmission
259             os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(WAIT_TIME), do_
    ↪send);
260 #endif
261             break;
262         case EV_LOST_TSYNC:
263             SERIALDEBUG_PRINTLN(F("EV_LOST_TSYNC"));
264             break;
265         case EV_RESET:
266             SERIALDEBUG_PRINTLN(F("EV_RESET"));
267             break;
268         case EV_RXCOMPLETE:
269             // data received in ping slot
270             SERIALDEBUG_PRINTLN(F("EV_RXCOMPLETE"));
271             break;
272         case EV_LINK_DEAD:
273             SERIALDEBUG_PRINTLN(F("EV_LINK_DEAD"));
274             break;
275         case EV_LINK_ALIVE:
276             SERIALDEBUG_PRINTLN(F("EV_LINK_ALIVE"));
277             break;
278          default:
279             SERIALDEBUG_PRINTLN(F("Unknown event"));
280             break;
281     }
282 }
283
284 void do_send(osjob_t* j){
285     // Check if there is not a current TX/RX job running
286     if (LMIC.opmode & OP_TXRXPEND) {
287         SERIALDEBUG_PRINTLN(F("OP_TXRXPEND, not sending"));
288     } else {
289         // Prepare upstream data transmission at the next possible time.
290
291         float temperature, humidity, measuredvbat;
292         int16_t int16_temperature, int16_humidity, int16_vbat;
293
294         // Start a measurement to update the sensor's internal temperature & humidity␣
    ↪reading.
295         // Note, that when fetching measurements from a DHT22 sensor, the reported
296         // values belong to the measurement BEFORE the current measurement.
297         // Therefore, in order to get current observations, we first perform a new␣
    ↪measurement
298         // and wait 2 secs (which is the minimum time between two sensor observations␣
    ↪for
```

(continues on next page)

(continued from previous page)

```
        // the DHT22) and then directly retrieve the observations again.
        temperature = dht.readTemperature();
#ifdef SLEEPMODE
        Watchdog.sleep(2000);
#else
        delay(2000);
#endif
        // Now read the recently measured temperature (2 secs ago) as Celsius (the default)
        temperature = dht.readTemperature();
        // Read the recently measured humidity (2 secs ago)
        humidity = dht.readHumidity();

        // Check if any reads failed and exit early (to try again).
        if (isnan(humidity) || isnan(temperature)) {
            SERIALDEBUG_PRINTLN("Failed to read from DHT sensor!");
            // blink the LED five times to indicate that the sensor values could not be read
            for (int i=0; i<5; i++) {
              digitalWrite(LED_BUILTIN, HIGH);    // turn the LED on by making the voltage HIGH
              delay(150);
              digitalWrite(LED_BUILTIN, LOW);    // turn the LED on by making the voltage HIGH
              delay(150);
            }
            // ok, then wait for another period and try it again
            os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(TX_INTERVAL), do_send);
        } else {
            SERIALDEBUG_PRINT("Humidity: ");
            SERIALDEBUG_PRINT(humidity);
            SERIALDEBUG_PRINT(" %\t");
            SERIALDEBUG_PRINT("Temperature: ");
            SERIALDEBUG_PRINT(temperature);
            SERIALDEBUG_PRINT(" °C ");

            int16_temperature = 100*temperature;
            int16_humidity = 100*humidity;
            mydata[0] = (byte) (int16_temperature >> 8);
            mydata[1] = (byte) (int16_temperature & 0x00FF);
            mydata[2] = (byte) (int16_humidity >> 8);
            mydata[3] = (byte) (int16_humidity & 0x00FF);

            measuredvbat = analogRead(VBATPIN);
            measuredvbat *= 2;    // we divided by 2, so multiply back
            measuredvbat *= 3.3;   // Multiply by 3.3V, our reference voltage
            measuredvbat /= 1023;  // convert to voltage
            int16_vbat = round(measuredvbat * 100);
            mydata[4] = (byte) (int16_vbat >> 8);
            mydata[5] = (byte) (int16_vbat & 0x00FF);
            SERIALDEBUG_PRINT(" \t");
            SERIALDEBUG_PRINT("Battery Voltage: ");
            SERIALDEBUG_PRINT(measuredvbat);
            SERIALDEBUG_PRINTLN(" V");

            // Send the 6 bytes payload to LoRaWAN port 7 and do not request an acknowledgement.
```

(continues on next page)

(continued from previous page)

```
            // The following call does not directly sends the data, but puts a "send job"
            // in the job queue. This job eventually is performed in the call "os_runloop_once();"
            // issued repeatedly in the "loop()" method below. After the transmission is
            // complete, the EV_TXCOMPLETE event is signaled, which is handled in the
            // event handler method "onEvent (ev_t ev)" above. In the EV_TXCOMPLETE branch
            // then a new call to the "do_send(osjob_t* j)" method is being prepared for
            // delayed execution with a waiting time of TX_INTERVAL seconds.
            LMIC_setTxData2(7, mydata, 6, 0);
            SERIALDEBUG_PRINTLN(F("Packet queued"));
            digitalWrite(LED_BUILTIN, HIGH);    // turn the LED on by making the voltage HIGH

            // Next TX is scheduled after TX_COMPLETE event.
        }
    }
}

void setup() {
    delay(5000);                    // give enough time to open serial monitor (if needed)

    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW

#ifdef SERIALDEBUG
    Serial.begin(115200);
    // while (!Serial);
#endif

    dht.begin();                    // initialize DHT22 sensor

    SERIALDEBUG_PRINTLN(F("Starting"));

    #ifdef VCC_ENABLE
    // For Pinoccio Scout boards
    pinMode(VCC_ENABLE, OUTPUT);
    digitalWrite(VCC_ENABLE, HIGH);
    delay(1000);
    #endif

    // LMIC init
    os_init();
    // Reset the MAC state. Session and pending data transfers will be discarded.
    LMIC_reset();
    LMIC_setClockError(MAX_CLOCK_ERROR * 1 / 100);

    // Set static session parameters. Instead of dynamically establishing a session
    // by joining the network, precomputed session parameters are be provided.
    #ifdef PROGMEM
    // On AVR, these values are stored in flash and only copied to RAM
    // once. Copy them to a temporary buffer here, LMIC_setSession will
    // copy them into a buffer of its own again.
```

(continues on next page)

(continued from previous page)

```
    uint8_t appskey[sizeof(APPSKEY)];
    uint8_t nwkskey[sizeof(NWKSKEY)];
    memcpy_P(appskey, APPSKEY, sizeof(APPSKEY));
    memcpy_P(nwkskey, NWKSKEY, sizeof(NWKSKEY));
    LMIC_setSession (0x1, DEVADDR, nwkskey, appskey);
    #else
    // If not running an AVR with PROGMEM, just use the arrays directly
    LMIC_setSession (0x1, DEVADDR, NWKSKEY, APPSKEY);
    #endif

    #if defined(CFG_eu868)
    // Set up the channels used by the Things Network, which corresponds
    // to the defaults of most gateways. Without this, only three base
    // channels from the LoRaWAN specification are used, which certainly
    // works, so it is good for debugging, but can overload those
    // frequencies, so be sure to configure the full frequency range of
    // your network here (unless your network autoconfigures them).
    // Setting up channels should happen after LMIC_setSession, as that
    // configures the minimal channel set.
    // NA-US channels 0-71 are configured automatically
    LMIC_setupChannel(0, 868100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(1, 868300000, DR_RANGE_MAP(DR_SF12, DR_SF7B), BAND_CENTI);
↪// g-band
    LMIC_setupChannel(2, 868500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(3, 867100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(4, 867300000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(5, 867500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(6, 867700000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(7, 867900000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(8, 868800000, DR_RANGE_MAP(DR_FSK,  DR_FSK),  BAND_MILLI);
↪// g2-band
    // TTN defines an additional channel at 869.525Mhz using SF9 for class B
    // devices' ping slots. LMIC does not have an easy way to define set this
    // frequency and support for class B is spotty and untested, so this
    // frequency is not configured here.
    #elif defined(CFG_us915)
    // NA-US channels 0-71 are configured automatically
    // but only one group of 8 should (a subband) should be active
    // TTN recommends the second sub band, 1 in a zero based count.
    // https://github.com/TheThingsNetwork/gateway-conf/blob/master/US-global_conf.
↪json
    LMIC_selectSubBand(1);
    #endif

    // Disable link check validation
    LMIC_setLinkCheckMode(0);

    // TTN uses SF9 for its RX2 window.
    LMIC.dn2Dr = DR_SF9;

```

(continues on next page)

(continued from previous page)

```
    // Set data rate and transmit power for uplink (note: txpow seems to be ignored by the library)
    LMIC_setDrTxpow(DR_SF7,14);

    // Start job. This will initiate the repetitive sending of data packets,
    // because after each data transmission, a delayed call to "do_send()"
    // is being scheduled again.
    do_send(&sendjob);
}

void loop() {
    os_runloop_once();
}
```

Listing 17: TTN payload decoder for Adafruit32u4 LoRa sensor node

```
function Decoder (bytes, port) {
  var result = {};
  var transformers = {};

  if (port==7) {
     transformers = {
      'temperature': function transform (bytes) {
          value=bytes[0]*256 + bytes[1];
          if (value>=32768) value=value-65536;
          return value/100.0;
        },
      'humidity': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/100.0;
        },
      'vbattery': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/100.0;
        },
    }

    result['temperature'] = {
      value: transformers['temperature'](bytes.slice(0, 2)),
      uom: 'Celsius',
    }

    result['humidity'] = {
      value: transformers['humidity'](bytes.slice(2, 4)),
      uom: 'Percent',
    }

    result['vbattery'] = {
      value: transformers['vbattery'](bytes.slice(4, 6)),
      uom: 'Volt',
    }
  }

  return result;
}
```

2.4.5 References

- Adafruit Feather 32u4 LoRa microntroller
- Adafruit Feather 32u4 LoRa tutorial
- IBM LMIC (LoraMAC-in-C) library for Arduino
- Using Adafruit Feather 32u4 RFM95 as an TTN Node - Stories - Labs
- TTN LoraWan Atmega32U4 based node – ABP version | Primal Cortex's Weblog
- node-workshop/lora32u4.md at master · kersing/node-workshop · GitHub
- Got Adafruit Feather 32u4 LoRa Radio to work and here is how - End Devices (Nodes) - The Things Network
- Adafruit Feather as LoRaWAN node | Wolfgang Klenk
- LMiC on Adafruit Lora Feather successfully sends message to TTN and then halts with "Packet queued" - End Devices (Nodes) - The Things Network
- GitHub - marcuscbehrens/loralife
- GPS-Tracker - Stories - Labs

On battery saving / using the deep sleep mode

- Adafruit Feather 32u4 LoRa - long transmission time after deep sleep - End Devices (Nodes) - The Things Network and this
- Full Arduino Mini LoraWAN and 1.3uA Sleep Mode - End Devices (Nodes) - The Things Network
- Adding Method to Adjust hal_ticks Upon Waking Up from Sleep · Issue #109 · matthijskooijman/arduino-lmic
- minilora-test/minilora-test.ino at cbe686826bd84fac8381de47b5f5b02dd47c2ca0 · tkerby/minilora-test
- Arduino-LMIC library with low power mode - Mario Zwiers

2.5 Adafruit 32u4 LoRa with Display

This tutorial is made to showcase the use of Adafruit 32u4 board to create a LoRaWAN enabled sensor node with a display and a case. In the following example, a temperature and humidity sensor was used with the Adafruit 32u4 board to create this tutorial.

2.5.1 Hardware

To build this sensor node we have used following hardware components:

- Adafruit Feather 32u4 LoRa module
- Grove - DHT-22 Temperature & Humidity Sensor
- LED Display
- Breadboard
- Battery
- Resistor: 4.7k to 10k Ohm
- 3d-Printed case

Microcontroller

The Adafruit Feather 32u4 LoRa module is operated by the 8bit ATmega32u4 microcontroller running at 8MHz. It has 32 KB flash memory (to store the program code) and 2 KB of RAM (to store variables, status information, and buffers). The operating voltage of the board is 3.3V (this is important when attaching sensors and other peripherals; they also must operate on 3.3V). The board offers 20 general purpose digital input/output pins (20 GPIOs) with 10 analog input pins (with 12bit analog digital converters (ADC)), one serial port (programmable Universal Asynchronous Receiver and Transmitter, UART), one I2C port, one SPI port, one USB port. The board comes with an embedded Lithium polymer battery management chip and status indicator led, which allows to directly connect a 3.7V LiPo rechargeable battery that will be automatically recharged when the board is powered over its USB connector. The Adafruit Feather 32u4 LoRa board is available in German shops from around 37 € to 45 €.

The LoRa transmitter and receiver is encapsulated within an RFM95 module from the company HopeRF. This module uses the LoRa chip SX1276 from the company Semtech and is dedicated to the 868 MHz frequency band. The RFM95 module is connected via SPI interface to the microcontroller. Most of the required connections of the LoRa transceiver pins with the microcontroller are already built-in on the Adafruit Feather 32u4 LoRa board. However, Digital Pin 6 of the microcontroller must be connected to DIO1 of the LoRa transceiver module in addition using a simple wire. Since the module only implements the LoRa physical layer, the LoRaWAN protocol stack must be implemented in software on the microcontroller. We are using the Arduino library LMIC for that purpose (see below). The implemented LoRaWAN functionality is compatible with LoRaWAN Class A/C.

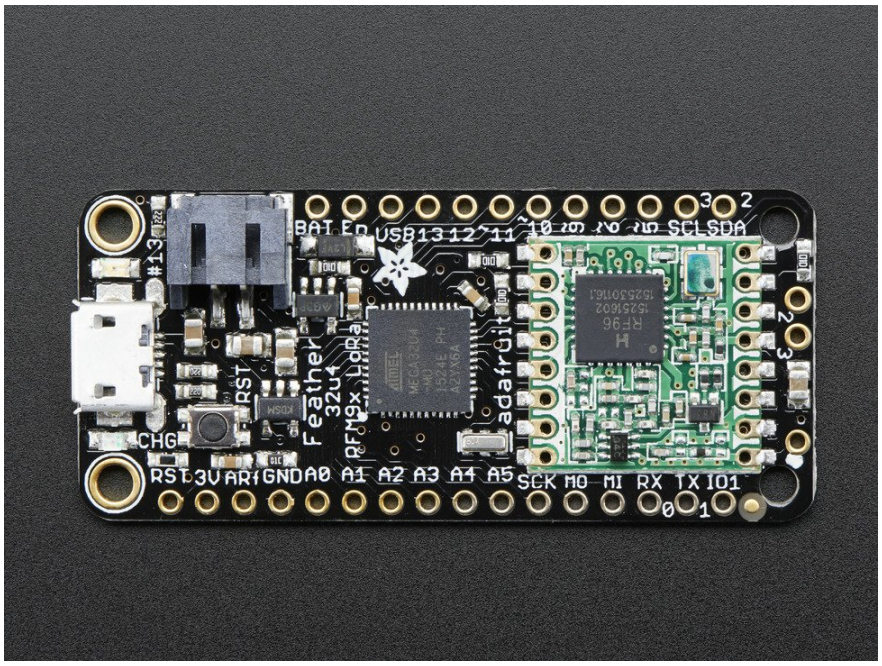


Fig. 15: Feather 32u4 with RFM95 LoRa Radio-868 MHz-RadioFruit from Adafruit. Feather 32u4 LoRa tutorial with explanations, datasheets, and examples.

Sensor

We have attached a DHT22 sensor to the microcontroller board, which measures air temperature and humidity. The minimal time interval between two measurements is 2 seconds. All data transfers between the DHT22 and the microcontroller use a single digital line. The sensor data pin is attached to a GPIO pin (here: Digital Pin 6) of the microcontroller. In addition, a so-called pull-up resistor of 4.7k to 10k Ohm must be connected between the data line and VCC (+3.3V). The DHT22 datasheet provides more technical details about the DHT22 Sensor. A tutorial on how

to use the DHT22 sensor with Arduino microcontrollers is provided here. The sensor is available in German shops for around 4 € to 10 €.

Display / Beeper

On top of the microcontroller board we have attached an Adafruit Display Wing with a 4 digit 14 segments LED display. It can show 0-4 numbers or letters (upper case and lower case). The display controller is using the I2C protocol and the I2C pins SDA and SCL are directly connected to the Adafruit Feather via the Wing connectors. The Wing is using the default I2C address (0x70). Also a 3.3V beeper is installed that is used to indicate that a new message was received and is now being displayed. The '+' pin of the beeper has to be connected to Digital Pin 12 and the '-' pin to GND. The display and the beeper can be used to notify a user with (very) short messages. The reason why we have included this is mostly to experiment with and to demonstrate the downlink capabilities of LoRaWAN. When a downlink message has been queued it will be transmitted to the node right after it has transmitted the next data packet (uplink data). Hence, it depends on the transmission time period how long it can take unless the node receives and displays a downlink message.

Case

The case was 3D printed using the design files provided by Adafruit. The case consists of three parts. Part 1 is the main enclosure (it does not have a switch holder or tabs, the design file is feather-case.stl). Part 2 is the battery holder (with a slide switch holder, the design file is feather-bat-switch.stl). Part 3 is the case topper (with a cutout for the Adafruit Feather Wing, the design file is feather-top-wing.stl). All design files can be downloaded from Thingiverse.

We have ordered the three parts from an online 3D printing service. The quality of the delivered parts was generally ok, but not good enough for snapping the three parts firmly together. It is not clear yet whether this is a problem of the design files or of the printing service. We used two rubber bands In order to fix the three parts together.

Once all these connection are made, the board is connected with a computer using a USB cable. Further, steps of *software part* needs to be followed. But, before that we need to *register a new device on the service* that we are using.

2.5.2 Software

The sensor node has been programmed using the Arduino IDE. Please note, that in the Arduino framework a program is called a 'Sketch'.

After the sketch has successfully established a connection to The Things Network it reports the air temperature, humidity, and the voltage of a (possibly) attached LiPo battery every 5 minutes. All three values are being encoded in two byte integer values each (in most significant byte order) and then sent as a 6 bytes data packet to the respective TTN application using LoRaWAN port 7. Please note, that LoRaWAN messages can be addressed to ports 1-255 (port 0 is reserved); these ports are similar to port numbers 0-65535 when using the Internet TCP/IP protocol. Voltage and humidity values are always greater or equal to 0, but the temperature value can also become negative. Negative values are represented as a two's complement; this must be considered in the Payload Decoding Function used in The Things Network (*see here*).

In between two sensor readings the microcontroller is going into deep sleep mode to save battery power. We still have to run some tests to find out for how long the system can run using the 2000 mAh LiPo battery and the current version of the sketch. Showing a received message on the display draws a considerable amount of power and will shorten battery life significantly. Hence, when running on battery it is recommended to clear a displayed message soon by sending a simple space character (0x20). (Further optimizations would be possible, for example, not switching on the LED on the microcontroller board during LoRa data transmissions.)

The employed RFM95 LoRa module does not provide built-in support of the LoRaWAN protocol. Thus, it has to be implemented on the ATmega32u4 microcontroller. We use the IBM LMIC (LoraMAC-in-C) library for Arduino. Since the ATmega32u4 microcontroller only has 32 KB of flash memory and the LMIC library is taking most of it,


LoRa

there is only very limited code space left for the application dealing with the sensors (about 2 KB). Nevertheless, this is sufficient to query some sensors like in our example the DHT22.

Now download and run the *Arduino Sketch for Adafruit32u4 LoRa with display sensor node* file in the Arduino IDE. This code was created by merging the example code of both the sensors and the ttn-otaa example from the lmic library. Some required changes were made while merging the example codes. The user should change the network session key, app session key and device address in the code before compiling. These keys can be obtained from the TTN account as shown in the *services section*.

Listing 18: Modify the keys in highlighted lines.

```
// This EUI must be in little-endian format, so least-significant-byte
// first. When copying an EUI from ttnctl output, this means to reverse
// the bytes. For TTN issued EUIs the last bytes should be 0xD5, 0xB3, 0x70.
static const u1_t PROGMEM APPEUI[8]={ 0x55, 0xC1, 0x00, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX }
↪;
void os_getArtEui (u1_t* buf) { memcpy_P(buf, APPEUI, 8);}

// This should also be in little endian format, see above.
static const u1_t PROGMEM DEVEUI[8]={ 0xF6, 0xE2, 0x10, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX }
↪;
void os_getDevEui (u1_t* buf) { memcpy_P(buf, DEVEUI, 8);}

// This key should be in big endian format (or, since it is not really a
// number but a block of memory, endianness does not really apply). In
// practice, a key taken from ttnctl can be copied as-is.
// The key shown here is the semtech default key.
static const u1_t PROGMEM APPKEY[16] = { 0xC2, 0x21, 0x2E, 0x7A, 0xXX, 0xXX, 0xXX,␣
↪0xXX, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX };
void os_getDevKey (u1_t* buf) {  memcpy_P(buf, APPKEY, 16);}
```

Following is the example code that can be used to measure the battery voltage of the sensor node:

Listing 19: Code for measuring the battery voltage

```
            measuredvbat = analogRead(VBATPIN);
            measuredvbat *= 2.0;      // we divided by 2, so multiply back
            measuredvbat *= 3.3;      // Multiply by 3.3V, our reference voltage
            measuredvbat /= 1023.0;   // convert to voltage
            int16_vbat = round(measuredvbat * 100.0);
            mydata[4] = (byte) (int16_vbat >> 8);
            mydata[5] = (byte) (int16_vbat & 0x00FF);
            SERIALDEBUG_PRINT(" \t");
            SERIALDEBUG_PRINT("Battery Voltage: ");
            SERIALDEBUG_PRINT(measuredvbat);
            SERIALDEBUG_PRINTLN(" V");
```

2.5.3 Services

The services used for this sensor-node are:

- *TheThingsNetwork* service for LoRaWAN network service.
- *TheThingsNetwork - OGC SensorWeb* integration for uploading LoRaWAN sensor data into OGC infrastructure.

Registration of the sensor node with The Things Network (TTN)

The LoRaWAN protocol makes use of a number of different identifiers, addresses, keys, etc. These are required to unambiguously identify devices, applications, as well as to encrypt and decrypt messages. The names and meanings are nicely explained on a dedicated TTN web page.

The sketch given above connects the sensor node with The Things Network (TTN) using the Over-the-Air-Activation (OTAA) mode. In this mode, we use the three keys AppEUI, DevEUI, AppKey. The DevEUI should be delivered with the sensor node by the manufacturer, the other two keys are created using the TTN console. Each sensor node must be manually registered in the TTN console before it can be started. This assumes that you already have a TTN user account (which needs to be created otherwise). In the TTN console create a new device and enter the DevEUI number that was shipped with the Adafruit Feather LoRa board. Note that the shipped number only consists of 6 bytes while LoRaWAN requires an 8 bytes DevEUI. We simply add 0x00 0x00 in the middle of the 6 bytes provided. If you have lost the provided DevEUI you can also let the TTN console create a new one. After the registration of the device the respective keys (AppEUI, DevEUI, AppKey) can be copied from the TTN console and must be pasted into the the proper places in the source code of the sketch above. Please make sure that you choose for each of the three keys the correct byte ordering (DevEUI, AppEUI in LSB; AppKey in MSB). A detailed explanation of these steps is given here. Then the sketch can be compiled and uploaded to the Adafruit Feather 32u4 LoRa microcontroller. Note that the three constants (AppEUI, DevEUI, AppKey) must be changed in the source code for every new sensor node.

Using the OTAA mode has the advantage over the ABP (activation by personalization) mode that during connection the session keys are newly created which improves security. Another advantage is that the packet counter is automatically reset to 0 both in the node and in the TTN application.

TTN Payload Decoding

Everytime a data packet is received by a TTN application a dedicated Javascript function is being called (Payload Decoder Function). This function can be used to decode the received byte string and to create proper Javascript objects or values that can directly be read by humans when looking at the incoming data packet. This is also useful to format the data in a specific way that can then be forwarded to an external application (e.g. a sensor data platform like MyDevices or Thingspeak). Such a forwarding can be configured in the TTN console in the "Integrations" tab. *TTN payload decoder for Adafruit32u4 LoRa with display sensor node* given here checks if a packet was received on LoRaWAN port 7 and then assumes that it consists of the 6 bytes encoded as described above. It creates the three Javascript objects 'temperature', 'humidity', and 'vbattery'. Each object has two fields: 'value' holds the value and 'uom' gives the unit of measure. The source code can simply be copied and pasted into the 'decoder' tab in the TTN console after having selected the application. Choose the option 'Custom' in the 'Payload Format' field. Note that when you also want to handle other sensor nodes sending packets on different LoRaWAN ports, then the Payload Decoder Function can be extended after the end of the if (port==7) {...} statement by adding else if (port==8) {...} else if (port==9) {...} etc.

The Things Network - OGC SensorWeb Integration

The presented Payload Decoder Function works also with the TTN-OGC SWE Integration for the 52° North Sensor Observation Service (SOS). This software component can be downloaded from this repository. It connects a TTN application with a running transactional Sensor Observation Service 2.0.0 (SOS). Data packets received from TTN are imported into the SOS. The SOS persistently stores sensor data from an arbitrary number of sensor nodes and can be queried for the most recent as well as for historic sensor data readings. The 52° North SOS comes with its own REST API and a nice web client allowing to browse the stored sensor data in a convenient way.

We are running an instance of the 52° North SOS and the TTN-OGC SWE Integration. The web client for this LoRaWAN sensor node can be accessed on this page. Here is a screenshot showing the webclient:

Fig. 17: Web client for data visualization

Sending a message to the Sensor Node (Downlink)

Using the TTN console we can send a message (i.e. a byte string) to the sensor node. In the TTN console application page click on the respective application. Then click on the 'Devices' tab and choose the proper sensor node (here: adafruit-feather-32u4-lora3). On the overview page scroll down to the 'Downlink' section. In the 'Payload' field enter 1 to 4 bytes. In order to show digits or letters on the LED display these must be ASCII encoded and have to be entered as hexadecimal numbers. When you click on the 'Send' button the message will be queued and the next time when the node sends its data packet (uplink) it will receive the message. The first 4 bytes will be shown on the display and the beeper indicates the reception of a new downlink message. In order to blank the display just send a one byte message with the value '20' (hexadecimal for 32, which is the ASCII code for a space). When the node receives just a single blank character it will not produce a beeping sound. There is a nice web page offering online encoding of text to ASCII numbers in hexadecimal encoding. For example, in order to display the text 'LoRa', the four hexadecimal numbers 4C 6F 52 61 have to be entered in the Payload entry field.

2.5.4 Code files

Listing 20: Arduino Sketch for Adafruit32u4 LoRa with display sensor node

```
/*******************************************************************************
 * Arduino Sketch for a LoRaWAN sensor node that is registered with
 * 'The Things Network' (TTN) www.thethingsnetwork.org
 *
 * Author:  Thomas H. Kolbe, thomas.kolbe@tum.de
 * Version: 1.0.0
 * Last update: 2018-12-09
```

(continues on next page)

(continued from previous page)

```
 *
 * The sensor node is based on the Adafruit Feather LoRa microcontroller board
 * with either the AVR ATmega32u4 or the ATSAMD21G18 ARM Cortex M0 microcontroller.
 * See https://learn.adafruit.com/adafruit-feather-32u4-radio-with-lora-radio-module/
 * or https://learn.adafruit.com/adafruit-feather-m0-radio-with-lora-radio-module/
 * The sensor node uses a DHT22 sensor measuring air temperature and humidity.
 * Also the voltage of an attached LiPo battery is monitored and sent as
 * an observation. All three values are encoded as 2 byte integer values each.
 * Hence, the total message payload is 6 bytes. Before the values are converted
 * to integers they are multiplied by 100 to preserve 2 digits after the decimal
 * point. Thus, the received values must be divided by 100 to obtain the measured
 * values. The payload is sent every 300s to LoRaWAN port 7. The following
 * Javascript function can be used as a payload decoding function in TTN:
 *
 * function Decoder(bytes, port) {
 *   // Decode an uplink message from a buffer
 *   // (array) of bytes to an object of fields.
 *   if (port==7) {
 *     var decoded = {
 *       "temperature": (bytes[0] << 8 | bytes[1]) / 100.0,
 *       "humidity": (bytes[2] << 8 | bytes[3]) / 100.0,
 *       "vbattery": (bytes[4] << 8 | bytes[5]) / 100.0
 *     };
 *   } else {
 *     var decoded = null;
 *   }
 *   return decoded;
 * }
 *
 * In between two data transmissions the microcontroller board can go
 * into sleep mode to reduce energy consumption for extended operation
 * time when running on battery. Usage of the sleep mode must be
 * explicitly configured below.
 *
 * Note, that the DHT22 data pin must be connected to Digital Pin 6 of the
 * microcontroller board (for the Feather 32u4) or Digital Pin 12 (for
 * the Feather M0). A resistor of 4.7k - 10k Ohm must be connected to
 * the data pin and VCC (+3.3V).
 *
 * Digital Pin 5 (for the Feather 32u4) must be connected to DIO1 of the
 * LoRa transceiver module using a simple wire.
 *
 * For this node we also attach an Adafruit Feather Wing with a four digit
 * 14-segments LED display. The display controller is using I2C and the
 * I2C pins SDA and SCL are directly connected to the Adafruit Feather
 * via the Wing connectors. The wing is using the default I2C address
 * (0x70). Any LoRaWAN downlink message sent to this node is shown on
 * the display (only the first 4 characters). We treat each byte of the
 * received payload as a character in ASCII code. Besides numbers and
 * letters in upper and lower case also some special characters are
 * supported. For further details on the Feather Display Wing see here:
 * https://learn.adafruit.com/14-segment-alpha-numeric-led-featherwing
 *
 * In order to notify persons standing nearby that a new text was received
 * we let the node beep a couple of times. Therefore, Digital Pin 12 (for
 * the Feather 32u4) should be connected to the '+' port of a 3.3V buzzer
 * module. The '-' port of the buzzer must be connected to GND.
```

(continues on next page)

(continued from previous page)

```
 * If a payload containing just a single space (character code 0x20) is
 * received, the display will be blanked without emitting beeps.
 *
 * Note that if the LED display shows some text this will draw a
 * significant amount of power. This will certainly reduce the operational
 * duration when running on battery.
 *
 * The code is based on the Open Source library LMIC implementing the LoRaWAN
 * protocol stack on top of a given LoRa transceiver module (here: RFM95 from
 * HopeRF, which uses the Semtech SX1276 LoRa chip). The library is originally
 * being developed by IBM and has been ported to the Arduino platform. See
 * notes below from the original developers.
 *
 *******************************************************************************
 * Copyright (c) 2015 Thomas Telkamp and Matthijs Kooijman
 *
 * Permission is hereby granted, free of charge, to anyone
 * obtaining a copy of this document and accompanying files,
 * to do whatever they want with them without any restriction,
 * including, but not limited to, copying, modification and redistribution.
 * NO WARRANTY OF ANY KIND IS PROVIDED.
 *
 * This uses OTAA (Over-the-air activation), where a DevEUI and
 * application key is configured, which are used in an over-the-air
 * activation procedure where a DevAddr and session keys are
 * assigned/generated for use with all further communication.
 *
 * Note: LoRaWAN per sub-band duty-cycle limitation is enforced (1% in
 * g1, 0.1% in g2), but not the TTN fair usage policy (which is probably
 * violated by this sketch when left running for longer)!
 *
 * To use this sketch, first register your application and device with
 * the things network, to set or generate an AppEUI, DevEUI and AppKey.
 * Multiple devices can use the same AppEUI, but each device has its own
 * DevEUI and AppKey.
 *
 * Do not forget to define the radio type correctly in config.h.
 *
 *******************************************************************************/

// If the following line is uncommented, messages are being printed out to the
// serial connection for debugging purposes. When using the Arduino Integrated
// Development Environment (Arduino IDE), these messages are displayed in the
// Serial Monitor selecting the proper port and a baudrate of 115200.

// #define SERIALDEBUG

#ifdef SERIALDEBUG
  #define SERIALDEBUG_PRINT(...) Serial.print(__VA_ARGS__)
  #define SERIALDEBUG_PRINTLN(...) Serial.println(__VA_ARGS__)
#else
  #define SERIALDEBUG_PRINT(...)
  #define SERIALDEBUG_PRINTLN(...)
#endif

// If the following line is uncommented, the sensor node goes into sleep mode
// in between two data transmissions. Also the 2secs time between the
```

(continues on next page)

(continued from previous page)

```
// initialization of the DHT22 sensor and the reading of the observations
// is spent in sleep mode.
// Note, that on the Adafruit Feather 32u4 LoRa board the Serial connection
// gets lost as soon as the board goes into sleep mode, and it will not be
// established again. Thus, the definition of SERIALDEBUG should be commented
// out above when using sleep mode with this board.

#define SLEEPMODE

#ifdef SLEEPMODE
  #include <Adafruit_SleepyDog.h>
#endif

#include <lmic.h>
#include <hal/hal.h>
#include <SPI.h>

#include <util/atomic.h>
#include <avr/power.h>

#include <DHT.h>

#ifdef __AVR_ATmega32U4__
  #define DHTPIN            6     // Arduino Digital Pin which is connected to the DHT sensor for Feather 32u4.
#endif
#ifdef ARDUINO_SAMD_FEATHER_M0
  #define DHTPIN            12    // Arduino Digital Pin which is connected to the DHT sensor for Feather M0.
#endif
#define DHTTYPE           DHT22    // DHT 22 (AM2302)

DHT dht(DHTPIN, DHTTYPE);          // create the sensor object


#ifdef __AVR_ATmega32U4__
   #define VBATPIN A9             // battery voltage is measured from Analog Input A9 for Feather 32u4
#endif
#ifdef ARDUINO_SAMD_FEATHER_M0
   #define VBATPIN A7             // battery voltage is measured from Analog Input A7 for Feather M0
#endif

#ifdef __AVR_ATmega32U4__
  extern volatile unsigned long timer0_overflow_count;
#endif

#include <Wire.h>
#include <Adafruit_GFX.h>
#include "Adafruit_LEDBackpack.h"

Adafruit_AlphaNum4 alpha4 = Adafruit_AlphaNum4();

#define BUZZERPIN 12               // Arduino Digital Pin which is connected to the buzzer module

```

(continues on next page)

(continued from previous page)

```

// The following three constants (AppEUI, DevEUI, AppKey) must be changed
// for every new sensor node. We are using the LoRaWAN OTAA mode (over the
// air activation). Each sensor node must be manually registered in the
// TTN console at https://console.thethingsnetwork.org before it can be
// started. In the TTN console create a new device and enter the DevEUI
// number that was shipped with the Adafruit Feather LoRa board. Note that
// the shipped number only consists of 6 bytes while LoRaWAN requires
// an 8 bytes DevEUI. We simply add 0x00 0x00 in the middle of the 6 bytes
// provided. If you have lost the provided DevEUI you can also let the
// TTN console create a new one. After the registration of the device the
// three values can be copied from the TTN console. A detailed explanation
// of these steps is given in
// https://learn.adafruit.com/the-things-network-for-feather?view=all

// This EUI must be in little-endian format, so least-significant-byte
// first. When copying an EUI from ttnctl output, this means to reverse
// the bytes. For TTN issued EUIs the last bytes should be 0xD5, 0xB3, 0x70.
static const u1_t PROGMEM APPEUI[8]={ 0x55, 0xC1, 0x00, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX }
↪;
void os_getArtEui (u1_t* buf) { memcpy_P(buf, APPEUI, 8);}

// This should also be in little endian format, see above.
static const u1_t PROGMEM DEVEUI[8]={ 0xF6, 0xE2, 0x10, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX }
↪;
void os_getDevEui (u1_t* buf) { memcpy_P(buf, DEVEUI, 8);}

// This key should be in big endian format (or, since it is not really a
// number but a block of memory, endianness does not really apply). In
// practice, a key taken from ttnctl can be copied as-is.
// The key shown here is the semtech default key.
static const u1_t PROGMEM APPKEY[16] = { 0xC2, 0x21, 0x2E, 0x7A, 0xXX, 0xXX, 0xXX,␣
↪0xXX, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX };
void os_getDevKey (u1_t* buf) {  memcpy_P(buf, APPKEY, 16);}


// The following array of bytes is a placeholder to contain the message payload
// which is transmitted to the LoRaWAN gateway. We are currently only using 6 bytes.
// Please make sure to extend the size of the array, if more sensors should be
// attached to the sensor node and the message payload becomes larger than 10 bytes.
static uint8_t mydata[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0xA};

static osjob_t sendjob;

// Schedule transmission every TX_INTERVAL seconds (might become longer due to duty
// cycle limitations). The total interval time is 2secs for the measurement
// plus 3secs for the LoRaWAN packet transmission plus SLEEP_TIME seconds
// plus SLEEP_TIME seconds (microcontroller in sleep mode)
const unsigned int TX_INTERVAL = 300;       // overall cycle time (send one set of␣
↪observations every 5mins)
const unsigned int TX_TIME = 22;            // rough estimate of transmission time of␣
↪a single packet
const unsigned int MEASURE_TIME = 2;        // seconds measuring time
const unsigned int SLEEP_TIME = TX_INTERVAL - TX_TIME - MEASURE_TIME;
const unsigned int WAIT_TIME = TX_INTERVAL - TX_TIME - MEASURE_TIME;

// Pin mapping of the LoRa transceiver. Please make sure that DIO1 is connected
```

(continues on next page)

(continued from previous page)

```
// to Arduino Digital Pin 6 using an external wire. DIO2 is left unconnected
// (it is only required, if FSK modulation instead of LoRa would be used).
#ifdef __AVR_ATmega32U4__
  const lmic_pinmap lmic_pins = {
    .nss = 8,
    .rxtx = LMIC_UNUSED_PIN,
    .rst = 4,
    .dio = {7, 5, LMIC_UNUSED_PIN},     // in the Feather 32u4 DIO0 is connected to Arduino Digital Pin 7
  };
#endif
#ifdef ARDUINO_SAMD_FEATHER_M0
  const lmic_pinmap lmic_pins = {
    .nss = 8,
    .rxtx = LMIC_UNUSED_PIN,
    .rst = 4,
    .dio = {3, 6, LMIC_UNUSED_PIN},     // in the Feather M0 DIO0 is connected to Arduino Digital Pin 3
  };
#endif

void onEvent (ev_t ev) {
    SERIALDEBUG_PRINT(os_getTime());
    SERIALDEBUG_PRINT(": ");
    switch(ev) {
        case EV_SCAN_TIMEOUT:
            SERIALDEBUG_PRINTLN(F("EV_SCAN_TIMEOUT"));
            break;
        case EV_BEACON_FOUND:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_FOUND"));
            break;
        case EV_BEACON_MISSED:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_MISSED"));
            break;
        case EV_BEACON_TRACKED:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_TRACKED"));
            break;
        case EV_JOINING:
            SERIALDEBUG_PRINTLN(F("EV_JOINING"));
            break;
        case EV_JOINED:
            SERIALDEBUG_PRINTLN(F("EV_JOINED"));

            // Disable link check validation (automatically enabled
            // during join, but not supported by TTN at this time).
//            LMIC_setLinkCheckMode(0);
            break;
        case EV_RFU1:
            SERIALDEBUG_PRINTLN(F("EV_RFU1"));
            break;
        case EV_JOIN_FAILED:
            SERIALDEBUG_PRINTLN(F("EV_JOIN_FAILED"));
            break;
        case EV_REJOIN_FAILED:
            SERIALDEBUG_PRINTLN(F("EV_REJOIN_FAILED"));
            break;
        case EV_TXCOMPLETE:
```

(continues on next page)

(continued from previous page)

```
            digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
            SERIALDEBUG_PRINTLN(F("EV_TXCOMPLETE (includes waiting for RX windows)"));
            if (LMIC.txrxFlags & TXRX_ACK)
              SERIALDEBUG_PRINTLN(F("Received ack"));
            if (LMIC.dataLen) {
#ifdef SERIALDEBUG
              SERIALDEBUG_PRINT(F("Received "));
              SERIALDEBUG_PRINT(LMIC.dataLen);
              SERIALDEBUG_PRINT(F(" bytes of payload: 0x"));
              for (int i=0; i<LMIC.dataLen; i++) {
                if (LMIC.frame[LMIC.dataBeg + i] < 0x10) {
                  SERIALDEBUG_PRINT(F("0"));
                }
                SERIALDEBUG_PRINT(LMIC.frame[LMIC.dataBeg + i], HEX);
              }
              SERIALDEBUG_PRINTLN();
#endif
              // add your code to handle a received downlink data packet here
              alpha4.clear();
              for (int i=0; i<LMIC.dataLen && i<4; i++) {
                  alpha4.writeDigitAscii(i, LMIC.frame[LMIC.dataBeg + i]);
              }
              alpha4.writeDisplay();
              if (!(LMIC.frame[LMIC.dataBeg]==' ' && LMIC.dataLen==1))
                  messagebeep();
            }

#ifdef SLEEPMODE
            // Schedule next transmission in 1ms second after the board returns from sleep mode
            os_setTimedCallback(&sendjob, os_getTime()+ms2osticks(1), do_send);

            SERIALDEBUG_PRINTLN("going to sleep now ... ");
            // lmic library sleeps automatically after transmission has been completed

            doSleep((uint32_t)SLEEP_TIME*1000);
/*
            int sleepcycles = (int)SLEEP_TIME / 8;
            int restsleep = (int)SLEEP_TIME % 8;
            for(int i=0; i < sleepcycles; i++) {
              Watchdog.sleep(8000); // maximum seems to be 8 seconds
              SERIALDEBUG_PRINT('.');
            }
            if (restsleep) {
              Watchdog.sleep(restsleep*1000);
              SERIALDEBUG_PRINT('*');
            }
            SERIALDEBUG_PRINTLN("... woke up again");

#ifdef __AVR_ATmega32U4__
            // The following statement is required to prevent that LMIC spends another
            // couple of seconds busy waiting for some RX packets. This is only required
            // when using SLEEPMODE, because during sleep mode the Arduino timer variables
```

(continues on next page)

(continued from previous page)

```
            // are not being incremented and LMIC job scheduling is based on this.
//            timer0_overflow_count += 3E6;

            ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
                extern volatile unsigned long timer0_millis;
                extern volatile unsigned long timer0_overflow_count;
                timer0_millis += SLEEP_TIME*1000;
                // timer0 uses a /64 prescaler and overflows every 256 timer ticks
                timer0_overflow_count += microsecondsToClockCycles((uint32_t)SLEEP_
↪TIME * 1000000) / (64 * 256);
            }
#endif
*/
            // We need to reset the duty cycle limits within the LMIC library.
            // The reason is that in sleep mode the Arduino system timers millis and␣
↪micros
            // do not get incremented. However, LMIC monitors the adherence to the
            // LoRaWAN duty cycle limitations using the system timers millis and␣
↪micros.
            // Since LMIC does not know that we have slept for a long time and duty
            // cycle requirements in fact are met, we must reset the respective LMIC␣
↪timers
            // in order to prevent the library to wait for some extra time (which␣
↪would
            // not use sleep mode and, thus, would waste battery energy).
            LMIC.bands[BAND_MILLI].avail = os_getTime();
            LMIC.bands[BAND_CENTI].avail = os_getTime();
            LMIC.bands[BAND_DECI].avail = os_getTime();
#else
            // Schedule next transmission
            os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(WAIT_TIME), do_
↪send);
#endif
                break;
        case EV_LOST_TSYNC:
            SERIALDEBUG_PRINTLN(F("EV_LOST_TSYNC"));
            break;
        case EV_RESET:
            SERIALDEBUG_PRINTLN(F("EV_RESET"));
            break;
        case EV_RXCOMPLETE:
            // data received in ping slot
            SERIALDEBUG_PRINTLN(F("EV_RXCOMPLETE"));
            break;
        case EV_LINK_DEAD:
            SERIALDEBUG_PRINTLN(F("EV_LINK_DEAD"));
            break;
        case EV_LINK_ALIVE:
            SERIALDEBUG_PRINTLN(F("EV_LINK_ALIVE"));
            break;
         default:
            SERIALDEBUG_PRINTLN(F("Unknown event"));
            break;
    }
}

void do_send(osjob_t* j){
```

(continues on next page)

(continued from previous page)

```
    // Check if there is not a current TX/RX job running
    if (LMIC.opmode & OP_TXRXPEND) {
        SERIALDEBUG_PRINTLN(F("OP_TXRXPEND, not sending"));
    } else {
        // Prepare upstream data transmission at the next possible time.

        float temperature, humidity, measuredvbat;
        int16_t int16_temperature, int16_humidity, int16_vbat;

        // Start a measurement to update the sensor's internal temperature & humidity reading.
        // Note, that when fetching measurements from a DHT22 sensor, the reported
        // values belong to the measurement BEFORE the current measurement.
        // Therefore, in order to get current observations, we first perform a new measurement
        // and wait 2 secs (which is the minimum time between two sensor observations for
        // the DHT22) and then directly retrieve the observations again.

        temperature = dht.readTemperature();
#ifdef SLEEPMODE
//        Watchdog.sleep(MEASURE_TIME * 1000UL);
        doSleep(MEASURE_TIME * 1000UL);
#else
        delay(MEASURE_TIME * 1000UL);
#endif

        // Now read the recently measured temperature (2 secs ago) as Celsius (the default)
        temperature = dht.readTemperature();

        // Read the recently measured humidity (2 secs ago)
        humidity = dht.readHumidity();

        // Check if any reads failed and exit early (to try again).
        if (isnan(humidity) || isnan(temperature)) {
            SERIALDEBUG_PRINTLN("Failed to read from DHT sensor!");
            // blink the LED five times to indicate that the sensor values could not be read
            for (int i=0; i<5; i++) {
              digitalWrite(LED_BUILTIN, HIGH);    // turn the LED on by making the voltage HIGH
              delay(150);
              digitalWrite(LED_BUILTIN, LOW);    // turn the LED on by making the voltage HIGH
              delay(150);
            }
            // ok, then wait for another period and try it again
            os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(TX_INTERVAL), do_send);
        } else {
            SERIALDEBUG_PRINT("Humidity: ");
            SERIALDEBUG_PRINT(humidity);
            SERIALDEBUG_PRINT(" %\t");
            SERIALDEBUG_PRINT("Temperature: ");
            SERIALDEBUG_PRINT(temperature);
            SERIALDEBUG_PRINT(" °C ");
```

(continues on next page)

(continued from previous page)

```
            int16_temperature = round(100.0*temperature);
            int16_humidity = round(100.0*humidity);
            mydata[0] = (byte) (int16_temperature >> 8);
            mydata[1] = (byte) (int16_temperature & 0x00FF);
            mydata[2] = (byte) (int16_humidity >> 8);
            mydata[3] = (byte) (int16_humidity & 0x00FF);

            measuredvbat = analogRead(VBATPIN);
            measuredvbat *= 2.0;      // we divided by 2, so multiply back
            measuredvbat *= 3.3;      // Multiply by 3.3V, our reference voltage
            measuredvbat /= 1023.0;   // convert to voltage
            int16_vbat = round(measuredvbat * 100.0);
            mydata[4] = (byte) (int16_vbat >> 8);
            mydata[5] = (byte) (int16_vbat & 0x00FF);
            SERIALDEBUG_PRINT(" \t");
            SERIALDEBUG_PRINT("Battery Voltage: ");
            SERIALDEBUG_PRINT(measuredvbat);
            SERIALDEBUG_PRINTLN(" V");

            // Send the 6 bytes payload to LoRaWAN port 7 and do not request an␣
↪acknowledgement.
            // The following call does not directly sends the data, but puts a "send␣
↪job"
            // in the job queue. This job eventually is performed in the call "os_
↪runloop_once();"
            // issued repeatedly in the "loop()" method below. After the transmission␣
↪is
            // complete, the EV_TXCOMPLETE event is signaled, which is handled in the
            // event handler method "onEvent (ev_t ev)" above. In the EV_TXCOMPLETE␣
↪branch
            // then a new call to the "do_send(osjob_t* j)" method is being prepared␣
↪for
            // delayed execution with a waiting time of TX_INTERVAL seconds.
            LMIC_setTxData2(7, mydata, 6, 0);
            SERIALDEBUG_PRINTLN(F("Packet queued"));
            digitalWrite(LED_BUILTIN, HIGH);    // turn the LED on by making the␣
↪voltage HIGH

            // Next TX is scheduled after TX_COMPLETE event.
        }
    }
}

void doSleep(uint32_t time) {
  ADCSRA &= ~(1 << ADEN);
  power_adc_disable();

  while (time > 0) {
    uint16_t slept;
    if (time < 8000)
      slept = Watchdog.sleep(time);
    else
      slept = Watchdog.sleep(8000);

    // Update the millis() and micros() counters, so duty cycle
    // calculations remain correct. This is a hack, fiddling with
```

(continues on next page)

(continued from previous page)

```
    // Arduino's internal variables, which is needed until
    // https://github.com/arduino/Arduino/issues/5087 is fixed.
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
      extern volatile unsigned long timer0_millis;
      extern volatile unsigned long timer0_overflow_count;
      timer0_millis += slept;
      // timer0 uses a /64 prescaler and overflows every 256 timer ticks
      timer0_overflow_count += microsecondsToClockCycles((uint32_t)slept * 1000) /␣
↪(64 * 256);
    }

    if (slept >= time)
      break;
    time -= slept;
  }

  power_adc_enable();
  ADCSRA |= (1 << ADEN);
}

void beep(bool longbeep) {
    digitalWrite(BUZZERPIN, HIGH);   // turn the BUZZER off by making the voltage LOW
    if (longbeep)
      delay(250);
    else
      delay(100);
    digitalWrite(BUZZERPIN, LOW);   // turn the BUZZER off by making the voltage LOW
    delay(100);
}

void messagebeep() {
    beep(false);
    beep(true);
    beep(false);
    beep(false);
    delay(200);
    beep(false);
    beep(true);
    beep(false);
    beep(false);
}

void setup() {
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW

    pinMode(BUZZERPIN, OUTPUT);
    digitalWrite(BUZZERPIN, LOW);   // turn the BUZZER off by making the voltage LOW

    alpha4.begin(0x70);             // pass in the I2C address of the display
    alpha4.clear();
    alpha4.writeDisplay();
    alpha4.writeDigitAscii(0, 'T');
    alpha4.writeDigitAscii(1, 'e');
    alpha4.writeDigitAscii(2, 's');
    alpha4.writeDigitAscii(3, 't');
    alpha4.writeDisplay();
```

(continues on next page)

(continued from previous page)

```
539
540     messagebeep();
541
542     delay(10000);                     // give enough time to open serial monitor (if
    ↪needed) or to start uploading of a new sketch
543
544     alpha4.clear();
545     alpha4.writeDisplay();
546
547 #ifdef SERIALDEBUG
548     Serial.begin(115200);
549     // while (!Serial);
550 #endif
551
552     dht.begin();                      // initialize DHT22 sensor
553
554     SERIALDEBUG_PRINTLN(F("Starting"));
555
556     #ifdef VCC_ENABLE
557     // For Pinoccio Scout boards
558     pinMode(VCC_ENABLE, OUTPUT);
559     digitalWrite(VCC_ENABLE, HIGH);
560     delay(1000);
561     #endif
562
563     // LMIC init
564     os_init();
565     // Reset the MAC state. Session and pending data transfers will be discarded.
566     LMIC_reset();
567     LMIC_setClockError(MAX_CLOCK_ERROR * 1 / 100);
568
569     #if defined(CFG_eu868)
570     // Set up the channels used by the Things Network, which corresponds
571     // to the defaults of most gateways. Without this, only three base
572     // channels from the LoRaWAN specification are used, which certainly
573     // works, so it is good for debugging, but can overload those
574     // frequencies, so be sure to configure the full frequency range of
575     // your network here (unless your network autoconfigures them).
576     // Setting up channels should happen after LMIC_setSession, as that
577     // configures the minimal channel set.
578     // NA-US channels 0-71 are configured automatically
579     LMIC_setupChannel(0, 868100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
    ↪// g-band
580     LMIC_setupChannel(1, 868300000, DR_RANGE_MAP(DR_SF12, DR_SF7B), BAND_CENTI);
    ↪// g-band
581     LMIC_setupChannel(2, 868500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
    ↪// g-band
582     LMIC_setupChannel(3, 867100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
    ↪// g-band
583     LMIC_setupChannel(4, 867300000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
    ↪// g-band
584     LMIC_setupChannel(5, 867500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
    ↪// g-band
585     LMIC_setupChannel(6, 867700000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
    ↪// g-band
586     LMIC_setupChannel(7, 867900000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
    ↪// g-band
```

(continues on next page)

(continued from previous page)

```
    LMIC_setupChannel(8, 868800000, DR_RANGE_MAP(DR_FSK,  DR_FSK),  BAND_MILLI);
↪// g2-band
    // TTN defines an additional channel at 869.525Mhz using SF9 for class B
    // devices' ping slots. LMIC does not have an easy way to define set this
    // frequency and support for class B is spotty and untested, so this
    // frequency is not configured here.
    #elif defined(CFG_us915)
    // NA-US channels 0-71 are configured automatically
    // but only one group of 8 should (a subband) should be active
    // TTN recommends the second sub band, 1 in a zero based count.
    // https://github.com/TheThingsNetwork/gateway-conf/blob/master/US-global_conf.
↪json
    LMIC_selectSubBand(1);
    #endif

    // Disable link check validation
//    LMIC_setLinkCheckMode(0);

    // TTN uses SF9 for its RX2 window.
    LMIC.dn2Dr = DR_SF9;

    // Set data rate and transmit power for uplink (note: txpow seems to be ignored␣
↪by the library)
    LMIC_setDrTxpow(DR_SF9,14);

    // Start job. This will initiate the repetitive sending of data packets,
    // because after each data transmission, a delayed call to "do_send()"
    // is being scheduled again.
    do_send(&sendjob);

    // The following settings should further reduce energy consumption. I have not
    // tested them yet, they are taken from a post in the TTN forum. See
    // https://www.thethingsnetwork.org/forum/t/adafruit-lora-feather-gateway/2440/50
    /*
    power_adc_disable();
    power_usart0_disable();
    power_twi_disable();
    power_timer1_disable();
    power_timer2_disable();
    power_timer3_disable();
    power_usart1_disable();
    power_usb_disable();
    USBCON |= (1 << FRZCLK);
    PLLCSR &= ~(1 << PLLE);
    USBCON &= ~(1 << USBE );
    clock_prescale_set(clock_div_2);
    */
}

void loop() {
    os_runloop_once();
}
```

Listing 21: TTN payload decoder for Adafruit32u4 LoRa with display sensor node

```
function Decoder (bytes, port) {
  var result = {};
  var transformers = {};

  if (port==7) {
     transformers = {
      'temperature': function transform (bytes) {
          value=bytes[0]*256 + bytes[1];
          if (value>=32768) value=value-65536;
          return value/100.0;
        },
      'humidity': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/100.0;
        },
      'vbattery': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/100.0;
        },
    }

    result['temperature'] = {
      value: transformers['temperature'](bytes.slice(0, 2)),
      uom: 'Celsius',
    }

    result['humidity'] = {
      value: transformers['humidity'](bytes.slice(2, 4)),
      uom: 'Percent',
    }

    result['vbattery'] = {
      value: transformers['vbattery'](bytes.slice(4, 6)),
      uom: 'Volt',
    }
  }

  return result;
}
```

2.5.5 References

- Adafruit Feather 32u4 LoRa microntroller
- Adafruit Feather 32u4 LoRa tutorial
- IBM LMIC (LoraMAC-in-C) library for Arduino
- Using Adafruit Feather 32u4 RFM95 as an TTN Node - Stories - Labs
- TTN LoraWan Atmega32U4 based node – ABP version | Primal Cortex's Weblog
- node-workshop/lora32u4.md at master · kersing/node-workshop · GitHub
- Got Adafruit Feather 32u4 LoRa Radio to work and here is how - End Devices (Nodes) - The Things Network
- Adafruit Feather as LoRaWAN node | Wolfgang Klenk

- LMiC on Adafruit Lora Feather successfully sends message to TTN and then halts with “Packet queued” - End Devices (Nodes) - The Things Network
- GitHub - marcuscbehrens/loralife
- GPS-Tracker - Stories - Labs

On battery saving / using the deep sleep mode

- Adafruit Feather 32u4 LoRa - long transmission time after deep sleep - End Devices (Nodes) - The Things Network and this
- Full Arduino Mini LoraWAN and 1.3uA Sleep Mode - End Devices (Nodes) - The Things Network
- Adding Method to Adjust hal_ticks Upon Waking Up from Sleep · Issue #109 · matthijskooijman/arduino-lmic
- minilora-test/minilora-test.ino at cbe686826bd84fac8381de47b5f5b02dd47c2ca0 · tkerby/minilora-test
- Arduino-LMIC library with low power mode - Mario Zwiers

2.6 Adafruit M0 LoRa

2.6.1 Hardware

Microcontroller

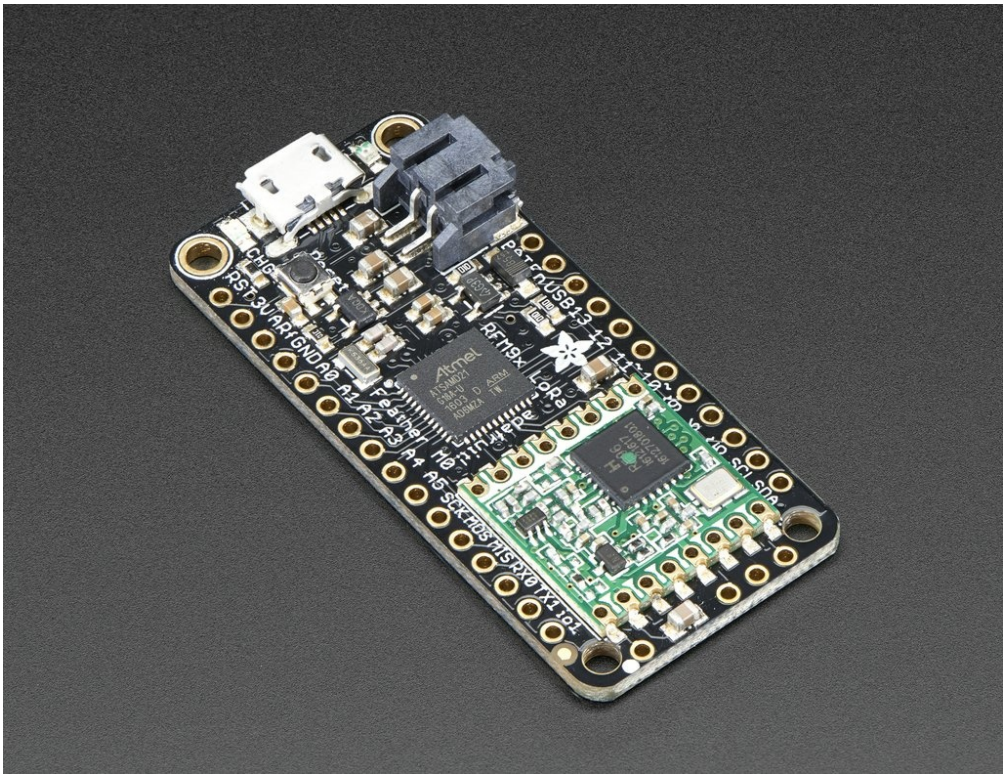


Fig. 18: Feather M0 with RFM95 LoRa Radio - 900 MHz - RadioFruit from Adafruit. Feather M0 LoRa tutorial with explanations, datasheets, and examples.

The Adafruit Feather M0 LoRa board is operated by the 32bit ATSAMD21G18 ARM Cortex M0 microcontroller running at 48MHz. It has 256 KB flash memory (to store the program code) and 32 KB of RAM (to store variables, status information, and buffers). The operating voltage of the board is 3.3V (this is important when attaching sensors and other peripherals; they also must operate on 3.3V). The board offers 20 general purpose digital input/output pins (20 GPIOs) with 10 analog input pins (with 12bit analog digital converters (ADC)), one analog output pin, one serial port (programmable Universal Asynchronous Receiver and Transmitter, UART), one I2C port, one SPI port, one USB port. The board comes with an embedded Lithium polymer battery management chip and status indicator led, which allows to directly connect a 3.7V LiPo rechargeable battery that will be automatically recharged when the board is powered over its USB connector. The Adafruit Feather M0 LoRa board is available in German shops from around 37 € to 45 €.

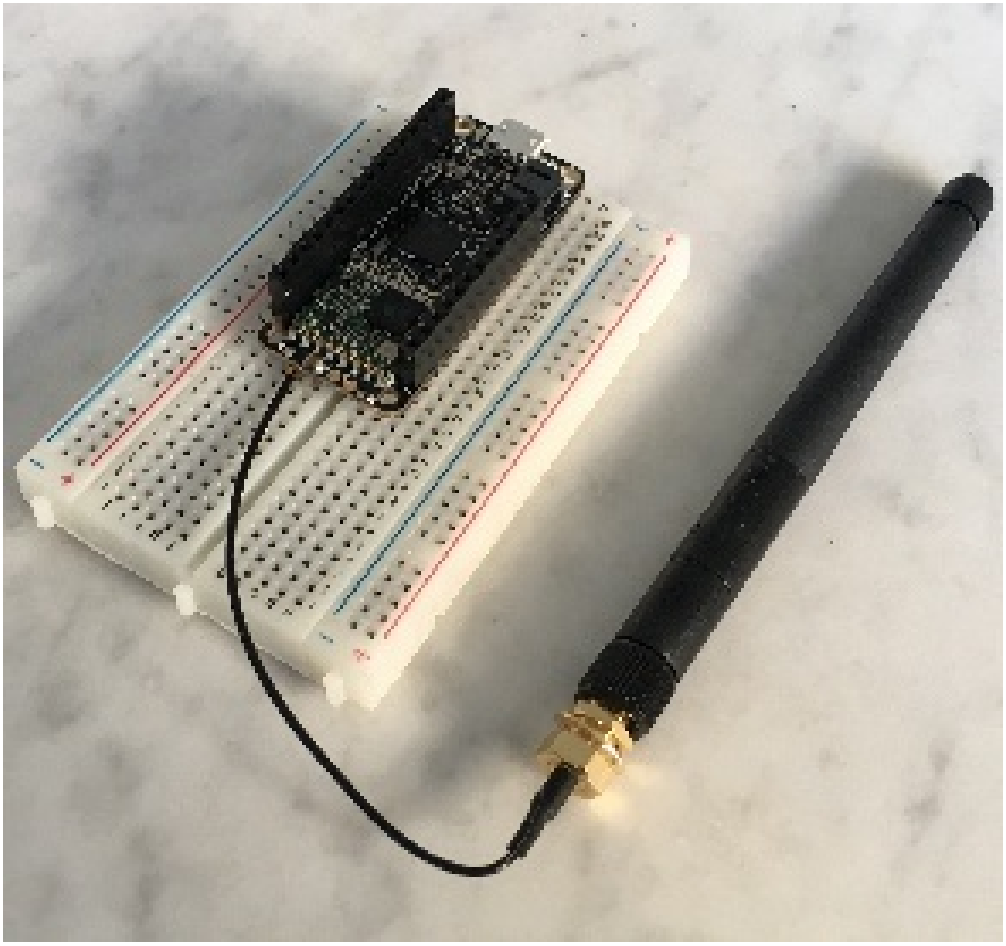


Fig. 19: The Adafruit Feather M0 RFM95 LoRa with attached antenna placed onto a prototyping breadboard. (On this photo the DHT22 sensor and the LiPo battery are missing; we will upload a new photo in the future)

The LoRa transmitter and receiver is encapsulated within an RFM95 module from the company HopeRF. This module uses the LoRa chip SX1276 from the company Semtech and is dedicated to the 868 MHz frequency band. The RFM95 module is connected via SPI interface to the microcontroller. Most of the required connections of the LoRa transceiver pins with the microcontroller are already built-in on the Adafruit Feather M0 LoRa board. However, Digital Pin 6 of the microcontroller must be connected to DIO1 of the LoRa transceiver module in addition using a simple wire. Since the module only implements the LoRa physical layer, the LoRaWAN protocol stack must be implemented in software on the microcontroller. We are using the Arduino library LMIC for that purpose (see below). The implemented LoRaWAN functionality is compatible with LoRaWAN Class A/C.

Sensor

We have attached a DHT22 sensor to the microcontroller board, which measures air temperature and humidity. The minimal time interval between two measurements is 2 seconds. All data transfers between the DHT22 and the microcontroller use a single digital line. The sensor data pin is attached to a GPIO pin (here: Digital Pin 12) of the microcontroller. In addition, a so-called pull-up resistor of 4.7k to 10k Ohm must be connected between the data line and VCC (+3.3V). The DHT22 datasheet can be accessed here. A tutorial on how to use the DHT22 sensor with Arduino microcontrollers is provided on this page. The sensor is available in German shops for around 4 € to 10 €.

2.6.2 Software

The sensor node has been programmed using the Arduino IDE. Please note, that in the Arduino framework a program is called a 'Sketch'.

Now download and run the *Arduino Sketch for Adafruit M0 LoRa sensor node* file in the Arduino IDE. After the sketch has successfully established a connection to The Things Network it reports the air temperature, humidity, and the voltage of a (possibly) attached LiPo battery every 5 minutes. All three values are being encoded in two byte integer values each and then sent as a 6 bytes data packet to the respective TTN application using LoRaWAN port 7. Please note, that LoRaWAN messages can be addressed to ports 1-255 (port 0 is reserved); these ports are similar to port numbers 0-65535 when using the Internet TCP/IP protocol. Voltage and humidity values are always greater or equal to 0, but the temperature value can also become negative. Negative values are represented as a two's complement; this must be considered in the Payload Decoding Function used in The Things Network (see *below*).

In between two sensor readings the microcontroller is going into deep sleep mode to save battery power. With a 2000 mAh LiPo battery and the current version of the sketch the system can run for at least 3 months. (Further optimizations would be possible, for example, not switching on the LED on the microcontroller board during LoRa data transmissions.)

The employed RFM95 LoRa module does not provide built-in support of the LoRaWAN protocol. Thus, it has to be implemented on the ARM Cortex M0 microcontroller. We use the IBM LMIC (LoraMAC-in-C) library for Arduino, which can be downloaded from this repository. The ARM Cortex M0 microcontroller has 256 KB of flash memory, which is plenty enough for the LMIC library, the code dealing with the sensors, and even some sophisticated analysis tasks (if required). The source code is given in the following listing:

Note, that the source code is very similar to the source code for the Adafruit Feather 32u4 LoRa board given on the Wiki page LoRaWAN Node - Adafruit 32u4 LoRa. The source code for the Adafruit Feather 32u4 LoRa board has also more detailed comments. It is planned to merge them into a single source code that can be used and compiled for both types of microcontrollers (ATmega32u4 and ARM Cortex M0). The merged source code is already available from LoRaWAN Node - Adafruit 32u4 LoRa, but was not tested with the M0 microcontroller board yet.

Note also, that there is an open issue regarding the deep sleep mode on the ARM Cortex M0 microcontroller in the source code above. During deep sleep mode the (software) timers of the LMIC library are not incremented and after wake-up the library does not recognize that enough time has passed to allow sending another data packet. This built-in mechanism of the LMIC library should ensure that the sensor node does not exceed the maximum duty cycle for LoRaWAN of 1%. This somehow also affects the waiting time for a possible downlink data packet coming from the gateway. As a consequence, the sensor node is not only active for around 2.5 seconds (0.5 seconds to submit the most recent datapacket to the gateway (uplink) and 2 seconds to wait for possible downlink data packets), but sometimes for about 5-6 seconds before it goes back into deep sleep mode (this can be seen from the duration the red LED is activated on the board). These extra seconds awake (with the LED and the LoRa transceiver module switched on) reduce battery lifetime significantly. The ATmega32u4 microcontroller does not have these problems and can go faster back to deep sleep mode. As a result the Adafruit Feather 32u4 LoRa board can run with a 1000 mAh LiPo battery for 5 months and the Adafruit Feather M0 LoRa board with a 2000 mAh LiPo battery for only 3 months.

2.6.3 Services

The services used for this sensor-node are:

- *TheThingsNetwork* service for LoRaWAN network service.
- *TheThingsNetwork - OGC SensorWeb* integration for uploading LoRaWAN sensor data into OGC infrastructure.

Registration of the sensor node with The Things Network (TTN)

The LoRaWAN protocol makes use of a number of different identifiers, addresses, keys, etc. These are required to unambiguously identify devices, applications, as well as to encrypt and decrypt messages. The names and meanings are nicely explained on a dedicated TTN web page.

The sketch given above connects the sensor node with The Things Network (TTN) using the Activation-by-Personalisation (ABP) mode. In this mode, the required keys for data encryption and session management are created manually using the TTN console window and must be pasted into the source code of the sketch provided in *software section* . In order to get this running, you will need to create a new device <https://www.thethingsnetwork.org/docs/devices/registration.html>‘_ in the TTN console window. This assumes that you already have a TTN user account (which needs to be created otherwise). In the settings menu of the newly created device the ABP mode must be selected and the settings must be saved. Then copy the DevAddr, the NwkSKey, and the AppSKey from the TTN console web page of the newly registered device and paste them into the proper places in the sketch above. Please make sure that you choose for each of the three keys the correct byte ordering (MSB for all three keys). A detailed explanation of these steps is given here. Then the sketch can be compiled and uploaded to the Adafruit Feather M0 LoRa microcontroller.

Important hint: everytime the sensor node is reset or being started again, make sure to reset the frame counter of the registered sensor in the TTN console web page of the registered device. The reason is that in LoRaWAN all transmitted data packets have a frame counter, which is incremented after each data frame being sent. This way a LoRaWAN application can avoid receiving and using the same packet again (replay attack). When TTN receives a data packet, it checks if the frame number is higher than the last one received before. If not, the received packet is considered to be old or a replay attack and is discarded. When the sensor node is reset or being started again, its frame counter is also reset to 0, hence, the TTN application assumes that all new packages are old, because their frame counter is lower than the last frame received (before the reset). A manual frame counter reset is only necessary when registering the node using ABP mode. In OTAA mode the frame counter is automatically reset in the sensor node and the TTN network server.

TTN Payload Decoding

Everytime a data packet is received by a TTN application a dedicated Javascript function is being called (Payload Decoder Function). This function can be used to decode the received byte string and to create proper Javascript objects or values that can directly be read by humans when looking at the incoming data packet. This is also useful to format the data in a specific way that can then be forwarded to an external application (e.g. a sensor data platform like MyDevices or Thingspeak). Such a forwarding can be configured in the TTN console in the “Integrations” tab. *TTN payload decoder for Adafruit M0 LoRa sensor node* given here checks if a packet was received on LoRaWAN port 7 and then assumes that it consists of the 6 bytes encoded as described above. It creates the three Javascript objects ‘temperature’, ‘humidity’, and ‘vbattery’. Each object has two fields: ‘value’ holds the value and ‘uom’ gives the unit of measure. The source code can simply be copied and pasted into the ‘decoder’ tab in the TTN console after having selected the application. Choose the option ‘Custom’ in the ‘Payload Format’ field. Note that when you also want to handle other sensor nodes sending packets on different LoRaWAN ports, then the Payload Decoder Function can be extended after the end of the if (port==7) {. . . } statement by adding else if (port==8) {. . . } else if (port==9) {. . . } etc.

The Things Network - OGC SensorWeb Integration

The presented Payload Decoder Function works also with the TTN-OGC SWE Integration for the 52° North Sensor Observation Service (SOS). This software component can be downloaded from this repository. It connects a TTN application with a running transactional Sensor Observation Service 2.0.0 (SOS). Data packets received from TTN are imported into the SOS. The SOS persistently stores sensor data from an arbitrary number of sensor nodes and can be queried for the most recent as well as for historic sensor data readings. The 52° North SOS comes with its own REST API and a nice web client allowing to browse the stored sensor data in a convenient way.

We are running an instance of the 52° North SOS and the TTN-OGC SWE Integration. The web client for this LoRaWAN sensor node can be accessed on this page. Here is a screenshot showing the webclient: (Note that the sensor node was wrongly registered with TTN using the name adafruit-feather-32u4-lora - it should have been adafruit-feather-m0-lora. Hence, while the legend says it is a 32u4 microcontroller in fact it is the M0)

Fig. 20: Web client for data visualization

2.6.4 Code files

Listing 22: Arduino Sketch for Adafruit M0 LoRa sensor node

```
/*******************************************************************************
 * Copyright (c) 2015 Thomas Telkamp and Matthijs Kooijman
 *
 * Permission is hereby granted, free of charge, to anyone
 * obtaining a copy of this document and accompanying files,
 * to do whatever they want with them without any restriction,
 * including, but not limited to, copying, modification and redistribution.
 * NO WARRANTY OF ANY KIND IS PROVIDED.
 *
```

(continues on next page)

(continued from previous page)

```
 * This example sends a valid LoRaWAN packet with payload "Hello,
 * world!", using frequency and encryption settings matching those of
 * the The Things Network.
 *
 * This uses ABP (Activation-by-personalisation), where a DevAddr and
 * Session keys are preconfigured (unlike OTAA, where a DevEUI and
 * application key is configured, while the DevAddr and session keys are
 * assigned/generated in the over-the-air-activation procedure).
 *
 * Note: LoRaWAN per sub-band duty-cycle limitation is enforced (1% in
 * g1, 0.1% in g2), but not the TTN fair usage policy (which is probably
 * violated by this sketch when left running for longer)!
 *
 * To use this sketch, first register your application and device with
 * the things network, to set or generate a DevAddr, NwkSKey and
 * AppSKey. Each device should have their own unique values for these
 * fields.
 *
 * Do not forget to define the radio type correctly in config.h.
 *
 *******************************************************************************/

// #define SERIALDEBUG

#ifdef SERIALDEBUG
  #define SERIALDEBUG_PRINT(...) Serial.print(__VA_ARGS__)
  #define SERIALDEBUG_PRINTLN(...) Serial.println(__VA_ARGS__)
#else
  #define SERIALDEBUG_PRINT(...)
  #define SERIALDEBUG_PRINTLN(...)
#endif


#include <lmic.h>
#include <hal/hal.h>
#include <SPI.h>

#include <Adafruit_SleepyDog.h>

// #include <Adafruit_Sensor.h>
#include <DHT.h>
// #include <DHT_U.h>

#define DHTPIN            12        // Pin which is connected to the DHT sensor.
#define DHTTYPE           DHT22     // DHT 22 (AM2302)

// DHT_Unified dht(DHTPIN, DHTTYPE);
DHT dht(DHTPIN, DHTTYPE);

#define VBATPIN A7

// LoRaWAN NwkSKey, network session key
// This should be in big-endian (aka msb).
static const PROGMEM u1_t NWKSKEY[16] = {NETWORK_SESSION_KEY_HERE_IN_MSB_FORMAT};

// LoRaWAN AppSKey, application session key
// This should also be in big-endian (aka msb).
```

(continues on next page)

(continued from previous page)

```
static const u1_t PROGMEM APPSKEY[16] = {APPLICATION_SESSION_KEY_HERE_IN_MSB_FORMAT};

// LoRaWAN end-device address (DevAddr)
// See http://thethingsnetwork.org/wiki/AddressSpace
// The library converts the address to network byte order as needed, so this should be in big-endian (aka msb) too.
static const u4_t DEVADDR = 0x260XXXXX   ; // <-- Change this address for every node!

// These callbacks are only used in over-the-air activation, so they are
// left empty here (we cannot leave them out completely unless
// DISABLE_JOIN is set in config.h, otherwise the linker will complain).
void os_getArtEui (u1_t* buf) { }
void os_getDevEui (u1_t* buf) { }
void os_getDevKey (u1_t* buf) { }

static uint8_t mydata[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0xA};
static osjob_t sendjob;

// Schedule TX every this many seconds (might become longer due to duty
// cycle limitations).
const unsigned TX_INTERVAL = 1;        // seconds transmit cycle plus ...
const unsigned SLEEP_TIME = 60*4+55;   // seconds sleep time plus ...
const unsigned MEASURE_TIME = 2;       // seconds measuring time should lead to ...
                                       // 5 minute(s) total cycle time

// Pin mapping
const lmic_pinmap lmic_pins = {
    .nss = 8,
    .rxtx = LMIC_UNUSED_PIN,
    .rst = 4,
    .dio = {3, 6, LMIC_UNUSED_PIN},
};


void onEvent (ev_t ev) {
//    Serial.print(os_getTime());
//    Serial.print(": ");
    SERIALDEBUG_PRINT(os_getTime());
    SERIALDEBUG_PRINT(": ");
    switch(ev) {
        case EV_SCAN_TIMEOUT:
            SERIALDEBUG_PRINTLN(F("EV_SCAN_TIMEOUT"));
            break;
        case EV_BEACON_FOUND:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_FOUND"));
            break;
        case EV_BEACON_MISSED:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_MISSED"));
            break;
        case EV_BEACON_TRACKED:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_TRACKED"));
            break;
        case EV_JOINING:
            SERIALDEBUG_PRINTLN(F("EV_JOINING"));
            break;
        case EV_JOINED:
            SERIALDEBUG_PRINTLN(F("EV_JOINED"));
```

(continues on next page)

(continued from previous page)

```
            break;
        case EV_RFU1:
            SERIALDEBUG_PRINTLN(F("EV_RFU1"));
            break;
        case EV_JOIN_FAILED:
            SERIALDEBUG_PRINTLN(F("EV_JOIN_FAILED"));
            break;
        case EV_REJOIN_FAILED:
            SERIALDEBUG_PRINTLN(F("EV_REJOIN_FAILED"));
            break;
        case EV_TXCOMPLETE:
            digitalWrite(LED_BUILTIN, LOW);     // turn the LED off by making the voltage LOW
            SERIALDEBUG_PRINTLN(F("EV_TXCOMPLETE (includes waiting for RX windows)"));
            if (LMIC.txrxFlags & TXRX_ACK)
              SERIALDEBUG_PRINTLN(F("Received ack"));
            if (LMIC.dataLen) {
              SERIALDEBUG_PRINT(F("Received "));
              SERIALDEBUG_PRINT(LMIC.dataLen);
              SERIALDEBUG_PRINTLN(F(" bytes of payload"));
            }
            // Schedule next transmission
            os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(TX_INTERVAL), do_send);

            SERIALDEBUG_PRINTLN("going to sleep now ... ");
            // lmic library sleeps automatically after transmission has been completed
            for(int i= 0; i < SLEEP_TIME / 16; i++) {
              Watchdog.sleep(16000); // maximum seems to be 16 seconds
              SERIALDEBUG_PRINT('.');
            }
            if (SLEEP_TIME % 16) {
              Watchdog.sleep((SLEEP_TIME % 16)*1000);
              SERIALDEBUG_PRINT('*');
            }
            SERIALDEBUG_PRINTLN("... woke up again");

            break;
        case EV_LOST_TSYNC:
            SERIALDEBUG_PRINTLN(F("EV_LOST_TSYNC"));
            break;
        case EV_RESET:
            SERIALDEBUG_PRINTLN(F("EV_RESET"));
            break;
        case EV_RXCOMPLETE:
            // data received in ping slot
            SERIALDEBUG_PRINTLN(F("EV_RXCOMPLETE"));
            break;
        case EV_LINK_DEAD:
            SERIALDEBUG_PRINTLN(F("EV_LINK_DEAD"));
            break;
        case EV_LINK_ALIVE:
            SERIALDEBUG_PRINTLN(F("EV_LINK_ALIVE"));
            break;
         default:
            SERIALDEBUG_PRINTLN(F("Unknown event"));
            break;
```

(continues on next page)

(continued from previous page)

```
    }
}

void do_send(osjob_t* j){
    // Check if there is not a current TX/RX job running
    if (LMIC.opmode & OP_TXRXPEND) {
        SERIALDEBUG_PRINTLN(F("OP_TXRXPEND, not sending"));
    } else {
        // Prepare upstream data transmission at the next possible time.

        float temperature, humidity, measuredvbat;
        int16_t int16_temperature, int16_humidity, int16_vbat;

        // Start a measurement to update the sensor's internal temperature & humidity reading
        SERIALDEBUG_PRINTLN("Start measurement...");
        temperature = dht.readTemperature();
        // delay(2000);
        Watchdog.sleep(2000);
        // Now read the recently measured temperature (2 secs ago) as Celsius (the default)
        temperature = dht.readTemperature();
        // Read the recently measured humidity (2 secs ago)
        humidity = dht.readHumidity();
        SERIALDEBUG_PRINTLN("... finished!");

        // Check if any reads failed and exit early (to try again).
        if (isnan(humidity) || isnan(temperature)) {
            SERIALDEBUG_PRINTLN("Failed to read from DHT sensor!");
            for (int i=0; i<5; i++) {
              digitalWrite(LED_BUILTIN, HIGH);    // turn the LED on by making the voltage HIGH
              delay(150);
              digitalWrite(LED_BUILTIN, LOW);    // turn the LED on by making the voltage HIGH
              delay(150);
            }
            // ok, then wait for another period and try it again
            os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(TX_INTERVAL), do_send);
        } else {
            SERIALDEBUG_PRINT("Humidity: ");
            SERIALDEBUG_PRINT(humidity);
            SERIALDEBUG_PRINT(" %\t");
            SERIALDEBUG_PRINT("Temperature: ");
            SERIALDEBUG_PRINT(temperature);
            SERIALDEBUG_PRINT(" *C ");

            int16_temperature = 100*temperature;
            int16_humidity = 100*humidity;
            mydata[0] = (byte) (int16_temperature >> 8);
            mydata[1] = (byte) (int16_temperature & 0x00FF);
            mydata[2] = (byte) (int16_humidity >> 8);
            mydata[3] = (byte) (int16_humidity & 0x00FF);

            measuredvbat = analogRead(VBATPIN);
            measuredvbat *= 2;    // we divided by 2, so multiply back
```

(continues on next page)

(continued from previous page)

```
            measuredvbat *= 3.3;   // Multiply by 3.3V, our reference voltage
            measuredvbat /= 1023;  // convert to voltage
            int16_vbat = round(measuredvbat * 100);
            mydata[4] = (byte) (int16_vbat >> 8);
            mydata[5] = (byte) (int16_vbat & 0x00FF);
            SERIALDEBUG_PRINT(" %\t");
            SERIALDEBUG_PRINT("Battery Voltage: ");
            SERIALDEBUG_PRINTLN(measuredvbat);

//            LMIC_setTxData2(1, mydata, sizeof(mydata)-1, 0);

            // send the 6 bytes payload to LoRaWAN port 7
            LMIC_setTxData2(7, mydata, 6, 0);
            SERIALDEBUG_PRINTLN(F("Packet queued"));
            digitalWrite(LED_BUILTIN, HIGH);    // turn the LED on by making the
↪voltage HIGH
        }

        // LMIC_setTxData2(1, mydata, sizeof(mydata)-1, 0);
        // Serial.println(F("Packet queued"));
    }
    // Next TX is scheduled after TX_COMPLETE event.
}

void setup() {
    delay(5000);

    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW

#ifdef SERIALDEBUG
    Serial.begin(115200);
    // while (!Serial);
#endif

    dht.begin();

    SERIALDEBUG_PRINTLN(F("Starting"));

    #ifdef VCC_ENABLE
    // For Pinoccio Scout boards
    pinMode(VCC_ENABLE, OUTPUT);
    digitalWrite(VCC_ENABLE, HIGH);
    delay(1000);
    #endif

    // LMIC init
    os_init();
    // Reset the MAC state. Session and pending data transfers will be discarded.
    LMIC_reset();
    LMIC_setClockError(MAX_CLOCK_ERROR * 1 / 100);

    // Set static session parameters. Instead of dynamically establishing a session
    // by joining the network, precomputed session parameters are be provided.
    #ifdef PROGMEM
    // On AVR, these values are stored in flash and only copied to RAM
    // once. Copy them to a temporary buffer here, LMIC_setSession will
```

(continues on next page)

(continued from previous page)

```
    // copy them into a buffer of its own again.
    uint8_t appskey[sizeof(APPSKEY)];
    uint8_t nwkskey[sizeof(NWKSKEY)];
    memcpy_P(appskey, APPSKEY, sizeof(APPSKEY));
    memcpy_P(nwkskey, NWKSKEY, sizeof(NWKSKEY));
    LMIC_setSession (0x1, DEVADDR, nwkskey, appskey);
    #else
    // If not running an AVR with PROGMEM, just use the arrays directly
    LMIC_setSession (0x1, DEVADDR, NWKSKEY, APPSKEY);
    #endif

    #if defined(CFG_eu868)
    // Set up the channels used by the Things Network, which corresponds
    // to the defaults of most gateways. Without this, only three base
    // channels from the LoRaWAN specification are used, which certainly
    // works, so it is good for debugging, but can overload those
    // frequencies, so be sure to configure the full frequency range of
    // your network here (unless your network autoconfigures them).
    // Setting up channels should happen after LMIC_setSession, as that
    // configures the minimal channel set.
    // NA-US channels 0-71 are configured automatically
    LMIC_setupChannel(0, 868100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(1, 868300000, DR_RANGE_MAP(DR_SF12, DR_SF7B), BAND_CENTI);
↪// g-band
    LMIC_setupChannel(2, 868500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(3, 867100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(4, 867300000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(5, 867500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(6, 867700000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(7, 867900000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(8, 868800000, DR_RANGE_MAP(DR_FSK,  DR_FSK),  BAND_MILLI);
↪// g2-band
    // TTN defines an additional channel at 869.525Mhz using SF9 for class B
    // devices' ping slots. LMIC does not have an easy way to define set this
    // frequency and support for class B is spotty and untested, so this
    // frequency is not configured here.
    #elif defined(CFG_us915)
    // NA-US channels 0-71 are configured automatically
    // but only one group of 8 should (a subband) should be active
    // TTN recommends the second sub band, 1 in a zero based count.
    // https://github.com/TheThingsNetwork/gateway-conf/blob/master/US-global_conf.
↪json
    LMIC_selectSubBand(1);
    #endif

    // Disable link check validation
    LMIC_setLinkCheckMode(0);

    // TTN uses SF9 for its RX2 window.
    LMIC.dn2Dr = DR_SF9;
```

(continues on next page)

(continued from previous page)

```
    // Set data rate and transmit power for uplink (note: txpow seems to be ignored by the library)
    LMIC_setDrTxpow(DR_SF7,14);

    // Start job
    do_send(&sendjob);
}

void loop() {
    os_runloop_once();
}
```

Listing 23: TTN payload decoder for Adafruit M0 LoRa sensor node

```
function Decoder (bytes, port) {
  var result = {};
  var transformers = {};

  if (port==7) {
     transformers = {
      'temperature': function transform (bytes) {
          value=bytes[0]*256 + bytes[1];
          if (value>=32768) value=value-65536;
          return value/100.0;
        },
      'humidity': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/100.0;
        },
      'vbattery': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/100.0;
        },
    }

    result['temperature'] = {
      value: transformers['temperature'](bytes.slice(0, 2)),
      uom: 'Celsius',
    }

    result['humidity'] = {
      value: transformers['humidity'](bytes.slice(2, 4)),
      uom: 'Percent',
    }

    result['vbattery'] = {
      value: transformers['vbattery'](bytes.slice(4, 6)),
      uom: 'Volt',
    }
  }

  return result;
}
```

2.6.5 References

- Adafruit Feather M0 LoRa microntroller
- Adafruit Feather M0 LoRa tutorial
- IBM LMIC (LoraMAC-in-C) library for Arduino
- Adafruit feather m0 lora 900 end-to-end instructions - End Devices (Nodes) - The Things Network
- Getting Started with AdaFruit Feather M0 LoRa - Telenor Start IoT
- GitHub - mcci-catena/arduino-lorawan: User-friendly library for using Feather M0 LoRa with The Things Network and LoRaWAN™
- GitHub - marcuscbehrens/loralife: source code associated with https://www.meetup.com/Internet-of-Things-IoT-LoRaWan-Infrastruktur-4-RheinNeckar/
- Workshop — LoRaTAS
- mikenz/Feather_M0_LoRa: Example Arduino code of using an Adafruit Feather M0 LoRa module to send sensor data
- TTN Ulm - LoRaWAN und LoRa in Ulm | Verkehrszählung mit LoRaWAN und TTN

On battery saving / using the deep sleep mode

- Full Arduino Mini LoraWAN and 1.3uA Sleep Mode - End Devices (Nodes) - The Things Network
- Adding Method to Adjust hal_ticks Upon Waking Up from Sleep · Issue #109 · matthijskooijman/arduino-lmic
- minilora-test/minilora-test.ino at cbe686826bd84fac8381de47b5f5b02dd47c2ca0 · tkerby/minilora-test

2.7 Dragino LoRa Arduino Shield

This tutorial is made to showcase the use of Dragino LoRa Arduino board to create a LoRaWAN enabled sensor node. In the following example, a temperature and humidity sensor was used with the Dragino LoRa board.

2.7.1 Hardware

Microcontroller

The employed microcontroller board is an Arduino Uno R3 variant (i.e. it is a cheap clone of the Arduino Uno R3). It is operated by the 8bit ATmega328 microcontroller running at 16MHz. It has 32 KB flash memory (to store the program code), 1 KB EEPROM (to store configuration data), and 2 KB of RAM (to store variables, status information, and buffers). The operating voltage of the board is 5V (this is important when attaching sensors and other peripherals; they also must operate on 5V). The board offers 20 general purpose digital input/output pins (20 GPIOs) of which 6 can be used as analog input pins (with 10bit analog digital converters (ADC)) and 6 as PWM outputs, one serial port (programmable Universal Asynchronous Receiver and Transmitter, UART), one I2C port, one SPI port, one USB port (which is attached to a USB/Serial converter that is connected to the hardware serial port). Arduino Uno R3 compatible boards are available in German shops from around 5 € to 10 €. The original Arduino Uno R3 board costs around 22 €.

The Dragino LoRa/GPS Shield runs on 5V and is directly attached to the connectors of the Arduino Uno R3 microcontroller board. It comes with a built-in LoRa transmitter and receiver chip SX1276 from the company Semtech that is dedicated to the 868 MHz frequency band. The SX1276 module is connected via SPI interface to the microcontroller. For that purpose, Lora CLK, Lora D0, and Lora DI must be jumpered to SCK, MISO, and MOSI respectively (on the left side of the Dragino shield when looking on the top side of the shield with the Antenna connectors showing to the

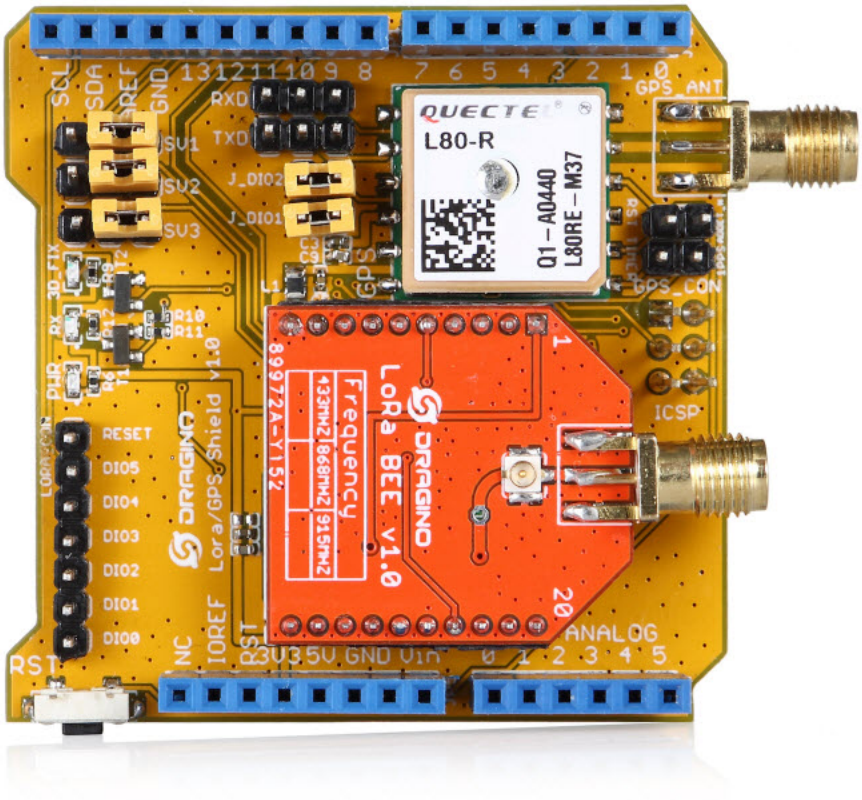


Fig. 21: LoRa/GPS Shield from Dragino. LoRa/GPS Shield Wiki with explanations, datasheets, and examples.

right). Lora DIO1 and Lora DIO2 must be jumpered to Arduino Digital Pin 6 and Pin 7 respectively. Since the module only implements the LoRa physical layer, the LoRaWAN protocol stack must be implemented in software on the microcontroller. We are using the Arduino library LMIC for that purpose (see below). The implemented LoRaWAN functionality is compatible with LoRaWAN Class A/C.

The board also contains a Quectel L80 GPS module (based on the MTK MT3339 GPS receiver) with a built-in antenna. According to the Dragino Wiki "this GPS module can calculate and predict orbits automatically using the ephemeris data (up to 3 days) stored in internal flash memory, so the shield can fix position quickly even at indoor signal levels with low power consumption". The GPS module has a serial UART interface that can be connected in different ways to the Arduino microcontroller. The default data transmission rate is 9600 baud, the default position reporting rate is 1s (1 Hz). The module is capable to report up to 10 positions per second (10 Hz). Supported protocols are NMEA 0183 and MediaTek PMTK. Note that the ATmega328 microcontroller has only one hardware serial UART interface and this is already connected via a USB/Serial converter to the USB port of the Arduino board. In order to attach the serial interface of the GPS module to the microcontroller two general purpose IO lines (GPIOs) are being used and the serial protocol is implemented in software. The GPS_RXD pin on the Dragino Shield must be connected to Arduino Digital Pin 4 and the GPS_TXD pin to Digital Pin 3 using two wires. No jumpers must be present for GPS_RXD and GPS_TXD (besides the two wires mentioned above to Digital Pins 4 and 3). The Dragino LoRa/GPS Shield is available in German shops for around 34 € to 40 €.

Fig. 22: Solar Charger Shield V2.2 from Seeedstudio.

Since the Arduino Uno R3 board normally has to be powered externally via the USB port or the power connector, we have added the Solar Charger Shield V2.2 from the company Seeedstudio. This shield is directly attached to the connectors of the Arduino Uno R3 microcontroller board and sits in-between the Arduino board (bottom) and the LoRa/GPS Shield (top). A lithium polymer LiPo battery with 3.7V can be attached to the shield. The 3.7V of the battery is transformed to 5V as required by the Arduino microcontroller board. The battery is automatically recharged when the Arduino board is powered externally (over USB or the power connector). Also a photovoltaic panel with 4.8-6V can be attached to the shield to recharge the battery. The Solar Charger Shield V2.2 can report the current battery voltage level. For that purpose we had to solder a bridge on the shield at the connector marked as 'R7'. Over

a voltage divider the battery anode is connected to Analog Pin A0 and can be queried using the built-in analog/digital converter. The Solar Charger Shield V2.2 is available in German shops for around 12 € to 18 €.

Sensor

We have attached a DHT22 sensor to the microcontroller board, which measures air temperature and humidity. The minimal time interval between two measurements is 2 seconds. All data transfers between the DHT22 and the microcontroller use a single digital line. The sensor data pin is attached to a GPIO pin (here: Digital Pin 5) of the microcontroller. In addition, a so-called pull-up resistor of 4.7k to 10k Ohm must be connected between the data line and VCC (+3.3V). The DHT22 datasheet provides more technical details about the DHT22 Sensor. A tutorial on how to use the DHT22 sensor with Arduino microcontrollers is provided here. The sensor is available in German shops for around 4 € to 10 €.

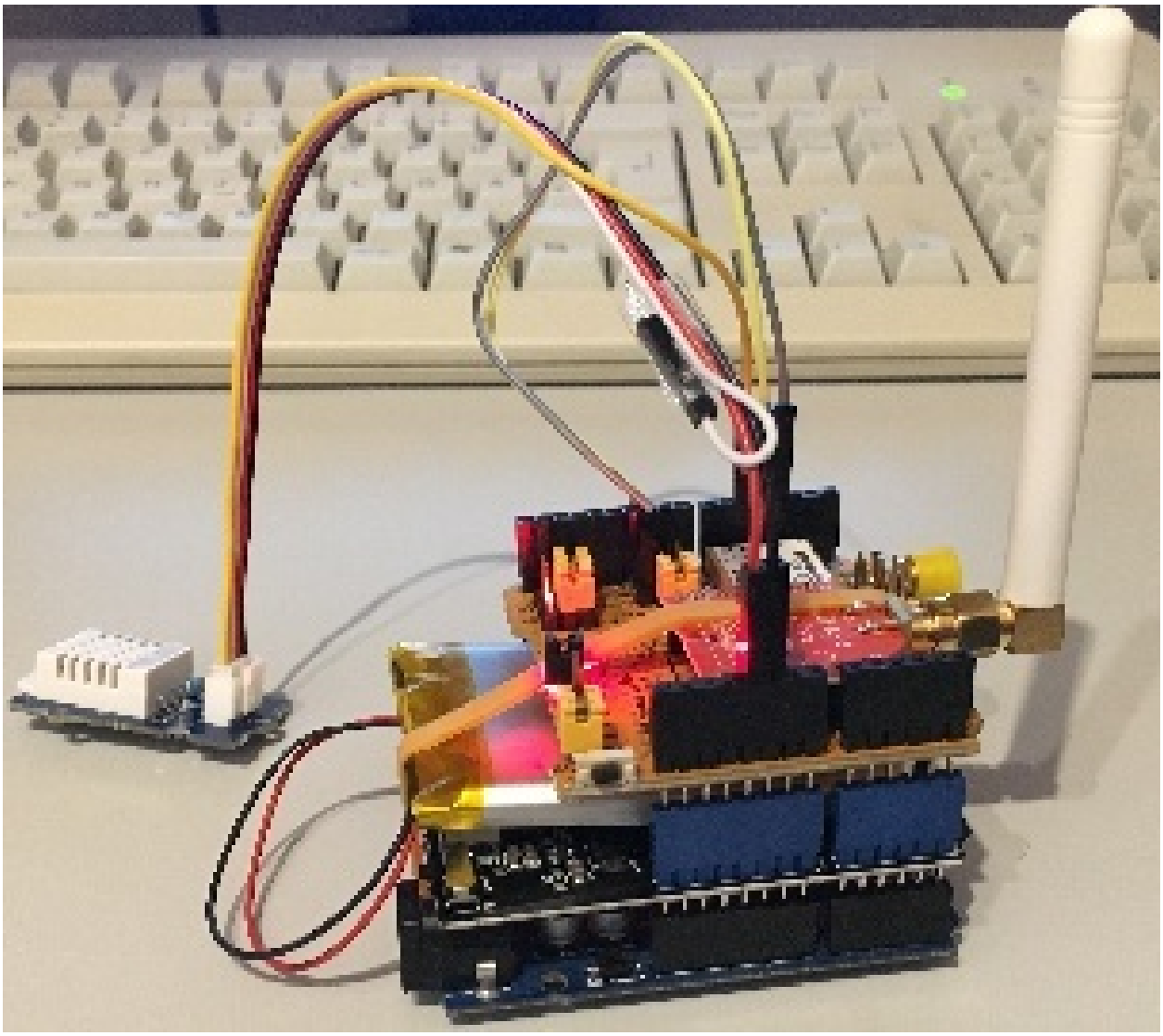


Fig. 23: The Arduino Uno R3 (bottom) with attached Solar Charger Shield and a 2000 mAh lithium polymer LiPo battery (middle), the Dragino LoRa/GPS Shield with attached antenna (top), and an attached DHT22 temperature / humidity sensor (white box on the left).

2.7.2 Software

The sensor node has been programmed using the Arduino IDE. Please note, that in the Arduino framework a program is called a 'Sketch'.

After the sketch has successfully established a connection to The Things Network it reports the air temperature, humidity, and the voltage of a (possibly) attached LiPo battery every 5 minutes. All three values are being encoded in two byte integer values each (in most significant byte order) and then sent as a 6 bytes data packet to the respective TTN application using LoRaWAN port 7. Please note, that LoRaWAN messages can be addressed to ports 1-255 (port 0 is reserved); these ports are similar to port numbers 0-65535 when using the Internet TCP/IP protocol. Voltage and humidity values are always greater or equal to 0, but the temperature value can also become negative. Negative values are represented as a two's complement; this must be considered in the Payload Decoding Function used in The Things Network (see *this section*).

The next eight bytes contain two 32 bit integer values (MSB) for latitude and longitude . In order to a) provide enough precision and b) avoid negative values, the original angles (given as decimal fractions) are first added with an offset (90.0 degrees for the latitude and 180.0 degrees for the longitude) and then multiplied by 1,000,000. These transformations have to be reverted in the Payload Decoding Function. The next two bytes represent a 16 bit integer value for the altitude (MSB). The next byte contains the current number of satellites seen by the GPS receiver. Note that only when this number is greater or equal to 4 the provided GPS position is a current one. Finally, the last two bytes contain a 16 bit integer value (MSB) for the battery voltage in centivolts (this value will be divided by 100 in the Payload Decoding Function to provide volts). The entire data packet is sent to the respective TTN application using LoRaWAN port 9. Please note, that LoRaWAN messages can be addressed to ports 1-255 (port 0 is reserved); these ports are similar to port numbers 0-65535 when using the Internet TCP/IP protocol.

Currently we are not making use of the sleep mode, because we have to find out how to deal with the GPS receiver in conjunction with deep sleep mode. This means that the board is constantly drawing a significant amount of power reducing battery life considerably. Using the current sketch the sensor node can operate roughly 6 hours on battery power before it has to be recharged. Besides software improvements there are also other possibilities to reduce power consumption: the Arduino board and the Dragino LoRa/GPS Shield have power LEDs which are constantly lit during operation. Furthermore, the Dragino LoRa/GPS Shield has an indicator LED that blinks when the GPS module is successfully receiving position fixes. These LEDs could be desoldered to reduce the energy consumption of the sensor node.

The employed SX1276 LoRa module on the Dragino LoRa/GPS shield does not provide built-in support of the LoRaWAN protocol. Thus, it has to be implemented on the ATmega328 microcontroller. We use the IBM LMIC (LoraMAC-in-C) library for Arduino, which can be downloaded from this repository. Since the ATmega328 microcontroller only has 32 KB of flash memory and the LMIC library is taking most of it, there is only very limited code space left for the application dealing with the sensors (about 2 KB). Nevertheless, this is sufficient to query some sensors like in our example the DHT22 and to decode the GPS data. The source code is given in the following section: *Arduino Sketch for Dragino LoRa sensor node*

2.7.3 Services

The services used for this sensor-node are:

- *TheThingsNetwork* service for LoRaWAN network service.
- *TheThingNetwork- OGC SensorWeb Integration*

Registration of the sensor node with The Things Network (TTN)

The LoRaWAN protocol makes use of a number of different identifiers, addresses, keys, etc. These are required to unambiguously identify devices, applications, as well as to encrypt and decrypt messages. The names and meanings are nicely explained on a dedicated TTN web page.

The sketch given above connects the sensor node with The Things Network (TTN) using the Activation-by-Personalisation (ABP) mode. In this mode, the required keys for data encryption and session management are created manually using the TTN console window and must be pasted into the source code of the sketch below. In order to get this running, you will need to create a new device in the TTN console window. This assumes that you already have a TTN user account (which needs to be created otherwise). In the settings menu of the newly created device the ABP mode must be selected and the settings must be saved. Then copy the DevAddr, the NwkSKey, and in the AppSKey from the TTN console web page of the newly registered device and paste them into the proper places in the sketch above. Please make sure that you choose for each of the three keys the correct byte ordering (MSB for all three keys). A detailed explanation of these steps is given here. Then the sketch can be compiled and uploaded to the Arduino Uno R3 microcontroller.

Important hint: everytime the sensor node is reset or being started again, make sure to reset the frame counter of the registered sensor in the TTN console web page of the registered device. The reason is that in LoRaWAN all transmitted data packets have a frame counter, which is incremented after each data frame being sent. This way a LoRaWAN application can avoid receiving and using the same packet again (replay attack). When TTN receives a data packet, it checks if the frame number is higher than the last one received before. If not, the received packet is considered to be old or a replay attack and is discarded. When the sensor node is reset or being started again, its frame counter is also reset to 0, hence, the TTN application assumes that all new packages are old, because their frame counter is lower than the last frame received (before the reset). A manual frame counter reset is only necessary when registering the node using ABP mode. In OTAA mode the frame counter is automatically reset in the sensor node and the TTN network server.

TTN Payload Decoding

Everytime a data packet is received by a TTN application a dedicated Javascript function is being called (*TTN payload decoder for Dragino LoRa sensor node*). This function can be used to decode the received byte string and to create proper Javascript objects or values that can directly be read by humans when looking at the incoming data packet. This is also useful to format the data in a specific way that can then be forwarded to an external application (e.g. a sensor data platform like MyDevices or Thingspeak). Such a forwarding can be configured in the TTN console in the "Integrations" tab. The Payload Decoder Function given below checks if a packet was received on LoRaWAN port 9 and then assumes that it consists of the 17 bytes encoded as described above. It creates the seven Javascript objects 'temperature', 'humidity', 'lat', 'lon', 'altitude', 'sat', and 'vbattery'. Each object has two fields: 'value' holds the value and 'uom' gives the unit of measure. The source code can simply be copied and pasted into the 'decoder' tab in the TTN console after having selected the application. Choose the option 'Custom' in the 'Payload Format' field. Note that when you also want to handle other sensor nodes sending packets on different LoRaWAN ports, then the Payload Decoder Function can be extended after the end of the if (port==9) {...} statement by adding else if (port==7) {...} else if (port==8) {...} etc.

The Things Network - OGC SensorWeb Integration

The presented Payload Decoder Function works also with the TTN-OGC SWE Integration for the 52° North Sensor Observation Service (SOS). This software component can be downloaded from this repository. It connects a TTN application with a running transactional Sensor Observation Service 2.0.0 (SOS). Data packets received from TTN are imported into the SOS. The SOS persistently stores sensor data from an arbitrary number of sensor nodes and can be queried for the most recent as well as for historic sensor data readings. The 52° North SOS comes with its own REST API and a nice web client allowing to browse the stored sensor data in a convenient way.

We are running an instance of the 52° North SOS and the TTN-OGC SWE Integration. The web client for this LoRaWAN sensor node can be accessed on this page. Here is a screenshot showing the webclient:

2.7.4 Code files

Fig. 24: Web client for data visualization

Listing 24: Arduino Sketch for Dragino LoRa sensor node

```
/*******************************************************************************
 * Arduino Sketch for a LoRaWAN sensor node that is registered with
 * 'The Things Network' (TTN) www.thethingsnetwork.org
 *
 * Author:  Thomas H. Kolbe, thomas.kolbe@tum.de
 * Version: 0.4
 * Last update: 2018-11-28
 *
 * The sensor node is based on the Arduino Uno3 microcontroller board
 * and the Dragino Lora Shield with GPS receiver. Also a Seeedstudio
 * Solar Charger Shield V2.2 is connected to provide a battery power
 * supply with the possibility to use a small PV panel for recharging.
 * See https://wiki.dragino.com/index.php?title=Lora/GPS_Shield
 * and http://wiki.seeedstudio.com/Solar_Charger_Shield_V2.2/
 *
 * The sensor node uses a DHT22 sensor measuring air temperature and humidity.
 * The GPS receiver of the Dragino Lora Shield is used to locate the node.
 * The voltage of an attached LiPo battery is monitored and sent as an
 * additional observation.
 *
 * All three values are encoded as 2 byte integer values each.
 * Hence, the total message payload is 6 bytes. Before the values are converted
 * to integers they are multiplied by 100 to preserve 2 digits after the decimal
 * point. Thus, the received values must be divided by 100 to obtain the measured
 * values. The payload is sent every 60s to LoRaWAN port 9. The following
 * Javascript function can be used as a payload decoding function in TTN:
```

(continues on next page)

(continued from previous page)

```
 *
 * function Decoder(bytes, port) {
 *   // Decode an uplink message from a buffer
 *   // (array) of bytes to an object of fields.
 *   if (port==7) {
 *     var decoded = {
 *       "temperature": (bytes[0] << 8 | bytes[1]) / 100.0,
 *       "humidity": (bytes[2] << 8 | bytes[3]) / 100.0,
 *       "vbattery": (bytes[4] << 8 | bytes[5]) / 100.0
 *     };
 *   } else {
 *     var decoded = null;
 *   }
 *   return decoded;
 * }
 *
 * In between two data transmissions the microcontroller board can go
 * into sleep mode to reduce energy consumption for extended operation
 * time when running on battery. Usage of the sleep mode must be
 * explicitly configured below.
 *
 * Important hint: everytime the sensor node is reset or being started again,
 * make sure to reset the frame counter of the registered sensor in the
 * TTN console at https://console.thethingsnetwork.org. The reason is that
 * in LoRaWAN all transmitted packets have a frame counter, which is
 * incremented after each data frame being sent. This way a LoRaWAN application
 * can avoid receiving and using the same packet again (replay attack). When
 * TTN receives a data packet, it checks if the frame number is higher than
 * the last one received before. If not, the received packet is considered
 * to be old or a replay attack and is discarded. When the sensor node is
 * reset or being started again, its frame counter is also reset to 0, hence,
 * the TTN application assumes that all new packages are old, because their
 * frame counter is lower than the last frame received (before the reset).
 *
 * Note, that the DHT22 data pin must be connected to Digital Pin 5 of the
 * Arduino board. A resistor of 4.7k - 10k Ohm must be connected to
 * the data pin and VCC (+5V). The GPS_RXD pin on the Dragiono Shield must
 * be connected to Arduino Digital Pin 4 and the GPS_TXD pin to Digital Pin 3.
 * Lora CLK, Lora D0, and Lora DI must be jumpered to SCK, MISO, and MOSI
 * respectively (on the left side of the Dragino shield when looking on the
 * top side of the shield with the Antenna connectors shwoing to the right).
 * Lora DIO1 and Lora DIO2 must be jumpered to Arduino Digital Pin 6 and
 * Pin 7 respectively. No jumpers must be present for GPS_RXD and GPS_TXD
 * (besides the two wires mentioned above to Digital Pins 4 and 3).
 *
 * The code is based on the Open Source library LMIC implementing the LoRaWAN
 * protocol stack on top of a given LoRa transceiver module (here: RFM95 from
 * HopeRF, which uses the Semtech SX1276 LoRa chip). The library is originally
 * being developed by IBM and has been ported to the Arduino platform. See
 * notes below from the original developers.
 *
 *******************************************************************************
 * Copyright (c) 2015 Thomas Telkamp and Matthijs Kooijman
 *
 * Permission is hereby granted, free of charge, to anyone
 * obtaining a copy of this document and accompanying files,
 * to do whatever they want with them without any restriction,
```

(continues on next page)

(continued from previous page)

```
 * including, but not limited to, copying, modification and redistribution.
 * NO WARRANTY OF ANY KIND IS PROVIDED.
 *
 * This uses ABP (Activation-by-personalisation), where a DevAddr and
 * Session keys are preconfigured (unlike OTAA, where a DevEUI and
 * application key is configured, while the DevAddr and session keys are
 * assigned/generated in the over-the-air-activation procedure).
 *
 * Note: LoRaWAN per sub-band duty-cycle limitation is enforced (1% in
 * g1, 0.1% in g2), but not the TTN fair usage policy (which is probably
 * violated by this sketch when left running for longer)!
 *
 * To use this sketch, first register your application and device with
 * the things network, to set or generate a DevAddr, NwkSKey and
 * AppSKey. Each device should have their own unique values for these
 * fields.
 *
 * Do not forget to define the radio type correctly in config.h.
 *
 *******************************************************************************/

// If the following line is uncommented, messages are being printed out to the
// serial connection for debugging purposes. When using the Arduino Integrated
// Development Environment (Arduino IDE), these messages are displayed in the
// Serial Monitor selecting the proper port and a baudrate of 115200.

// #define SERIALDEBUG

#ifdef SERIALDEBUG
  #define SERIALDEBUG_PRINT(...) Serial.print(__VA_ARGS__)
  #define SERIALDEBUG_PRINTLN(...) Serial.println(__VA_ARGS__)
#else
  #define SERIALDEBUG_PRINT(...)
  #define SERIALDEBUG_PRINTLN(...)
#endif

// If the following line is uncommented, the sensor node goes into sleep mode
// in between two data transmissions. Also the 2secs time between the
// initialization of the DHT22 sensor and the reading of the observations
// is spent in sleep mode.
// Note, that on the Adafruit Feather 32u4 LoRa board the Serial connection
// gets lost as soon as the board goes into sleep mode, and it will not be
// established again. Thus, the definition of SERIALDEBUG should be commented
// out above when using sleep mode with this board.

// #define SLEEPMODE

#ifdef SLEEPMODE
  #include <Adafruit_SleepyDog.h>
#endif

#include <lmic.h>
#include <hal/hal.h>
#include <SPI.h>

#include <DHT.h>
#define DHTPIN            5         // Arduino Digital Pin which is connected to the␣
↪DHT sensor for Arduino.
```

(continues on next page)

(continued from previous page)

```
#define DHTTYPE           DHT22     // DHT 22 (AM2302)
DHT dht(DHTPIN, DHTTYPE);           // create the sensor object

#include <TinyGPS.h>
TinyGPS gps;
bool newGPSdata = false;

#include <SoftwareSerial.h>
SoftwareSerial SWSerial(3, 4);

#define VBATPIN A0                  // battery voltage is measured from Analog Input A0 for Seeed Solar Shield V2.2

// The following three constants (NwkSKey, AppSKey, DevAddr) must be changed
// for every new sensor node. We are using the LoRaWAN ABP mode (activation by
// personalisation) which means that each sensor node must be manually registered
// in the TTN console at https://console.thethingsnetwork.org before it can be
// started. In the TTN console create a new device and choose ABP mode in the
// settings of the newly created device. Then, let TTN generate the NwkSKey and
// and the AppSKey and copy them (together with the device address) from the webpage
// and paste them below.

// LoRaWAN NwkSKey, network session key
// This should be in big-endian (aka msb).
static const PROGMEM u1_t NWKSKEY[16] = {NETWORK_SESSION_KEY_HERE_IN_MSB_FORMAT};

// LoRaWAN AppSKey, application session key
// This should also be in big-endian (aka msb).
static const u1_t PROGMEM APPSKEY[16] = {APPLICATION_SESSION_KEY_HERE_IN_MSB_FORMAT};

// LoRaWAN end-device address (DevAddr)
// See http://thethingsnetwork.org/wiki/AddressSpace
// The library converts the address to network byte order as needed, so this should be in big-endian (aka msb) too.
static const u4_t DEVADDR = 0x260XXXXX   ; // <-- Change this address for every node!

// These callbacks are only used in over-the-air activation, so they are
// left empty here (we cannot leave them out completely unless
// DISABLE_JOIN is set in config.h, otherwise the linker will complain).
void os_getArtEui (u1_t* buf) { }
void os_getDevEui (u1_t* buf) { }
void os_getDevKey (u1_t* buf) { }

// The following array of bytes is a placeholder to contain the message payload
// which is transmitted to the LoRaWAN gateway. We are currently only using 6 bytes.
// Please make sure to extend the size of the array, if more sensors should be
// attached to the sensor node and the message payload becomes larger than 10 bytes.
static uint8_t mydata[17] = {0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0xE, 0xF, 0x10};

static osjob_t sendjob;

// Schedule transmission every TX_INTERVAL seconds (might become longer due to duty
// cycle limitations). The total interval time is 2secs for the measurement
// plus 3secs for the LoRaWAN packet transmission plus TX_INTERVAL_AFTER_SLEEP seconds
// plus SLEEP_TIME seconds (microcontroller in sleep mode)
const unsigned TX_INTERVAL = 300;       // overall cycle time (send one set of observations every 5 mins)
```

(continues on next page)

(continued from previous page)

```
// const unsigned TX_INTERVAL = 30;       // overall cycle time (send one set of observations every 30 secs)
const unsigned TX_TIME = 3;             // rough estimate of transmission time of a single packet
const unsigned MEASURE_TIME = 2;        // seconds measuring time
const unsigned SLEEP_TIME = TX_INTERVAL - TX_TIME - MEASURE_TIME;
const unsigned WAIT_TIME = TX_INTERVAL - TX_TIME - MEASURE_TIME;

// Pin mapping of the LoRa transceiver. Please make sure that DIO1 is connected
// to Arduino Digital Pin 6 using an external wire. DIO2 is left unconnected
// (it is only required, if FSK modulation instead of LoRa would be used).
const lmic_pinmap lmic_pins = {
    .nss = 10,
    .rxtx = LMIC_UNUSED_PIN,
    .rst = 9,
    .dio = {2, 6, 7},
};

void onEvent (ev_t ev) {
    SERIALDEBUG_PRINT(os_getTime());
    SERIALDEBUG_PRINT(": ");
    switch(ev) {
        case EV_SCAN_TIMEOUT:
            SERIALDEBUG_PRINTLN(F("EV_SCAN_TIMEOUT"));
            break;
        case EV_BEACON_FOUND:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_FOUND"));
            break;
        case EV_BEACON_MISSED:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_MISSED"));
            break;
        case EV_BEACON_TRACKED:
            SERIALDEBUG_PRINTLN(F("EV_BEACON_TRACKED"));
            break;
        case EV_JOINING:
            SERIALDEBUG_PRINTLN(F("EV_JOINING"));
            break;
        case EV_JOINED:
            SERIALDEBUG_PRINTLN(F("EV_JOINED"));
            break;
        case EV_RFU1:
            SERIALDEBUG_PRINTLN(F("EV_RFU1"));
            break;
        case EV_JOIN_FAILED:
            SERIALDEBUG_PRINTLN(F("EV_JOIN_FAILED"));
            break;
        case EV_REJOIN_FAILED:
            SERIALDEBUG_PRINTLN(F("EV_REJOIN_FAILED"));
            break;
        case EV_TXCOMPLETE:
            digitalWrite(LED_BUILTIN, LOW);     // turn the LED off by making the voltage LOW
            SERIALDEBUG_PRINTLN(F("EV_TXCOMPLETE (includes waiting for RX windows)"));
            if (LMIC.txrxFlags & TXRX_ACK)
              SERIALDEBUG_PRINTLN(F("Received ack"));
            if (LMIC.dataLen) {
#ifdef SERIALDEBUG
```

(continues on next page)

(continued from previous page)

```
              SERIALDEBUG_PRINT(F("Received "));
              SERIALDEBUG_PRINT(LMIC.dataLen);
              SERIALDEBUG_PRINT(F(" bytes of payload: 0x"));
              for (int i=0; i<LMIC.dataLen; i++) {
                if (LMIC.frame[LMIC.dataBeg + i] < 0x10) {
                  SERIALDEBUG_PRINT(F("0"));
                }
                SERIALDEBUG_PRINT(LMIC.frame[LMIC.dataBeg + i], HEX);
              }
              SERIALDEBUG_PRINTLN();
#endif
              // add your code to handle a received downlink data packet here
            }

#ifdef SLEEPMODE
            // Schedule next transmission in 1ms second after the board returns from sleep mode
            os_setTimedCallback(&sendjob, os_getTime()+ms2osticks(1), do_send);

            SERIALDEBUG_PRINTLN("going to sleep now ... ");
            // lmic library sleeps automatically after transmission has been completed
            for(int i= 0; i < SLEEP_TIME / 8; i++) {
              Watchdog.sleep(8000); // maximum seems to be 8 seconds
              SERIALDEBUG_PRINT('.');
            }
            if (SLEEP_TIME % 8) {
              Watchdog.sleep((SLEEP_TIME % 8)*1000);
              SERIALDEBUG_PRINT('*');
            }
            SERIALDEBUG_PRINTLN("... woke up again");

            // We need to reset the duty cycle limits within the LMIC library.
            // The reason is that in sleep mode the Arduino system timers millis and micros
            // do not get incremented. However, LMIC monitors the adherence to the
            // LoRaWAN duty cycle limitations using the system timers millis and micros.
            // Since LMIC does not know that we have slept for a long time and duty
            // cycle requirements in fact are met, we must reset the respective LMIC timers
            // in order to prevent the library to wait for some extra time (which would
            // not use sleep mode and, thus, would waste battery energy).
            LMIC.bands[BAND_MILLI].avail = os_getTime();
            LMIC.bands[BAND_CENTI].avail = os_getTime();
            LMIC.bands[BAND_DECI].avail = os_getTime();
#else
            // Schedule next transmission
            os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(WAIT_TIME), do_send);
#endif
            break;
        case EV_LOST_TSYNC:
            SERIALDEBUG_PRINTLN(F("EV_LOST_TSYNC"));
            break;
        case EV_RESET:
            SERIALDEBUG_PRINTLN(F("EV_RESET"));
```

(continues on next page)

(continued from previous page)

```
            break;
        case EV_RXCOMPLETE:
            // data received in ping slot
            SERIALDEBUG_PRINTLN(F("EV_RXCOMPLETE"));
            break;
        case EV_LINK_DEAD:
            SERIALDEBUG_PRINTLN(F("EV_LINK_DEAD"));
            break;
        case EV_LINK_ALIVE:
            SERIALDEBUG_PRINTLN(F("EV_LINK_ALIVE"));
            break;
         default:
            SERIALDEBUG_PRINTLN(F("Unknown event"));
            break;
    }
}

void do_send(osjob_t* j){
    // Check if there is not a current TX/RX job running
    if (LMIC.opmode & OP_TXRXPEND) {
        SERIALDEBUG_PRINTLN(F("OP_TXRXPEND, not sending"));
    } else {
        // Prepare upstream data transmission at the next possible time.

        float temperature, humidity, measuredvbat, lat, lon, alt;
        int16_t int16_temperature, int16_humidity, int16_vbat, int16_alt;
        int32_t int32_lat, int32_lon;
        unsigned long age;
        byte sat=0;

        // Start a measurement to update the sensor's internal temperature & humidity␣
↪reading.
        // Note, that when fetching measurements from a DHT22 sensor, the reported
        // values belong to the measurement BEFORE the current measurement.
        // Therefore, in order to get current observations, we first perform a new␣
↪measurement
        // and wait 2 secs (which is the minimum time between two sensor observations␣
↪for
        // the DHT22) and then directly retrieve the observations again.
        temperature = dht.readTemperature();
//        temperature = 23;
#ifdef SLEEPMODE
        Watchdog.sleep(2000);
#else
        delay(2000);
#endif
        // Now read the recently measured temperature (2 secs ago) as Celsius (the␣
↪default)
        temperature = dht.readTemperature();
//        temperature = 23;
        // Read the recently measured humidity (2 secs ago)
        humidity = dht.readHumidity();
//        humidity = 66;

        // Check if any reads failed and exit early (to try again).
        if (isnan(humidity) || isnan(temperature)) {
            SERIALDEBUG_PRINTLN("Failed to read from DHT sensor!");
```

(continues on next page)

(continued from previous page)

```
            // blink the LED five times to indicate that the sensor values could not be read
            for (int i=0; i<5; i++) {
              digitalWrite(LED_BUILTIN, HIGH);    // turn the LED on by making the voltage HIGH
              delay(150);
              digitalWrite(LED_BUILTIN, LOW);    // turn the LED on by making the voltage HIGH
              delay(150);
            }
            // ok, then wait for another period and try it again
            os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(TX_INTERVAL), do_send);
        } else {
            SERIALDEBUG_PRINT("Humidity: ");
            SERIALDEBUG_PRINT(humidity);
            SERIALDEBUG_PRINT(" %\t");
            SERIALDEBUG_PRINT("Temperature: ");
            SERIALDEBUG_PRINT(temperature);
            SERIALDEBUG_PRINT(" °C ");

            int16_temperature = round(100.0*temperature);
            int16_humidity = round(100.0*humidity);
            mydata[0] = (byte) (int16_temperature >> 8);
            mydata[1] = (byte) (int16_temperature & 0x00FF);
            mydata[2] = (byte) (int16_humidity >> 8);
            mydata[3] = (byte) (int16_humidity & 0x00FF);

            if (newGPSdata) {
                gps.f_get_position(&lat, &lon, &age);
                int32_lat = round(1000000.0*(lat+90.0));
                int32_lon = round(1000000.0*(lon+180.0));
                alt = gps.f_altitude();
                int16_alt = round(alt);
                sat = gps.satellites();
                mydata[4] = (byte) (int32_lat >> 24);
                mydata[5] = (byte) ((int32_lat >> 16) & 0x00FF);
                mydata[6] = (byte) ((int32_lat >> 8) & 0x0000FF);
                mydata[7] = (byte) (int32_lat & 0x000000FF);
                mydata[8] = (byte) (int32_lon >> 24);
                mydata[9] = (byte) ((int32_lon >> 16) & 0x00FF);
                mydata[10] = (byte) ((int32_lon >> 8) & 0x0000FF);
                mydata[11] = (byte) (int32_lon & 0x000000FF);
                mydata[12] = (byte) (int16_alt >> 8);
                mydata[13] = (byte) (int16_alt & 0x00FF);
                mydata[14] = sat;
            } else {
                mydata[14] = 0;
            }

#ifdef VBATPIN
            measuredvbat = analogRead(VBATPIN);
            measuredvbat *= 2.0;      // we divided by 2, so multiply back
            measuredvbat *= 5.0;      // Multiply by 5V, our reference voltage
            measuredvbat /= 1023.0;   // convert to voltage
#else
            measuredvbat = 0.0;
```

(continues on next page)

(continued from previous page)

```
#endif
            int16_vbat = round(measuredvbat * 100.0);
            mydata[15] = (byte) (int16_vbat >> 8);
            mydata[16] = (byte) (int16_vbat & 0x00FF);
            SERIALDEBUG_PRINT(" \t");
            SERIALDEBUG_PRINT("Battery Voltage: ");
            SERIALDEBUG_PRINT(measuredvbat);
            SERIALDEBUG_PRINTLN(" V");

            // Send the 17 bytes payload to LoRaWAN port 9 and do not request an acknowledgement.
            // The following call does not directly sends the data, but puts a "send job"
            // in the job queue. This job eventually is performed in the call "os_runloop_once();"
            // issued repeatedly in the "loop()" method below. After the transmission is
            // complete, the EV_TXCOMPLETE event is signaled, which is handled in the
            // event handler method "onEvent (ev_t ev)" above. In the EV_TXCOMPLETE branch
            // then a new call to the "do_send(osjob_t* j)" method is being prepared for
            // delayed execution with a waiting time of TX_INTERVAL seconds.
            LMIC_setTxData2(9, mydata, 17, 0);
            SERIALDEBUG_PRINTLN(F("Packet queued"));
            newGPSdata=false;
            digitalWrite(LED_BUILTIN, HIGH);    // turn the LED on by making the voltage HIGH

            // Next TX is scheduled after TX_COMPLETE event.
        }
    }
}

void setup() {
    delay(5000);                    // give enough time to open serial monitor (if needed)

    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW

#ifdef SERIALDEBUG
    Serial.begin(115200);
    // while (!Serial);
#endif

    dht.begin();                    // initialize DHT22 sensor

    SERIALDEBUG_PRINTLN(F("Starting"));

    #ifdef VCC_ENABLE
    // For Pinoccio Scout boards
    pinMode(VCC_ENABLE, OUTPUT);
    digitalWrite(VCC_ENABLE, HIGH);
    delay(1000);
    #endif

```

(continues on next page)

(continued from previous page)

```
    SWSerial.begin(9600);

    // LMIC init
    os_init();
    // Reset the MAC state. Session and pending data transfers will be discarded.
    LMIC_reset();
    LMIC_setClockError(MAX_CLOCK_ERROR * 1 / 100);

    // Set static session parameters. Instead of dynamically establishing a session
    // by joining the network, precomputed session parameters are be provided.
    #ifdef PROGMEM
    // On AVR, these values are stored in flash and only copied to RAM
    // once. Copy them to a temporary buffer here, LMIC_setSession will
    // copy them into a buffer of its own again.
    uint8_t appskey[sizeof(APPSKEY)];
    uint8_t nwkskey[sizeof(NWKSKEY)];
    memcpy_P(appskey, APPSKEY, sizeof(APPSKEY));
    memcpy_P(nwkskey, NWKSKEY, sizeof(NWKSKEY));
    LMIC_setSession (0x1, DEVADDR, nwkskey, appskey);
    #else
    // If not running an AVR with PROGMEM, just use the arrays directly
    LMIC_setSession (0x1, DEVADDR, NWKSKEY, APPSKEY);
    #endif

    #if defined(CFG_eu868)
    // Set up the channels used by the Things Network, which corresponds
    // to the defaults of most gateways. Without this, only three base
    // channels from the LoRaWAN specification are used, which certainly
    // works, so it is good for debugging, but can overload those
    // frequencies, so be sure to configure the full frequency range of
    // your network here (unless your network autoconfigures them).
    // Setting up channels should happen after LMIC_setSession, as that
    // configures the minimal channel set.
    // NA-US channels 0-71 are configured automatically
    LMIC_setupChannel(0, 868100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(1, 868300000, DR_RANGE_MAP(DR_SF12, DR_SF7B), BAND_CENTI);
↪// g-band
    LMIC_setupChannel(2, 868500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(3, 867100000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(4, 867300000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(5, 867500000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(6, 867700000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(7, 867900000, DR_RANGE_MAP(DR_SF12, DR_SF7),  BAND_CENTI);
↪// g-band
    LMIC_setupChannel(8, 868800000, DR_RANGE_MAP(DR_FSK,  DR_FSK),  BAND_MILLI);
↪// g2-band
    // TTN defines an additional channel at 869.525Mhz using SF9 for class B
    // devices' ping slots. LMIC does not have an easy way to define set this
    // frequency and support for class B is spotty and untested, so this
    // frequency is not configured here.
    #elif defined(CFG_us915)
```

(continues on next page)

(continued from previous page)

```
    // NA-US channels 0-71 are configured automatically
    // but only one group of 8 should (a subband) should be active
    // TTN recommends the second sub band, 1 in a zero based count.
    // https://github.com/TheThingsNetwork/gateway-conf/blob/master/US-global_conf.json
    LMIC_selectSubBand(1);
    #endif

    // Disable link check validation
    LMIC_setLinkCheckMode(0);

    // TTN uses SF9 for its RX2 window.
    LMIC.dn2Dr = DR_SF9;

    // Set data rate and transmit power for uplink (note: txpow seems to be ignored by the library)
//    LMIC_setDrTxpow(DR_SF7,14);
    LMIC_setDrTxpow(DR_SF9,14);

    // Start job. This will initiate the repetitive sending of data packets,
    // because after each data transmission, a delayed call to "do_send()"
    // is being scheduled again.
    do_send(&sendjob);
}

void loop() {
/*
    // read from port 1, send to port 0:
    if (Serial.available()) {
      int inByte = Serial.read();
      SWSerial.write(inByte);
    }

    // read from port 0, send to port 1:
    if (SWSerial.available()) {
      int inByte = SWSerial.read();
      Serial.write(inByte);
    }
*/
    unsigned long chars;
    unsigned short sentences, failed;

    // For one second we parse GPS data and report some key values
//    for (unsigned long start = millis(); millis() - start < 1000;)
//    {
      while (SWSerial.available())
      {
        char c = SWSerial.read();
        // Serial.write(c); // uncomment this line if you want to see the GPS data flowing
        if (gps.encode(c)) // Did a new valid sentence come in?
          newGPSdata = true;
      }
//    }

/*
    if (newGPSdata)
```

(continues on next page)

(continued from previous page)

```
    {
      float flat, flon;
      unsigned long age;
      gps.f_get_position(&flat, &flon, &age);
      Serial.print("LAT=");
      Serial.print(flat == TinyGPS::GPS_INVALID_F_ANGLE ? 0.0 : flat, 6);
      Serial.print(" LON=");
      Serial.print(flon == TinyGPS::GPS_INVALID_F_ANGLE ? 0.0 : flon, 6);
      Serial.print(" ALT=");
      Serial.print(gps.f_altitude() == TinyGPS::GPS_INVALID_F_ALTITUDE ? 0.0 : gps.f_
↪altitude(), 6);
      Serial.print(" SAT=");
      Serial.print(gps.satellites() == TinyGPS::GPS_INVALID_SATELLITES ? 0 : gps.
↪satellites());
      Serial.print(" PREC=");
      Serial.print(gps.hdop() == TinyGPS::GPS_INVALID_HDOP ? 0 : gps.hdop());
    }

    gps.stats(&chars, &sentences, &failed);
    Serial.print(" CHARS=");
    Serial.print(chars);
    Serial.print(" SENTENCES=");
    Serial.print(sentences);
    Serial.print(" CSUM ERR=");
    Serial.println(failed);
    if (chars == 0)
      Serial.println("** No characters received from GPS: check wiring **");
*/
    os_runloop_once();
}
```

Listing 25: TTN payload decoder for Dragino LoRa sensor node

```
function Decoder(bytes, port) {
  var result = {};
  var transformers = {};

  if (port == 9) {
    transformers = {
      temperature: function transform(bytes) {
        value = bytes[0] * 256 + bytes[1];
        if (value >= 32768) value = value - 65536;
        return value / 100.0;
      },
      humidity: function transform(bytes) {
        return (bytes[0] * 256 + bytes[1]) / 100.0;
      },
      lat: function transform(bytes) {
        return (
          (bytes[0] * 16777216 + bytes[1] * 65536 + bytes[2] * 256 + bytes[3]) /
            1000000.0 -
          90.0
        );
      },
      lon: function transform(bytes) {
        return (
```

(continues on next page)

(continued from previous page)

```
          (bytes[0] * 16777216 + bytes[1] * 65536 + bytes[2] * 256 + bytes[3]) /
            1000000.0 -
          180.0
        );
      },
      altitude: function transform(bytes) {
        return bytes[0] * 256 + bytes[1];
      },
      sat: function transform(bytes) {
        return bytes[0];
      },
      vbattery: function transform(bytes) {
        return (bytes[0] * 256 + bytes[1]) / 100.0;
      }
    };

    result["temperature"] = {
      value: transformers["temperature"](bytes.slice(0, 2)),
      uom: "Celsius"
    };

    result["humidity"] = {
      value: transformers["humidity"](bytes.slice(2, 4)),
      uom: "Percent"
    };

    result["lat"] = {
      value: transformers["lat"](bytes.slice(4, 8)),
      uom: "Degree"
    };

    result["lon"] = {
      value: transformers["lon"](bytes.slice(8, 12)),
      uom: "Degree"
    };

    result["altitude"] = {
      value: transformers["altitude"](bytes.slice(12, 14)),
      uom: "Meter"
    };

    result["sat"] = {
      value: transformers["sat"](bytes.slice(14, 15)),
      uom: "Count"
    };

    result["vbattery"] = {
      value: transformers["vbattery"](bytes.slice(15, 17)),
      uom: "Volt"
    };

    return result;
  }
}
```

2.7.5 References

- Arduino Uno R3 microcontroller
- FAQ on Arduino microcontrollers from Adafruit
- Dragino Lora/GPS Shield Wiki
- Dragino Lora/GPS Shield github
- Seeedstudio Solar Charger Shield V2.2
- IBM LMIC (LoraMAC-in-C) library for Arduino
- Connect to TTN - Wiki for Dragino Project
- dragino/Arduino-Profile-Examples/Arduino_LMIC.ino GitHub
- dragino/Arduino-Profile-Examples/tinygps_example.ino GitHub
- goodcheney/ttn_mapper/gps_shield at master GitHub

On battery saving / using the deep sleep mode (these are written for other microcontroller boards, but do apply for the Arduino Uno R3 and the Dragino Lora/GPS Shield, too):

- Adafruit Feather 32u4 LoRa - long transmission time after deep sleep - End Devices (Nodes) - The Things Network
- Full Arduino Mini LoraWAN and 1.3uA Sleep Mode - End Devices (Nodes) - The Things Network
- Adding Method to Adjust hal_ticks Upon Waking Up from Sleep · Issue #109 · matthijskooijman/arduino-lmic
- minilora-test/minilora-test.ino at cbe686826bd84fac8381de47b5f5b02dd47c2ca0 · tkerby/minilora-test
- Arduino-LMIC library with low power mode - Mario Zwiers

2.8 Pycom LoPy4

This tutorial is made to showcase the use of Pycom LoPy4 board to create a LoRaWAN enabled sensor node. In the following example, a temperature and humidity sensor was used with the Pycom LoPy4 board.

2.8.1 Hardware

Microcontroller

The Pycom LoPy4 is a microcontroller board offering many radio frequency (RF) connection options, namely LoRa (and LoRaWAN), SIGFOX, Bluetooth (Classic and Low Energy, BLE), and WiFi. In contrast to most other microcontroller boards the LoPy4 is programmed in MicroPython, which is a special subset of the Python 3 programming language and libraries for microcontrollers. The module is operated by the Espressif ESP32 microcontroller board, which contains a dual-core Xtensa 32bit LX6 processor running with up to 240MHz, 8 MB of flash memory (to store the program code and some files within a file system), and 520 KB of RAM (to store variables, status information, and buffers). The ESP32 module also has built-in WiFi and Bluetooth LE connectivity. In addition, the LoPy4 has 4 MB of PSRAM (pseudo static RAM) that is used as a memory extension for the ESP32. The operating voltage of the board is 3.3V (this is important when attaching sensors and other peripherals; they also must operate on 3.3V). The board offers 18 general purpose input/output pins (18 GPIOs), from which up to 12 can be used as analog input pins (with 12bit analog digital converters (ADC)) and two as analog output pins (8bit digital analog converter (DAC)). Most GPIO pins can be configured for specific hardware protocols. In total 3 serial ports (programmable Universal Asynchronous Receiver and Transmitter, UART), 2 I2C ports, 3 SPI ports, 1 CAN bus, 1 PWM channel, and an I2S

Fig. 25: LoPy4 from Pycom. LoPy4 pinout, documentation, example code.

port can be utilized. The board has a built-in RGB LED that can be programmed by the user. The LoPy4 is available from the manufacturer for around 35 €.

The LoPy4 needs to be operated on 3.5V – 5.5V put to the VIN pin. The onboard regulator brings it down to 3.3V. The 3.3V pin can only be used as an output. Do not feed 3.3V into this pin as this could damage the regulator. The board can be programmed over the serial interface, or via WiFi using a telnet or FTP connection. By default, the LoPy4 acts as a WiFi access point (SSID: lopy4–wlan–XXXX, Password: www.pycom.io) and a user can connect to the module after joining the WiFi network in order to upload user programs and data files.

A WiFi and Bluetooth antenna is mounted on the LoPy board, but also an external antenna can be connected via an SMA-type connector. The LoRa or SIGFOX antenna has to be connected via an SMA-type connector. The LoRa transmitter and receiver is encapsulated within a LoRa module. It uses the LoRa chip SX1276 from the company Semtech and can be configured to work either in the 433 MHz, 868 MHz, or 915 MHz frequency band. The LoRa module is connected via SPI interface to the microcontroller and all of the required connections of the LoRa transceiver pins with the microcontroller are already built-in on the LoPy4 board. Since the module only implements the LoRa physical layer, the LoRaWAN protocol stack is implemented in software on the microcontroller. The implemented LoRaWAN functionality is compatible with LoRaWAN Class A/C.

Expansion board 3.0

The LoPy4 can be attached to the Pycom Expansion Board 3.0. The board offers a USB port that is connected internally via a USB/Serial converter to the default serial port (UART) of the LoPy4. This allows to easily upload programs and data files to the LoPy4 from a developer computer over USB connection. The expansion board also comes with a connector for a 3.7V lithium polymer (LiPo) battery with an additional battery charger circuit. When the expansion board is connected via USB to a developer computer or to an USB charger, an attached battery will be automatically charged. The battery voltage can be monitored via the LoPy4 analog to digital converter (ADC). The board also comes with a user LED, a user switch, and a MicroSD card slot to read and write files from the LoPy4, for example, to log recorded data. The LoPy Epansion Board 3.0 is available from the manufacturer for 16 €.

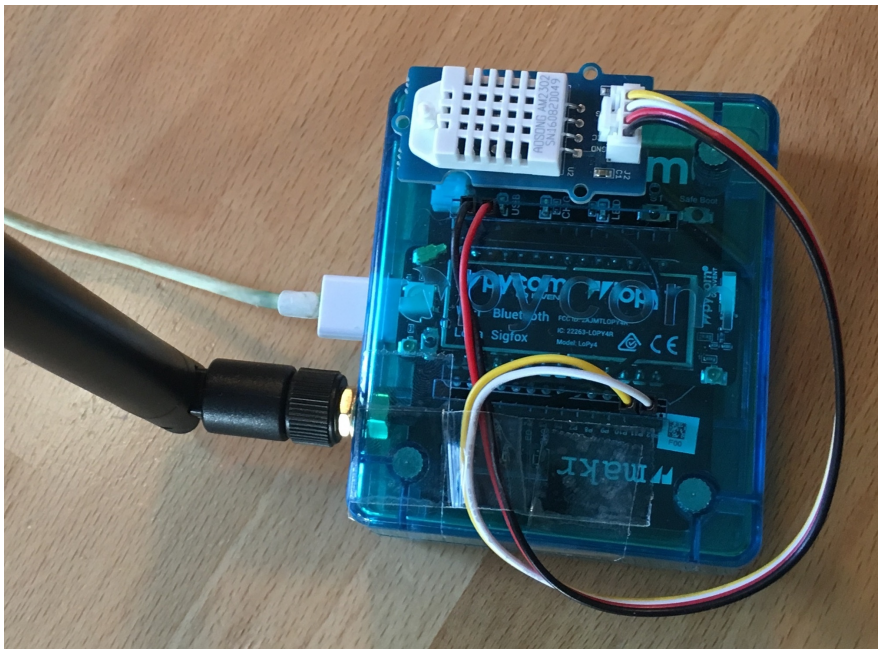


Fig. 26: The Pycom LoPy4 with the Expansion Board 3 (inside the blue case) and an externally attached DHT22 temperature and humidity sensor.

Sensor

We have attached a Grove DHT22 sensor module to the expansion board, which measures air temperature and humidity. The minimal time interval between two measurements is 2 seconds. All data transfers between the DHT22 and the microcontroller use a single digital line. The sensor data pin is attached to a GPIO pin (here: GPIO22) of the expansion board. The 5V pin of the Grove module is connected to 3V3 of the expansion board, and the GND of the Grove module to GND of the expansion board. The DHT22 datasheet can be accessed here. The sensor is available in German shops for around 4 € to 10 €.

2.8.2 Software

The sensor node has been programmed in the MicroPython language. We use Microsoft's Visual Studio Code platform with the Pymakr plugin to edit and upload the program. The Pymakr plugin is developed by Pycom and can be used with either Visual Studio Code or the Atom Text Editor. Both IDEs can be downloaded free of charge; Atom is also Open Source software. Note that MicroPython programs do not need to be compiled (like Java or C/C++ programs). The source code is interpreted by the Microcontroller instead.

The source code consists of the following two files. *main.py* and *boot.py*. They must be copied into the base folder on the LoPy4. We use a library for the DHT22 written by Erik de Lange. It can be downloaded from the following link but is also provided here *dht22.py*. The library has to be copied into the subdirectory "lib" on the LoPy4.

After the program has successfully established a connection to The Things Network it reports the air temperature, humidity, and the voltage of an attached LiPo battery every 5 minutes. Since we are running the device on an USB charger, the program does not check the battery level and the transferred value is always set to 0 V. All three values are being encoded in two byte integer values each (in most significant byte order) and then sent as a 6 bytes data packet to the respective TTN application using LoRaWAN port 7. Please note, that LoRaWAN messages can be addressed to ports 1-255 (port 0 is reserved); these ports are similar to port numbers 0-65535 when using the Internet TCP/IP protocol. Voltage and humidity values are always greater or equal to 0, but the temperature value can also become negative. Negative values are represented as a two's complement ; this must be considered in the Payload Decoding

Function used in The Things Network (see *below*).

The program as given above does not make use of the deep sleep mode or any other power saving method. In between two sensor readings the microcontroller is busy 'doing nothing' until the waiting time before the next measurement is over. When the LoPy4 should be operated on battery, the power saving modes of the LoPy4 should be investigated. Note that this will require to restructure the *main.py* program significantly.

2.8.3 Services

The services used for this sensor-node are:

- *TheThingsNetwork* service for LoRaWAN network service.
- *TheThingsNetwork - OGC SensorWeb* integration for uploading LoRaWAN sensor data into OGC infrastructure.

Registration of the sensor node with The Things Network (TTN)

The LoRaWAN protocol makes use of a number of different identifiers, addresses, keys, etc. These are required to unambiguously identify devices, applications, as well as to encrypt and decrypt messages. The names and meanings are nicely explained on a dedicated TTN web page.

The program given above connects the sensor node with The Things Network (TTN) using the Over-the-Air-Activation (OTAA) mode. In this mode, we use the three keys AppEUI, DevEUI, AppKey. The DevEUI is pre-programmed into the LoPy4. In order to register the device with TTN, you first need to fetch the DevEUI from the LoPy4 board. This is explained in the LoPy4 documentation. Each sensor node must be manually registered in the TTN console before it can be started. This assumes that you already have a TTN user account and have created an application in the user account (both need to be created otherwise). In the TTN console create a new device using the DevEUI value that was previously determined. After the registration of the device the two generated keys (AppEUI, AppKey) can be copied from the TTN console and must be pasted into the the proper places in the source code of the program above. Please make sure that you choose for both keys the correct byte ordering (all are in MSB, i.e. in the same ordering as given in the TTN console). A detailed explanation of these steps is given here. Then the program can be uploaded to the LoPy4 microcontroller. Note that the two constants (AppEUI, AppKey) must be changed in the source code for every new sensor node (the DevEUI is different for each node anyway).

Using the OTAA mode has the advantage over the ABP (activation by personalization) mode that during connection the session keys are newly created which improves security. Another advantage is that the packet counter is automatically reset to 0 both in the node and in the TTN application.

TTN Payload Decoding

Everytime a data packet is received by a TTN application a dedicated Javascript function is being called (Payload Decoder Function). This function can be used to decode the received byte string and to create proper Javascript objects or values that can directly be read by humans when looking at the incoming data packet. This is also useful to format the data in a specific way that can then be forwarded to an external application (e.g. a sensor data platform like MyDevices or Thingspeak). Such a forwarding can be configured in the TTN console in the "Integrations" tab. *TTN payload decoder* given here checks if a packet was received on LoRaWAN port 7 and then assumes that it consists of the 6 bytes encoded as described above. It creates the three Javascript objects 'temperature', 'humidity', and 'vbattery'. Each object has two fields: 'value' holds the value and 'uom' gives the unit of measure. The source code can simply be copied and pasted into the 'decoder' tab in the TTN console after having selected the application. Choose the option 'Custom' in the 'Payload Format' field. Note that when you also want to handle other sensor nodes sending packets on different LoRaWAN ports, then the Payload Decoder Function can be extended after the end of the if (port==7) {. . . } statement by adding else if (port==8) {. . . } else if (port==9) {. . . } etc.

The Things Network - OGC SensorWeb Integration

The presented Payload Decoder Function works also with the TTN-OGC SWE Integration for the 52° North Sensor Observation Service (SOS). This software component can be downloaded from this repository. It connects a TTN application with a running transactional Sensor Observation Service 2.0.0 (SOS). Data packets received from TTN are imported into the SOS. The SOS persistently stores sensor data from an arbitrary number of sensor nodes and can be queried for the most recent as well as for historic sensor data readings. The 52° North SOS comes with its own REST API and a nice web client allowing to browse the stored sensor data in a convenient way.

We are running an instance of the 52° North SOS and the TTN-OGC SWE Integration. The web client for this LoRaWAN sensor node can be accessed on this page. Here is a screenshot showing the webclient:

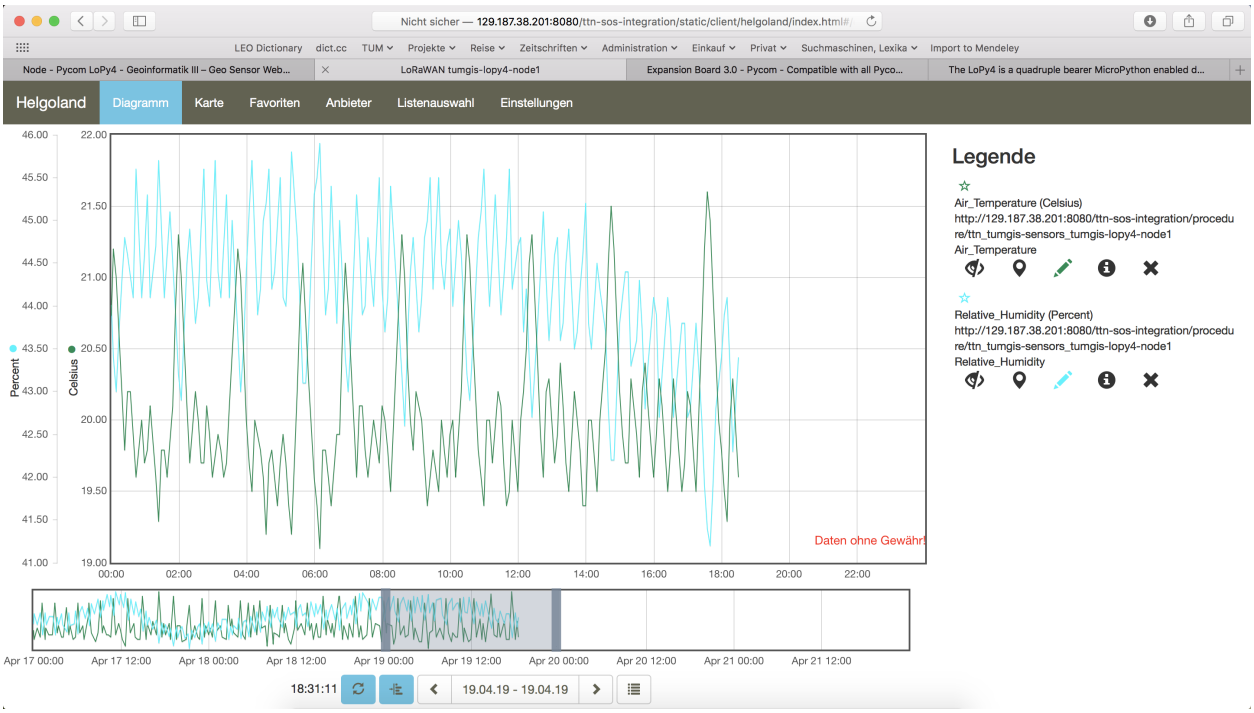


Fig. 27: Web client for data visualization

2.8.4 Code files

Listing 26: boot.py

```
from network import LoRa
import socket
import time
import ubinascii
import pycom
from machine import Pin
import dht22

# Initialise LoRa in LORAWAN mode.
# Please pick the region that matches where you are using the device:
# Asia = LoRa.AS923
# Australia = LoRa.AU915
```

(continues on next page)

(continued from previous page)

```
# Europe = LoRa.EU868
# United States = LoRa.US915
lora = LoRa(mode=LoRa.LORAWAN, region=LoRa.EU868)

# Create the OTAA authentication parameters:
# directly copy the values from the Things Network Console and replace the
# xxxx's and yyyy's by these values (do not prepend anything like '0x' or similar)
app_eui = ubinascii.unhexlify('xxxxxxxxxxxxxxxx')
app_key = ubinascii.unhexlify('yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy')

print("Initializing DHT22 Sensor... ",end='')
dht = dht22.device(Pin.exp_board.G22)
print("ready!\n")

pycom.heartbeat(False)
pycom.rgbled(0xFF0000)

# join a network using OTAA (Over the Air Activation)
lora.join(activation=LoRa.OTAA, auth=(app_eui, app_key), timeout=0)

# wait until the module has joined the network
while not lora.has_joined():
    time.sleep(2.5)
    print('Not yet joined...')

pycom.rgbled(0x00FF00)
time.sleep(1)

# create a LoRa socket
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)

# set the LoRaWAN data rate
s.setsockopt(socket.SOL_LORA, socket.SO_DR, 5)

# set the LoRaWAN port number for transmitted packets
s.bind(7)

pycom.heartbeat(True)
```

Listing 27: main.py

```
from network import LoRa
import socket
import time
import ubinascii
import pycom
from machine import Pin
import dht22

while (True):
    pycom.heartbeat(False)
    pycom.rgbled(0x800000)

    # start a new measurement (taking 2 seconds) to ensure that the next
    # retrieval of readings has current values
    dht.trigger()
```

(continues on next page)

(continued from previous page)

```
    time.sleep(0.2)

    # now start a 2nd measurements which - according to the DHT22 datasheet -
    # delivers the measured values from the previous reading
    hasreading=False
    numtrials=0

    while(not hasreading):
        hasreading=dht.trigger()
        numtrials=numtrials+1
        if hasreading:
            print("RH = {}%  T = {}C".format(dht.humidity, dht.temperature))
        else:
            print(dht.status)

    hum_msb=int(dht.humidity*100/256)
    hum_lsb=int(dht.humidity*100%256)

    tmp_int=int(dht.temperature*100)
    # if temperature value is negative, then represent it by its 2's complement (16 bit)
    if (tmp_int<0):
        tmp_int=65536+tmp_int

    tmp_msb=int(tmp_int/256)
    tmp_lsb=int(tmp_int%256)
    print("RH = {} {}  T = {} {}".format(hum_msb, hum_lsb, tmp_msb, tmp_lsb))
    pycom.rgbled(0x000040)

    # make the socket blocking
    # (waits for the data to be sent and for the 2 receive windows to expire)
    s.setblocking(True)

    # send some data
    s.send(bytes([tmp_msb, tmp_lsb, hum_msb, hum_lsb, 0, 0]))

    # make the socket non-blocking
    # (because if there's no data received it will block forever...)
    s.setblocking(False)

    # get any data received (if any...)
    data = s.recv(64)
    print(data)

    pycom.heartbeat(True)
    # wait for such a time period that we have one measurement every 300 seconds
    time.sleep(300-numtrials*4-3)
```

Listing 28: dht22.py

```

# dht22.py
#
# Class file for accessing the DHT22 temperature and humidity sensor using a WiPy 2.0.
#
# 2018 - Erik de Lange
```

(continues on next page)

(continued from previous page)

```

from machine import Pin

import pycom
import time

class device:

    def __init__(self, pin):
        self.temperature = None
        self.humidity = None
        self.status = "NoConversionStartedError"
        self.pin = Pin(pin, mode=Pin.OPEN_DRAIN)

    def trigger(self):
        self.pin(1)
        time.sleep(2)  # enforce two second read interval

        self.pin(0)  # send start signal (1ms low).
        time.sleep_ms(1)

        pulses = pycom.pulses_get(self.pin, 100)  # capture communication

        self.pin.init(Pin.OPEN_DRAIN)

        if len(pulses) != 82:  # 40 data bit plus one acknowledge expected
            self.status = "ReadError - received {} only pulses".format(len(pulses))
            return False

        bits = []

        for level, duration in pulses[1:]:
            if level == 1:
                bits.append(0 if duration < 50 else 1)  # convert to 0 or 1

        data = []

        for n in range(5):
            byte = 0
            for i in range(8):  # shift 8 bits into a byte
                byte <<= 1
                byte += bits[n * 8 + i]
            data.append(byte)

        int_rh, dec_rh, int_t, dec_t, csum = data

        if ((int_rh + dec_rh + int_t + dec_t) & 0xFF) != csum:
            self.status = "Checksum Error"
            return False

        self.humidity = ((int_rh * 256) + dec_rh) / 10
        self.temperature = (((int_t & 0x7F) * 256) + dec_t) / 10
        if (int_t & 0x80) > 0:
            self.temperature *= -1

        self.status = "OK"
        return True
```

(continues on next page)

(continued from previous page)

```

if __name__ == "__main__":
    dht = device(Pin.exp_board.G22)

    for _ in range(5):
        if dht.trigger() == True:
            print("RH = {}%  T = {}C".format(dht.humidity, dht.temperature))
        else:
            print(dht.status)
```

Listing 29: TTN payload decoder

```
function Decoder (bytes, port) {
  var result = {};
  var transformers = {};

  if (port==7) {
     transformers = {
      'temperature': function transform (bytes) {
          value=bytes[0]*256 + bytes[1];
          if (value>=32768) value=value-65536;
          return value/100.0;
        },
      'humidity': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/100.0;
        },
      'vbattery': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/100.0;
        },
    }

    result['temperature'] = {
      value: transformers['temperature'](bytes.slice(0, 2)),
      uom: 'Celsius',
    }

    result['humidity'] = {
      value: transformers['humidity'](bytes.slice(2, 4)),
      uom: 'Percent',
    }

    result['vbattery'] = {
      value: transformers['vbattery'](bytes.slice(4, 6)),
      uom: 'Volt',
    }
  }

  return result;
}
```

2.8.5 References

- Pycom LoPy4 product homepage
- LoPy4 specification document

- LoPy4 pinout specification
- LoPy4 online documentation (incl. description of software libraries)
- MicroPython language and library reference
- LoPy4 Getting Started (hardware & software setup, installation of Pymakr IDE)

On the Expansion Board 3.0

- Pycom Expansion Board 3.0 product homepage
- Expansion Board 3.0 documentation (pinout, datasheet)

On the DHT22 sensor

- MicroPython library for the DHT22 sensor
- DHT22 datasheet

2.9 Seeeduino LoRaWAN

2.9.1 Hardware

Microcontroller

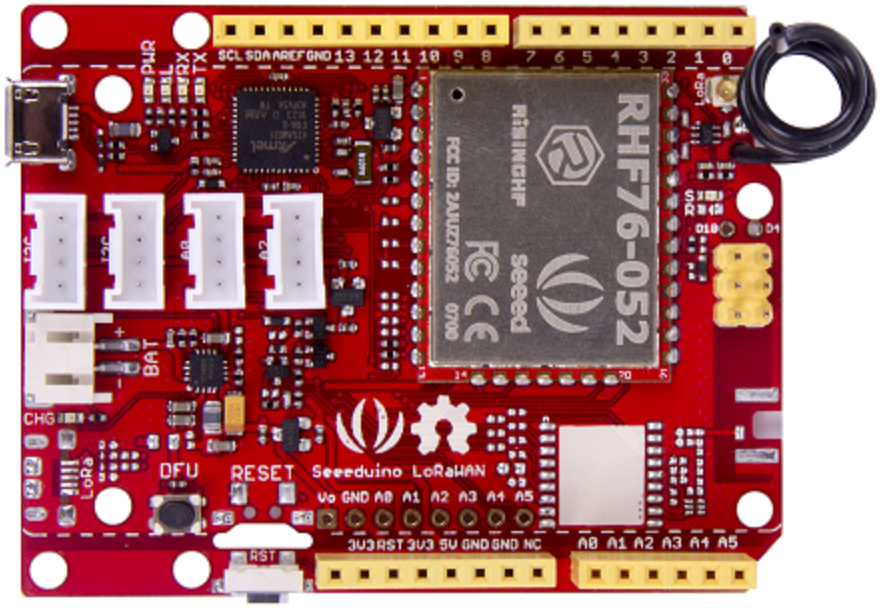


Fig. 28: Seeeduino LoRaWAN microcontroller from Seeed Studio.

The Seeeduino LoRaWAN module is operated by the 32bit microcontroller ATSAMD21G18 (ARM® Cortex®-M0+) running at 48MHz. It has 256 KB flash memory (to store the program code) and 32 KB of RAM (to store variables, status information, and buffers). The operating voltage of the board is 3.3V (this is important when attaching sensors

and other peripherals; they also must operate on 3.3V). The board offers 20 general purpose digital input/output pins (20 GPIOs), 6 analog input pins (with 12bit analog digital converters (ADC)), 1 analog output pin (with 10bit digital analog converter (DAC)), 2 serial ports (2 programmable Universal Asynchronous Receiver and Transmitters, UARTs). The board comes with an embedded lithium battery management chip and status indicator led, which allows to directly connect a 3.7V LiPo rechargeable battery that will be automatically recharged when the board is powered over its USB connector. The battery voltage level can be queried from analog input A4, the charging status (charging, full) from analog input A5. The Seeeduino LoRaWAN (without GPS module) is available in German shops for around 37 €.

The LoRa transmitter and receiver is encapsulated within an RHF76-052AM module from the Chinese company RisingHF. The RF module contains its own microcontroller, which implements the LoRaWAN protocol. The module is connected via the serial interface to the ATSAMD21G18 microcontroller and can be controlled by sending so-called 'AT' commands. The implemented LoRaWAN functionality is compatible with LoRaWAN Class A/C. The explanation of all supported commands as well as a number of examples on how to use the Seeeduino LoRaWAN are given on the Seeeduino LoRaWAN Wiki.

The board has 4 on-board Grove connectors. 'Grove' is a framework developed by the company Seeed Studio standardizing the connectors, operating voltages, and pin configurations for attaching peripherals like sensors, actuators, and displays to microcontrollers. Note that the Grove modules need to be able to operate (also) on 3.3V (instead of only with 5V), because the Seeeduino LoRaWAN board only provides 3.3V to the Grove connectors. Important hint: if you want to use the Grove ports, make sure to include the command "digitalWrite(38, HIGH)" in the setup() routine of your program. A low level on that pin deactivates the power supply of the four Grove ports.

The board has also the typical Arduino UNO connectors allowing to attach so-called Arduino shields (however, please note that the shields must be working with 3.3V; the normal operating voltage for the Arduino UNO microcontroller and its shields is 5V).

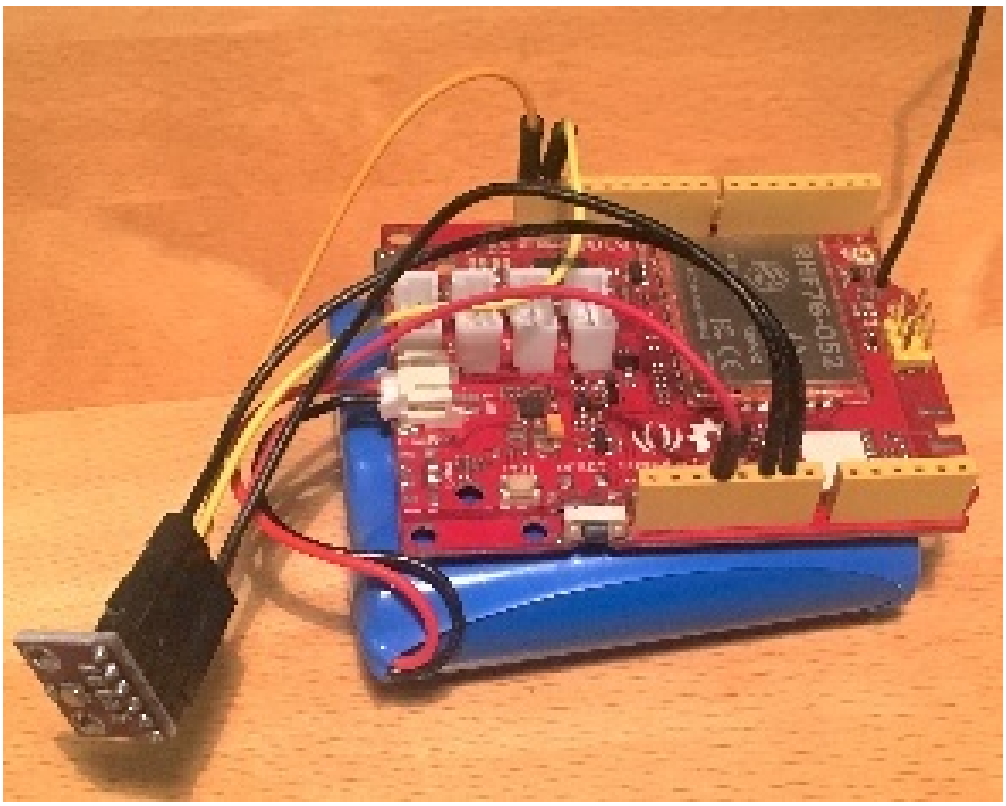


Fig. 29: The Seeeduino LoRaWAN GPS microcontroller with a 6600 mAh lithium polymer (LiPo) battery (bottom), and an attached BME280 temperature / humidity / barometer sensor module.

Sensor

We attached a Bosch BME280 sensor module to the extension connectors of the microcontroller board using 5 wires. The employed BME280 sensor board is a cheap no-name product. VCC and GND are connected to 3.3V and GND of the microcontroller board respectively. SCL and SDA from the sensor board are connected to SCL and SDA of the microcontroller board. SDO from the sensor board is also connected to GND of the microcontroller; it selects 0x76 as the I2C device address (a high level, i.e. 3.3V, would set the device address to 0x77 - this is relevant, if two sensor modules should be operated on the same I2C bus). Note that there is also a Seeed Grove BME280 module available which alternatively can be used and connected to the first I2C Grove connector of the Seeeduino LoRaWAN board. The BME280 measures temperature in the range -40 - 85 °C, with ±1.0°C accuracy; 0% - 100% relative humidity with ±3% accuracy; and atmospheric pressure in the range 300 - 1100 hPa (1 hPa= one hundred Pa) with ±1.0 hPa accuracy. It offers the two interface standards I2C and SPI (we are using I2C here and the default I2C address 0x76). The atmospheric pressure changes with altitude, hence, the BME280 can also be used to measure the approximate altitude of a place.

2.9.2 Software

The sensor node has been programmed using the Arduino IDE. Please note, that in the Arduino framework a program is called a 'Sketch'.

In order to support the "Seeeduino LoRaWAN" board with the Arduino IDE, make sure to have installed the package "Seeed SAMD boards by Seeed Studio" in version 1.3.0 using the board manager in the Arduino IDE. This is also explained on a dedicated webpage from Seeed Studio. The sketch requires the software libraries "RTCZero", "Arduino_BME280", "Adafruit_Sensor", "Wire", and "LoRaWAN". The first three have to be installed using the library manager of the Arduino IDE, the fourth library is already installed with the Arduino IDE and the latter library comes with the "Seeeduino LoRaWAN" board installation.

After the sketch has successfully established a connection to The Things Network it reports the air temperature, relative humidity, air pressure, altitude, and the voltage of a (possibly) attached LiPo battery every 5 minutes. All five values are being encoded in two byte integer values each and then sent as a 10 bytes data packet to the respective TTN application using LoRaWAN port 33. Please note, that LoRaWAN messages can be addressed to ports 1-255 (port 0 is reserved); these ports are similar to port numbers 0-65535 when using the Internet TCP/IP protocol. Voltage, pressure, altitude, and humidity values are always greater or equal to 0, but the temperature value can also become negative. Negative values are represented as a two's complement; this must be considered in the Payload Decoding Function used in The Things Network (see below).

In between two sensor readings the microcontroller, the LoRaWAN module, and the sensor module are going into deep sleep mode to save battery power. During LoRaWAN data transmission the device draws up to 65mA current. When in sleep mode the entire node only draws around 0.06 mA power. Hence, with a 6600 mAh 3.7V LiPo battery and the current version of the sketch the system should be able to run for many years before recharging (not taking into account the self-discharging rate of the battery).

The source code is provided in the following section *Arduino Sketch for Seeeduino LoRaWAN sensor node*

2.9.3 Services

The services used for this sensor-node are:

- *TheThingsNetwork* service for LoRaWAN network service.
- *TheThingsNetwork - OGC SensorWeb* integration for uploading LoRaWAN sensor data into OGC infrastructure.

Registration of the sensor node with The Things Network (TTN)

The LoRaWAN protocol makes use of a number of different identifiers, addresses, keys, etc. These are required to unambiguously identify devices, applications, as well as to encrypt and decrypt messages. The names and meanings are nicely explained on a dedicated TTN web page.

The sketch given above connects the sensor node with The Things Network (TTN) using the Over-the-Air-Activation (OTAA) mode. In this mode, we use the three keys AppEUI, DevEUI, AppKey. The DevEUI should normally be delivered with the sensor node by the manufacturer. However, it seems that there is no explicit DevEUI provided with the Seeeduino LoRaWAN module. Therefore, it has to be generated automatically together with the other two keys using the TTN console. Each sensor node must be manually registered in the TTN console before it can be started. This assumes that you already have a TTN user account (which needs to be created otherwise). In the TTN console create a new device with also the DevEUI being automatically generated. After the registration of the device the respective keys (AppEUI, DevEUI, AppKey) can be copied from the TTN console and must be pasted into the the proper places in the source code of the sketch above. Please make sure that you choose for each of the three keys are in the correct byte ordering (all are in MSB, i.e. in the same ordering as given in the TTN console). A detailed explanation of these steps is given on this page. Then the sketch can be compiled and uploaded to the Seeeduino LoRaWAN microcontroller. Note that the three constants (AppEUI, DevEUI, AppKey) must be changed in the source code for every new sensor node.

Using the OTAA mode has the advantage over the ABP (activation by personalization) mode that during connection the session keys are newly created which improves security. Another advantage is that the packet counter is automatically reset to 0 both in the node and in the TTN application.

TTN Payload Decoding

Everytime a data packet is received by a TTN application a dedicated Javascript function is being called (Payload Decoder Function). This function can be used to decode the received byte string and to create proper Javascript objects or values that can directly be read by humans when looking at the incoming data packet. This is also useful to format the data in a specific way that can then be forwarded to an external application (e.g. a sensor data platform like MyDevices or Thingspeak).

Such a forwarding can be configured in the TTN console in the "Integrations" tab. *TTN payload decoder for Seeeduino LoRaWAN sensor node* given here checks if a packet was received on LoRaWAN port 33 and then assumes that it consists of the 10 bytes encoded as described above. It creates the five Javascript objects 'temperature', 'humidity', 'pressure', 'altitude', and 'vbattery'. Each object has two fields: 'value' holds the value and 'uom' gives the unit of measure. The source code can simply be copied and pasted into the 'decoder' tab in the TTN console after having selected the application. Choose the option 'Custom' in the 'Payload Format' field. Note that when you also want to handle other sensor nodes sending packets on different LoRaWAN ports, then the Payload Decoder Function can be extended after the end of the if (port==33) {...} statement by adding else if (port==7) {...} else if (port==8) {...} etc.

The Things Network - OGC SensorWeb Integration

The presented Payload Decoder Function works also with the TTN-OGC SWE Integration for the 52° North Sensor Observation Service (SOS). This software component can be downloaded from this repository. It connects a TTN application with a running transactional Sensor Observation Service 2.0.0 (SOS). Data packets received from TTN are imported into the SOS. The SOS persistently stores sensor data from an arbitrary number of sensor nodes and can be queried for the most recent as well as for historic sensor data readings. The 52° North SOS comes with its own REST API and a nice web client allowing to browse the stored sensor data in a convenient way.

We are running an instance of the 52° North SOS and the TTN-OGC SWE Integration. The web client for this LoRaWAN sensor node can be accessed on this webpage. Here is a screenshot showing the webclient:

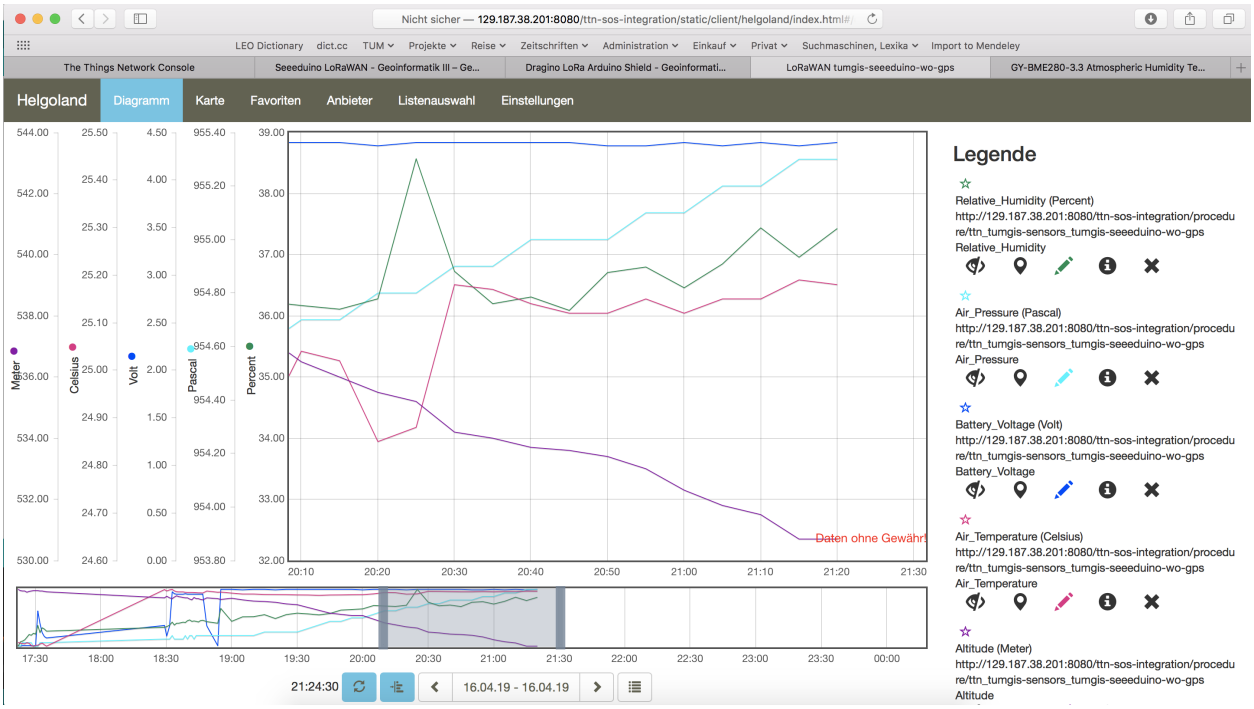


Fig. 30: Web client for data visualization

2.9.4 Code files

Listing 30: Arduino Sketch for Seeeduino LoRaWAN sensor node

```
/**********************************************************************************
 * Arduino Sketch for a LoRaWAN sensor node that is registered with
 * 'The Things Network' (TTN) www.thethingsnetwork.org
 *
 * Filename: Seeeduino_LoRaWAN_GPS_BME280_OTAA_Sleep_Adafruit_V2.ino
 *
 * Author:  Thomas H. Kolbe, thomas.kolbe@tum.de
 * Version: 1.0.1
 * Last update: 2019-04-17
 *
 * This sketch works with a Seeeduino LoRaWAN microcontroller board (with or
 * without embedded GPS module). See http://wiki.seeedstudio.com/Seeeduino_LoRAWAN/
 * It requires a Seeed Grove BME280 air temperature, relative humidity,
 * and air pressure sensor module attached to the I2C Grove connector of
 * the microcontroller board. The current configuration assumes that
 * the BME280 is configured to I2C device address 0x76 (default).
 * The sketch makes a connection to The Things Network (TTN) using
 * LoRaWAN in OTAA mode. It then sends a data packet of 10 bytes to
 * LoRaWAN port 33 around every 5 minutes. The packet contains the
 * following 5 integer values (16 bit, most significant byte (MSB) first):
 *   1. temperature in Celsius (signed, multiplied by 100)
 *   2. relative humidity in percent (unsigned, multiplied by 100)
 *   3. air pressure in Pascal (unsigned, divided by 10)
 *   4. current altitude in Meters (unsigned, multiplied by 10)
 *   5. battery voltage in millivolt (unsigned)
```

(continues on next page)

(continued from previous page)

```
 * These values have to be decoded by the LoRaWAN network controller
 * using a proper "payload decoder function" written in Javascript.
 *
 * Note that when the board is powered over the USB connector and
 * no battery is connected, the measured battery voltage is incorrect.
 *
 * If the board shall be running on a lithium polymer (LiPo) battery,
 * it is recommended to remove the green power LED from the board or
 * to cut the connection between the LED and the resistor lying above
 * of it as the LED constantly draws around 8mW of power. In order to
 * save energy the sketch puts the GPS module on the board to standby
 * mode right from the beginning. After each measurement and data transfer
 * the LoRaWAN module and the sensor is put to standby mode, too, and the
 * microcontroller goes into deep sleep mode. All components require
 * a total current of around 0.34mA during sleep mode and up to 65mA
 * during LoRa transmission for the board version with GPS. The board
 * version without GPS only requires 0.06mA during sleep mode. Since the
 * entire system is mostly sleeping, the GPS board should be running
 * around 2 years on a 6600mAh LiPo battery before recharging
 * (6600mAh / 0.34mA / 24 = 808 days). The non GPS board version should
 * even run for more than 10 years...
 *
 * This code is based on example code given on the Seeeduino LoRaWAN
 * wiki page. It utilizes the Open Source libraries "Adafruit_BME280"
 * and "Adafruit_Sensor" provided by the company Adafruit and the
 * library "LoRaWan.h" provided by Seeed Studio.
 ******************************************************************************/

#include <RTCZero.h>
#include <LoRaWan.h>
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>

// Keep the following line, if the board is a Seeeduino LoRaWAN with GPS,
// otherwise comment the line out

// #define HAS_GPS 1

#define BME280_ADDRESS       (0x76)   // I2C device address of the BME280 sensor

// The barometer of the BME280 can also be used to estimate the current
// altitude of the device, if the air pressure at sea level (NN) is known.
// The following value has to be set to the current air pressure at NN (in hPa)
// in order to give reasonable altitude estimations. Note that this value is
// slowly changing over time. For Munich the current value can be obtained
// from https://www.meteo.physik.uni-muenchen.de/mesomikro/stadt/messung.php

#define SEALEVELPRESSURE_HPA (1017.8)

Adafruit_BME280 bme280;

RTCZero rtc;

unsigned char data[10];                 // buffer for the LoRaWAN data packet to be
↪transferred
char buffer[256];                       // buffer for text messages received from the
↪LoRaWAN module for display
```

(continues on next page)

(continued from previous page)

```

void setup(void)
{
    digitalWrite(38, HIGH);             // Provide power to the 4 Grove connectors of the board

    for(int i = 0; i < 26; i ++)        // Set all pins to HIGH to save power (reduces the
    {                                   // current drawn during deep sleep by around 0.7mA).
        if (i!=13) {                    // Don't switch on the onboard user LED (pin 13).
          pinMode(i, OUTPUT);
          digitalWrite(i, HIGH);
        }
    }

    delay(5000);                        // Wait 5 secs after reset/booting to give time for potential upload
                                        // of a new sketch (sketches cannot be uploaded when in sleep mode)
    SerialUSB.begin(115200);            // Initialize USB/serial connection
    delay(500);
    // while(!SerialUSB);
    SerialUSB.println("Seeeduino LoRaWAN board started!");

    if(!bme280.begin(BME280_ADDRESS)) { // Initialize the BME280 sensor module
      SerialUSB.println("BME280 device error!");
    }

    // Set the BME280 to a very low power operation mode (c.f. chapter 3.5
    // "Recommended modes of operation" in the BME280 datasheet. See
    // https://cdn-shop.adafruit.com/datasheets/BST-BME280_DS001-10.pdf );
    // proper values can only be queried every 60s
    bme280.setSampling(Adafruit_BME280::MODE_FORCED,
                    Adafruit_BME280::SAMPLING_X16,  // temperature
                    Adafruit_BME280::SAMPLING_X16,  // pressure
                    Adafruit_BME280::SAMPLING_X16,  // humidity
                    Adafruit_BME280::FILTER_OFF   );

    // nrgSave.begin(WAKE_RTC_ALARM);
    // rtc.begin(TIME_H24);

#ifdef HAS_GPS
    Serial.begin(9600);                 // Initialize serial connection to the GPS module
    delay(500);
    Serial.write("$PMTK161,0*28\r\n");  // Switch GPS module to standby mode as we don't use it in this sketch
#endif

    lora.init();                        // Initialize the LoRaWAN module

    memset(buffer, 0, 256);             // clear text buffer
    lora.getVersion(buffer, 256, 1);
    memset(buffer, 0, 256);             // We call getVersion() two times, because after a reset the LoRaWAN module can be
```

(continues on next page)

(continued from previous page)

```
    lora.getVersion(buffer, 256, 1);    // in sleep mode and then the first call only wakes it up and will not be performed.
    SerialUSB.print(buffer);

    memset(buffer, 0, 256);
    lora.getId(buffer, 256, 1);
    SerialUSB.print(buffer);

    // The following three constants (AppEUI, DevEUI, AppKey) must be changed
    // for every new sensor node. We are using the LoRaWAN OTAA mode (over the
    // air activation). Each sensor node must be manually registered in the
    // TTN console at https://console.thethingsnetwork.org before it can be
    // started. In the TTN console create a new device with the DevEUI also
    // being automatically generated. After the registration of the device the
    // three values can be copied from the TTN console. A detailed explanation
    // of these steps is given in
    // https://learn.adafruit.com/the-things-network-for-feather?view=all

    // The EUIs and the AppKey must be given in big-endian format, i.e. the
    // most-significant-byte comes first (as displayed in the TTN console).
    // For TTN issued AppEUIs the first bytes should be 0x70, 0xB3, 0xD5.

    // void setId(char *DevAddr, char *DevEUI, char *AppEUI);
    lora.setId(NULL, "xxxxxxxxxxxxxxxx", "yyyyyyyyyyyyyyyy");

    // setKey(char *NwkSKey, char *AppSKey, char *AppKey);
    lora.setKey(NULL, NULL, "zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz");

    lora.setDeciveMode(LWOTAA);           // select OTAA join mode (note that setDeciveMode is not a typo; it is misspelled in the library)
    // lora.setDataRate(DR5, EU868);         // SF7, 125 kbps (highest data rate)
    lora.setDataRate(DR3, EU868);         // SF9, 125 kbps (medium data rate and range)
    // lora.setDataRate(DR0, EU868);         // SF12, 125 kbps (lowest data rate, highest max. distance)

    // lora.setAdaptiveDataRate(false);
    lora.setAdaptiveDataRate(true);       // automatically adapt the data rate

    lora.setChannel(0, 868.1);
    lora.setChannel(1, 868.3);
    lora.setChannel(2, 868.5);
    lora.setChannel(3, 867.1);
    lora.setChannel(4, 867.3);
    lora.setChannel(5, 867.5);
    lora.setChannel(6, 867.7);
    lora.setChannel(7, 867.9);

    // The following two commands can be left commented out;
    // TTN works with the default values. (It also works when
    // uncommenting the commands, though.)
    // lora.setReceiceWindowFirst(0, 868.1);
    // lora.setReceiceWindowSecond(869.525, DR0);

    lora.setDutyCycle(false);             // for debugging purposes only - should normally be activated
    lora.setJoinDutyCycle(false);         // for debugging purposes only - should normally be activated
```

(continues on next page)

(continued from previous page)

```

    lora.setPower(14);                      // LoRa transceiver power (14 is the maximum for the 868 MHz band)

    // while(!lora.setOTAAJoin(JOIN));
    while(!lora.setOTAAJoin(JOIN,20));      // wait until the node has successfully joined TTN

    lora.setPort(33);                       // all data packets are sent to LoRaWAN port 33
}

void loop(void)
{
    bool result = false;
    float temperature, altitude, pressure, humidity;
    int16_t int16_temperature, int16_humidity, int16_pressure, int16_altitude, int16_vbat;

    bme280.takeForcedMeasurement();         // wake the sensor up for the next readings

    //get and print temperatures
    SerialUSB.print("Temp: ");
    SerialUSB.print(temperature = bme280.readTemperature());
    SerialUSB.print("C  ");

    //get and print atmospheric pressure data
    SerialUSB.print("Pressure: ");
    SerialUSB.print(pressure = bme280.readPressure());
    SerialUSB.print("Pa  ");

    //get and print altitude data
    SerialUSB.print("Altitude: ");
    SerialUSB.print(altitude = bme280.readAltitude(SEALEVELPRESSURE_HPA));
    SerialUSB.print("m  ");

    //get and print humidity data
    SerialUSB.print("Humidity: ");
    SerialUSB.print(humidity = bme280.readHumidity());
    SerialUSB.print("%  ");

    //get and print battery voltage
    SerialUSB.print("VBat: ");
    SerialUSB.print(int16_vbat=lora.getBatteryVoltage());
    SerialUSB.println("mV");

    int16_temperature = temperature*100.0;
    int16_humidity = humidity*100.0;
    int16_pressure = pressure/10.0;
    int16_altitude = altitude*10.0;

    data[0] = (byte) (int16_temperature >> 8);
    data[1] = (byte) (int16_temperature & 0x00FF);
    data[2] = (byte) (int16_humidity >> 8);
    data[3] = (byte) (int16_humidity & 0x00FF);
    data[4] = (byte) (int16_pressure >> 8);
    data[5] = (byte) (int16_pressure & 0x00FF);
```

(continues on next page)

(continued from previous page)

```
    data[6] = (byte) (int16_altitude >> 8);
    data[7] = (byte) (int16_altitude & 0x00FF);
    data[8] = (byte) (int16_vbat >> 8);
    data[9] = (byte) (int16_vbat & 0x00FF);

    result = lora.transferPacket(data, 10, 5);   // send the data packet (10 bytes) with a default timeout of 5 secs

    if(result)
    {
        short length;
        short rssi;

        memset(buffer, 0, 256);
        length = lora.receivePacket(buffer, 256, &rssi);

        if(length)
        {
            SerialUSB.print("Length is: ");
            SerialUSB.println(length);
            SerialUSB.print("RSSI is: ");
            SerialUSB.println(rssi);
            SerialUSB.print("Data is: ");
            for(unsigned char i = 0; i < length; i ++)
            {
                SerialUSB.print("0x");
                SerialUSB.print(buffer[i], HEX);
                SerialUSB.print(" ");
            }
            SerialUSB.println();
        }
    }

    lora.setDeviceLowPower();     // bring the LoRaWAN module to sleep mode
    doSleep((5*60-8)*1000);       // deep sleep for 292 secs (+ 3 secs transmission time + 5 secs timeout = 300 secs period)
    lora.setPort(33);             // send some command to wake up the LoRaWAN module again
}

// The following function implements deep sleep waiting. When being called the
// CPU goes into deep sleep mode (for power saving). It is woken up again by
// the CPU-internal real time clock (RTC) after the configured time.
//
// A similar function would also be available in the standard "ArduinoLowPower" library.
// However, in order to be able to use that library with the Seeeduino LoRaWAN board,
// four files in the package "Seeed SAMD boards by Seeed Studio Version 1.3.0" that is
// installed using the Arduino IDE board manager need to be patched. The reason is that
// Seeed Studio have not updated their files to a recent Arduino SAMD version yet
// and the official "ArduinoLowPower" library provided by the Arduino foundation is
// referring to some missing functions. For further information see here:
// https://forum.arduino.cc/index.php?topic=603900.0 and here:
// https://github.com/arduino/ArduinoCore-samd/commit/b9ac48c782ca4b82ffd7e65bf2c956152386d82b

```

(continues on next page)

(continued from previous page)

```
void doSleep(uint32_t millis) {
    if (!rtc.isConfigured()) {    // if called for the first time,
        rtc.begin(false);         // then initialize the real time clock (RTC)
    }

    uint32_t now = rtc.getEpoch();
    rtc.setAlarmEpoch(now + millis/1000);
    rtc.enableAlarm(rtc.MATCH_HHMMSS);

    rtc.standbyMode();            // bring CPU into deep sleep mode (until woken up
↪by the RTC)
}
```

Listing 31: TTN payload decoder for Seeeduino LoRaWAN sensor node

```
function Decoder (bytes, port) {
  var result = {};
  var transformers = {};

  if (port==33) {
    transformers = {
      'temperature': function transform (bytes) {
          value=bytes[0]*256 + bytes[1];
          if (value>=32768) value=value-65536;
          return value/100.0;
        },
      'humidity': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/100.0;
        },
      'pressure': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/10.0;
        },
      'altitude': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/10.0;
        },
      'vbattery': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/1000.0;
        }
    }

    result['temperature'] = {
      value: transformers['temperature'](bytes.slice(0, 2)),
      uom: 'Celsius',
    }

     result['humidity'] = {
      value: transformers['humidity'](bytes.slice(2, 4)),
      uom: 'Percent',
    }

    result['pressure'] = {
      value: transformers['pressure'](bytes.slice(4, 6)),
      uom: 'hPa',
    }

    result['altitude'] = {
```

(continues on next page)

(continued from previous page)

```
      value: transformers['altitude'](bytes.slice(6, 8)),
      uom: 'Meter',
    }

    result['vbattery'] = {
      value: transformers['vbattery'](bytes.slice(8, 10)),
      uom: 'Volt',
    }
  }

  return result;
}
```

2.9.5 References

- Seeeduino LoRaWAN microcontroller
- Seeeduino LoRaWAN Wiki with instructions
- A short presentation on LoRaWAN basics and using the Seeeduino LoRaWAN board

On the RisingHF RHF76-052AM LoRaWAN module

- Product homepage
- LoRaWAN Class A/C AT Command Specification

On the Bosch BME280 sensor

- Product details
- Datasheet
- Adafruit_BME280 library for the Arduino platform
- Adafruit_Sensor library for the Arduino platform
- Instructions from Adafruit Industries how to use the BME280 and the library

2.10 Seeeduino LoRaWAN with GPS

2.10.1 Hardware

Micro-controller

The Seeeduino LoRaWAN module is operated by the 32bit microcontroller ATSAMD21G18 (ARM® Cortex®-M0+) running at 48MHz. It has 256 KB flash memory (to store the program code) and 32 KB of RAM (to store variables, status information, and buffers). The operating voltage of the board is 3.3V (this is important when attaching sensors and other peripherals; they also must operate on 3.3V). The board offers 20 general purpose digital input/output pins (20 GPIOs), 6 analog input pins (with 12bit analog digital converters (ADC)), 1 analog output pin (with 10bit digital analog converter (DAC)), 2 serial ports (2 programmable Universal Asynchronous Receiver and Transmitters, UARTs). The board comes with an embedded lithium battery management chip and status indicator led, which allows to directly connect a 3.7V LiPo rechargeable battery that will be automatically recharged when the board is powered over its USB connector. The battery voltage level can be queried from analog input A4, the charging status (charging, full) from analog input A5. There is an on-board L70 GPS receiver module from the company Quectel Wireless Solutions and a small chip antenna. The Seeeduino LoRaWAN GPS module is available in German shops from around 37 € to 45 €.

Fig. 31: Seeeduino LoRaWAN GPS microcontroller from Seeed Studio.

The LoRa transmitter and receiver is encapsulated within an RHF76-052AM module from the Chinese company RisingHF. The RF module contains its own microcontroller, which implements the LoRaWAN protocol. The module is connected via the serial interface to the ATSAMD21G18 microcontroller and can be controlled by sending so-called 'AT' commands. The implemented LoRaWAN functionality is compatible with LoRaWAN Class A/C. The explanation of all supported commands as well as a number of examples on how to use the Seeeduino LoRaWAN are given on the Seeeduino LoRaWAN Wiki.

The board has 4 on-board Grove connectors. 'Grove' is a framework developed by the company Seeed Studio standardizing the connectors, operating voltages, and pin configurations for attaching peripherals like sensors, actuators, and displays to microcontrollers. The board has also the typical Arduino UNO connectors allowing to attach so-called Arduino shields (however, please note that the shields must be working with 3.3V; the normal operating voltage for the Arduino UNO microcontroller and its shields is 5V).

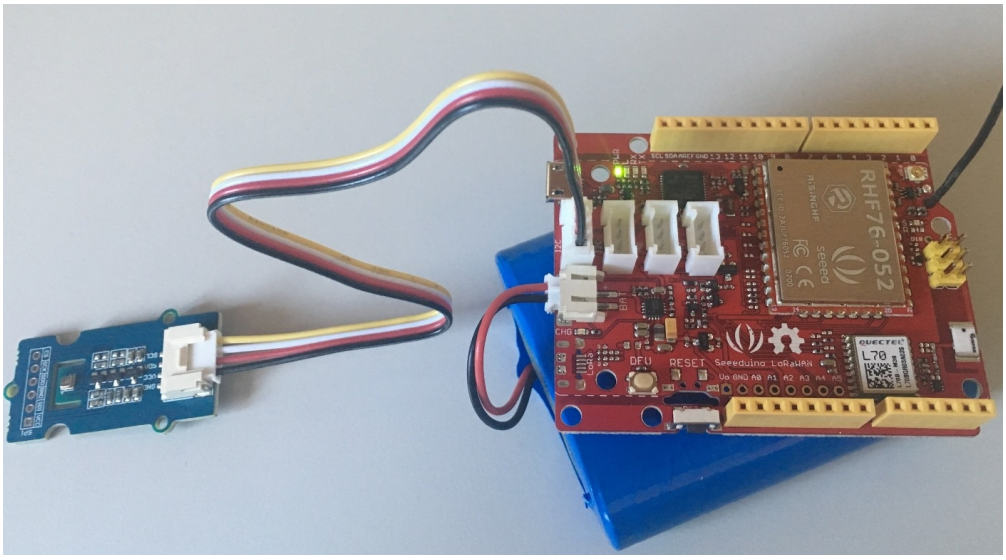


Fig. 32: The Seeeduino LoRaWAN GPS microcontroller with a 6600 mAh lithium polymer (LiPo) battery (bottom), and an attached BME280 temperature / humidity / barometer sensor module.

Sensor

We attached a Seeed BME280 Grove module with a Bosch BME280 sensor to the first I2C Grove connector of the Seeeduino LoRaWAN board. The BME280 measures temperature in the range -40 - 85 °C, with ±1.0°C accuracy; 0%-100% relative humidity with ±3% accuracy; and atmospheric pressure in the range 300 - 1100 hPa (1 hPa= one hundred Pa) with ±1.0 hPa accuracy. It offers the two interface standards I2C and SPI (we are using I2C here and the default I2C address 0x76). The atmospheric pressure changes with altitude, hence, the BME280 can also be used to measure the approximate altitude of a place.

2.10.2 Software

The sensor node has been programmed using the Arduino IDE. Please note, that in the Arduino framework a program is called a 'Sketch'.

In order to support the "Seeeduino LoRaWAN" board with the Arduino IDE, make sure to have installed the package "Seeed SAMD boards by Seeed Studio" in version 1.3.0 using the board manager in the Arduino IDE. This is also explained on a dedicated webpage from Seeed Studio. The sketch requires the software libraries "RTCZero", "Arduino_BME280", "Adafruit_Sensor", "Wire", and "LoRaWAN". The first three have to be installed using the library

manager of the Arduino IDE, the fourth library is already installed with the Arduino IDE and the latter library comes with the "Seeeduino LoRaWAN" board installation.

After the sketch has successfully established a connection to The Things Network it measures the air temperature, humidity, and pressure every 30 seconds. From the measured values also the altitude of the device (in meters above ground) is estimated. All four values are being encoded in two byte integer values each and then sent as an 8 bytes data packet to the respective TTN application using LoRaWAN port 8. Please note, that LoRaWAN messages can be addressed to ports 1-255 (port 0 is reserved); these ports are similar to port numbers 0-65535 when using the Internet TCP/IP protocol. Voltage, pressure, altitude, and humidity values are always greater or equal to 0, but the temperature value can also become negative. Negative values are represented as a two's complement; this must be considered in the Payload Decoding Function used in The Things Network (see below).

In between two sensor readings the microcontroller, the LoRaWAN module, and the sensor module are going into deep sleep mode to save battery power. During LoRaWAN data transmission the device draws up to 65mA current. When in sleep mode the entire node only draws around 0.06 mA power. Hence, with a 6600 mAh 3.7V LiPo battery and the current version of the sketch the system should be able to run for many years before recharging (not taking into account the self-discharging rate of the battery).

The source code is provided in the following section *Arduino Sketch for Seeeduino LoRaWAN with GPS sensor node*

2.10.3 Services

The services used for this sensor-node are:

- *TheThingsNetwork* service for LoRaWAN network service.
- *TheThingsNetwork - OGC SensorWeb* integration for uploading LoRaWAN sensor data into OGC infrastructure.

Registration of the sensor node with The Things Network (TTN)

The LoRaWAN protocol makes use of a number of different identifiers, addresses, keys, etc. These are required to unambiguously identify devices, applications, as well as to encrypt and decrypt messages. The names and meanings are nicely explained on a dedicated TTN web page.

The sketch given above connects the sensor node with The Things Network (TTN) using the Over-the-Air-Activation (OTAA) mode. In this mode, we use the three keys AppEUI, DevEUI, AppKey. The DevEUI should normally be delivered with the sensor node by the manufacturer. However, it seems that there is no explicit DevEUI provided with the Seeeduino LoRaWAN module. Therefore, it has to be generated automatically together with the other two keys using the TTN console. Each sensor node must be manually registered in the TTN console before it can be started. This assumes that you already have a TTN user account (which needs to be created otherwise). In the TTN console create a new device with also the DevEUI being automatically generated. After the registration of the device the respective keys (AppEUI, DevEUI, AppKey) can be copied from the TTN console and must be pasted into the the proper places in the source code of the sketch above. Please make sure that you choose for each of the three keys are in the correct byte ordering (all are in MSB, i.e. in the same ordering as given in the TTN console). A detailed explanation of these steps is given on this page. Then the sketch can be compiled and uploaded to the Seeeduino LoRaWAN microcontroller. Note that the three constants (AppEUI, DevEUI, AppKey) must be changed in the source code for every new sensor node.

Using the OTAA mode has the advantage over the ABP (activation by personalization) mode that during connection the session keys are newly created which improves security. Another advantage is that the packet counter is automatically reset to 0 both in the node and in the TTN application.

TTN Payload Decoding

Everytime a data packet is received by a TTN application a dedicated Javascript function is being called (Payload Decoder Function). This function can be used to decode the received byte string and to create proper Javascript objects or values that can directly be read by humans when looking at the incoming data packet. This is also useful to format the data in a specific way that can then be forwarded to an external application (e.g. a sensor data platform like MyDevices or Thingspeak).

Such a forwarding can be configured in the TTN console in the "Integrations" tab. *TTN payload decoder for Seeeduino LoRaWAN with GPS sensor node* given here checks if a packet was received on LoRaWAN port 33 and then assumes that it consists of the 10 bytes encoded as described above. It creates the five Javascript objects 'temperature', 'humidity', 'pressure', 'altitude', and 'vbattery'. Each object has two fields: 'value' holds the value and 'uom' gives the unit of measure. The source code can simply be copied and pasted into the 'decoder' tab in the TTN console after having selected the application. Choose the option 'Custom' in the 'Payload Format' field. Note that when you also want to handle other sensor nodes sending packets on different LoRaWAN ports, then the Payload Decoder Function can be extended after the end of the if (port==33) {...} statement by adding else if (port==7) {...} else if (port==8) {...} etc.

The Things Network - OGC SensorWeb Integration

The presented Payload Decoder Function works also with the TTN-OGC SWE Integration for the 52° North Sensor Observation Service (SOS). This software component can be downloaded from this repository. It connects a TTN application with a running transactional Sensor Observation Service 2.0.0 (SOS). Data packets received from TTN are imported into the SOS. The SOS persistently stores sensor data from an arbitrary number of sensor nodes and can be queried for the most recent as well as for historic sensor data readings. The 52° North SOS comes with its own REST API and a nice web client allowing to browse the stored sensor data in a convenient way.

We are running an instance of the 52° North SOS and the TTN-OGC SWE Integration. The web client for this LoRaWAN sensor node can be accessed on this webpage. Here is a screenshot showing the webclient:

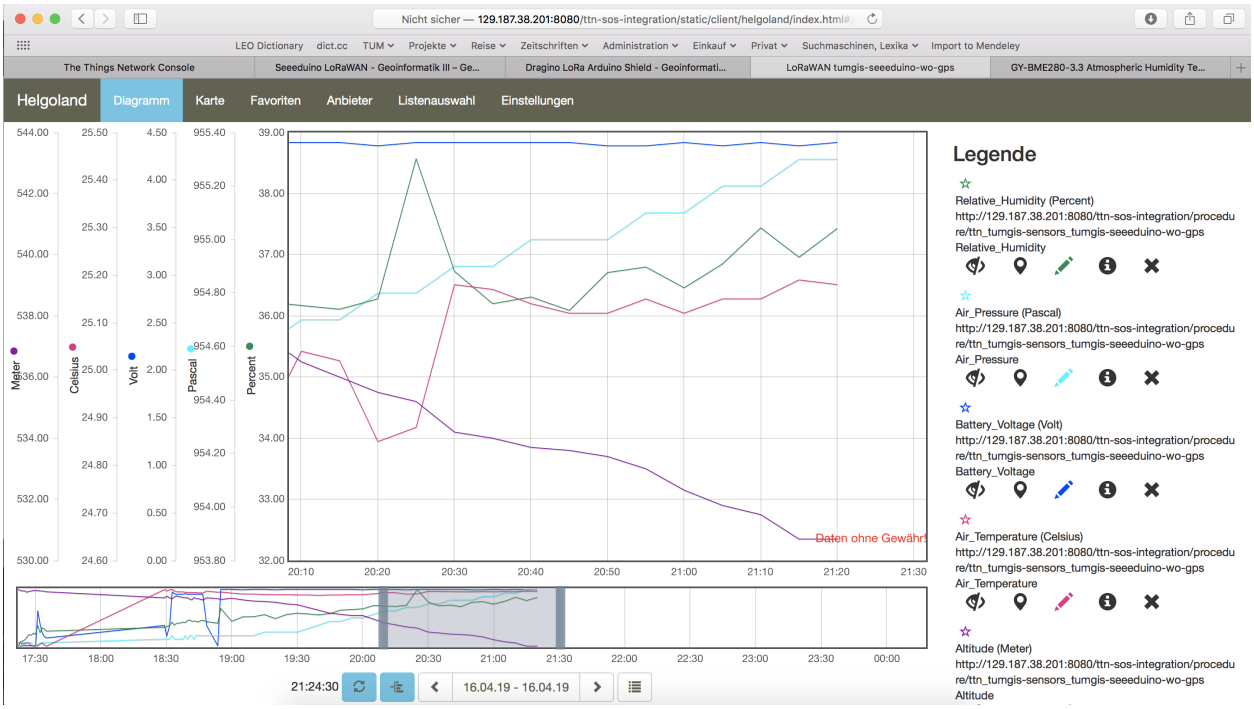


Fig. 33: Web client for data visualization

2.10.4 Code files

Listing 32: Arduino Sketch for Seeeduino LoRaWAN with GPS sensor node

```
/*******************************************************************************
 * Arduino Sketch for a LoRaWAN sensor node that is registered with
 * 'The Things Network' (TTN) www.thethingsnetwork.org
 *
 * Filename: Seeeduino_LoRaWAN_GPS_BME280_OTAA_Sleep_Adafruit_V2.ino
 *
 * Author:  Thomas H. Kolbe, thomas.kolbe@tum.de
 * Version: 1.0.1
 * Last update: 2019-04-17
 *
 * This sketch works with a Seeeduino LoRaWAN microcontroller board (with or
 * without embedded GPS module). See http://wiki.seeedstudio.com/Seeeduino_LoRAWAN/
 * It requires a Seeed Grove BME280 air temperature, relative humidity,
 * and air pressure sensor module attached to the I2C Grove connector of
 * the microcontroller board. The current configuration assumes that
 * the BME280 is configured to I2C device address 0x76 (default).
 * The sketch makes a connection to The Things Network (TTN) using
 * LoRaWAN in OTAA mode. It then sends a data packet of 10 bytes to
 * LoRaWAN port 33 around every 5 minutes. The packet contains the
 * following 5 integer values (16 bit, most significant byte (MSB) first):
 *   1. temperature in Celsius (signed, multiplied by 100)
 *   2. relative humidity in percent (unsigned, multiplied by 100)
 *   3. air pressure in Pascal (unsigned, divided by 10)
 *   4. current altitude in Meters (unsigned, multiplied by 10)
 *   5. battery voltage in millivolt (unsigned)
 * These values have to be decoded by the LoRaWAN network controller
 * using a proper "payload decoder function" written in Javascript.
 *
 * Note that when the board is powered over the USB connector and
 * no battery is connected, the measured battery voltage is incorrect.
 *
 * If the board shall be running on a lithium polymer (LiPo) battery,
 * it is recommended to remove the green power LED from the board or
 * to cut the connection between the LED and the resistor lying above
 * of it as the LED constantly draws around 8mW of power. In order to
 * save energy the sketch puts the GPS module on the board to standby
 * mode right from the beginning. After each measurement and data transfer
 * the LoRaWAN module and the sensor is put to standby mode, too, and the
 * microcontroller goes into deep sleep mode. All components require
 * a total current of around 0.34mA during sleep mode and up to 65mA
 * during LoRa transmission for the board version with GPS. The board
 * version without GPS only requires 0.06mA during sleep mode. Since the
 * entire system is mostly sleeping, the GPS board should be running
 * around 2 years on a 6600mAh LiPo battery before recharging
 * (6600mAh / 0.34mA / 24 = 808 days). The non GPS board version should
 * even run for more than 10 years...
 *
 * This code is based on example code given on the Seeeduino LoRaWAN
 * wiki page. It utilizes the Open Source libraries "Adafruit_BME280"
 * and "Adafruit_Sensor" provided by the company Adafruit and the
 * library "LoRaWan.h" provided by Seeed Studio.
 *******************************************************************************/
```

(continues on next page)

(continued from previous page)

```

#include <RTCZero.h>
#include <LoRaWan.h>
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>

// Keep the following line, if the board is a Seeeduino LoRaWAN with GPS,
// otherwise comment the line out

#define HAS_GPS 1

#define BME280_ADDRESS       (0x76)   // I2C device address of the BME280 sensor

// The barometer of the BME280 can also be used to estimate the current
// altitude of the device, if the air pressure at sea level (NN) is known.
// The following value has to be set to the current air pressure at NN (in hPa)
// in order to give reasonable altitude estimations. Note that this value is
// slowly changing over time. For Munich the current value can be obtained
// from https://www.meteo.physik.uni-muenchen.de/mesomikro/stadt/messung.php

#define SEALEVELPRESSURE_HPA (1017.8)

Adafruit_BME280 bme280;

RTCZero rtc;

unsigned char data[10];                 // buffer for the LoRaWAN data packet to be transferred
char buffer[256];                       // buffer for text messages received from the LoRaWAN module for display


void setup(void)
{
    digitalWrite(38, HIGH);             // Provide power to the 4 Grove connectors of the board

    for(int i = 0; i < 26; i ++)        // Set all pins to HIGH to save power (reduces the
    {                                   // current drawn during deep sleep by around 0.7mA).
        if (i!=13) {                    // Don't switch on the onboard user LED (pin 13).
          pinMode(i, OUTPUT);
          digitalWrite(i, HIGH);
        }
    }

    delay(5000);                        // Wait 5 secs after reset/booting to give time for potential upload
                                        // of a new sketch (sketches cannot be uploaded when in sleep mode)
    SerialUSB.begin(115200);            // Initialize USB/serial connection
    delay(500);
    // while(!SerialUSB);
    SerialUSB.println("Seeeduino LoRaWAN board started!");
```

(continues on next page)

(continued from previous page)

```
102
103     if(!bme280.begin(BME280_ADDRESS)) { // Initialize the BME280 sensor module
104       SerialUSB.println("BME280 device error!");
105     }
106
107     // Set the BME280 to a very low power operation mode (c.f. chapter 3.5
108     // "Recommended modes of operation" in the BME280 datasheet. See
109     // https://cdn-shop.adafruit.com/datasheets/BST-BME280_DS001-10.pdf );
110     // proper values can only be queried every 60s
111     bme280.setSampling(Adafruit_BME280::MODE_FORCED,
112                     Adafruit_BME280::SAMPLING_X16,  // temperature
113                     Adafruit_BME280::SAMPLING_X16,  // pressure
114                     Adafruit_BME280::SAMPLING_X16,  // humidity
115                     Adafruit_BME280::FILTER_OFF   );
116
117     // nrgSave.begin(WAKE_RTC_ALARM);
118     // rtc.begin(TIME_H24);
119
120 #ifdef HAS_GPS
121     Serial.begin(9600);                 // Initialize serial connection to the GPS␣
    ↪module
122     delay(500);
123     Serial.write("$PMTK161,0*28\r\n");  // Switch GPS module to standby mode as we don
    ↪'t use it in this sketch
124 #endif
125
126     lora.init();                        // Initialize the LoRaWAN module
127
128     memset(buffer, 0, 256);             // clear text buffer
129     lora.getVersion(buffer, 256, 1);
130     memset(buffer, 0, 256);             // We call getVersion() two times, because␣
    ↪after a reset the LoRaWAN module can be
131     lora.getVersion(buffer, 256, 1);    // in sleep mode and then the first call only␣
    ↪wakes it up and will not be performed.
132     SerialUSB.print(buffer);
133
134     memset(buffer, 0, 256);
135     lora.getId(buffer, 256, 1);
136     SerialUSB.print(buffer);
137
138     // The following three constants (AppEUI, DevEUI, AppKey) must be changed
139     // for every new sensor node. We are using the LoRaWAN OTAA mode (over the
140     // air activation). Each sensor node must be manually registered in the
141     // TTN console at https://console.thethingsnetwork.org before it can be
142     // started. In the TTN console create a new device with the DevEUI also
143     // being automatically generated. After the registration of the device the
144     // three values can be copied from the TTN console. A detailed explanation
145     // of these steps is given in
146     // https://learn.adafruit.com/the-things-network-for-feather?view=all
147
148     // The EUIs and the AppKey must be given in big-endian format, i.e. the
149     // most-significant-byte comes first (as displayed in the TTN console).
150     // For TTN issued AppEUIs the first bytes should be 0x70, 0xB3, 0xD5.
151
152     // void setId(char *DevAddr, char *DevEUI, char *AppEUI);
153     lora.setId(NULL, "xxxxxxxxxxxxxxxx", "yyyyyyyyyyyyyyyy");
154
```

(continues on next page)

(continued from previous page)

```
    // setKey(char *NwkSKey, char *AppSKey, char *AppKey);
    lora.setKey(NULL, NULL, "zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz");

    lora.setDeciveMode(LWOTAA);           // select OTAA join mode (note that setDeciveMode is not a typo; it is misspelled in the library)
    // lora.setDataRate(DR5, EU868);          // SF7, 125 kbps (highest data rate)
    lora.setDataRate(DR3, EU868);         // SF9, 125 kbps (medium data rate and range)
    // lora.setDataRate(DR0, EU868);          // SF12, 125 kbps (lowest data rate, highest max. distance)

    // lora.setAdaptiveDataRate(false);
    lora.setAdaptiveDataRate(true);       // automatically adapt the data rate

    lora.setChannel(0, 868.1);
    lora.setChannel(1, 868.3);
    lora.setChannel(2, 868.5);
    lora.setChannel(3, 867.1);
    lora.setChannel(4, 867.3);
    lora.setChannel(5, 867.5);
    lora.setChannel(6, 867.7);
    lora.setChannel(7, 867.9);

    // The following two commands can be left commented out;
    // TTN works with the default values. (It also works when
    // uncommenting the commands, though.)
    // lora.setReceiceWindowFirst(0, 868.1);
    // lora.setReceiceWindowSecond(869.525, DR0);

    lora.setDutyCycle(false);             // for debugging purposes only - should normally be activated
    lora.setJoinDutyCycle(false);         // for debugging purposes only - should normally be activated

    lora.setPower(14);                    // LoRa transceiver power (14 is the maximum for the 868 MHz band)

    // while(!lora.setOTAAJoin(JOIN));
    while(!lora.setOTAAJoin(JOIN,20));    // wait until the node has successfully joined TTN

    lora.setPort(33);                     // all data packets are sent to LoRaWAN port 33
}

void loop(void)
{
    bool result = false;
    float temperature, altitude, pressure, humidity;
    int16_t int16_temperature, int16_humidity, int16_pressure, int16_altitude, int16_vbat;

    bme280.takeForcedMeasurement();       // wake the sensor up for the next readings

    //get and print temperatures
    SerialUSB.print("Temp: ");
    SerialUSB.print(temperature = bme280.readTemperature());
```

(continues on next page)

(continued from previous page)

```
    SerialUSB.print("C  ");

    //get and print atmospheric pressure data
    SerialUSB.print("Pressure: ");
    SerialUSB.print(pressure = bme280.readPressure());
    SerialUSB.print("Pa  ");

    //get and print altitude data
    SerialUSB.print("Altitude: ");
    SerialUSB.print(altitude = bme280.readAltitude(SEALEVELPRESSURE_HPA));
    SerialUSB.print("m  ");

    //get and print humidity data
    SerialUSB.print("Humidity: ");
    SerialUSB.print(humidity = bme280.readHumidity());
    SerialUSB.print("%  ");

    //get and print battery voltage
    SerialUSB.print("VBat: ");
    SerialUSB.print(int16_vbat=lora.getBatteryVoltage());
    SerialUSB.println("mV");

    int16_temperature = temperature*100.0;
    int16_humidity = humidity*100.0;
    int16_pressure = pressure/10.0;
    int16_altitude = altitude*10.0;

    data[0] = (byte) (int16_temperature >> 8);
    data[1] = (byte) (int16_temperature & 0x00FF);
    data[2] = (byte) (int16_humidity >> 8);
    data[3] = (byte) (int16_humidity & 0x00FF);
    data[4] = (byte) (int16_pressure >> 8);
    data[5] = (byte) (int16_pressure & 0x00FF);
    data[6] = (byte) (int16_altitude >> 8);
    data[7] = (byte) (int16_altitude & 0x00FF);
    data[8] = (byte) (int16_vbat >> 8);
    data[9] = (byte) (int16_vbat & 0x00FF);

    result = lora.transferPacket(data, 10, 5);   // send the data packet (10 bytes)␣
↪with a default timeout of 5 secs

    if(result)
    {
        short length;
        short rssi;

        memset(buffer, 0, 256);
        length = lora.receivePacket(buffer, 256, &rssi);

        if(length)
        {
            SerialUSB.print("Length is: ");
            SerialUSB.println(length);
            SerialUSB.print("RSSI is: ");
            SerialUSB.println(rssi);
            SerialUSB.print("Data is: ");
            for(unsigned char i = 0; i < length; i ++)
```

(continues on next page)

(continued from previous page)

```
            {
                SerialUSB.print("0x");
                SerialUSB.print(buffer[i], HEX);
                SerialUSB.print(" ");
            }
            SerialUSB.println();
        }
    }

    lora.setDeviceLowPower();      // bring the LoRaWAN module to sleep mode
    doSleep((5*60-8)*1000);        // deep sleep for 292 secs (+ 3 secs transmission time + 5 secs timeout = 300 secs period)
    lora.setPort(33);              // send some command to wake up the LoRaWAN module again
}

// The following function implements deep sleep waiting. When being called the
// CPU goes into deep sleep mode (for power saving). It is woken up again by
// the CPU-internal real time clock (RTC) after the configured time.
//
// A similar function would also be available in the standard "ArduinoLowPower" library.
// However, in order to be able to use that library with the Seeeduino LoRaWAN board,
// four files in the package "Seeed SAMD boards by Seeed Studio Version 1.3.0" that is
// installed using the Arduino IDE board manager need to be patched. The reason is that
// Seeed Studio have not updated their files to a recent Arduino SAMD version yet
// and the official "ArduinoLowPower" library provided by the Arduino foundation is
// referring to some missing functions. For further information see here:
// https://forum.arduino.cc/index.php?topic=603900.0 and here:
// https://github.com/arduino/ArduinoCore-samd/commit/b9ac48c782ca4b82ffd7e65bf2c956152386d82b

void doSleep(uint32_t millis) {
    if (!rtc.isConfigured()) {     // if called for the first time,
        rtc.begin(false);          // then initialize the real time clock (RTC)
    }

    uint32_t now = rtc.getEpoch();
    rtc.setAlarmEpoch(now + millis/1000);
    rtc.enableAlarm(rtc.MATCH_HHMMSS);

    rtc.standbyMode();             // bring CPU into deep sleep mode (until woken up by the RTC)
}
```

Listing 33: TTN payload decoder for Seeeduino LoRaWAN with GPS sensor node

```
function Decoder (bytes, port) {
  var result = {};
  var transformers = {};

  if (port==33) {
    transformers = {
      'temperature': function transform (bytes) {
```

(continues on next page)

(continued from previous page)

```
          value=bytes[0]*256 + bytes[1];
          if (value>=32768) value=value-65536;
          return value/100.0;
        },
      'humidity': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/100.0;
        },
      'pressure': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/10.0;
        },
      'altitude': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/10.0;
        },
      'vbattery': function transform (bytes) {
          return (bytes[0]*256 + bytes[1])/1000.0;
        }
    }

    result['temperature'] = {
      value: transformers['temperature'](bytes.slice(0, 2)),
      uom: 'Celsius',
    }

     result['humidity'] = {
      value: transformers['humidity'](bytes.slice(2, 4)),
      uom: 'Percent',
    }

    result['pressure'] = {
      value: transformers['pressure'](bytes.slice(4, 6)),
      uom: 'hPa',
    }

    result['altitude'] = {
      value: transformers['altitude'](bytes.slice(6, 8)),
      uom: 'Meter',
    }

    result['vbattery'] = {
      value: transformers['vbattery'](bytes.slice(8, 10)),
      uom: 'Volt',
    }
  }

  return result;
}
```

2.10.5 References

- Seeeduino LoRaWAN GPS microcontroller
- Seeeduino LoRaWAN Wiki with instructions
- A short presentation on LoRaWAN basics and using the Seeeduino LoRaWAN board
- L70 GPS receiver module from the company Quectel Wireless Solutions

- Adafruit GPS library (can be used with the L70 GPS module)

On battery saving / using the deep sleep mode

- Adafruit Feather 32u4 LoRa - long transmission time after deep sleep - End Devices (Nodes) - The Things Network
- Full Arduino Mini LoraWAN and 1.3uA Sleep Mode - End Devices (Nodes) - The Things Network
- Adding Method to Adjust hal_ticks Upon Waking Up from Sleep · Issue #109 · matthijskooijman/arduino-lmic
- minilora-test/minilora-test.ino at cbe686826bd84fac8381de47b5f5b02dd47c2ca0 · tkerby/minilora-test
- Arduino-LMIC library with low power mode - Mario Zwiers

2.11 Sodaq ONE

2.11.1 Hardware

Micro-controller

Fig. 34: SODAQ ONE-EU-RN2483-V3 from SODAQ. SODAQ ONE tutorial with explanations, schematics, datasheets, and examples. SODAQ ONE module pinout

The SODAQ ONE-EU-RN2483-V3 is a very compact module operated by the 32bit ATSAMD21G18 microcontroller running at 48 MHz. It has 256 KB flash memory (to store the program code) and 32 KB of RAM (to store variables, status information, and buffers), and up to 16 KB of emulated EEPROM (to store configuration data). The operating voltage of the board is 3.3V (this is important when attaching sensors and other peripherals; they also must operate on 3.3V). The board offers 14 general purpose analog/digital input/output pins (14 GPIOs) all of which can also be used for PWM output, and one of which can be used as an analog output pin (with 10bit digital analog converter (DAC)). The microcontroller has three serial ports (programmable Universal Asynchronous Receiver and Transmitter, UART); the first is connected internally via a USB/Serial converter to the USB port of the board, the second is connected internally to the LoRaWAN module, and the third is freely usable using GPIO pins D12 (TX) and D13 (RX). Furthermore, the board has one I2C port and four of the GPIO lines can be used as SPI port. The SODAQ ONE comes with an embedded Lithium polymer battery management chip and status indicator led, which allows to directly connect a 3.7V LiPo rechargeable battery that will be automatically recharged when the board is powered over its USB connector. Above, a solar charge controller is embedded allowing to directly connect a photovoltaic panel (4.5V to 6V) to recharge the battery. The board also features an RGB LED, a LSM303AGR module (3 axis magnetometer and 3 axis accelerometer), and an uBlox EVA 8M GPS module. The SODAQ ONE board is available from the manufacturer in Rotterdam, The Netherlands, for around 100 €.

The LoRa transmitter and receiver is encapsulated within a Microchip RN2483 LoRaWAN module. It uses the LoRa chip SX1276 from the company Semtech and is dedicated to the 868 MHz frequency band. The RF module contains its own microcontroller, which implements the LoRaWAN protocol. The module is connected via the serial interface to the ATSAMD21G18 microcontroller and can be controlled by sending text commands. The implemented LoRaWAN functionality is compatible with LoRaWAN Class A. The detailed explanation of the module is given in the RN2483 datasheet and all supported commands in the RN2483 Command Reference.

The SODAQ ONE is installed on a ONE Base board offering 7 on-board Grove connectors. 'Grove' is a framework developed by the company Seeed Studio standardizing the connectors, operating voltages, and pin configurations for attaching peripherals like sensors, actuators, and displays to microcontrollers. Please note that grove modules to be used with the SODAQ ONE must be working with 3.3V (the normal operating voltage for the Arduino UNO microcontroller and its shields is 5V).

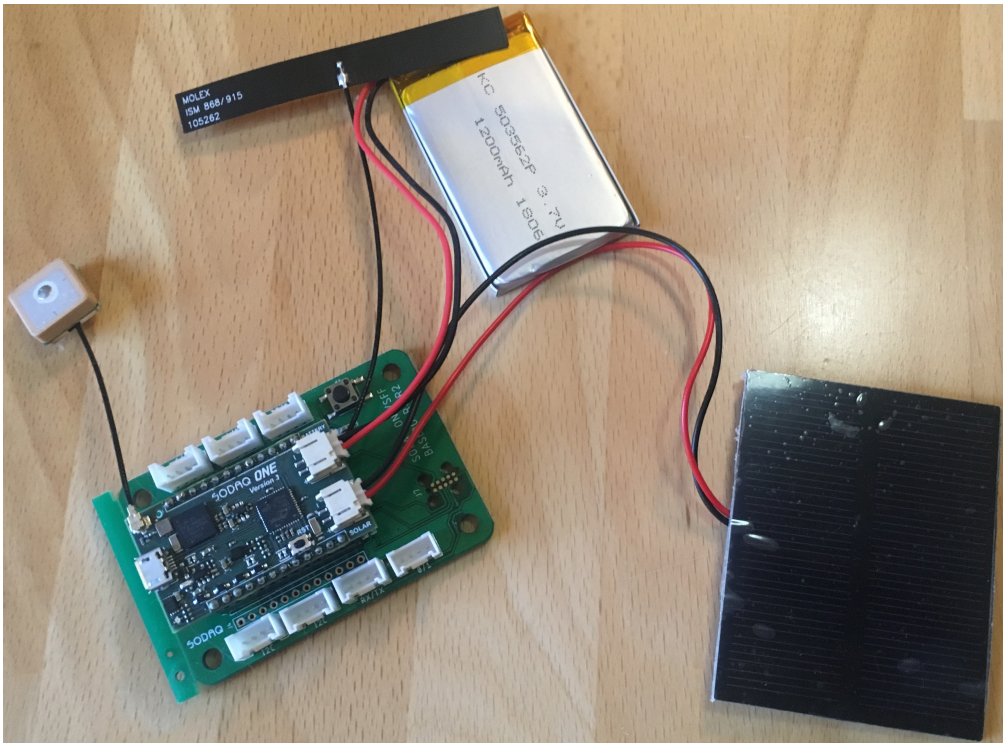


Fig. 35: The SODAQ ONE-EU-RN2483-V3 installed on a ONE Base board with attached GPS antenna (left), LoRa patch antenna (top), a 1200 mAh lithium polymer (LiPo) battery (top), and an attached 0.5W photovoltaic panel.

Sensor

The embedded LSM303AGR is an ultra-low-power 3D electronic compass featuring linear acceleration full scales of ±2g/±4g/±8g/±16g and a magnetic field dynamic range of ±50 gauss. The module can be configured to generate an interrupt signal for free-fall, motion detection and magnetic field detection. It is connected to the microcontroller via I2C bus. When the SODAQ ONE is used to track moving objects (like bikes, people, or animals), the LSM303AGR sensor can be used to wake up the microcontroller and the GPS from power saving mode (deep sleep). This way, the power consumption of the SODAQ ONE can be minimized (only when a movement has been detected, a new GPS position fix has to be acquired). The detailed explanation of the module is given in the LSM303AGR datasheet.

The embedded uBlox EVA 8M GPS module is a standard precision GNSS receiver with 72 channels supporting GPS and GLONASS. The module is capable to report up to 18 positions per second (18 Hz). It is connected to the microcontroller via I2C bus and supports different power saving modes. A detailed explanation of the module is given in the uBlox EVA 8M datasheet.

2.11.2 Software

The section is still to be written.

2.11.3 Services

The section is still to be written.

2.11.4 Code files

Listing 34: Arduino Sketch for Sodaq ONE sensor node

```
No source code yet!
```

Listing 35: TTN payload decoder for Sodaq ONE sensor node

```
Yet to be written
```

2.11.5 References

- SODAQ ONE-EU-RN2483-V3 microcontroller board
- SODAQ ONE tutorial, schematics, libraries, examples
- SODAQ ONE Base board
- SODAQ main github page
- Microchip RN2483 LoRaWAN module datasheet
- Microchip RN2483 LoRaWAN module command reference
- SODAQ RN2483 library
- RN2483 Firmware Updater
- LSM303AGR module datasheet
- SODAQ LSM303AGR library
- uBlox EVA 8M GPS module
- SODAQ uBlox GPS library
- SODAQ Universal Tracker
- SODAQ ONE Tracker v3

2.12 Wemos TTGO T-Beam

2.12.1 Hardware

Fig. 36: TTGO T-Beam from Wemos. TTGO T-Beam pinout, example code.

Micro-controller

The Wemos TTGO T-Beam is especially suited for mobile operations, because it has an on-board battery holder for an 18650 lithium polymer (LiPo) battery, a GPS receiver, a LoRa transceiver module (using the LoRa chip SX1276) dedicated to the 868 MHz frequency band, and an embedded battery charger. The module is operated by the Espressif ESP32 microcontroller board, which contains a dual-core Xtensa 32bit LX6 processor running with up to 240MHz, 4 MB of flash memory (to store the program code and some files within a file system), and 520 KB of RAM (to store variables, status information, and buffers). The ESP32 module also has built-in WiFi and Bluetooth LE connectivity. In addition, the TTGO T-Beam has 4 MB of PSRAM (pseudo static RAM) that is used as a memory extension for the ESP32. The operating voltage of the board is 3.3V (this is important when attaching sensors and other peripherals; they also must operate on 3.3V). The board offers 18 general purpose input/output pins (18 GPIOs), from which up to 12 can be used as analog input pins (with 12bit analog digital converters (ADC)) and one as analog output pin (8bit digital analog converter (DAC)). Some GPIO pins can be used as serial port (programmable Universal Asynchronous Receiver and Transmitter, UART), I2C port, SPI port, and I2S port. The USB port is connected internally via a USB/Serial converter to another serial port (UART). The WiFi and Bluetooth antenna are mounted on the TTGO T-Beam board. A small GPS antenna is connected via a pigtail to an U.FL / IPX connector. The LoRa antenna has to be connected via an SMA-type connector. The TTGO T-Beam is available from Chinese sellers for around 23 € (I have not found a European shop where it can be bought yet).

The LoRa transmitter and receiver is encapsulated within a LoRa module. It uses the LoRa chip SX1276 from the company Semtech and is dedicated to the 868 MHz frequency band. The LoRa module is connected via SPI interface to the microcontroller and all of the required connections of the LoRa transceiver pins with the microcontroller are already built-in on the TTGO T-Beam board. Since the module only implements the LoRa physical layer, the LoRaWAN protocol stack must be implemented in software on the microcontroller. We are using the Arduino library LMIC for that purpose (see below). The implemented LoRaWAN functionality is compatible with LoRaWAN Class A/C.

Sensor

While the GPS module of the TTGO T-Beam is labelled as a uBlox NEO-6 module, this does not seem to be true. The module seems to be of a Chinese brand instead that is not fully compatible with the uBlox NEO-6.

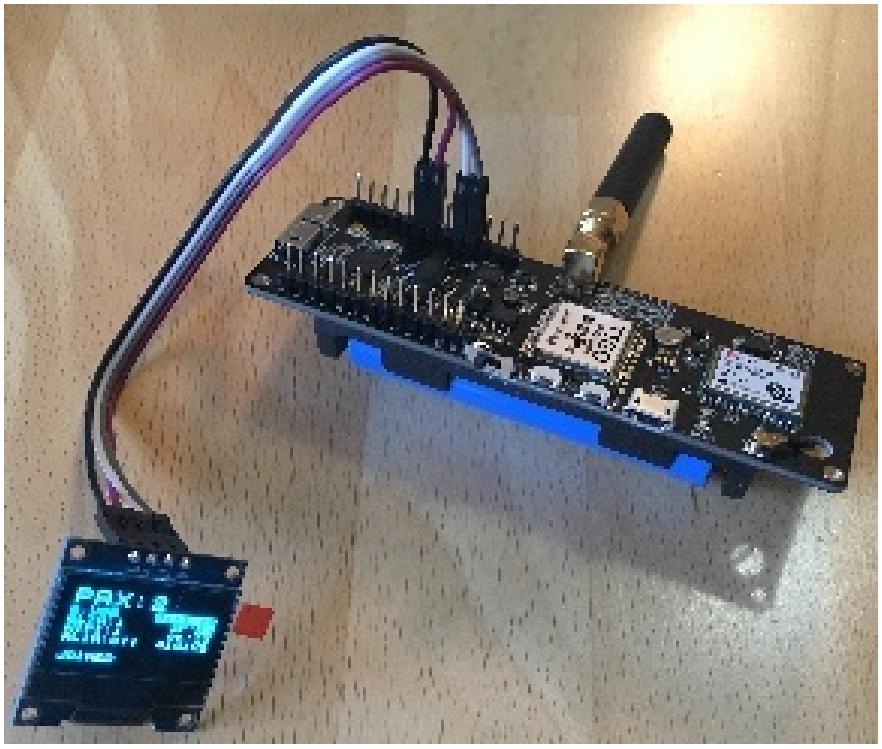


Fig. 37: The Wemos TTGO T-Beam with a 2200 mAh 18650 lithium polymer (LiPo) battery (in the battery holder on the backside of the board) with attached OLED display.

The embedded uBlox EVA 8M GPS module is a standard precision GNSS receiver with 72 channels supporting GPS and GLONASS. The module is capable to report up to 18 positions per second (18 Hz). It is connected to the microcontroller via I2C bus and supports different power saving modes. A detailed explanation of the module is given in the uBlox EVA 8M datasheet.

Display

We have attached a 0.96 in monochrome OLED display to the I2C bus of the TTGO T-Beam module. The display is using an SD1306 controller and has a resolution of 64 x 64 pixels.

2.12.2 Software

The section is still to be written.

2.12.3 Services

The section is still to be written.

2.12.4 Code files

Listing 36: Arduino Sketch for Wemos TTGO T-Beam sensor node

```
No source code yet!
```

Listing 37: TTN payload decoder for Wemos TTGO T-Beam sensor node

```
Yet to be written
```

2.12.5 References

- Wemos TTGO T-Beam pinout, libraries, examples
- TTGO T-Beam TinyMicros Wiki (links to schematics, GPS datasheet)
- TTGO T-Beam – Kompakter Knochen zum Mappen
- TTNMapper on the TTGO T-Beam
- TTNMapper on the TTGO T-Beam with Deep Sleep
- Wifi & BLE driven passenger flow metering with cheap ESP32 boards

CHAPTER 3

Indices and tables

- genindex
- search