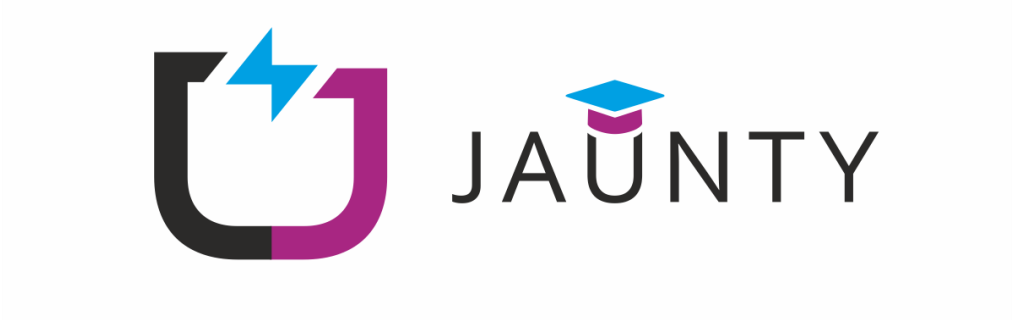
# Introduction to Modbus TCP




**Author:** Public Power Corporation S.A

Co-funded by the
Erasmus+ Programme
of the European Union

## Copyright



Consisting of

| | | |
|---|---|---|
| Coordinator: | Technical University of Sofia | Bulgaria |
| Partners: | University of Western Macedonia | Greece |
| | International Hellenic University | Greece |
| | Public Power Corporation S.A. | Greece |
| | University of Cyprus | Cyprus |
| | K3Y Ltd | Bulgaria |
| | Software Company EOOD | Bulgaria |



Co-funded by the Erasmus+ Programme of the European Union

## Table of Contents


1. Abbreviations ..... 3
2. Scope ..... 4
   2.1 Specific outcomes ..... 4
   2.2 General description ..... 4
   2.3 Lab configuration ..... 8
3. Exercise 1: Implementation of a Modbus TCP master ..... 8
   Step 1: The basic structure ..... 8
   Step 2: Defining the Modbus attributes ..... 9
   Step 3: Establishing reliable Modbus TCP connection ..... 9
   Step 4: Reading the contents of Modbus addresses ..... 10
   Step 5: Handling errors ..... 11
   Step 6: Prepare JSON format – Final script ..... 11
4. Exercise 2: Emulate a PLC by implementing a Modbus TCP slave ..... 11
   Step 1: The basic structure ..... 11
   Step 2: Initialise the data structures ..... 12
   Step 3: Worker process for updating addresses contents ..... 13
   Step 4: Run the Modbus server – Final script ..... 14
   Step 5: Validate using the Modbus TCP master ..... 15
5. Exercise 3: Implement a custom Modbus TCP slave for Adafruit SHT40 ..... 15
6. Exam Questions ..... 16

References ..... 17

7. Contacts ..... 18

# 1. Abbreviations

| | |
|---|---|
| TCP | Transmission Control Protocol |
| IIoT | Industrial Internet of Things |
| RTU | Remote Terminal Unit |
| PLC | Programmable Logic Device |
| SCADA | Supervisory Control and Data Acquisition |
| IP | Internet Protocol |
| MTU | Master Terminal Unit |
| DI | Digital Input |
| HR | Holding Register |
| UDP | User Datagram Protocol |

## 2. Scope

The scope of this laboratory exercise is to provide a comprehensive introduction to the Modbus TCP protocol, including the development of Modbus slaves and masters by utilising the pymodbus Python library.

### 2.1 Specific outcomes

Upon completion of this exercise, individuals will be able to:

- Understand the Modbus TCP communication protocol.
- Design and implement Modbus TCP slaves using Python.
- Implement Modbus TCP master to retrieve data from real Modbus TCP devices.

### 2.2 General description

Modbus is a data communications protocol that was originally published by Modicon (now Schneider Electric) in 1979. Modbus is a very popular protocol in Industrial IoT (IIoT) networks due to its simplicity on deployment and maintenance compared to other standards. Modbus was originally used to connect industrial assets (e.g., Remote Terminal Units – RTUs) to Supervisory Control and Data Acquisition (SCADA) via serial cables (RS232 or RS485). Therefore, the term "Modbus RTU" refers to the Modbus version that utilises serial communication lines for data transmission. However, Modbus messages can also be transmitted via standard Ethernet-based medium over TCP/IP. This Modbus variant is called "Modbus TCP".

According to the Modbus specification [1], Modbus devices are distinguished into Modbus masters and Modbus slaves. Modbus slaves correspond to servers (e.g., Programmable Logic Controllers – PLCs, RTUs, energy meters, etc) that provide measurements and receive commands upon request. On the other hand, Modbus masters correspond to clients, which initiate Modbus requests to the Modbus slaves. A Master Terminal Unit (MTU) usually has the role of Modbus master, undertaking to collect measurements from Modbus slaves and either visualise them or store them in databases.

Modbus messages are used to read or write the contents of memory registries, commonly referred to as Modbus registers. By accessing the Modbus registers, a Modbus master can access measurements or remotely control the Modbus slave. The following types of Modbus registers are defined according to the standard [1]:

- Coils: 1-bit readable and writable memory. They are used to open/close trips.
- Discrete input: 1-bit readable memory. They are used to indicate proper operation or possible faults.
- Input registers: 16-bit readable memory. They are used mainly to retrieve measurements and statuses from the field devices.

- Holding registers: 16-bit readable and writable memory. They are used to change configuration values (e.g., IP address, transformer factor, etc)

A set of function codes are defined by the standard, that apply different operations on the Modbus addresses. The most relevant Modbus function codes for this lab are depicted in Table 1.

**Table 1: Most common function codes in Modbus TCP**

| Function name | Function code (decimal) | Memory type |
|---|---|---|
| **Read Discrete Inputs** | 2 | Discrete input (read-only memory type) |
| **Read Coils** | 1 | Coil (read/write memory type) |
| **Write Single Coil** | 5 | |
| **Write Multiple Coils** | 15 | |
| **Read Input Registers** | 4 | Input register (read-only memory type) |
| **Read Multiple Holding Registers** | 3 | Holding register (read/write memory type) |
| **Write Single Holding Register** | 6 | |
| **Write Multiple Holding Registers** | 16 | |

Finally, in the context of this lab, interaction will be established with a real Modbus TCP PLC that is installed in PPC premises to retrieve the operational status of a 17.5 MW surge generator. The utilised PLC is illustrated in Figure 1. The inner structure of the PLC cabinet is depicted in Figure 2.


**Figure 1: The Unitronics Visio700 PLC | The front side HMI of the PLC in PPC**

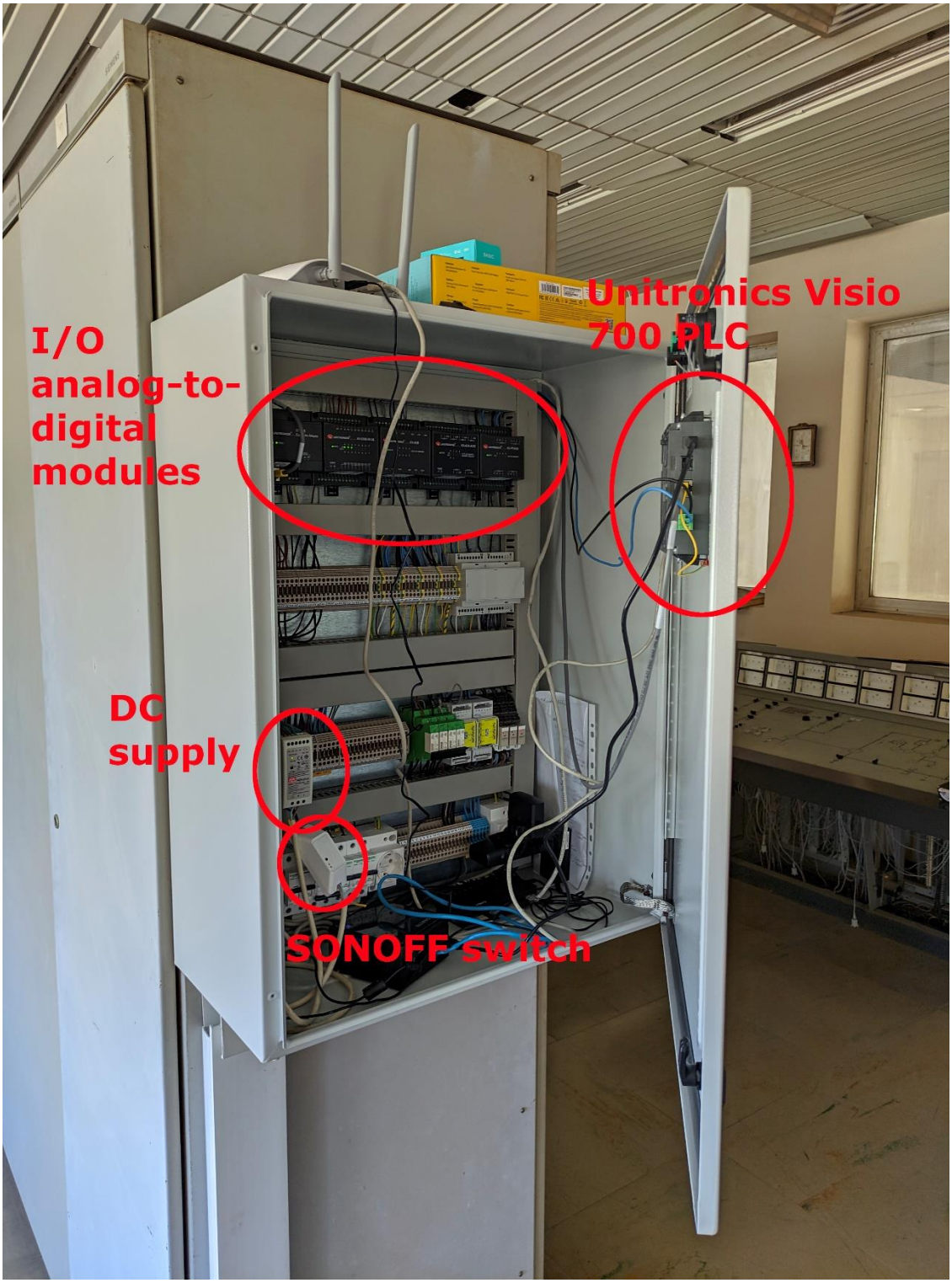



**Figure 2: The inner structure of a PLC cabinet**

Finally, to establish Modbus communication with the PLC, the Modbus register mapping of Table 2 is the main reference documentation. This mapping depends solely on the device manufacturer, for embedded devices, or the programmer in case of programmable devices, like PLCs and RTUs.

**Table 2: The Modbus register map of the PLC in PPC premises**

| Slave ID: 1 | | | | | |
|---|---|---|---|---|---|
| **ID** | **Title** | **Description** | **Modbus Address** | **Function code** | **Normal values** |
| 1 | Power grid 20 KV – Phase R | Indicates that voltage exists on the L1 phase | 400 | Read Discrete Inputs | 1 |
| 2 | Power grid 20 KV – Phase S | Indicates that voltage exists on the L2 phase | 401 | | 1 |
| 3 | Power grid 20 KV – Phase T | Indicates that voltage exists on the L3 phase | 402 | | 1 |
| 4 | Main MG Set Nn | The generator has acquired rated rounds per minutes (rpms) | 403 | | 1 |
| 5 | Exciter MG Set Nn | The exciter has acquired rated rpms | 404 | | 1 |
| 6 | Overvoltage AC Main Generator | Indicates that overvoltage on the main generator exists | 405 | | 0 |
| 7 | Overcurrent Main Generator | Indicates that overcurrent on the main generator exists | 406 | | 0 |
| 8 | 24 V Batteries voltage | - | 400 | Read Holding Registers | 0 – 24V |
| 9 | 60 V Batteries voltage | Measurement not available yet | 401 | | - |
| 10 | Generator motor speed | - | 402 | | 1000 rpm +-5 |
| 11 | Generator motor voltage | - | 403 | | 400 V +- 30 |
| 12 | Generator motor current | - | 404 | | 0 – 850 A |
| 13 | Exciter motor voltage | - | 405 | | 400 V +-30 |
| 14 | Exciter motor current | - | 406 | | 0 – 350 A |
| 15 | Incoming Cooling water | Temperature of incoming cooling water | 450 | | 10 - 40 °C |
| 16 | Main generator status winding 2 | Temperature of generator winding at point 2 | 451 | | 10 – 35 °C |
| 17 | Main generator outlet air | Temperature of outlet air | 452 | | 10 – 35 °C |
| 18 | Exciter MG set bearing 2 | Temperature of exciter winding at point 2 | 453 | | 40 – 66 °C |
| 19 | Remote command | We have not assigned to this command any specific role. This command could open or close a relay or circuit. | 450 | Write Single Coil / Read Coil | NaN |

## 2.3 Lab configuration

The following parameters are provided by the lab instructor:

| Parameter | Value |
|---|---|
| **Workstation IP address** | |
| **Access method** | SSH and/or VNC |
| **Workstation Username (also for SSH)** | |
| **Workstation Password (also for SSH and VNC)** | |
| **IP address of the PLC** | |
| **Modbus TCP port of the PLC** | 502 |
| **IP address of the Raspberry Pi 4B (RPI4B)** | |
| **RPI4B Username** | |
| **RPI4B Password** | |

# 3. Exercise 1: Implementation of a Modbus TCP master

Goal of this exercise is to implement a Modbus TCP master software that retrieves and displays the measurements from the real PLC located in the physical laboratory. To this aim, the pymodbus Python library will be utilised.

## Step 1: The basic structure

Start by preparing a new virtual Python environment on your workstation (Refer to section 2.3 for access details). Then, create an empty file and integrate the source code provided bellow:

```
1  from pymodbus.client.sync import ModbusTcpClient
2  from datetime import datetime
3  import time
4  import random
5  import json
6
7  if __name__ == "__main__":
8
```

The required libraries are imported in lines 1-5. In particular:

- The synchronous ModbusTcpClient class is imported from the pymodbus.client.sync module. This class is utilised to represent synchronous Modbus TCP clients and perform basic operations on them (establishing connections, sending Modbus messages, etc).
- `datetime` class is used to create accurate timestamps.
- `time` is used to create time delay between subsequent operations.
- `random` is used to create random numbers.
- `json` is used to create JSON messages.

The source code of the Modbus master will be placed inside the if clause (line 8).

## Step 2: Defining the Modbus attributes

Add the following source code in the main function:[1]

```
1  if __name__ == "__main__":
2      DI_ADDRESSES = (400, 401, 402, 403, 404, 405, 406)
3      HR_ADDRESSES_1 = (400, 401, 402, 403, 404, 405, 406)
4      HR_ADDRESSES_2 = (450, 451, 452, 453)
5      COIL_ADDRESS = 450
6      INTERARRIVAL = 5
7      UNIT_ID = 1
8      MB_SLAVE_IP_ADDR = "PLC IP ADDRESS"
9      MB_SLAVE_TCP_PORT = 502
```

The following attributes are defined:

- In line 2, the discrete input addresses are defined, according to the Modbus mapping (Table 2).
- Then, in line 3, the first set of sequential holding register addresses are defined, according to Table 2.
- The second set of sequential holding register addresses are defined in line 4.
- The interarrival is defined in line 5, i.e., how often the Modbus master will query for the contents of the Modbus addresses.
- The unit ID is defined in line 7, also known as slave ID.
- Finally, the IP address and the Modbus slave TCP port are defined in lines 8, 9 respectively. Refer to section 2.3 for the correct value.

## Step 3: Establishing reliable Modbus TCP connection

Expand the source code of the main function as follows:

```
1  if __name__ == "__main__":
2
3      # <source code of step 2>
4
5      slave = ModbusTcpClient(MB_SLAVE_IP_ADDR, port=MB_SLAVE_TCP_PORT)
6      while True:    # main loop
7          while True:    # loop for establishing Modbus connection
8              if not slave.is_socket_open():
9                  successful = slave.connect()
10                 if successful:
11                     print("Connection with " + MB_SLAVE_IP_ADDR + "
   successful.")
12                     break
13                 else:
14                     print("Connection not successful. Retrying after
   10 seconds"
```

[1] DI = Digital Input, HR = Holding Registers

```
15                          time.sleep(10)
16                          continue
17          # <source code of step 4>
18          # <source code of step 5>
19
```

The above source code creates a Python object that represents the PLC (line 5) and aims to establish reliable TCP connection with that device. Two loops are defined. The outer loop (line 6) corresponds to the repeated reading of measurements from the Modbus PLC. The inner loop (line 7-16) corresponds to the establishment of the TCP connection with the Modbus PLC. In more detail:

- First, it is checked whether a connection with the PLC has already been established (line 8). If a connection already exists, then the script process to retrieving the measurements.
- If no connection has been established, then the script attempts to establish a new TCP session (line 9). Return value of the connection method is a Boolean value that indicates whether the connection has been established successfully.
- If the return value is True, then the connection has been established successfully, and the loop breaks (line 11-12). Otherwise, if the connection has not been established, the script freezes for 10 seconds (line 15) and the inner loop continues (line 16).

## Step 4: Reading the contents of Modbus addresses

Expand the script of step 3 as follows:

```
1   if __name__ == "__main__":
2
3       # <source code of step 2>
4
5       # <source code of step 3>
6
7       response_di = slave.read_discrete_inputs(DI_ADDRESSES[0],
    len(DI_ADDRESSES)m unit=UNIT_ID)
8       response_coil = slave.read_coils(COIL_ADDRESS, unit=UNIT_ID)
9       response_hr_1 = slave.read_holding_registers(HR_ADDRESSES_1[0],
    len(HR_ADDRESSES_1), unit=UNIT_ID)
10      response_hr_2 = slave.read_holding_registers(HR_ADDRESSES_2[0],
    len(HR_ADDRESSES_2), unit=UNIT_ID)
11
```

In this step, the contents of the previously defined Modbus addresses are read from the PLC in a synchronous way. In this context, synchronous means that the script flow is blocked until each TCP transaction is completed. Thus, line 7 initiates a TCP transaction to retrieve the contents of the discrete input Modbus addresses. When this transaction is completed, lines 8, 9, and 10 retrieve the contents of the coil and holding register addresses respectively. Since the holding register addresses are not continuous, two separate requests are required.

## Step 5: Handling errors

After the Modbus requests, add the following code:

```
1  if __name__ == "__main__":
2
3      # <source code of step 2>
4      # <source code of step 3>
5      # <source code of step 4>
6
7      if response_di.isError() or response_coil.isError() or
       response_hr_1.isError() or response_hr_2.isError():
8          print("Modbus error occurred. Retrying after 60 seconds...")
9          slave.close()
10         time.sleep(60)
11         continue
```

The source code of this steps aims to handle unexpected errors, including invalid requests, invalid address map or internal errors caused by the PLC. Until now, pymodbus does not raise exceptions for such errors, therefore, the response is checked manually after executing the requests. In case any error is detected, the script gracefully closes the TCP session (line 9) and retries after 1 minute.

## Step 6: Prepare JSON format – Final script

Unify the source code of the previous steps to a single script, that repeatedly queries the Modbus PLC for the contents of the Modbus addresses. The results should be converted in JSON format. In addition, the current timestamp should be added in the JSON object, as additional key/value pair. Finally, after printing the results, the script should pause for the interval specified in step 2 / line 6 before proceeding to the next iteration.

# 4. Exercise 2: Emulate a PLC by implementing a Modbus TCP slave

Goal of this exercise is to implement a Modbus TCP slave, utilising the pymodbus Python library. The Modbus TCP slave of this exercise will imitate the real PLC in PPC premises, by using the same Modbus addresses and function codes as the real devices [2].

## Step 1: The basic structure

The main structure of the Python source code for this exercise is provided bellow.

```
1  from pymodbus.version import version
2  from pymodbus.server.asynchronous import StartTcpServer
3  from pymodbus.device import ModbusDeviceIdentification
4  from pymodbus.datastore import ModbusSequentialDataBlock,
   ModbusSparseDataBlock, ModbusServerContext
```

```
5  from threading import Timer
6
7  import logging
8  logging.basicConfig()
9  log = logging.getLogger()
10 log.setLevel(logging.DEBUG)
11
12
13 def worker_function(a):
14
15
16 if __name__ == "__main__":
17
```

The following pymodbus modules are imported in lines 1-4:

- pymodbus.version is used to retrieve the version of Pymodbus library.
- The StartTcpServer class of the pymodbus.server.asynchronous module that implements the Modbus TCP server in an asynchronous way.
- The ModbusDeviceIdentification of the pymodbus.device module supplies the Modbus TCP slave with the additional information for device identification. Function code 43 is used by a Modbus master in order to read this information. Whilst the Modbus address mapping of the PPC's PLC does not include asset information, it is a good practice to implement this function code on Modbus slaves.
- The ModbusSequentialDataBlock class of the pymodbus.datastore module is utilised to define the virtual memory stores, where measurements are stored by the Modbus TCP slave device. The blocks correspond to Modbus addresses that a master uses to retrieve that information.
- The ModbusSlaveContext class of the pymodbus.datastore module defines the function codes and slave IDs that the Modbus TCP slave device should support. An attribute of the ModbusSlaveContext is the address space provided by the ModbusSequentialDataBlock.

The Threading class imported in line 5 is used to regularly execute the worker function as a separate thread in parallel with the main server thread, to update the values of the Modbus addresses.

In lines 7-10, the logging capability of the Modbus slave is configured. It is considered a good practise to show log messages via the Python logger instead of the print() function. At a latter stage, the developer could configure the logger once to apply a specific format to all logs, or to save them in a rotated log file.

The source code of the worker function is placed in line 14.

Finally, the main source code that spawns the Modbus TCP server is placed in line 17.

## Step 2: Initialise the data structures

Update the main function as follows:

```
1  if __name__ == "__main__":
```

```
2    slave_id = 0x01
3    store = {
4      slave_id: ModbusSlaveContext(
5                  di=ModbusSequentialDataBlock(400, [1]*7)
6                  hr=ModbusSparseDataBlock({400: [0]*7, 450: [23]*4})
7                  co=ModbusSequentialDataBlock(450, 0))
8    }
9    context = ModbusServerContext(slaves=slaves, single=False)
10
11   identity = ModbusDeviceIdentification()
12   identity.VendorName = "Unitronics"
13   identity.ProductCode = "Vision700"
14   identity.VendorUrl = "https://www.unitronicsplc.com/"
15   identity.ModelName = "V700-T20BJ"
16   identity.MajorMinorRevision = version.short()
```

The Modbus data storage is defined in lines 3-8. The storage is defined as a JSON object that consists of the slave ID (unit ID) and the corresponding slave context. The ModbusSlaveContext constructor consists of the di, hr and co arguments that correspond to digital inputs, holding registers and coils respectively. Assigning a data block to one of the above arguments, the corresponding function codes are activated. In particular:

- A sequential address space of digital input addresses is defined in line 3. Starting address is 400, while the block totally spans 7 addresses (i.e., 400 to 406). The address contents are initialised with the value 1.
- A sparse (non-sequential) address space of holding registers is defined in line 4. The first block starts at 400 and spans until 406 addresses. The second block starts at 450 and ends at 403.
- A single address is defined as coils in line 5, at address 450 with zero initial value.

The slave context is finally defined in line 9. By setting single to False, only the provided slave ID returns the slave context.

The server information is initialised in lines 11-16, including typical information according to the Modbus standard.

## Step 3: Worker process for updating addresses contents

The following source code updates the contents of the Modbus addresses, by increasing the existing values by 1 each time.

```
1  def updating_data(context):
2    log.debug("Updating the context")
3    function_code = 'hr'
4    slave_id = 1
5    address = 400
6    values = context[slave_id].getValues(function_code, address, count=7)
7    values = [v+1 for v in values]
```

| | |
|---|---|
| 8 | log.debug("New values: " + str(values)) |
| 9 | context[slave_id].setValues(function_code, address, values) |

In lines 3-5, static values are initialised for the operation, including the alphanumeric representation of the function code (hr for holding registers), the unit/slave ID and the starting Modbus address.

The current values are retrieved in line 6. Then, the values are increased by 1 in line 7.

Finally, the new values are being stored to the Modbus addresses in line 9.

## Step 4: Run the Modbus server – Final script

The final source code of the Modbus server is provided bellow:

```
1  from pymodbus.version import version
2  from pymodbus.server.asynchronous import StartTcpServer
3  from pymodbus.device import ModbusDeviceIdentification
4  from pymodbus.datastore import ModbusSequentialDataBlock,
   ModbusSparseDataBlock, ModbusServerContext
5  from threading import Timer
6
7  import logging
8  logging.basicConfig()
9  log = logging.getLogger()
10 log.setLevel(logging.DEBUG)
11
12 slave_id = 1
13
14 def updating_data(context):
15     log.debug("Updating the context")
16     function_code = 'hr'
17     start_address = 400
18     values = context[slave_id].getValues(function_code, start_address,
   count=7)
19     values = [v+1 for v in values]
20     log.debug("New values: " + str(values))
21
22
23 if __name__ == "__main__":
24     store = {
25       slave_id: ModbusSlaveContext(
26                 di=ModbusSequentialDataBlock(400, [1]*7)
27                 hr=ModbusSparseDataBlock({400: [0]*7, 450: [23]*4})
28                 co=ModbusSequentialDataBlock(450, 0))
29     }
30     context = ModbusServerContext(slaves=slaves, single=False)
31
32     identity = ModbusDeviceIdentification()
33     identity.VendorName = "Unitronics"
34     identity.ProductCode = "Vision700"
35     identity.VendorUrl = "https://www.unitronicsplc.com/"
36     identity.ModelName = "V700-T20BJ"
```

```
37      identity.MajorMinorRevision = version.short()
38
39      frequency = 5
40      thread = Timer(frequency, updating_data, args=(context,))
41      thread.start()
42      StartTcpServer(context, identity=identity, address=("0.0.0.0", 502))
```

The lines 39-42 have been added in this step. In particular, the Timer function is used in order to create a Python thread that is executed repeatedly. The thread is started in line 41. Finally, the Modbus TCP server is started in asynchronous (non-blocking) mode.

Remember that processes not run by the root user cannot bind TCP/UDP ports bellow 1024. This means that a Modbus TCP server binding port 502 should be run as sudo. If you are not able to run the Python script with sudo privileges, please replace 502 with 5020.

### Step 5: Validate using the Modbus TCP master

Try to use the source code of Exercise 2 to communicate with the Modbus slave implemented in this exercise. You should be able to successfully communicate with the emulated Modbus PLC

## 5. Exercise 3: Implement a custom Modbus TCP slave for Adafruit SHT40

Objective of this exercise is to implement a Modbus TCP interface for retrieving the measurements of Adafruit SHT40 in real-time.

Based on Exercise 2, try to implement a Modbus TCP slave that runs on the Raspberry Pi 4B device (access details in section 2.3). The Modbus TCP slave must be able to provide the humidity and temperature measurements of the SHT40 sensor. Moreover, information like manufacturer ID and device ID should also be provided by the Modbus TCP slave, utilising function code 47. To implement this exercise, the following points should be considered:

- Custom Modbus addresses should be reserved for the attributes/measurements (entirely your choice).
- The appropriate function codes should be configured for the chosen Modbus addresses. Choose the right function code(s), based on the supported operations.

## 6. Exam Questions

1. What is the difference between a Modbus TCP and a Modbus RTU device?
2. What information would you need to access the measurements provided by a Modbus TCP device?
3. What is the difference between an input register and a holding register?
4. Which type of Modbus address would you use to allow you read and write a Boolean variable, i.e., a variable that can be either true or false?
5. Which type of Modbus address would you use to allow you read integer values?
6. As a developer, since only integers can be stored in holding registers, how would you accommodate the storage of a float variable (e.g., temperature) in a holding register address?

## References

[1] Modbus Organization, Inc, “MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b3,” 26 April 2012. [Online]. Available: https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf. [Accessed 28 April 2022].

[2] Pymodbus Community, “Pymodbus Documentation,” 2017. [Online]. Available: https://pymodbus.readthedocs.io/en/latest/source/example/updating_server.html. [Accessed 28 April 2022].

## 7. Contacts

**Project Coordinator:**

- Name: Technical University of Sofia
- Address:
    - Technical University of Sofia,
      Kliment Ohridsky Bd 8
      1000, Sofia, Bulgaria
- Phone: +3592623073

**Output 2 Leader:**

- Name: FOSS Research Centre for Sustainable Energy, University of Cyprus
- Address:
    - University of Cyprus,
      Panepistimiou 1 Avenue
      P.O. Box 20537
      1678, Nicosia, Cyprus
- Email: foss@ucy.ac.cy
- Phone: +357 22 894288