Modbus Script Documentation

Leaf Controller

[16th February 2021]

Document Control Number:

Revision Number: 0.1

Expiration Date:

Author: Georg Kordowich

Approval: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Revision History

|  |  |  |
| --- | --- | --- |
| Date | Revision Number | Pages Affected |
| 20th February 2021 | 0.1 | Initial release |

Table of Contents

[1. Using the Modbus Script 2](#_Toc64445686)

[1.1 Using the Modbus Script with command line arguments 2](#_Toc64445687)

[1.1.1 Setting the IP-Address and Port of the Clients 3](#_Toc64445688)

[1.1.2 Reading devices from the command line 4](#_Toc64445689)

[1.1.3 Advanced Settings (Cyclic Reading, Server function). 4](#_Toc64445690)

[1.1.4 Reading and writing single devices manually 5](#_Toc64445691)

[1.2 Using the Modbus Script with a config file 7](#_Toc64445692)

[2. Internal structure of the script 9](#_Toc64445693)

[2.1 Settings and Configuration 9](#_Toc64445694)

[2.2 Main Components 9](#_Toc64445695)

[2.3 Definition of devices 10](#_Toc64445696)

[2.4 Definition of devices in the server context 10](#_Toc64445697)

# Using the Modbus Script

There are two options how to use the Modbus Script. The first one is using it with command line arguments. This is recommended for testing. The other option is using a config file. This is recommended for usage in production.

The script has two main functions: The client functionality and the server functionality.

The client functionality is used to read values from and write values to clients. Those clients can be devices like a Chiller, HVAC etc.

The server functionality is used so the script can receive values from a superordinate entity (for example the core controller). The server runs in an asynchronous thread. The received values are added to a queue and can then be written to the clients when there is time.

## Using the Modbus Script with command line arguments

Generally, a python script can be started the following way:

controllino\_modbus.py

Or, If python is not added to the path:

python3 controllino\_modbus.py

All arguments which are presented in the following can be combined.

### Setting the IP-Address and Port of the Clients

When using the script, it may be useful to change the Ip-Address and port of the clients. If this is desired, the first command-line argument must be a valid Ip-Address and the second one the Port.

Examples:

controllino\_modbus.py 192.168.0.1 502 <additional arguments>

controllino 192.168.0.1 502 controllino notserver

The Ip-Address of the server will be set automatically, as it is identical with the Ip-Address of the device which the script is running on.

### Reading devices from the command line

Note that the Ip-Address and Port can be set as desired as explained before.

A device can be read in the following way:

controllino\_modbus.py <device>

The following devices can be read:

* controllino
* chiller1
* chiller2
* hvac1
* hvac2
* osensa
* dl10
* flowmeter1
* ups
* flowmeter2
* virtualslave
* hvacTelco
* upsTelco

Example:

controllino\_modbus.py controllino

It is also possible to read multiple devices:

controllino\_modbus.py controllino flowmeter2 chiller1

It is also possible to read all devices:

controllino\_modbus.py all

It is also possible to read all devices except one:

controllino\_modbus.py all notcontrollino

It is also possible to read all devices except multiple:

controllino\_modbus.py all notcontrollino nothvac1

### Advanced Settings (Cyclic Reading, Server function).

The server functionality can be activated using the following command:

controllino\_modbus.py server

And deactivated using the following command:

controllino\_modbus.py notserver

The client functionality can be activated using the following command (it is activated per default though)

controllino\_modbus.py clients

And deactivated using the following command (Note: No clients will be read when deactivating the client functionality):

controllino\_modbus.py notclients

If desired, the clients can be read not only once, but cyclic. This is especially useful when also using the server.

controllino\_modbus.py cyclic

The sleep time, which is the time that the script waits between two cycles, can be set in the following way (Note: Unit is seconds)

controllino\_modbus.py sleepTime 3

The mode in which the script operates can also be changed. There are three modes: The production mode will not ask for any inputs from the user and will print only the most important information. The manual mode will ask for user input and print more information. The testing mode will not read devices, but instead simulate the reading of devices by using random values. This can be useful when not having real devices available.

controllino\_modbus.py mode Testing

controllino\_modbus.py mode Manual

controllino\_modbus.py mode Production

### Reading and writing single devices manually

The script can be used to read from and write to devices manually. Note that this is a different operating mode, as the script will only execute the single operation and exit afterwards.

#### Reading

Reading one single register:  
First Argument after readsingle is the device (unit Id can also be given instead), the second is the symbolic register name:

controllino\_modbus.py readsingle chiller1 systemOnOff

Alternatively, instead of the symbolic register name, the register type (co, di, ir, hr) and address can be given:

controllino\_modbus.py readsingle 1 ir 1024

It is also possible to read multiple registers:  
In this case, the first register and the last register that should be read must be given:

controllino\_modbus readmultiple chiller1 systemOnOff waterTempSet

Again, instead of the symbolic register name, the register type (co, di, ir, hr) and start address and end-address can be given:

controllino\_modbus readmultiple 42 ir 3 5

#### Writing

Writing one single register:  
First Argument after ‘write’ is the device (unit Id can also be given instead), the second is the symbolic register name, the last one is the value that shall be written. Optionally, the ‘force’ keyword can be used before the ‘write’ keyword write even without write-permission. This can lead to unexpected behaviour:

controllino\_modbus.py write chiller1 systemOnOff 8

controllino\_modbus.py force write chiller1 systemOnOff 8

Alternatively, instead of the symbolic register name, the register type (co, di, ir, hr) and address can be given:

controllino\_modbus.py write chiller1 hr 1024 8

controllino\_modbus.py force write chiller1 hr 1024 8

## Using the Modbus Script with a config file

A config file can be used with the script. Note that the values from the config file will be completely ignored when starting the script with any command line arguments. An example configuration can be found below. The name of the config file shall be controllino\_modbus.conf. The script will look for the config file automatically in the current working directory of the script and in the hardcoded default path ("/usr/local/share/fluence/controllino\_modbus.conf").

Alternatively the path to the config file can be defined by using “-f” in the command line.

Example:

controllino\_modbus.py -f /home/usr/local/

The ‘#’ sign can be used to comment lines, so they will be ignored.

Important: When using command line arguments, most settings are overwritten! Only Unit IDs and special Ip-Addresses will not be overwritten.

Example Configuration:

# Config File for the controllino\_modbus Script

# Specifies on which controller the Script runs. 'Cube' and 'OCTE' are currently the only two options.

controllerType = Cube

# The mode determines the behaviour of the script:

# 'Testing' will simulate a modbus connection by using random values, if connection is impossible

# 'Manual' will use input values from the user.

# 'Production' won't request any input from the user and will not print out any values, only errors

mode = Manual

# IP-Adresses and Ports

clientIpAddress = 192.168.2.3

clientPort: 502

#serverIpAddress: 172.16.1.111 #Usually the script will choose the correct Ip-Address.

serverPort: 1502

#Enable/disable the server functionality of the script.

ServerActive: True

#If set to false, no client will be read, if set to true all enabled clients will be read.

ClientsActive: True

# If set to false, every client will be read only once.

cyclicRead: True

#enable/disable single devices

controllinoRead: False

chiller1Read: False

chiller2Read: False

hvac1Read: False

hvac2Read: False

osensaRead: False

dl10Read: False

flowmeter1Read: False

upsRead: False

flowmeter2Read: False

virtualSlaveRead: True

hvacTelcoRead: False

upsTelcoRead: False

# Time between read cycles in seconds

sleepTime: 2

#Modbus IDs

controllinoUnitId = 42

chiller1UnitId = 1

chiller2UnitId = 2

hvac1UnitId = 3

hvac2UnitId = 4

osensaUnitId = 10

dl10UnitId = 5

flowmeter1UnitId = 20

upsUnitId = 21

flowmeter2UnitId = 22

virtualSlaveUnitId = 100

hvacTelcoUnitId = 47

upsTelcoUnitId = 48

# Client Ip Addresses (only necessary when they differ from the normal client Ip Address)

upsTelcoIp = 127.0.0.1

upsTelcoPort = 502

### Special Configuration when using the Config File (Ip-Addresses)

When using the config file, different Ip-Addresses can be used for different clients. It is important, that both an Ip-Address and a Port are given, otherwise it will be ignored. If no Ip-Address or Port is given for a device, the normal Client Ip-Address will be used.

# Internal structure of the script

The following section is useful when changing code inside the script.

## Settings and Configuration

Line 74   
The logging level can be set to Critical or Debug. Debug may be useful especially when debugging server functionality.

log.setLevel(logging.CRITICAL)

Lines 85 – 127  
The default values are set here. These values are especially useful when starting the script with command line arguments.

controllino\_modbus readmultiple 42 ir 3 5

All settings are stored in an instance “scriptSettings” of the class “modbusScriptSettings”, which is defined in controllino\_modbus\_utils.py

## Main Components

In Line 1513 the Thread in which the clients are read and written is defined. In line 1522, the Modbus TCP server is started in the Main Thread. Both threads are started in line 1539 with the reactor.run() call.

reactor.callInThread(loop\_transfer\_data\_client\_to\_server)

StartTcpServer(context, identity = identity, address = (scriptSettings.serverIpAddress, scriptSettings.serverPort), defer\_reactor\_run = True)

In the “Client-Thread”, at first all clients are read and the values are printed. Afterwards, the queue, in which the Server-Thread puts write-requests is checked and the write requests are processed. Please be aware that this is a potential bottleneck, as it may take a long time to read all clients and therefor it may take a long time until the write-requests are processed. This problem could be circumvented by checking the queue after every single device read, but this has other drawbacks.

Server Thread

External Source (e.g. superordinate controller)

External Source (e.g. superordinate controller)

Client Thread

Queue

Reads/writes

Server Thread writes Write-Requests to Queue

Clients Thread gets Write-Requests from Queue and processes them

Modbus Device  
(e.g. Chiller)

Modbus Device  
(e.g. HVAC)

Client Thread writes read values to Server Thread storage

Reads/writes

## Definition of devices

To make the definition of new devices easier, a class called “ModbusDevice” has been defined in the file “controllino\_generic\_modbus\_device”. It can be initialized using a Unit Id, and a List of Registers. During the initialization, a dataframe will be created, in which the register names, addresses, values and read/write permissions are stored. In the class, two methods are defined: “readAll” and “printDataFrames”. The readAll-Function will read all registers, that were previously defined in the List of Registers. It will try to read as many registers consecutively as possible (maximum 30 for stability reasons). The printDataFrames function will print all defined dataframes. It may be useful to overwrite the readAll function to implement a custom read function for special devices. One example for that is the UPS. It is useful to create a child class of the ModbusDevice class for each device-type.

The devices are mainly used for the client side and for the ability to print humanly readable output. In the function readClientsWriteToServer the data will be transferred to the server context. Note the parameter setQueue = False. This parameter is necessary to see, if the values in the server-context have been set by an external source (in which case setQueue = True) or by the script itself. If setQueue==True, the script will assume new values have been set by an external source and therefore write the new values to the modbus device.

## Definition of devices in the server context

The slaves of the server are defined in lines 1405 ff. This is necessary, so that the values of the slaves can be read asynchronously. Each slave datastore is an instance of the class CustomModbusSlaveContext, which is a child class of the class ModbusSlaveContext. The slave contexts can be read and set asynchronously from an external source. If new values are set, a write request entry is put in the queue. This queue will be processed periodically in the function writeToClients. If there are any write requests in the queue, the values will be written to the according physical devices.

## The Virtual Device

The Virtual Device should not be thought of as a physical device, but as an interface to aggregate registers from multiple devices. In the virtual slave register list, the address and Unit Id are not the ones of the virtual device, but instead point to the correct register of the real device. This enables the script to pass values, which are written to the virtual device, directly to the correct device. However, some care must be taken, when writing to the virtual device, as some values (e.g. aggregated ones), can not simply be passed to the real device.

Virtual Device

External Source (e.g. superordinate controller)

Real Device (e.g. UPS)

Real Device (e.g. Chiller)

Real Device (e.g. HVAC)

Reads/writes

Reads/writes

Reads/writes

Reads/writes

## Adding a new Device

When adding a new device, the following steps must be followed. To find out how exactly each step is done, I recommend looking at how the chiller is implemented, as it is a very typical modbus device.

|  |  |
| --- | --- |
| Create a file with a register list and a class definition. The register list can usually be obtained from the manual of the device. | New File |
| Import the file | controllino\_modbus.py |
| Define the necessary settings in the class modbusScriptSettings | controllino\_modbus\_utils.py |
| Assign default settings | controllino\_modbus.py (lines 136ff) |
| Assign the correct client in the function getClient() | controllino\_modbus.py (lines 187ff) |
| Initialize the client in clientsInit() | controllino\_modbus.py (lines 269ff) |
| Create an instance of the previously defined class | controllino\_modbus.py (lines 337ff) |
| Add the device to the devDict | controllino\_modbus.py (lines 443ff) |
| Close the custom client when shutting down the script | controllino\_modbus.py (lines 1117ff) |
| Add device to the function readClients() and readClientsWriteToServer() | controllino\_modbus.py (1163ff) |
| Add device to the modbus server context | controllino\_modbus.py (1557ff) |
|  |  |
| (Optional) Add the necessary keywords where the command line arguments are processed | controllino\_modbus.py (lines 705ff) |
| (Optional) Add the necessary keywords where the config file is processed in readConfiFile() | controllino\_modbus\_utils.py (lines 545ff) |