Developer Manual

1.6.1

Envira Sostenible S.A.

Nanoenvi IAQ

envira IOT

Table of Contents

List of Figures

List of Tables

Chapter 1. Version control

Table 1.1. Version control

Version	Changes
V1.6.1	Drawing up of integration guide for Nanoenvi IAQ LoRa

Chapter 2. Scope of document

This document details aspects of the integration of Nanoenvi™ IAQ into LoRaWAN and/or the IoT platform, downlink message protocol and information on measurement deserialization.

Chapter 3. Introduction

Nanoenvi™ IAQ has a LoRaWAN communication variant. This allows the device to be installed in buildings with LoRaWAN network coverage, where real-time air quality measurement is required separately from general purpose networks such as WiFi networks or those without wired network infrastructures, such as MODBUS.

Nanoenvi™ IAQ requires installation in order to operate: installation of the device consists of setting up the connection and placing it in the location where measurements will be taken. It requires both the Network Server and Nanoenvi™ IAQ to have the same configuration. For further details on configuration in accordance with the Network Server used, please refer to the Network Server documentation or get in touch with your LoRaWAN network operator. For further details on the installation, configuration and operation process of the device, please refer to the Nanoenvi™ IAQ user manual.

After installation, during normal operation the LoRaWAN variant of Nanoenvi™ IAQ takes real-time indoor air quality data and sends it via the LoRaWAN network configured. The variables measured by the device are serialized in protobuf format.

Nanoenvi™ IAQ also supports RPC (remote procedure calls) messages sent via downlink messages to edit their configuration and see their status. The serialization format for both downlink and uplink messages is protobuf. (`https://developers.google.com/protocol-buffers`).

To make use of the data and/or send remote messages, device integration is required: integration consists of deploying the necessary services or configuring the Network Server so that the sensor measurements received in the Network Server can be used by a different system. For example, if sensor measurements are to be displayed on a visualization platform, messages received on the Network Server should be deserialized, converted to the format expected by the visualization platform and forwarded from the Network Server to the platform.

This manual documents all the information necessary for integrating Nanoenvi™ IAQ.

Chapter 4. Requirements for the integration of Nanoenvi™ IAQ LoRaWAN

Below is what you have to do in order to integrate Nanoenvi™ IAQ:

4.1. Usage of sensor data

In order to use the data taken and sent by the device you need:

- A device that deserializes the payload of the message sent by the device. The payload is serialized with protobuf.
- A device to forward Network Server data to the system on which it is to be used.

4.2. Acting on the device with RPCs

Acting on the device with RPCs is optional. It requires:

- The implementation of RPC serialization and composition.
- A device that implements the logic of sending an RPC and receiving the response compatible with the RPC communications model: wait to receive a message with sensor measurements; once you have received it, send the RPC. You should then wait for the response from the RPC, which will take a few seconds to be received.
- The implementation of the deserialization of RPC responses.

Chapter 5. Sensor measurement message format

5.1. Sensor measurement message format

The measurements taken by the Nanoenvi™ IAQ sensors are serialized with protobuf in a message with the following structure (described in proto3 language):

```
syntax = "proto3"; //Proto3 syntax

/**
 * Sensor measurements message format
 */
message SensorMeasurements {
    uint32 msg_ver = 1;  /// Message version to track changes. This number is included
                         // along all the SensorMeasurements versions and increased
                         // each time message format changes.
    float  co2     = 2;  /// CO2 measurement value in ppm units
    float  voc     = 3;  /// VOC measurement value in ppm units
    float  co      = 4;  /// CO measurement value in ppm units
    float  noise   = 5;  /// Noise level percentage
    float  pm10    = 6;  /// PM10 concentration in ug/m3
    float  pm2_5   = 7;  /// PM2.5 concentration in ug/m3
    float  temp    = 8;  /// Temperature in Celsius degrees
    float  hum     = 9;  /// Realtive humidity in % units
    float  prb     = 10; /// Barometric pressure in HPa units
    float  pm1     = 11; /// PM1 concentration in ug/m3
    float  pm4     = 12; /// PM4 concentration in ug/m3
    float  iaqi    = 13; /// Indoor air quality index
    float  tci     = 14; /// Thermal comfort index
    float  eiaqi   = 15; /// Environmental indoor air quality index
}
```

The CO and noise values are optional, depending on the device variant: due to hardware restrictions, Nanoenvi™ IAQ™ can only have one of the two sensors (either CO or noise).

The measurement message is sent approximately every 70 seconds and (just like any other type of LoRAWAN uplink message) is received on the Network Server. From the Network Server the message can be forwarded and/or deserialized. Please refer to the documentation of your Network Server.

Chapter 6. Configuration Messages

6.1. RPC

Nanoenvi™ IAQ implements remote procedure calls (RPC). Remote procedure calls are downlink messages that Nanoenvi™ IAQ can receive, interpret and execute. The messages implemented allow changes in the configuration, reading the configuration, calibrating the device and acting on it (e.g. resetting it).

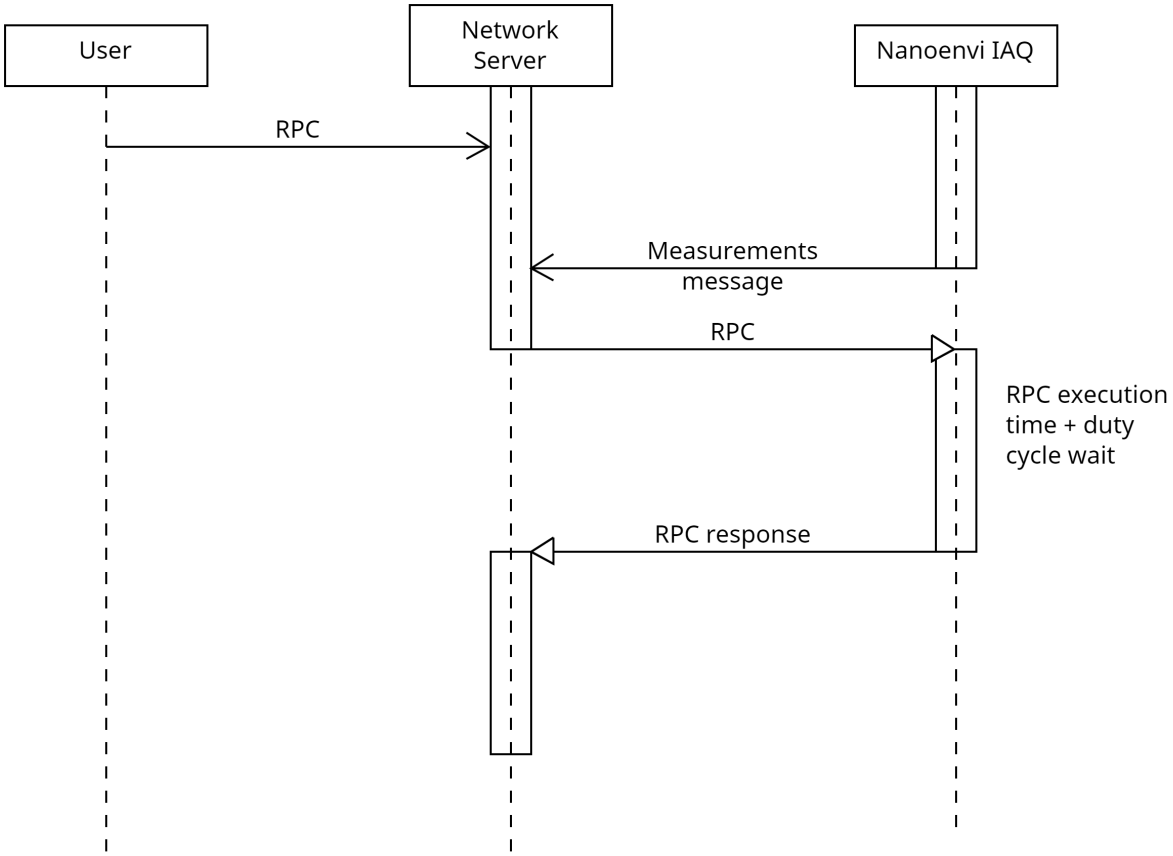
6.1.1. RPC communication model

Once connected to the LoRaWAN network, Nanoenvi™ IAQ takes sensor measurements and sends the results. As a class A LoRaWAN device, a window for listening to downlink messages opens immediately after a measurement is sent. If a downlink message is received, Nanoenvi™ IAQ interprets the message payload and if it contains a supported RPC with valid arguments and format, it executes it: RPCs can change the configuration, request information and act on the device. After the execution of the RPC, Nanoenvi™ IAQ generates a response (with the result of the RPC execution), waits the time corresponding to the duty cycle and sends this response through the LoRaWAN network (the response will therefore be received on the Network Server).

If the downlink message contains a payload with formatting errors, an unsupported RPC or a failure related to its interpretation or execution, the device will wait for the time corresponding to the duty cycle and then send an error message.

The following Figure shows the sequence of message sending between Network Serve and RPC and Nanoenvi™ IAQ when the user sends an RPC:

Figure 6.1. RPC communication model



Even though it is possible to send consecutive messages to the device before it responds to the first RPC sent, this is not recommended and may result in an unstable response from the device. It is therefore recommended to respect the communications model and wait for the response to each message before sending the next RPC.

RPC messages are processed in order of receipt and there are messages that cause the device to reset. It should be noted that after a reset the device will take some time to reconnect to the LoRaWAN network. .

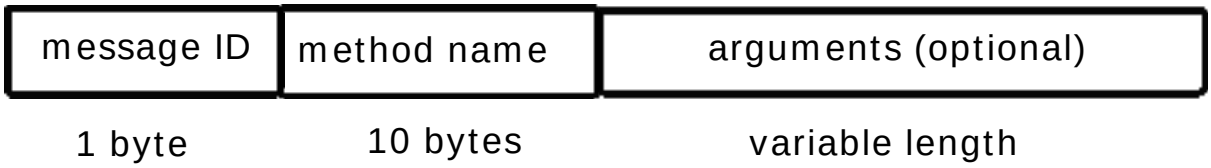
This communication model should be taken into account in order to integrate Nanoenvi™ IAQ into a Network Server. If you wish to allow the sending of RPCs to the device you should deploy a mechanism that allows the sending of an RPC right after receiving a message with sensor measurements, waiting for the RPC response, decoding it and reporting it.

6.2. RPC format

6.2.1. Format for RPCs sent to Nanoenvi™ IAQ

The payload of RPC messages in binary format that Nanoenvi™ IAQ expects to receive should have the following structure:

Figure 6.2. RPC message format

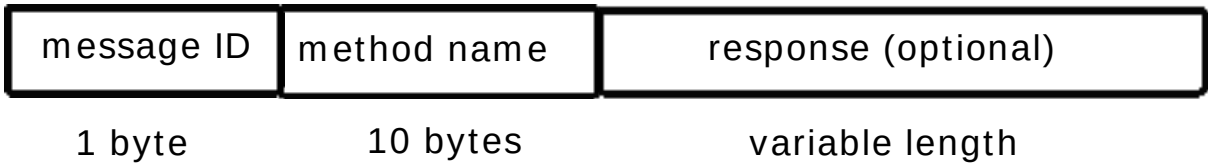


- ***message ID***: 1 byte corresponding to the message ID: a number between 0 and 255 that identifies the message. Its use is just to identify messages, but in practice it is not used by the device.
- **method name**: the name of the RPC method. It should be 10 bytes long. If the length of the method name is less than 10 bytes, the remaining bytes should be 0x01 in size.
- **arguments**: Contains the arguments of the RPC method serialized with Protobuf. Since some methods have no arguments, this field is optional.

6.2.2. Format for RPC responses sent by Nanoenvi™ IAQ

The response to RPC messages in binary format generated by Nanoenvi™ IAQ after receiving and processing an RPC message consists of a variable number of bytes in the following format:

Figure 6.3. RPC response format



- **message ID**: 1 byte corresponding to the ID of the message to which the device is responding.
- **method name**: the name of the RPC method. It should be 10 bytes long. If the length of the method name is less than 10 bytes, the remaining bytes should be 0x01 in size. It matches the name of the message received, processed and replied to by Nanoenvi™ IAQ.

- **response**: Values returned in the message as a response from the RPC method. They are variable in length and serialized with Protobuf.

6.2.2.1. Generic OK response

For some RPCs, a generic OK response is returned with the following values:

- **message ID**: byte with the same value as the RPC message being replied to.
- **method name**: 10 bytes with the same value as the method name of the RPC message being replied to.
- **response:** for a generic OK response, this field contains “OK” serialized with protobuf in a message with the following structure (described in proto3 language):

```
/**
 * Protobuf RPC genric OK response
 */
syntax = "proto3"; //Proto3 syntax

/**
 * RPC OK generic response
 */
message RpcOkArgs {
    bytes message =1; /// String to return OK response
}
```

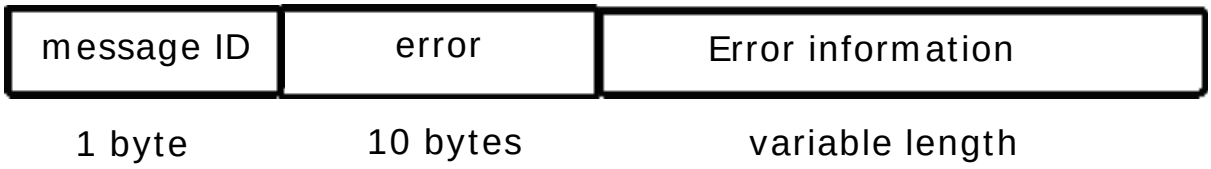
The response has only one field:

- **message:**always expected to contain “OK”

6.2.3. Error responses

If there is an error in the interpretation of an RPC Nanoenvi™ IAQ message, it will respond with different messages depending on the error. They all have the same structure:

Figure 6.4. Format of RPC error response



- **message ID**: 1 byte corresponding to the ID of the message that generated the error.
- **error**: 10 bytes showing that this is an error message. The value of the 10 bytes will be: `0x65 0x72 0x72 0x6F 0x72 0x01 0x01 0x01 0x01 0x01`
- **error information**: Information about the type of error serialized in protobuf format. Its length is variable but less than 30 bytes. It has two fields:
  - **code**: full 32-bit number with error code.
  - **message**: chain of characters holding information about the error.

Below is the structure of an error message described in proto3 language:

```
syntax = "proto3";
```

```
/**
 * RPC error response fields
 */
message RpcErrorArgs {
    int32 error_code = 1;  /// Error code
    bytes message =2; /// Error message (16 bytes max.)
}
```

The following table shows possible errors and their meaning:

error_code	message	error
-32602	bad params	The message received has the wrong parameters (out of range).
-32601	method not found	The device is not implementing the RPC method
-32700	parse error	The message received contains serialization errors or an incorrect format
-32600	inval request	The message received does not correspond to a valid RPC request
-32000	aborted	The process was aborted on processing and executing the RPC
-32001	busy	When the RPC was being executed use was requested of a resource that was busy
-32002	timeout	When the RPC was being executed there was a timeout
-32003	denied	The execution of the RPC was denied
-32004	not initalized	When the RPC was being executed use was requested of a resource that had not been initialized
-32005	failure	When the RPC was being executed there was a failure
-32006	forbidden	The RPC requested a forbidden operation
-32007	unknown	When the RPC was being executed there was an unknown error

6.3. List of public RPCs

The public RPC methods supported by the device are listed below:

- **Co2Frc**: The public RPC methods supported by the device are listed below:

- **reset**: resets the device.
- **SetOtaa**: edits the configuration of AppKey and AppEUI/JoinEUI.
- **FactoryCfg**: Resets the configuration to factory values.

If AppKey and/or AppEUI/JoinEUI as configured on the Network Server have values different from the default values, invoking this method will lead to the connection to the LoRaWAN network being lost.

- **FwData**: consults metadata of the firmware version.
- **DevInfo**: consults the date of manufacture and hardware version.

6.4. Public RPC documentation

6.4.1. Co2Frc

This method enables a reference concentration for the CO2 sensor in order to calibrate it. The device's CO2 sensor may give erroneous measurements if it is exposed to very high amounts of CO2 continuously for more than 7 days or if the device suffers an impact.

To recalibrate the sensor, you can either expose it to a known amount of CO2 or take the measurement from a calibrated reference meter and provide the actual CO2 concentration using the RPC Co2Frc method. If the CO2 concentration is not known (either because you do not have the ability to generate a reference CO2 concentration or because you do not have a reference meter), you can install the device outside for 5 minutes and assume that the actual CO2 concentration is 400 ppm.

6.4.1.1. Downlink message

Downlink message fields should have the following values:

- **Method name:** "Co2Frc". In hexadecimal: `0x43 0x6F 0x32 0x46 0x72 0x63 0x01 0x01 0x01 0x01`
- **Arguments:** The RPC arguments should be serialized in protobuf format. Following is the structure of the arguments described in proto3 language:

```
/**
 * SCD30 forced recalibration arguments structure definition
 */
syntax = "proto3"; //Proto3 syntax

message Scd30RpcFrcArgs
{
    uint32 c_ref_ppm = 1 ; // CO2 reference concentration.
                           // It should be the actual concentration that the
                           // sensor should be reading
}
```

Arguments only have one field:

- **c_ref_ppm**: The value should be between 400 and 2000 ppm.

6.4.1.2. Response

If there are no errors in the reception and execution of the method, a generic OK response is returned as specified in the corresponding section of this manual. If an error is found, the error response will be an error response in the format shown in the corresponding section of this manual.

6.4.2. reset

This method remotely resets the device. After receiving the RPC with this method, the device is not reset until the method response message is sent.

6.4.2.1. Downlink message

The downlink message fields should have the following values:

- **Method name:** “reset”. In hexadecimal: `0x72 0x65 0x73 0x65 0x74 0x01 0x01 0x01 0x01 0x01`
- **Arguments:** There are no arguments in this method.

6.4.2.2. Response

If reception and the execution of the method are completed with no errors, a generic OK response is returned as specified in the corresponding section of this manual. If an error is found, the error response will be an error response in the format shown in the corresponding section of this manual.

6.4.3. DevInfo

This RPC asks for the hardware version and device identifier.

6.4.3.1. Downlink message

The downlink message fields should have the following values:

- **Method name:** "Devinfo". In hexadecimal: `0x44 0x65 0x76 0x69 0x6E 0x66 0x6F 0x01 0x01 0x01 0x01`
- **Arguments:** There are no arguments in this method.

6.4.3.2. Response

If reception and the execution of the method are completed with no errors, the hardware version and device type are returned in the response, serialized in protobuf format with the following structure:

```
syntax = "proto3";

message HwVer
{
    uint32 major  = 1; // Hardware major version
    uint32 minor  = 2; // Hardware minor version
    uint32 patch  = 3; // Patch identifier
}

message DeviceType
{
    bytes device_type  = 1; // Device reference name
}

message RpcGetDeviceInfoArgs
{
    HwVer      version     = 1;
    DeviceType device_type = 2;
}
```

The metadata has the following fields:

- **HwVer**: Structure with three integer fields (major, minor and patch) identifying the hardware version of the device.

- **DeviceType**: ASCII-encoded character string with the device type identifier (Nanoenvi™ IAQ).

If an error is found, the error response will be an error response in the format shown in the relevant section of this manual.

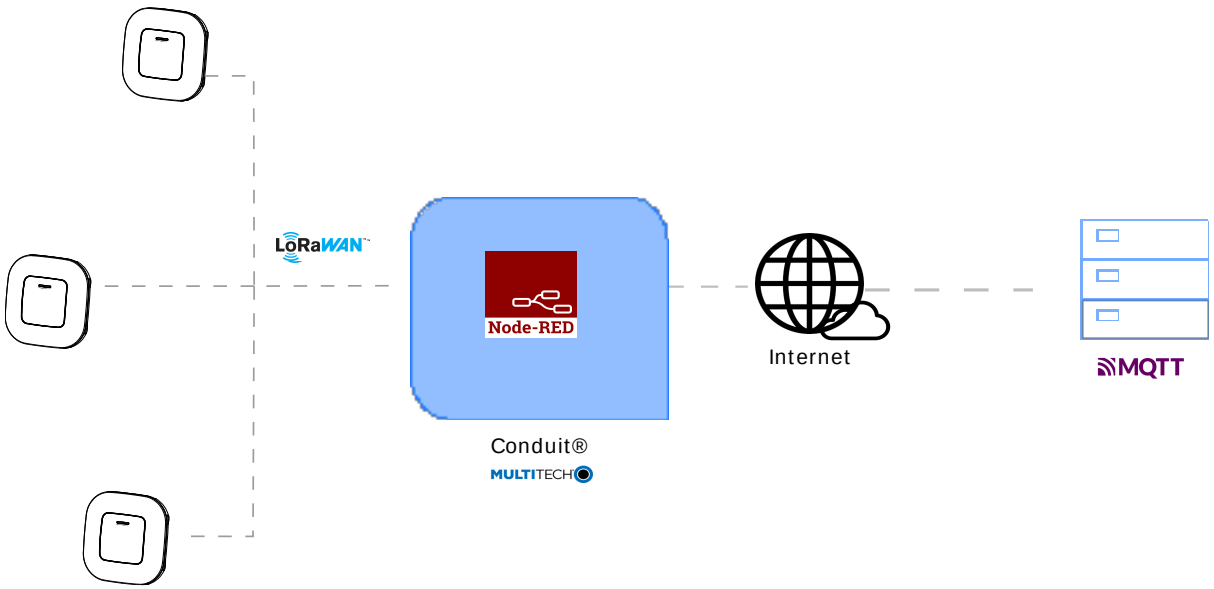
Chapter 7. Examples of the integration of Nanoenvi™ IAQ LoRaWAN

7.1. Example of resending data to MQTT broker

In this example, the use of Nanoenvi™ IAQ is presented in a LoRaWAN network deployed with a MultiTech branded Conduit™ running mPower 5.3.8 firmware.

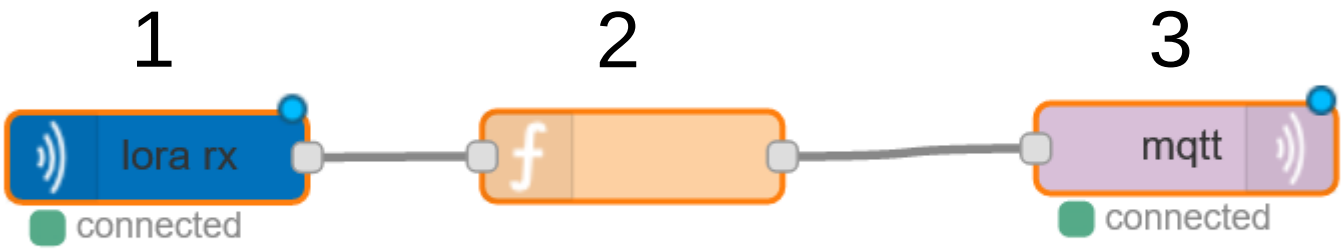
The Conduit™ integrates the functions of the gateway, Network Server and a Node-RED as the application server. The Node-RED flow is configured to redirect messages with Nanoenvi™ IAQ measurements to an MQTT broker. An image of the architecture in the example is shown below:

Figure 7.1. Overview of LoraWAN network redirected to Node-RED



The Node-RED flow is configured as shown in the diagram, in such a way that uplink messages from devices connected to the LoRaWAN network are redirected to the MQTT broker topic `lora/uplink/<devEUI>`:

Figure 7.2. NodeRED redirecting messages flow



The flow consists of three different elements:

1. *lora rx* node that receives the messages from the LoRaWAN network

2. Function node. It executes the following code:

```
var incoming_msg = msg;
var outgoing_msg = {};
outgoing_msg.payload = msg.payload;
outgoing_msg.topic = "lora/uplink/" + msg.eui;
return outgoing_msg;
```

3. MQTT publication node. It just configures the MQTT broker IP.

Once messages are received in the MQTT broker, services can be subscribed to the corresponding topic to collect, deserialize and use the data.

7.2. Example of deserialization of sensor messages using Node-RED

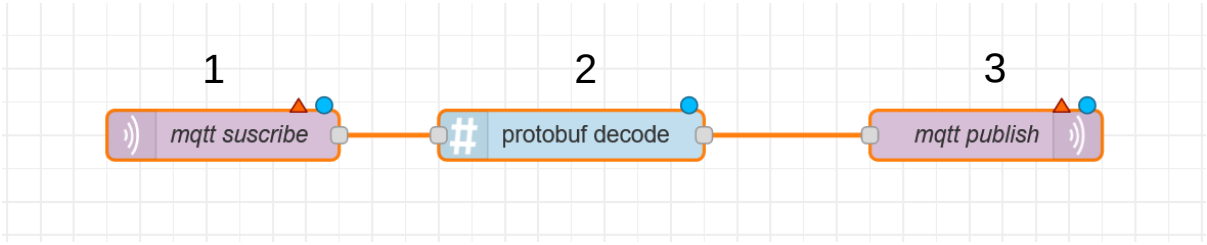
This example shows how to configure Node-RED to receive messages by subscribing to an MQTT broker topic with Nanoenvi™ IAQ measurements serialized with protobuf, convert them to a message in JSON format and publish them in another MQTT broker topic.

First of all you need the nodered-contrib-protobuf (`https://flows.nodered.org/ node/node-red-contrib-protobuf`) node. It can be installed by executing from the command line:

```
npm install node-red-contrib-protobuf
```

With this and the description of the message format in proto3 language, the flow shown in the diagram can be created:

Figure 7.3. Protobuf message deserialization flow

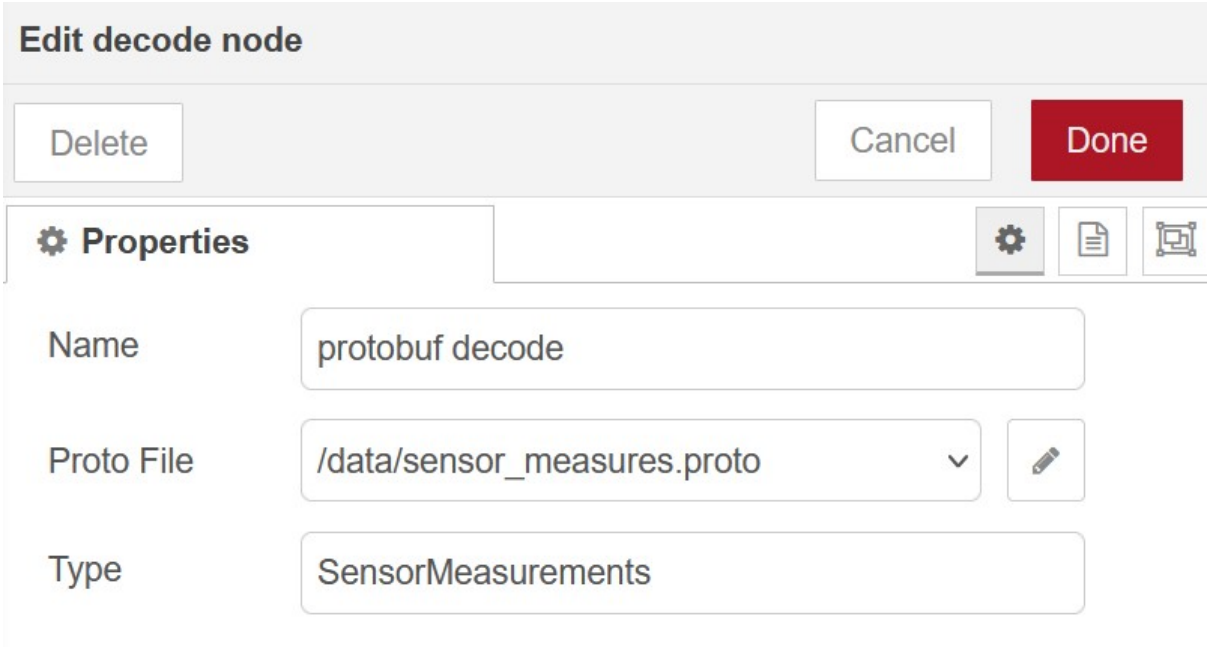


The flow contains the following elements:

1. MQTT subscription node: receives MQTT messages from Nanoenvi™ IAQ
2. Message decoding node with measurements from Nanoenvi™ IAQ.
3. Message publication node in JSON format in MQTT broker.

The protobuf decode node is configured as follows:

Figure 7.4. Configuration of protobuf decode node



The Proto File field contains the path to the file with the description of the Nanoenvi™ IAQ measurement message format included in this manual.

7.3. Example of deserialization of sensor messages using Python

This example shows how to write a Python script that converts the payload of a message with sensor measurements in protobuf format into JSON format.

First of all the grpcio-tools package should be installed, and executed from the command line:

```
pip install grpcio-tools
```

From the description of the measurement message format, the Python library that implements the serialization/deserialization should be generated by executing the following command in the same directory where the sensor measurement format description file is located:

```
protoc --python_out=. sensor_measures.proto
```

Executing the command will generate a file called sensor_measures_pb2.py with the following content:

```
# -*- coding: utf-8 -*-
# Generated by the protocol buffer compiler.  DO NOT EDIT!
# source: sensor_measures.proto
"""Generated protocol buffer code."""
from google.protobuf import descriptor as _descriptor
from google.protobuf import descriptor_pool as _descriptor_pool
from google.protobuf import message as _message
from google.protobuf import reflection as _reflection
from google.protobuf import symbol_database as _symbol_database
# @@protoc_insertion_point(imports)

_sym_db = _symbol_database.Default()



DESCRIPTOR = _descriptor_pool.Default().AddSerializedFile(b'\n\x15sensor_measures.proto
\"\xe3\x01\n\x12SensorMeasurements\x12\x0f\n\x07msg_ver\x18\x01 \x01(\r\x12\x0b\n
\x03\x63o2\x18\x02 \x01(\x02\x12\x0b\n\x03voc\x18\x03 \x01(\x02\x12\n\n\x02\x63o\x18\x04
 \x01(\x02\x12\r\n\x05noise\x18\x05 \x01(\x02\x12\x0c\n\x04pm10\x18\x06 \x01(\x02\x12\r
\n\x05pm2_5\x18\x07 \x01(\x02\x12\x0c\n\x04temp\x18\x08 \x01(\x02\x12\x0b\n\x03hum\x18\t
 \x01(\x02\x12\x0b\n\x03prb\x18\n \x01(\x02\x12\x0b\n\x03pm1\x18\x0b \x01(\x02\x12\x0b
\n\x03pm4\x18\x0c \x01(\x02\x12\x0c\n\x04iaqi\x18\r \x01(\x02\x12\x0b\n\x03tci\x18\x0e
 \x01(\x02\x12\r\n\x05\x65iaqi\x18\x0f \x01(\x02\x62\x06proto3')



_SENSORMEASUREMENTS = DESCRIPTOR.message_types_by_name['SensorMeasurements']
SensorMeasurements = _reflection.GeneratedProtocolMessageType('SensorMeasurements',
 (_message.Message,), {
  'DESCRIPTOR' : _SENSORMEASUREMENTS,
  '__module__' : 'sensor_measures_pb2'
  # @@protoc_insertion_point(class_scope:SensorMeasurements)
  })
_sym_db.RegisterMessage(SensorMeasurements)

if _descriptor._USE_C_DESCRIPTORS == False:

  DESCRIPTOR._options = None
  _SENSORMEASUREMENTS._serialized_start=26
  _SENSORMEASUREMENTS._serialized_end=253
# @@protoc_insertion_point(module_scope)
```

A script can be written that uses the code generated to read a file with the payload of a message sent by the device and converts it to JSON:

```
import argparse
import sys
import sensor_measures_pb2
from google.protobuf.json_format import MessageToDict

class NanoenviIaqMeasDeserializer:
    """ Class with method to convert Nanoenvi IAQ protobuf measures message to JSON """

    def deserialize(self, file_path):
        """ Receives as parameter the path of a file with Nanoenvi IAQ protobuf
```

```
        measures message, reads it converts it to JSON and returns it. """

        file_obj = open(file_path, 'rb')
        file_content = file_obj.read()
        file_obj.close()

        meas_obj = sensor_measures_pb2.SensorMeasurements()
        meas_string = meas_obj.ParseFromString(file_content)
        meas_dict = MessageToDict(meas_string)

        return meas_dict


if __name__ == "__main__":

    # Parse arguments
    try:
        parser = argparse.ArgumentParser()
        parser.add_argument(
            "-f",
            "--file",
            help="file containing Nanoenvi IAQ measurements serialized with protobuf",
            nargs="?",
            required=True,
        )

        args = parser.parse_args()
    except Exception as exception_object:
        print(exception_object)
        sys.exit(2)

    file_path = args.file

    # Instance NanoenviIaqMeasDeserializer
    deserializer = NanoenviIaqMeasDeserializer()
    # Deserialize file content
    result = deserializer.deserialize(file_path)
    # Print result
    print(result)
```

The script written can be invoked from the command line by passing a file with the payload of a message sent by Nanoenvi™ IAQ as an argument:

```
python main.py -f file_with_mesures_payload.bin
```

Nanoenvi