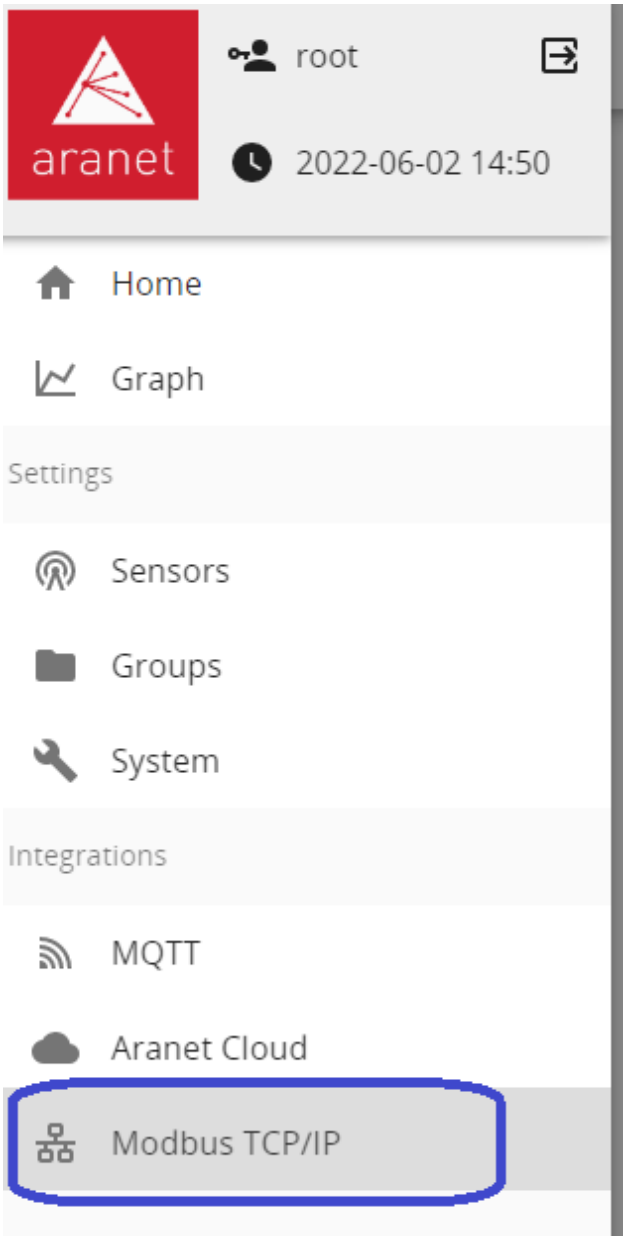
Aranet PRO Modbus TCP/IP server

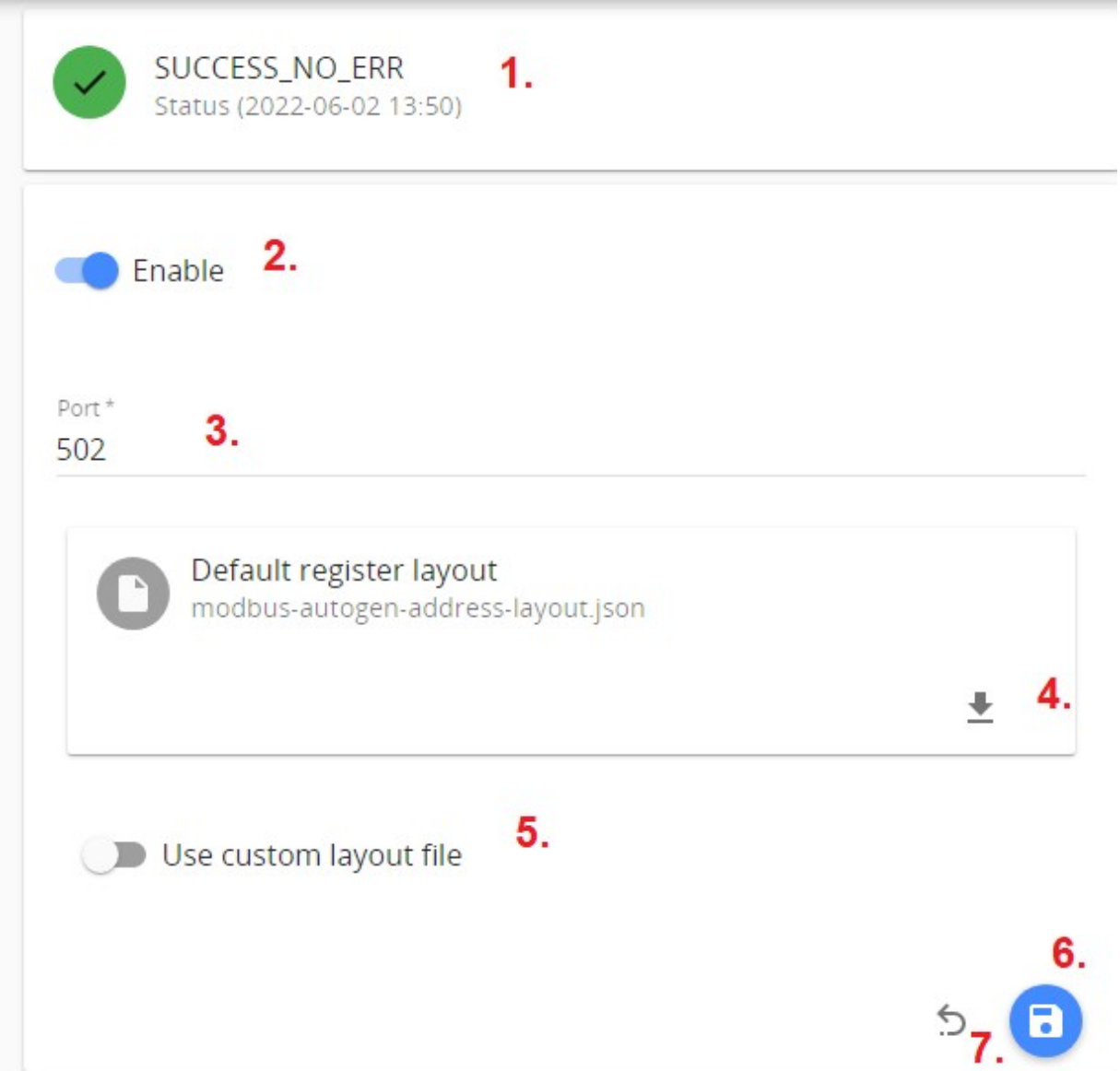
IMPORTANT: feature requires specific license to be uploaded on the base station.

In the web GUI main page Modbus TCP/IP server configuration is available under “Modbus TCP/IP”



Configuration controls

1. Status of Modbus TCP/IP server process;
2. Enable/Disable server process;
3. Specify TCP port number (default 502);
4. Download auto-generated address-to-measurements mapping configuration file;
5. Enable/Disable custom mapping configuration file usage. The feature allows to upload to the base station a user-specified address-to-measurements configuration file;
6. Save configuration changes;
7. Revert to the previous configuration.



Expected Modbus TCP/IP server process statuses and their description

- ***"SUCCESS_NO_ERR"*** - no error, Modbus TCP/IP service functions correctly;
- ***"ERR_CONTEXT_FAILED"*** – error occurred. Process failed to allocate memory for the Modbus TCP/IP server instance (not enough free memory);
- ***"ERR_MAPALLOC_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);
- ***"WARN_ADDR_COLLISION"*** – warning state. Registry address overlaps with other address (more info in the ***"Details"***). Modbus server continues to accept incoming requests and sends responses except for the registers with colliding addresses;
- ***"ERR_CONFIG_JSON_FAILED"*** – error occurred. Failed to load mapping configuration JSON file;
- ***"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_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 the ***"details"***.
- ***"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 invalid JSON structure;
- ***"SERV_SETTINGS_ERR_WRONG_PORT"*** - error occurred, incorrect port number in the service configuration;
- ***"ERR_OFFSET_MISSING"*** - error flag, offset field is missing or type is incorrect in mapping configuration (more info in the ***"details"***).

Address-to-measurements mapping configuration file description

Modbus object types

Modbus protocol has 4 object types (usage depends on desired functionality and data type):

- Coils (1 register = single bit value; read-write function),
- Discrete inputs (1 register = single bit value; read-only function),
- Input registers (1 register = 16-bit value; read-only function),
- Holding registers (1 register = 16-bit value; read-write function).

In configuration file is expected to have each of these 4 type objects named (accordingly):

- "coils"
- "discreteInputs"
- "inputRegisters"
- "holdingRegisters"

Each type has its 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.

NOTE: Current implementation supports "Discrete input" and "Input register" 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 " Discrete input") and **4** (read "Input register") are supported.

Discrete input type returns sensor's "packets lost" (RSSI alarm) state: 0 – alarm inactive; 1 – alarm is active (sensor is lost). Implementation is available starting from FW version 3.2.4

Example with Modbus object data type and address offset

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

For "inputRegisters" object type register addresses will start from 3000X

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.

Example with sensor mapped to Modbus data type having it's offset defined

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

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

Example with sensor’s ID (4194868) and its HEX ID (400234) mentioned in the comment

"**4194868**": {// Sensor **400234**; Address: 0

Address-to-sensor-to-measurement mapping

Measurement has its field named "offset" which is mandatory for each object and must have a **unique value** between a particular sensor's other measurement offset values (the same rule applies also to sensors – unique offset value between sensors). **Sum** of all **three offset values** determines the initial address for the register.

Example with sensor measurement to address mapping

```
{
      "inputRegisters": {
        "offset": 30000
      "1056849": {
          "offset": 0,
          "humidity": {
          "offset": 1          //for input registers, initial register address is 30001
        }
        "temperature": {
           "offset": 2         //second measurement's address for the same sensor is 30002
        }
      }, "4196581": {
          "offset": 10, "temperature": {
             "offset": 1 //register address of another sensor's measurement is 30011
          }
      }
   }
}
```

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 to 32767
int32	32-bit signed integer	2	-2147483648 to 2147483647
uint16	16-bit unsigned integer	1	0 to 65535
uint32	32-bit unsigned integer	2	0 to 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 it's *"dataType"* value is *uint32* then next measurement's offset value must be 3 (offset + registry size of the *dataType*). In case of address overlapping an error message will be reported in Modbus configuration GUI (see "Configuration controls") and both overlapping registries will not be available for Modbus requests (***ILLEGAL_DATA_ADDRESS*** exception will be received).

Example with sensor measurement registry data types

```
{
        "inputRegisters": {
          "offset": 30000
        "1056849": {
              "offset": 0,
           "humidity": {
              "offset": 1,                    //address 30001
              "dataType": "uint16" //humidity is one register long and can encode
values in range from 0 to 65535
           },
           "temperature": {
              "offset": 2,                    //address 30002
              "dataType": "int32"          //temperature is two register long and can encode  values in
range from -2147483648 to 2147483647
           }, "co2": {
              "offset": 4,                    //address 30004
              "dataType": "uint16" //co2 is one register long and can encode values in  range from 0 to
65535
           }
       }
```

```
                                    }
}
```

Measurement value multiplier

In the Modbus server side before value is being assigned to the registry, it is multiplied depending on measurement's precision. In order to retrieve original measurement value, received value must be divided by specified multiplier. Multipliers can be found in auto-generated registry mapping configuration file (see "Confguration controles", pt.4). Same multiplier values must be used in custom user-specified mapping configuration file.

Example with sensor measurement multipliers

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

NOTE: original value multiplied by the multiplier specified in the configuration determines which data type can be used in configuration, e.g., if original value of temperature is -40.5 and multiplier is 1000, then result will be -40500 which is out of the data type's ***int16*** defined range (so, data type *int32* must be used then).

Example of final configuration JSON structure

```
{
"5245646": {    // Sensor 500ACE; Address: 20040
      "offset": 40,
      "battCharge": {   // Address 20040: divide by 100 to get battCharge in fraction
        "offset": 0,
        "dataType": "int32",
        "multiplier": 100
      },
      "rssi": { // Address 20042: measurement rssi in dBm
        "offset": 2,
        "dataType": "int32",
        "multiplier": 1
      },
      "pulses": {   // Address 20044: measurement pulses in pulses
        "offset": 4,
        "dataType": "int32",
        "multiplier": 1
      },
      "time": { // Address 20046: measurement time in seconds
        "offset": 6,
        "dataType": "uint32",
        "multiplier": 1
      },
      "pulsescumulative": { // Address 20048: measurement pulsescumulative in pulses
        "offset": 8,
        "dataType": "int32",
        "multiplier": 1
      },
      "derivedp": { // Address 20050: divide by 1000000 to get derivedp weight in kg
        "offset": 10,
        "dataType": "int64",
        "multiplier": 1000000
      },
      "derivedpc": {    // Address 20054: divide by 1000000 to get derivedpc weight in kg
        "offset": 14,
        "dataType": "int64",
        "multiplier": 1000000
      }
    }
  }
```

Requesting data from Aranet PRO Modbus TCP/IP server

IMPORTANT NOTES FOR CLIENT SIDE:

- Register size: **16 bits**;
- Unit ID (slave ID) – not changeable: **1**
- Endianness - **big-endian** with **high word first**:
  - **high byte first**;
  - **high word first** (for 2 registries/32-bit values).
- Supports only:
  - Modbus functions: **read discrete inputs** (code: 2), **read input registers** (code: 4);

- Modbus data types: **discrete inputs**, **input registers** (any address can be used in supported range: 0-65535).

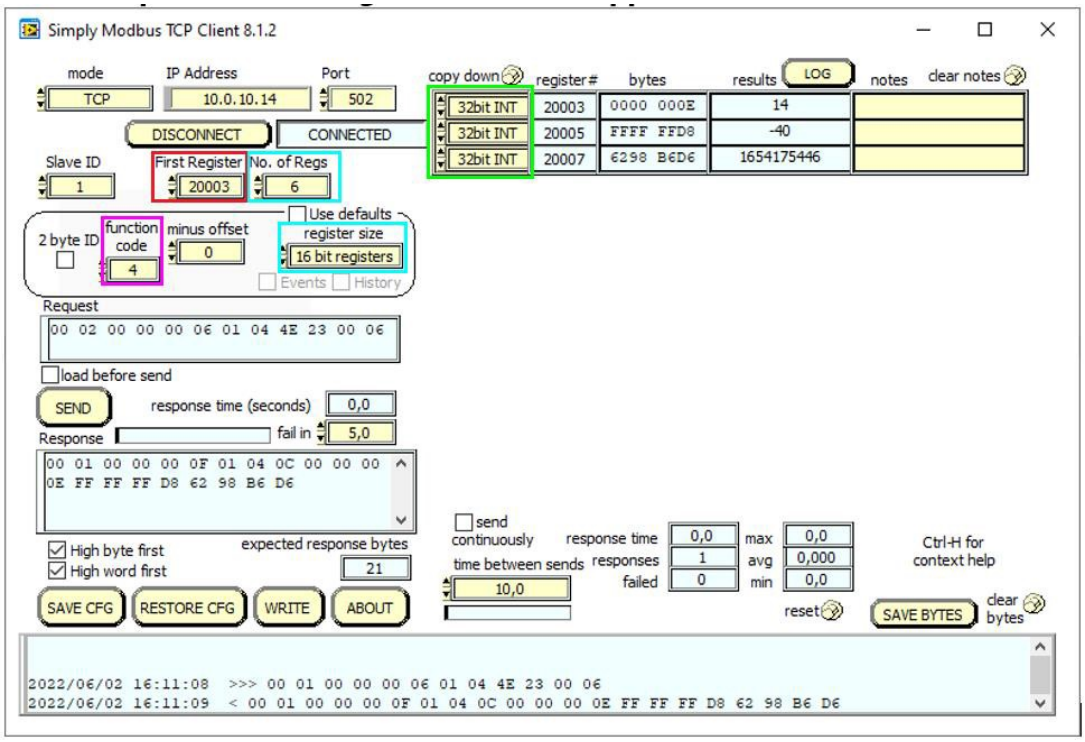
Used Modbus TCP exceptions

- In case if unsupported function is being requested, message with "***ILLEGAL_FUNCTION***" code will be replied.
- In case if unavailable registry address is being requested, 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, message with "***ILLEGAL_DATA_VALUE***" code will be replied. Address of the registry exists, but valid data of measurement has not been assigned yet.

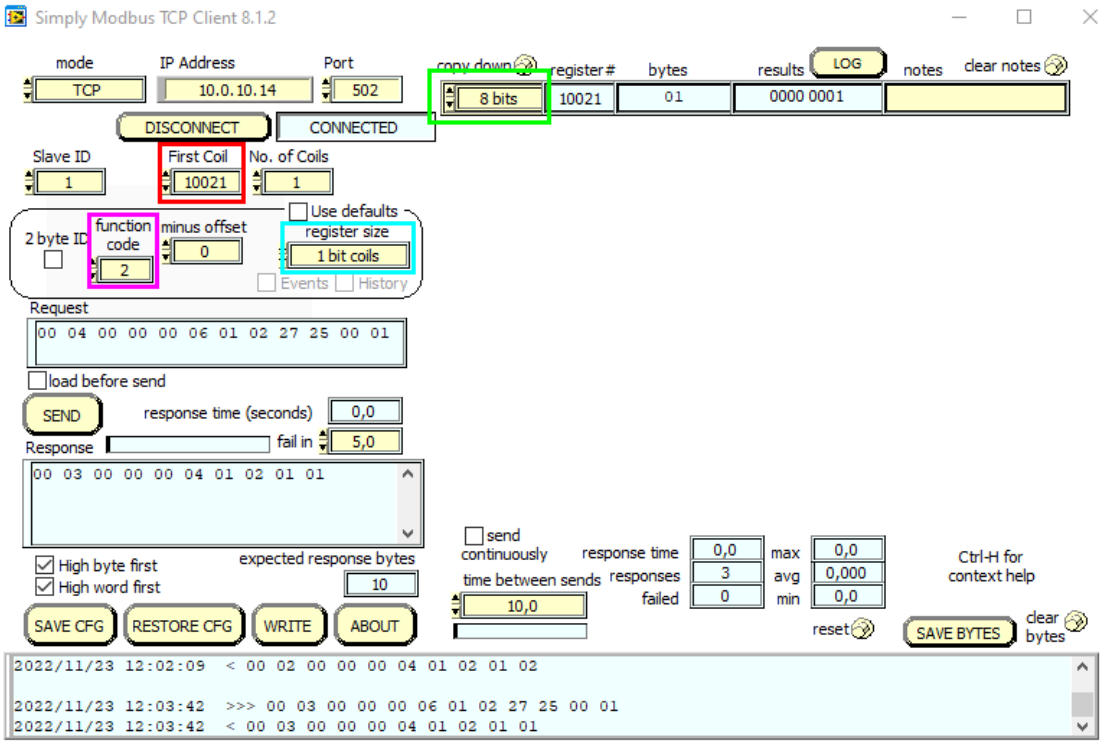
Uploaded custom address-to-measurements mapping configuration file JSON used for Input registries request example with three 2x16-bit wide registries:

```
{
   "inputRegisters": {                    // Input registry type
      "offset": 30000,
      "1056849": {
         "offset": 40,
         "battCharge": {                  // Registry address: 30044
            "offset": 4,
            "dataType": "int32", "multiplier": 100
         },
         "rssi": {                        // Registry address: 30046
            "offset": 6,
            "dataType": "int32", "multiplier": 1
         },
         "time": {                        // Registry address: 30048
            "offset": 8,
            "dataType": "uint32", "multiplier": 1
         }
      }
   }
}
```

Example with a request result using Modbus client application:



Uploaded custom address-to-measurements mapping configuration file JSON used for Discrete inputs request example:

```
{
   "discreteInputs": { "offset": 10000,
      "6292285": {
         "offset": 20,
         "rssiAlarm": {                    // Registry address: 10020
            "offset": 0,
            "dataType": "bit",
            "multiplier": 1
         }
      },
      "4196581": {
         "offset": 20,
         "rssiAlarm": {                    // Registry address: 10021
            "offset": 1,
            "dataType": "bit",
            "multiplier": 1
         }
      }
   }
}
```

Example with a request result using Modbus client application: