How to program MicroPython file for proprietary serial device

Moxa Technical Support Team

support@moxa.com

Contents

Released on Nov 22, 2024

About Moxa

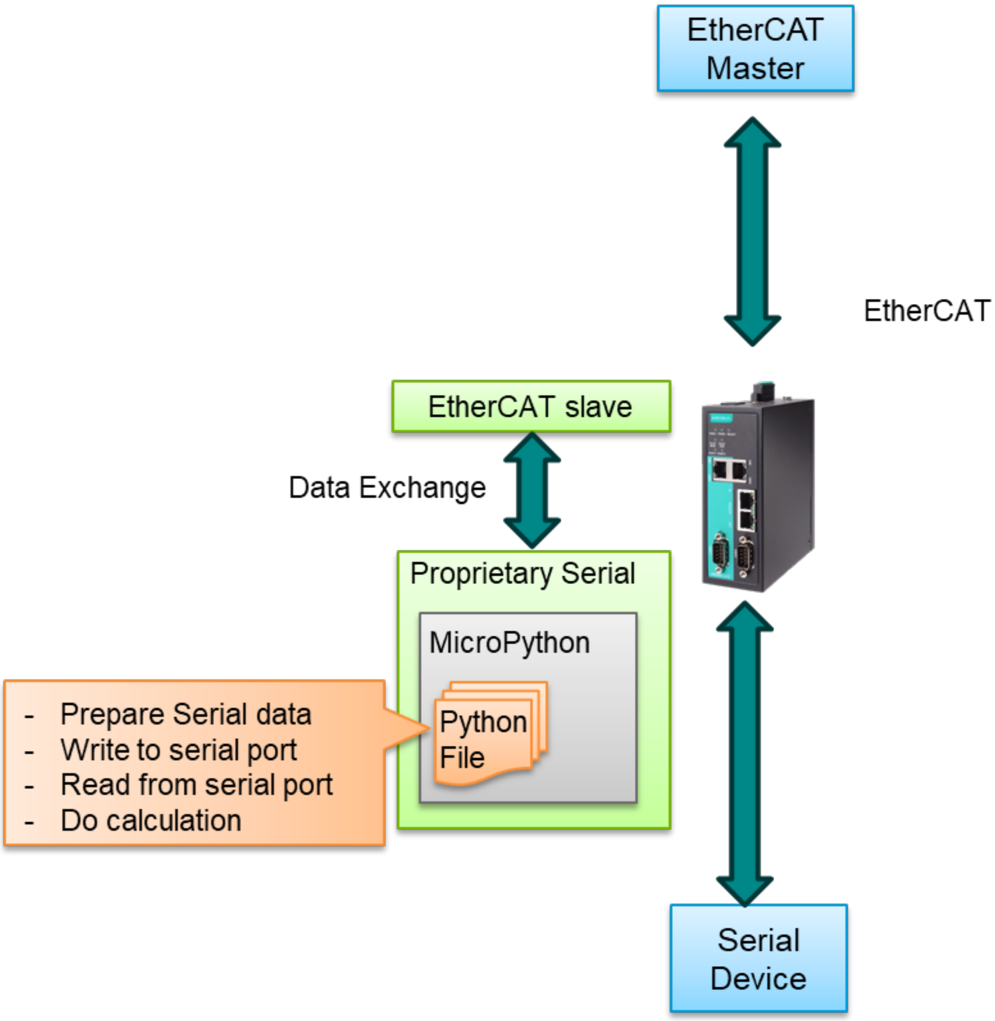
Moxa is a leading provider of edge connectivity, industrial computing, and network infrastructure solutions for enabling connectivity for the Industrial Internet of Things. With 35 years of industry experience, Moxa has connected more than 82 million devices worldwide and has a distribution and service network that reaches customers in more than 80 countries. Moxa delivers lasting business value by empowering industry with reliable networks and sincere service for industrial communications infrastructures. Information about Moxa's solutions is available at www.moxa.com.

1 Introduction

The MGate 5216 Series EtherCAT industrial Ethernet gateways convert data between Modbus RTU/ASCII, proprietary serial, and EtherCAT protocols. To integrate existing serial devices into an EtherCAT network, use the MGate 5216 as a serial master to collect data and exchange data with the EtherCAT host.

This document provides a tutorial on how to write a MicroPython file to communicate with serial devices, including preparing the request package with fixed data or data from EtherCAT host, sending and receiving serial bits and bytes through the serial port, handling the response, and transmitting the data to the EtherCAT host.

For configuring proprietary serial operation modes and settings, refer to the user manual.



2 Writing MicroPython Scripts

2.1 Programming tool

To write MicroPython scripts, which will be executed in the MGate 5216, you can use any text editor (e.g. Notepad++) or other programming tool supporting MicroPython syntax to check the validation of the file.

You can download the example codes and change them to fit the specific serial device you’re using.

2.2 Modules

The MGate 5216 supports executing a MicroPython file to communicate with a serial device. We provide a module that supports the following API to enable the MGate to send or receive serial data from a specific serial port. You can use this API in the MicroPython file to achieve this.

2.2.1 mx_sio

The module [mx_sio] can be used to read/write data from and to serial ports. Here are the functions in this module:

mx_sio.read(port, buffer, length, timeout)

This function reads data from a specific serial port.

port: The serial port number to read. Usually passes the input parameter of main().

buffer: A byte array to hold the read data. The size should be large enough to store [length] bytes of data.

length: The number of bytes to read.

timeout: Specifies the maximum time limit in milliseconds to read data before returning the number of bytes read upon timeout.

This function will try to read [length] bytes of data from [port] until [timeout] ms and place the data in [buffer]. Then it will return the number of bytes read.

mx_sio.write(port, data, length)

This function writes data to a specific serial port.

port: The serial port number to write. Usually passes the input parameter of main().

data: A byte array containing the data being written.

length: The number of bytes to write.

mx_sio.flush()

This function clears the input/output buffer of the serial port. When you receive data timeout or read/write fail, call flush to prevent receiving incomplete data.

mx_sio.endian_swap (type, buffer)

This function applies endian swap to the data buffer.

type: The endian swap type to be applied to the data buffer.

0: None

1: Byte

2: Word

3: Byte-and-word

Here is an example of each type:

The original data is 0x12345678.

The None endian swap will be 0x12345678.

The Byte endian swap will be 0x34127856.

The Word endian swap will be 0x56781234.

The Byte-and-word endian swap will be 0x78563412.

buffer: The data buffer to be applied for endian swap.

2.2.2 Denylist

We use MicroPython V1.20.0 to execute the MicroPython script. You can import other supported modules for calculations or data processing. Due to security concerns, certain libraries and functions are restricted from use. The denylist is as follows:

MicroPython standard libraries and micro-libraries:

- io
- os
- select
- socket
- ssl
- sys
- uasyncio
- zlib
- _thread

MicroPython-specific libraries: (all)

- bluetooth
- btree
- cryptolib
- framebuf
- machine
- micropython
- neopixel
- network
- uctypes
- WM8960

built-in functions:

- eval
- exec
- open

2.3 Main function

The MGate 5216 executes the [main] function to start the script. The format of the main() is fixed. Any script to be imported should have the same primary function, and here is the format:

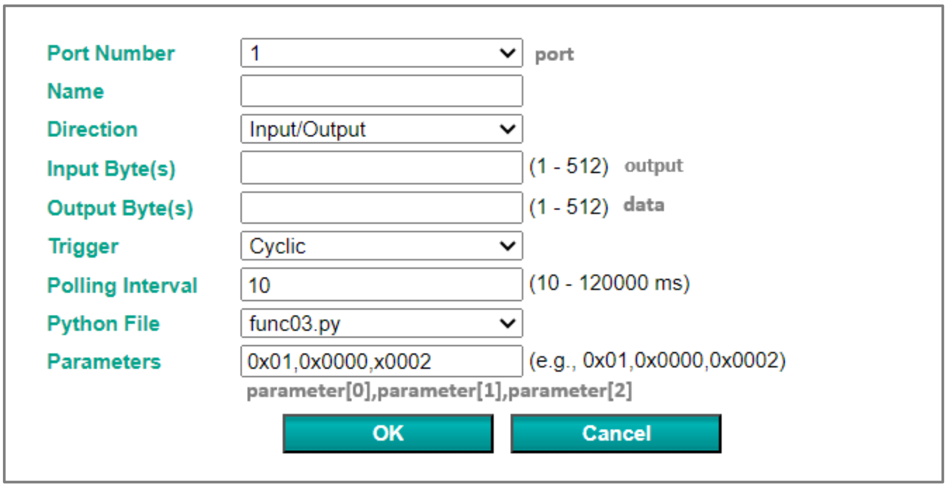
def main(port, parameter, data, output):

port: input parameter that is automatically set to the [Port Number] on the Proprietary Serial web setting page. Used in calling [mx_sio] functions.

parameter: input parameter that is automatically set to the [Parameters] on the Proprietary Serial web setting page. This is useful when you have several commands with the same packet format to send.

data: input parameter for when the script is used in the [Output] or [Input/Output] direction. The mapped data from EtherCAT will be placed in this input parameter.

output: output parameter for when the script is used in the [Input] or [Input/Output] direction. This is used to carry out the mapped data to EtherCAT.



Port Number	1 ▾	port
Name		
Direction	Input/Output ▾	
Input Byte(s)		(1 - 512) output
Output Byte(s)		(1 - 512) data
Trigger	Cyclic ▾	
Polling Interval	10	(10 - 120000 ms)
Python File	func03.py ▾	
Parameters	0x01,0x0000,x0002 parameter[0],parameter[1],parameter[2]	(e.g., 0x01,0x0000,0x0002)

2.4 Example: A simple example of reading Modbus device registers

For the rest of this chapter, we will show you step by step an example that sends a request to a Modbus device and reads values of registers. Please reference the example file **func03.py**

The example file contains comments. For a clearer understanding, we will explain line by line.

```
1.   import mx_sio
```

This is to import the pre-defined modules. [mx_sio] is the module we provide to do serial read/write in the MicroPython script. Normally, every script should import [mx_sio] to use the functions we provide for serial communication.

```
3.   LOW_BYTES = b'\
...
51. def add_crc(package: bytes) -> bytes:
52.     return package + crc16(package)
```

The constants and two functions are used to calculate the Cyclic Redundancy Check (CRC) that needs to be appended to request a package of Modbus serial. Error check bits can vary across different serial devices, and their implementation can also be achieved in multiple ways. This is just one way to implement CRC for Modbus serial.

```
54. def main(port, parameter, data, output):
```

The fixed entry point of the MGate 5216 executing the imported script. The function name and parameters should be the same as in the example. The parameters are described as:

port: input parameter that is automatically set to the [Port Number] on the Proprietary Serial web setting page. Used in calling [mx_sio] functions.

parameter: input parameter that is automatically set to the [Parameters] on the Proprietary Serial web setting page. This is useful when you have several commands with the same packet format to send. In this example, the server id, function code, start address and quantity are all set by the parameters.

data: input parameter for when the script is used in the [Output] or [Input/Output] direction. The mapped data from EtherCAT will be put in this input parameter. In this example, we send a read request, so no EtherCAT data is sent to the Modbus device. In later examples, we will see how to use the data.

output: output parameter for when the script is used in the [Input] or [Input/Output] direction. This is used to carry out the mapped data to EtherCAT. This example places the response data from the Modbus serial device into the output parameter, which is then sent to the EtherCAT side.

```
56.    # function 03 request len: 6 + 2  (fix)
57.    request_len = 6
58.    request = bytearray(request_len)
59.    request[0]   = 0x01                          # slave id
60.    request[1]   = 0x03                          # function code
61.    request[2:4] = bytearray([0x00, 0x00])       # start address
62.    request[4:6] = bytearray([0x00, 0x0A])       # quantity
63.    request_crc  = add_crc(request)              # crc (2 bytes)
```

In this paragraph, we’re preparing the request package of Modbus serial. The package format is defined in the Modbus protocol specification. We’re preparing to send a [read multiple registers (function 03)] request, so according to the specification, it will be a 6-byte (CRC not included) package. Line #57~#58 declares a byte array of 6 bytes. The server id is set to 0x01(Line #59), matching the connected Modbus device. The function code is set to 0x03 (read multiple registers, Line #60). The next two bytes are starting addresses we set to 0x0000 to read the registers at address 0 (Line #61), followed by two bytes of quantity, which are set to 0x0A to read 10 registers (Line #62).

Line #63 is calling a predefined CRC function to make the full package, including CRC value.

```
65.    mx_sio.write(port, request_crc, len(request_crc))
```

Now we can send the full package to a serial device by calling mx_sio.write. [port] is from input parameter by main(). [request_crc] is the generated full package data we create at Line #63, and finally the length of the data being sent. Check how many bytes were successfully sent by using the return value. To keep the example simple, we did not check here.

```
67.     # function 03 response len: 1(server id) + 1(function code) + 1(byte count) + 2*N + 2(crc)
68.     quan = 10
69.
70.     response = bytearray(3+2*quan+2)
71.     ret = mx_sio.read(port, response, len(response), 500)
```

Once the request is sent, the next step is to try to receive the response. The Modbus specification mandates a response format comprising a 1-byte server ID , a 1-byte function code, 1-byte count, data (2*N bytes with N being 10 in our example), and 2 bytes of CRC. A bytearray of sufficient size needs to be prepared to store the serial data read from the serial port (Line #70).

Now, call mx_sio.read() to receive the serial bytes. [port] is from the input parameter by main(), [response] is the buffer we created at Line #70, followed by the length to read, and finally the timeout in milliseconds. Here, we specify 500 ms so that the mx_sio.read either receives enough bytes of data and returns before 500 ms elapses, or waits 500 ms and returns the actual number being read. The return value may be used to check how many bytes were successfully read. To keep the example simple, we did not check here.

```
73.     output_data = response[3:3+quan*2]
74.     output.append(output_data)
```

The whole response package includes server id, function code, count, register data and CRC. Again, CRC or function code should be used to check if the response is normal and successful. For this example, we assume everything is working correctly in this example and immediately apply the data. So, the register data part is put in the [output_data]. To carry the data out to the MGate I/O memory and further to EtherCAT protocol side, the output parameter of main() should be used. Call output.append() to transfer the data to the function output. The MGate will handle the remaining data exchanges.

```
76.     if ret >= 0:
77.         return 0
78.     else:
79.         return -1
```

In short, the return value of main() signals whether the script executed successfully or encountered problems. If everything is done successfully, a return value of 0 should be returned. If the web console’s Proprietary Serial event logging is enabled and a non-zero value is returned, an event will be recorded. Because the return value is logged in the event, you can use different return values to identify specific error scenarios. This can help you figure out what went wrong.

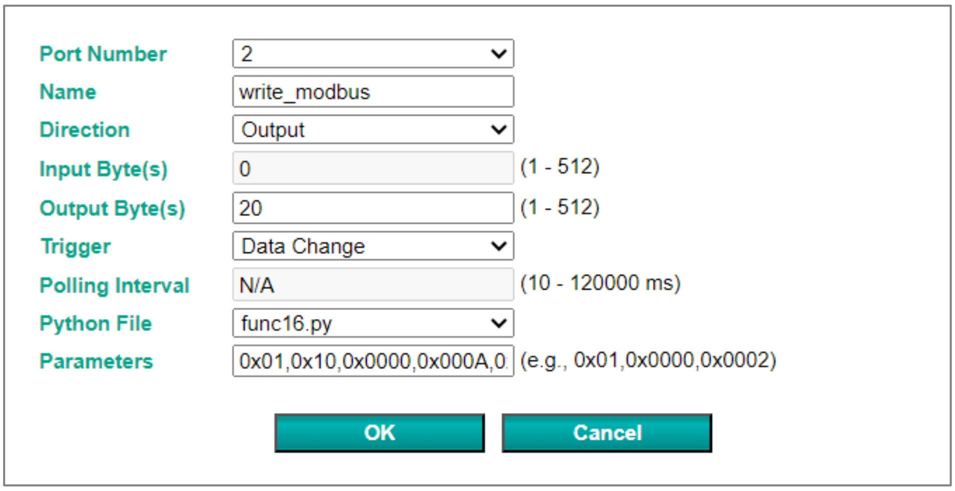
2.5 Example: A simple example of writing Modbus device registers with parameters

This chapter provides an example of writing Modbus device registers. Please reference the example file **func16.py**

The logic in this example is like the previous example. We'll introduce the major difference in this file, which includes the use of parameters and data from other protocols.

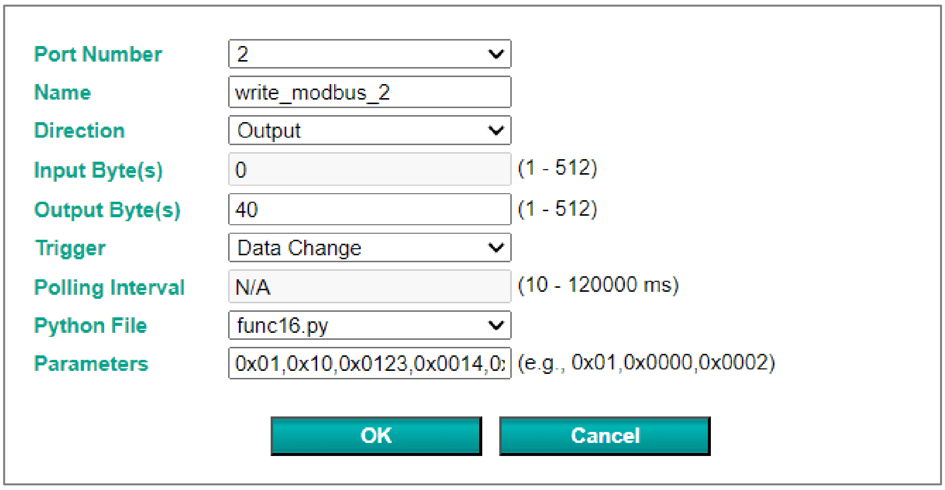
```
request[0:1] = parameter[0]    # server id
request[1:2] = parameter[1]    # function code
request[2:4] = parameter[2]    # start address
request[4:6] = parameter[3]    # quantity
request[6:7] = parameter[4]    # byte count
```

In this example, parameters set the request fields. During execution, the configuration setting will set the parameter on the Proprietary Serial protocol setting page, for example:



Port Number	2	
Name	write_modbus	
Direction	Output	
Input Byte(s)	0	(1 - 512)
Output Byte(s)	20	(1 - 512)
Trigger	Data Change	
Polling Interval	N/A	(10 - 120000 ms)
Python File	func16.py	
Parameters	0x01,0x10,0x0000,0x000A,0	(e.g., 0x01,0x0000,0x0002)

If we enter the parameters on the Proprietary Serial protocol setting page, then the execution will set the server id as 0x01, function code as 0x16, …., etc. This lets you use one MicroPython script file with different settings and parameters, avoiding the need for multiple files. Here, we add another data setting using the same MicroPython file.



Port Number	2	
Name	write_modbus_2	
Direction	Output	
Input Byte(s)	0	(1 - 512)
Output Byte(s)	40	(1 - 512)
Trigger	Data Change	
Polling Interval	N/A	(10 - 120000 ms)
Python File	func16.py	
Parameters	0x01,0x10,0x0123,0x0014,0	(e.g., 0x01,0x0000,0x0002)

Now, we have 2 Modbus commands running: writing to 0x0000 with 10 registers, and writing to 0x0123 with 20 registers. (The parameter[4] cut off in the screenshot is 0x0014 and 0x0028)

```
request[7:] = data            # data
```

You can easily include the data to be sent in the request using the input parameter “data”. The output bytes for expected values are set on the Proprietary Serial protocol settings page, while data mapping is configured on the I/O data mapping page. If all the settings are done, get the data from the EtherCAT side and then put it into the request to write to the serial device.

```
retry_max = 5
retry_count = 0

while retry_count < retry_max:

    mx_sio.write(port, request_crc, len(request_crc))

    # function 16 response len: 8 bytes (fix)
    response = bytearray(8)
    ret = mx_sio.read(port, response, len(response), 500)

    # if response success then break
    if ret == 8 and response[1] == request[1]:
        break;

    #else flush sio queue and retry (flush_func: 0=rx, 1=tx, 2=all)
    mx_sio.flush(port, 2)
    retry_count += 1
```

In the last example, we kept things simple by performing just one sio.write and then one sio.read. This example includes a retry mechanism. When the response is read, you can verify successful request handling by assessing the response’s length and the command it carries. In case of insufficient byte reading or an exception, we can call mx_sio.flush.If there’s not enough bytes read or an exception happens, we can call mx_sio.flush to clear the buffer of read/write and retry the same request. Exceeding the retry_count limit will trigger an error return.

2.6 Example: A simple example of reading/writing Modbus device registers with parameters

If you want to write data to a serial device and read data from a device with a single MicroPython file, here’s an example of a read/write request to a Modbus device. Please reference the example file **func23.py**

```
request_len = 12 + len(data)
request = bytearray(request_len)
request[0:1]   = parameter[0]     # server id
request[1:2]   = parameter[1]     # function code
request[2:4]   = parameter[2]     # read start address
request[4:6]   = parameter[3]     # read quantity
request[6:8]   = parameter[4]     # write start address
request[8:10]  = parameter[5]     # write quantity (n)
request[10:11] = parameter[6]     # write byte count (n*2)
request[12:] = data               # data
request_crc = add_crc(request)    # crc (2 bytes)
```

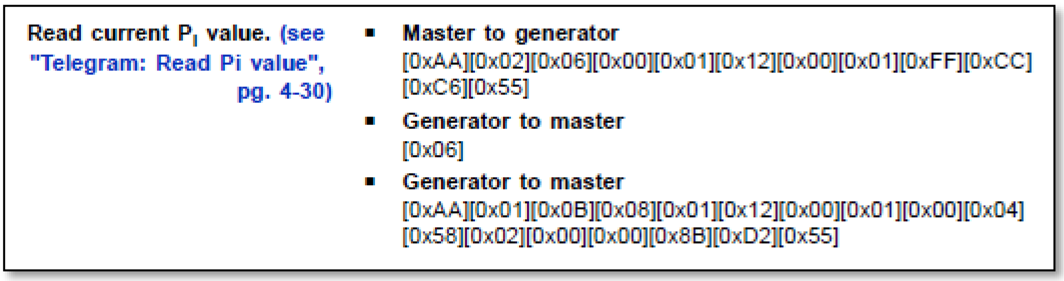
Like the last example, we use parameters to complete the request body. Since this is a read/write request, the request body has more fields to fill.

This example will look like the two examples combined. The data is from input and the response data will be appended to the output.

2.7 Example: Another example of reading a proprietary serial device

Here we have an example of reading data through a proprietary serial protocol from a serial device. The format may vary from Modbus, but the core concept and execution steps are similar. Refer to the example file **read_pi_value.py**

The request and response data for communication with the serial device is as follows:



Read current P_I value. (see "Telegram: Read Pi value", pg. 4-30)	▪ **Master to generator** [0xAA][0x02][0x06][0x00][0x01][0x12][0x00][0x01][0xFF][0xCC][0xC6][0x55] ▪ **Generator to master** [0x06] ▪ **Generator to master** [0xAA][0x01][0x0B][0x08][0x01][0x12][0x00][0x01][0x00][0x04][0x58][0x02][0x00][0x00][0x8B][0xD2][0x55]

Besides the format differing from Modbus, the behavior also has a slight variation. Once the serial device receives a request, it sends an acknowledgement (ACK) and then transmits the response packet. This example will demonstrate how to handle communication.

```
request = bytearray(12)
request[0:1]   = b'\xAA'                 # Start byte
request[1:2]   = b'\x02'                 # Address byte, always 0x02 with RS-232
request[2:3]   = b'\x06'                 # Number of bytes in the following data unit.
request[3:4]   = b'\x00'                 # [GS] Generator status, always 0x00 if it is sent by the master.
request[4:5]   = b'\x01'                 # [CMD] Read parameter
request[5:7]   = parameter[0]            # [IDX] Index | Parameter: set value Pi
request[7:8]   = parameter[1]            # [SUBIDX] Subindex
request[8:9]   = b'\xFF'                 # [STAT] Status value
crc = crc16(request, 14)
request[9:11] = crc.to_bytes(2, 'little')  # Checksum
request[11:12] = b'\x55'                 # Stop byte
```

The first step is to fill the request using the serial device specification format. We use parameters for [IDX] and [SUBIDX] so that this MicroPython file can be used to add multiple commands to read different addresses.

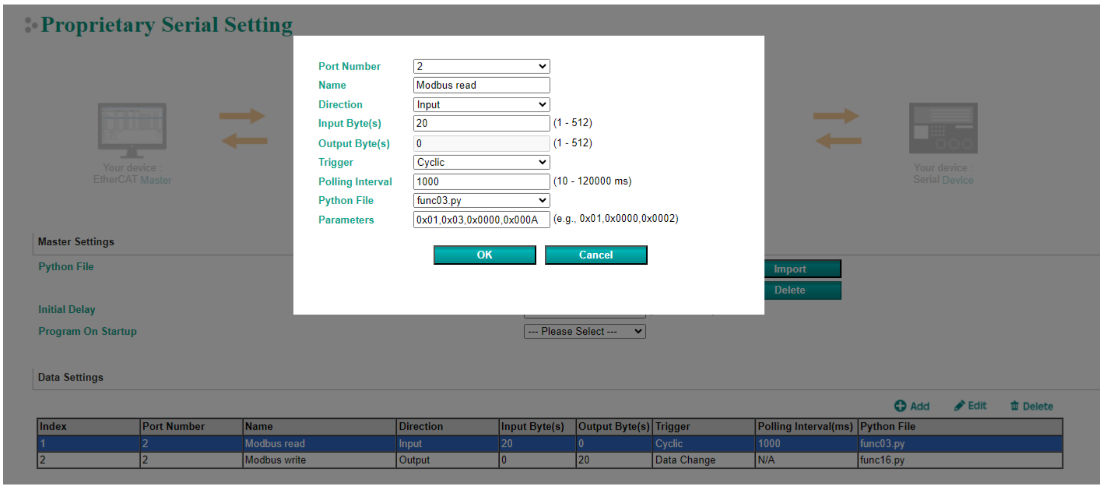
```
while retry_count < retry_max:

    # Master to generator (request)
    mx_sio.write(port, request, len(request))

    # Generator to master (ACK)
    ack = bytearray(1)
    ret = mx_sio.read(port, ack, len(ack), 1000)

    # Generator to master (resopnse)
    response = bytearray(17)
    ret = mx_sio.read(port, response, len(response), 1000)

    # Did not check response data for now, receive enough bytes will be a as successs
    if ret == 17:
        break;

    # not success, flush sio queue and retry (flush_func: 0=rx, 1=tx, 2=all)
    mx_sio.flush(port, 2)
    retry_count += 1

# append the response data to output_data, which will later be write to I/O share memory
output_data = response[10:14]
output.append(output_data)
```

Communication begins with a while loop. The loop operates by first transmitting the request to the device, then receiving a single byte to acknowledge the request, and finally reading the response data and validating its length. Otherwise, end the loop and add the data to the output.

This example illustrates how multiple read or write operations can be combined in one command to meet the needs of the protocol. Ensure the data is added to the output before you go back.

3 Verification

When you finish the MicroPython file, you may import and set it on the MGate 5216's Proprietary Serial Setting web page.



Proprietary Serial Setting

Port Number: 2
Name: Modbus read
Direction: Input
Input Byte(s): 20 (1 - 512)
Output Byte(s): 0 (1 - 512)
Trigger: Cyclic
Polling Interval: 1000 (10 - 120000 ms)
Python File: func03.py
Parameters: 0x01,0x03,0x0000,0x000A (e.g., 0x01,0x0000,0x0002)

Master Settings

Python File

Initial Delay

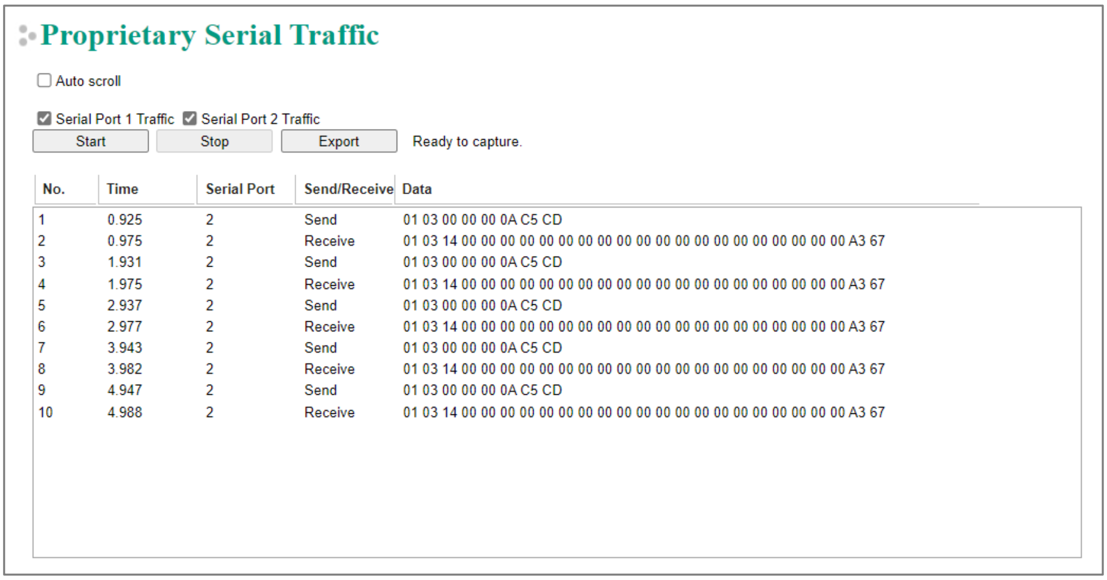
Program On Startup: --- Please Select ---

Data Settings

Index	Port Number	Name	Direction	Input Byte(s)	Output Byte(s)	Trigger	Polling Interval(ms)	Python File
1	2	Modbus read	Input	20	0	Cyclic	1000	func03.py
2	2	Modbus write	Output	0	20	Data Change	N/A	func16.py

The complete guide and details can be found in MGate 5216's user manual. Once you finish setting the commands, browse a few pages to see if your MicroPython script works as intended.

The following three pages are under **System Monitoring-> Protocol Status** of the MGate 5216's web console. First, go to the **Proprietary Serial Traffic** page.



Proprietary Serial Traffic

□ Auto scroll

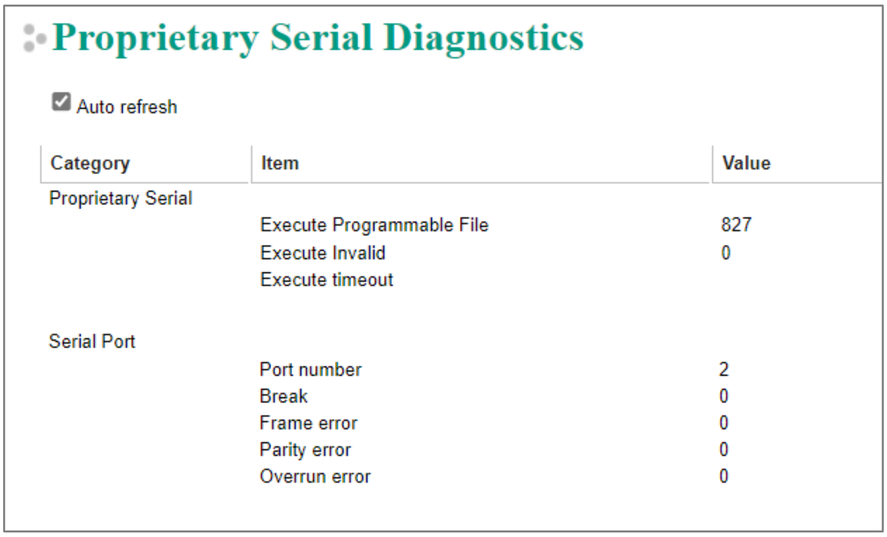
☑ Serial Port 1 Traffic ☑ Serial Port 2 Traffic

Ready to capture.

No.	Time	Serial Port	Send/Receive	Data
1	0.925	2	Send	01 03 00 00 00 0A C5 CD
2	0.975	2	Receive	01 03 14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A3 67
3	1.931	2	Send	01 03 00 00 00 0A C5 CD
4	1.975	2	Receive	01 03 14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A3 67
5	2.937	2	Send	01 03 00 00 00 0A C5 CD
6	2.977	2	Receive	01 03 14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A3 67
7	3.943	2	Send	01 03 00 00 00 0A C5 CD
8	3.982	2	Receive	01 03 14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A3 67
9	4.947	2	Send	01 03 00 00 00 0A C5 CD
10	4.988	2	Receive	01 03 14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A3 67

This method is the most reliable way to verify if your program's communication. The raw data will show in Hexadecimal (HEX) so you can check if the behavior is as expected and if the data bytes are correct. If a byte is incorrect, either review the MicroPython file or adjust the parameter settings.

In the absence of traffic data, check the **Proprietary Serial Diagnostics** to verify program execution. Invalid execution or timeouts suggest a potential logic error within the MicroPython file.

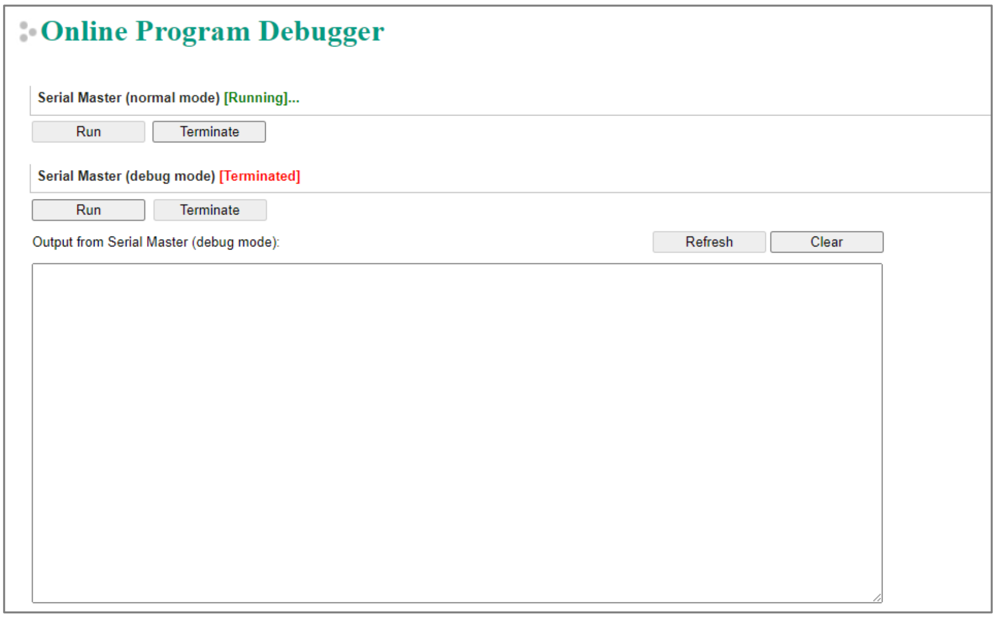


Proprietary Serial Diagnostics

☑ Auto refresh

Category	Item	Value
Proprietary Serial		
	Execute Programmable File	827
	Execute Invalid	0
	Execute timeout	
Serial Port		
	Port number	2
	Break	0
	Frame error	0
	Parity error	0
	Overrun error	0

Finally, for convenience, we provide the **Online Program Debugger** page to quickly edit file content and test.



Online Program Debugger

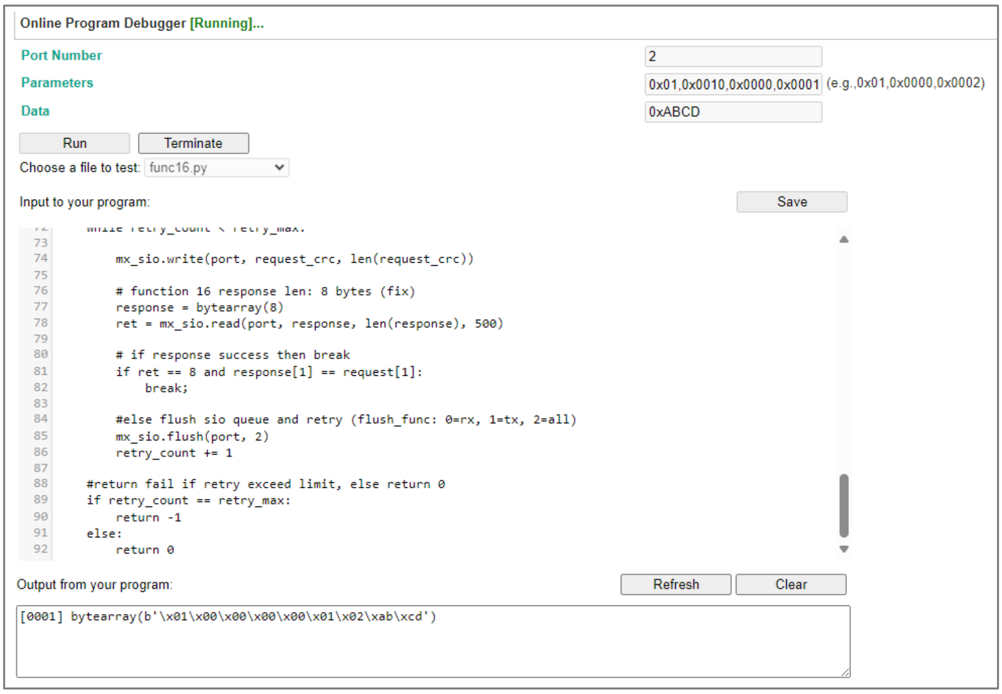
Serial Master (normal mode) [Running]...

Serial Master (debug mode) [Terminated]

Output from Serial Master (debug mode):

The MGate normally operates in normal mode, but you can stop normal mode and switch to debug mode. In debug mode, the output of the program will be shown in the text box. You can add prints in MicroPython file to check the status of the running program. In addition, any errors that the executor can identify will be displayed in the text box.

At the bottom of this page, you will find the Online Program Debugger.



Online Program Debugger [Running]...

Port Number: 2

Parameters: 0x01,0x0010,0x0000,0x0001 (e.g.,0x01,0x0000,0x0002)

Data: 0xABCD

Choose a file to test: func16.py

Input to your program:

```
    while retry_count < retry_max:

        mx_sio.write(port, request_crc, len(request_crc))

        # function 16 response len: 8 bytes (fix)
        response = bytearray(8)
        ret = mx_sio.read(port, response, len(response), 500)

        # if response success then break
        if ret == 8 and response[1] == request[1]:
            break;

        #else flush sio queue and retry (flush_func: 0=rx, 1=tx, 2=all)
        mx_sio.flush(port, 2)
        retry_count += 1

    #return fail if retry exceed limit, else return 0
    if retry_count == retry_max:
        return -1
    else:
        return 0
```

Output from your program:

```
[0001] bytearray(b'\x01\x00\x00\x00\x00\x01\x02\xab\xcd')
```

Edit, run, and view the output of the program here. When running the program debugger, you must provide the port, parameters, and data (if used in the program), unlike in debug mode. Once you've edited the MicroPython file, you can fill in the port, parameter, and data to your specific testing requirements. You'll see the output in the text box. Remember to save if you want to keep the changes you've made to the file. The saved file will overwrite the file you select.