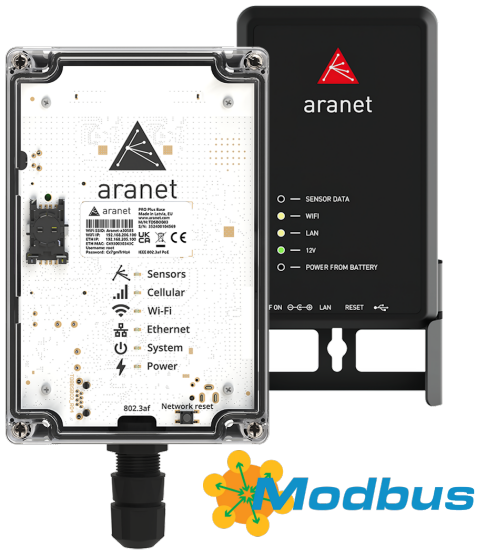
Base station Modbus TCP/IP server feature

Seamlessly integrating with Modbus, the industry-standard communication protocol, our Base station empowers you to effortlessly connect your Aranet sensors to existing automation and control systems, enhancing efficiency and streamlining operations like never before.



Modbus TCP/IP server menu and configuration settings

To enable Modbus TCP/IP feature on the Aranet base station, in the WebGUI navigate to the main menu sidebar item shown in Fig. 1. Please note that use Modbus TCP/IP server feature requires appropriate licence file to be uploaded on the base station.

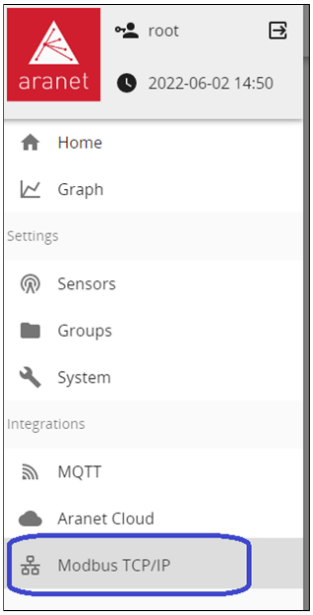


Figure 1: WebGUI main sidebar menu.

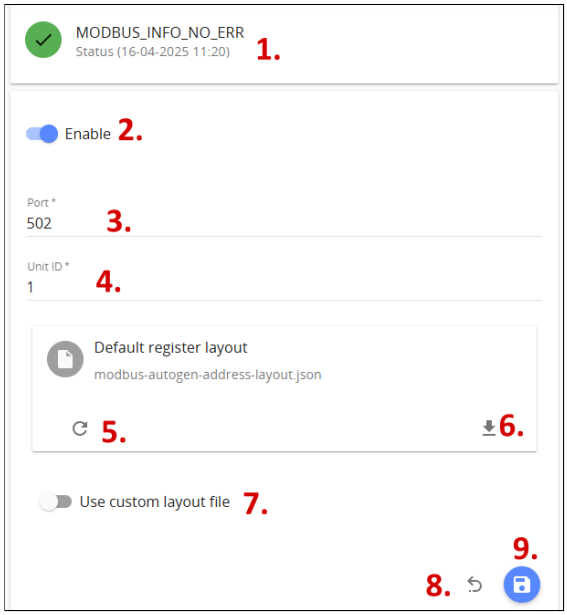


Figure 2: TCP/IP configuration menu.

The following items are shown in the Modbus TCP/IP configuration panel as indicated in Fig. 2:

(1) — Status of Modbus TCP/IP server process.

(2) — Switch to enable or disable server process.

(3) — Field to specify TCP port number (default is 502).

(4) — Set Modbus TCP/IP server Unit ID (a.k.a. Device ID). This must be the same as one set in Modbus

TCP/IP Client side. In some Client (Master) applications, this setting may be referred to as ”Slave ID”. Client requests with incorrect Unit ID on the Modbus TCP/IP server side will be ignored (communication timeout);

(5) — Reset auto-generated address-to-measurements mapping configuration file. **WARNING:** all addresses used will be reset and may differ from those assigned in previous default mapping file. This request does not affect addressing defined by “Custom layout file”.

(6) — Download auto-generated address-to-measurements mapping configuration file.

(7) — Switch to enable or disable custom mapping configuration file usage. (This feature allows to upload a user-specified address-to-measurements configuration file.)

(8) — Button to save configuration changes.

(9) — Button to revert the previous configuration without saving changes.

Process status description

`SUCCESS_NO_ERR`	No error. Modbus TCP/IP service functions correctly.
`MODBUS_DISABLED_BY_USER`	Service is disabled by the user. Service is not working.
`MODBUS_DISABLED_BY_LICENCE`	An appropriate license is not available to run this service. Service is not working.
`WARN_ADDR_COLLISION`	Warning state. Registry address overlaps with other address (more info in “details”). Modbus server continues to accept incoming requests and sends responses except for the registers with colliding addresses.
`WARN_OFFSET_OR_ADDRESS_OUT_OF_RANGE`	Warning state. One or more used offset values or resulting registry addresses (sum of all three offsets per measurement) are out of the expected range (0–65535). Modbus server continues to accept incoming requests and sends responses except for the registers with invalid offset values or resulting registry addresses (more info in “details”).
`ERR_CONTEXT_FAILED`	Error occurred. Process failed to allocate memory for the Modbus TCP/IP server instance (not enough free memory).
`ERR_MAP_ALLOC_FAILED`	Error occurred. Modbus address mapping instance allocation failed (not enough free memory).
`ERR_CONTEXT_NULL`	Error occurred. Error on opening Modbus TCP/IP server socket.
`ERR_LISTEN_FAILED`	Error occurred. Failed to listen for incoming Modbus TCP/IP connections (port already in use).

ERR_CONFIG_JSON_FAILED	Error occurred. Failed to load mapping configuration JSON file.
SERV_SETTINGS_ERR_CONFIG_LOAD	Error occurred. Failed to load settings configuration file (in order to load Modbus service settings).
SERV_SETTINGS_ERR_JSON_EMPTY	Error occurred. The settings configuration file has an invalid JSON structure.
SERV_SETTINGS_ERR_WRONG_PORT	Error occurred. Incorrect port number in the service configuration.
ERR_OFFSET_MISSING	Error occurred. Offset field is missing or type is incorrect in mapping configuration (more info in “details”).

Address-to-measurements mapping configuration file

Modbus object types

Modbus protocol has four object types (usage depends on desired functionality and data type). It is expected that in the configuration file there are all four of these objects defined.

Object name	Amount of registers	Amount of bits	Read function	Write function
coils	1	1	Yes	Yes
discreteInputs	1	1	Yes	No
inputRegisters	1	16	Yes	No
holdingRegisters	1	16	Yes	Yes

Each type has its own address range. Each range has an offset defined by the field named `offset`. The address value is limited to two bytes (0–65535). There is no binding between each object type and specific address range, e.g., coils can have a range 0–9999, discrete inputs 10000–19999, input registers 20000–29999, holding registers 30000–39999, or any other configuration.

Current implementation supports `discreteInputs` and `inputRegisters` object types only. Addresses can be in the range 0–65535. All other types can be defined in the configuration file, but they will be ignored. It refers also to function codes. Only function codes 2 (read `discreteInputs`) and 4 (read `inputRegisters`) are supported. Here is an example of the use:

```
{
    "inputRegisters": {
        "offset": 30000
    }
}
```

Discrete input type returns sensors “packets lost” (RSSI alarm) state: 0 (alarm inactive) or 1 (alarm active, sensor is lost). Implementation is available starting from firmware version 3.2.4. Starting from version 4.8.x, measurement threshold alarm states can also be read from discrete input registers.

Object type to sensor mapping

Each Modbus object type can have a set of sensors as nested JSON objects. For each sensor object, there is expected to have its own address `offset` within its parent Modbus object type address range. Here is an example with sensor mapped to Modbus data type having its offset defined:

```
{
    "inputRegisters": {
        "offset": 30000,
        "1056849": {"offset": 0}, // register address will start from 3000X
        "4196581": {"offset": 40} // register address will start from 3004X
    }
}
```

Numbers “1056849” and “4196581” are internal sensor identificators. Sensors HEX identificator is mentioned in the comment of the auto-generated address-to-measurements mapping configuration file (see “Confguration controls”, Pt. 4) in the same line next to the sensor identificator.

Address-to-sensor-to-measurement mapping

Measurement has its field named `offset` which is mandatory for each object and must have unique value between particular sensor’s other measurement offset values (same rule applies also to sensors — unique offset value between sensors). Sum of all three offset values determines initial address for register. Here is an example with sensor measurement to address mapping:

```
{
    "inputRegisters": {
        "offset": 30000,
        "1056849": {
            "offset": 0,
            "humidity": {
                "offset": 1
            },
            "temperature": {
                "offset": 2
            }
        },
        "4196581": {
            "offset": 10,
            "temperature": {
                "offset": 1
            }
        }
```

```
    }
}
```

Measurement data types

Data type helps to determine how to interpret received bytes on the Modbus client side. If incorrect data type is used in case of user custom mapping configuration, e.g., unsigned type for negative values, it may lead to incorrect value interpretation on Modbus client side. Also, incorrect length (16-bit instead of 32-bit) can be a source of incorrect data interpretation (value cannot be encoded correctly as it is out of the data type's specific value range).

`dataType` value	Description	Registry size	Value range
`int16`	16-bit signed integer	1	-32768–32767
`int32`	32-bit signed integer	2	-2147483648–2147483647
`uint16`	16-bit unsigned integer	1	0–65535
`uint32`	32-bit unsigned integer	2	0–4294967295

As for the data types `uint32`, `int32` registry size is 2 it also affects the next address value in the sequence of addressing, e.g., if sensor's measurement initial "offset" value is 1 and its `dataType` value is `uint32` then next measurement offset value must be 3 (`offset` plus registry size of the `dataType`). In case of address overlapping an error message will be reported in Modbus configuration GUI (see "Confguration controls") and both overlapping registries will not be available for Modbus requests (`ILLEGAL_DATA_ADDRESS` exception will be received). Here is an example of the use of these types:

```
{
    "inputRegisters": {
        "offset": 30000,
        "1056849": {

            // CO2: 1 register long, values 0--65535
            "co2": {
                "dataType": "uint16",
                "offset": 4            // address 30004
            },

            // RH: 1 register long, values 0--65535
            "humidity": {
                "dataType": "uint16",
                "offset": 1            // address 30001
            },
            "offset": 0,

            // CO2: 2 registers long, values -2147483648--2147483647
            "temperature": {
```

```
                "dataType": "int32",
                "offset": 2            // address 30002
            }
        }
    }
}
```

Measurement value multiplier

On the Modbus server side, before the value is assigned to the registry, it is multiplied depending on the measurement precision. To retrieve the original measurement value, the received value must be multiplied by the specified constant. Multipliers can be found in the auto-generated registry mapping configuration file (see “Confguration controls”). Same multiplier values must be used in a custom user-specified mapping configuration file.

```
{
    "inputRegisters": {
        "offset": 30000,
        "1056849": {
            "co2": {
                "dataType": "uint16",
                "multiplier": 1,      // original measurement value received
                "offset": 4
            },
            "humidity": {
                "dataType": "uint16",
                "multiplier": 10,     // divide by 10 to get humidity
                "offset": 1
            },
            "offset": 0,
            "temperature": {
                "dataType": "int32",
                "multiplier": 1000,   // divide by 1000 to get temperature
                "offset": 2
            }
        },
    }
}
```

The original value multiplied by the multiplier specified in the configuration determines which data type can be used in the configuration, e.g., if the original value of temperature is -40.5 and the multiplier is 1000, then the result will be -40500 which is out of the data type (`int16`) defined range (so, data type `int32` should be used in that case). Here is an additional example:

```
{
    "inputRegisters": {
        "6292285": {               // Sensor 60033D; Address: 20000
```

```
            "battCharge": {         // Address 20003: divide by 100 to get battCharge in
                fraction
                "dataType": "int32",
                "multiplier": 100,
                "offset": 3
            },
            "offset": 0,
            "rssi": {               // Address 20005: measurement rssi in dBm
                "dataType": "int32",
                "multiplier": 1,
                "offset": 5
            },
            "time": {               // Address 20007: measurement time in seconds
                "dataType": "uint32",
                "multiplier": 1,
                "offset": 7
            },
            "vwc": {                // Address 20001: divide by 1000 to get VWC in fraction
                "dataType": "int32",
                "multiplier": 1000,
                "offset": 1
            }
        },
        "offset": 20000
    }
}
```

Sensor UID readable from Input registers (from v4.9.x)

One of the common sensor registry objects is the sensor's UID (unique identifier) which can be read from the Input registers. It represents the sensor's HEX UID in decimal format. In the sensor's object it is nested as an object with a name "uid".

Discrete Inputs for Threshold alarm states (from v4.8.4)

RSSI alarm

Sensor's packet loss alarm state (`rssiAlarm`) can be read from discrete inputs registers. Each sensor has it's object nested inside `discreteInputs` type object. And each sensor object has it's RSSI alarm object.

NOTE: If the RSSI alarm is active for the sensor, all the measurement requests (from Input registers) except the time will return exception `ILLEGAL_DATA_VALUE`. Time reading will return a Unix timestamp since RSSI alarm was set as active.

Measurement threshold alarms (from v4.8.4)

Measurement threshold alarm states can be read from Discrete Inputs. Each measurement has Min and Max thresholds. In case if measurement value trespasses one or another threshold, appropriate threshold alarm state is being set to ”active” state (Discrete Input value: 1; ”inactive state”: 0).

In Modbus Auto-generated registry address mapping configuration file Min and Max threshold alarm state register JSON objects are nested inside the `discreteInputs` type object. For example, if measurement is ”temperature” then appropriate threshold alarm state JSON object name for Min and Max thresholds will be:

- `MinTemperatureAlarm`
- `MaxTemperatureAlarm`

For the 4T Transmitter measurement names will end with the probe ID:

- For measurement temperature1:

  `MinTemperatureAlarm:1`

  `MaxTemperatureAlarm:1`

- For measurement temperature2:

  `MinTemperatureAlarm:2`

  `MaxTemperatureAlarm:2`

Temperature threshold alarm state objects inside ”discreteInputs” object example:

```
{
        "discreteInputs": {
                "offset": 10000,
                "4196581": {    // Sensor 4008E5; Name: 4008E5; Group name: roomEnv;
                    Address: 10000
                        "offset": 0,
                        "MinTemperatureAlarm": {    // Address 10001: measurement
                            MinTemperatureAlarm
                                "offset": 1,
                                "dataType": "bit",
                                "address": 10001    // read-only
                        },
                        "MaxTemperatureAlarm": {    // Address 10002: measurement
                            MaxTemperatureAlarm
                                "offset": 2,
                                "dataType": "bit",
                                "address": 10002    // read-only
                        }
                }
        }
}
```

Battery low alarm (from v4.9.x)

Sensor's low battery alarm state (`batteryAlarm`) can be read from discrete inputsregisters. Each sensor has it's object nested inside `discreteInputs` type object. And each sensor object has it's `batteryAlarmobject`.

Example with low batteryalarm object in discreteInputs:

```
{
"discreteInputs":{
        "offset":10000,"
        4196581":{                      //Sensor4008E5
                "offset":0,
                "batteryAlarm":{        //Address10000:measurementbatteryAlarm
                        "offset":0,
                        "dataType":"bit",
                        "address":10000 //read-only
                        }
                }
        }
}
```

In the mapping example shown above request to address 10000 will return sensor's 4008E5low battery alarm state: 0 –inactive, 1 active.

Offset value assignment indefault auto-generated JSON(from v4.9.x)

Register type `discreteInputs` fixed offset value is 10000 and for `inputRegisters` –30000.

For `discreteInputs` sensor object starts with an offset 0 and for each next paired sensor offset value is increased by the step 20.

For `inputRegisters` first object is `baseStation` with an offset 0. Offset for the first paired sensor will be 20 and for each next paired sensor –offset increment is 20.

Common sensor measurements and their offsets

Common measurement objects for sensor objects nested in `discreteInputs` (in fixed sequence throughout for all sensor objects):

1. `batteryAlarm` (offset: 0)
2. `rssiAlarm` (offset: 1)
3. threshold alarms for sensor specificmeasurements (sorted by name in alphabetical order):

    min threshold (offset: <previous offset>+1);

    max threshold (offset: <previous offset>+1).

For `inputRegisters` initial nested object is `baseStation` with offset 0 having measurement

`rxMsgCounter` with offset 0 which makes the address for this measurement 30000 (30000 + 0 + 0)

Common measurement objects for sensor objects nested in `inputRegisters` (in fixed sequence throughout for all

sensor objects):

1. uid (offset: 0);
2. battery (offset: 2);
3. rssi (offset: 3);
4. time (offset: 5);
5. sensor specific measurements (sorted by name in alphabetical order)

   offset: <previous offset> + <previous measurement data type registry size>

Requesting data from Aranet PRO Modbus TCP/IP server

Important notes for the client side

Register size	16 bits
Unit ID (slave ID)	1 (not changeable)
Supported functions	Read discrete inputs (code: 2), read input registers (code: 4)
Supported data types	Discrete inputs, input registers (any address in the supported range of 0–65535)
Endianness	Big-endian with high word first (high byte first or high word first for 2 registers/32-bit values)

Used Modbus TCP exceptions

- In case if an unsupported function is requested, a message with `ILLEGAL_FUNCTION` code will be replied.
- In case if unavailable registry address is being requested, a message with `ILLEGAL_DATA_ADDRESS` code will be replied. This exception is sent also in case if particular address is colliding with any other registry address.
- In case if uninitialized registry address is being requested, a message with `ILLEGAL_DATA_VALUE` code will be replied. The address of the registry exists, but valid data of measurement has not been assigned yet.

Custom address-to-measurements mapping: Input registers

Below is an example of an uploaded custom address-to-measurements mapping configuration JSON file used for the input registers request example with three 2×16-bit wide registries. Representation in the Modbus client application can be seen in Fig. 3.

```
{
    "inputRegisters": {              // Input registry type
        "1056849": {
            "battCharge": {          // Registry address: 30044
                "dataType": "int32",
                "multiplier": 100,
                "offset": 4
            },
```

```
            "offset": 40,
            "rssi": {                // Registry address: 30046
                "dataType": "int32",
                "multiplier": 1,
                "offset": 6
            },
            "time": {                // Registry address: 30048
                "dataType": "uint32",
                "multiplier": 1,
                "offset": 8
            }
        },
        "offset": 30000
    }
}
```

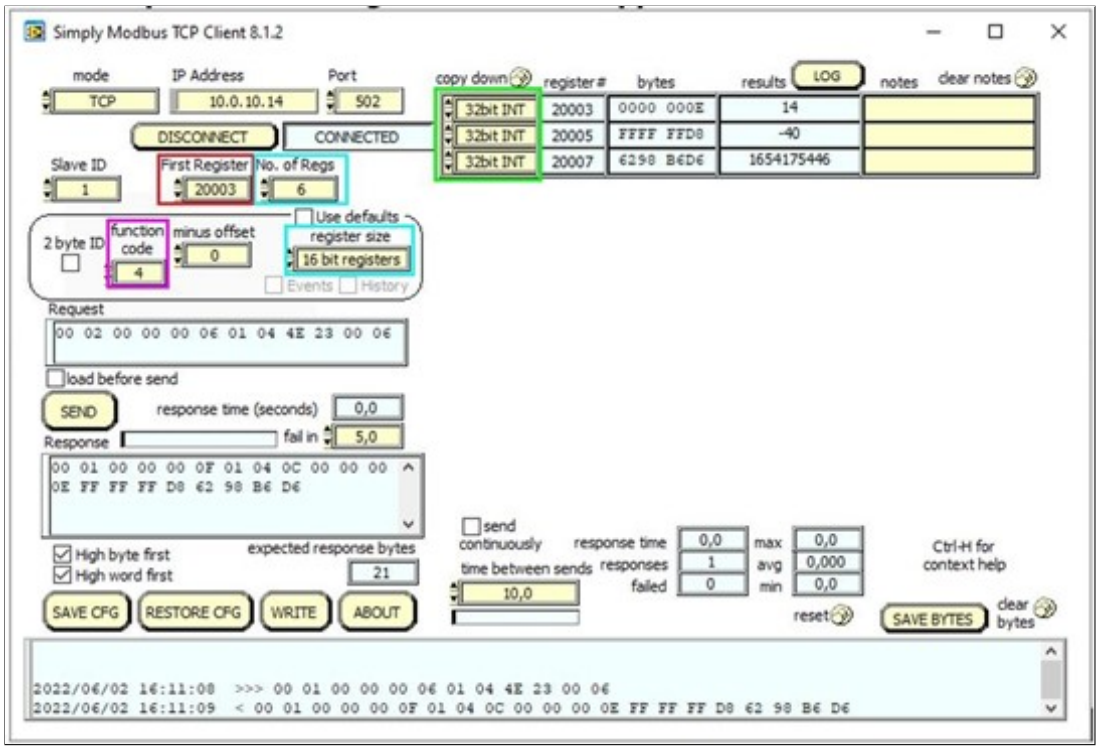


Figure 3: Modbus client application view with example configuration.

Custom address-to-measurements mapping: Dicrete inputs

Below is an example of uploaded custom address-to-measurements mapping configuration file JSON used for Discrete inputs request. Representation in the Modbus client application can be seen in Fig. 4.

```
{
    "discreteInputs": {
        "4196581": {
            "offset": 20,
```

```
            "rssiAlarm": {          // Registry address: 10020
                "dataType": "bit",
                "multiplier": 1,
                "offset": 1
            }
        },
        "6292285": {
            "offset": 20,
            "rssiAlarm": {          // Registry address: 10021
                "dataType": "bit",
                "multiplier": 1,
                "offset": 0
            }
        },
        "offset": 10000
    }
}
```

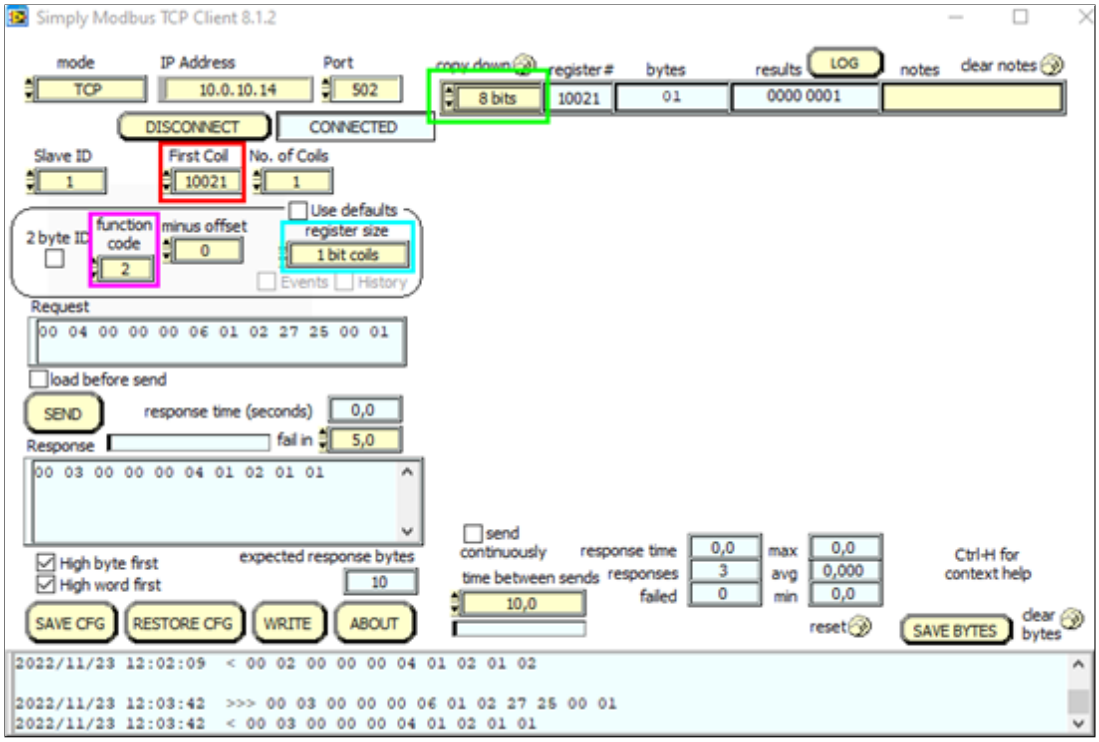


Figure 4: Modbus client application view with example configuration.