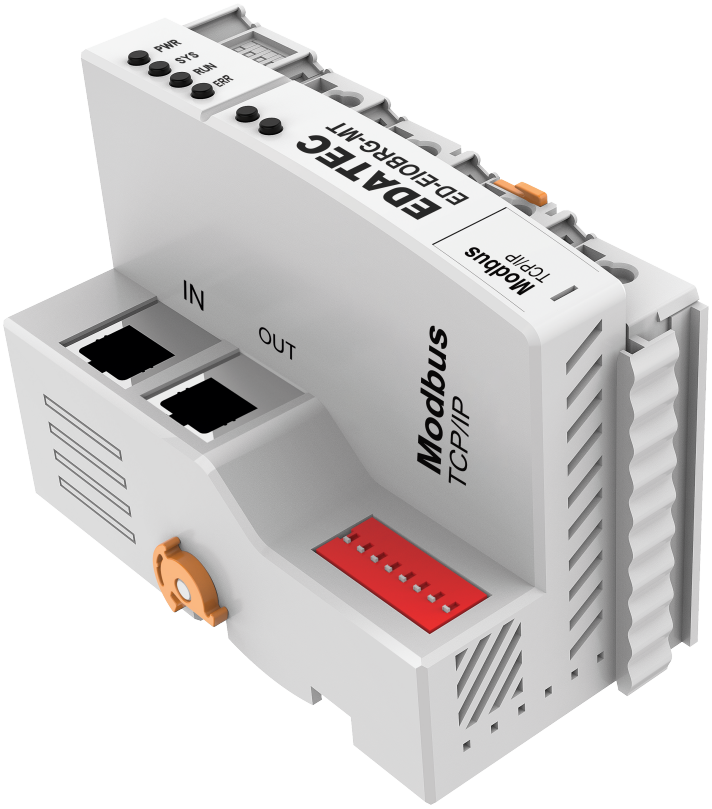


Modbus TCP Coupler

User Manual

by EDA Technology Co., Ltd

built: 2026-09-11

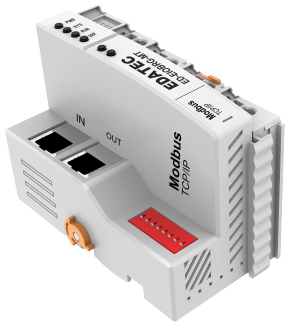
1 Quick Start Guide

This chapter provides an overview of the Modbus TCP coupler, including networking, wiring, network configuration, other parameter settings, and I/O module settings.

1.1 Overview

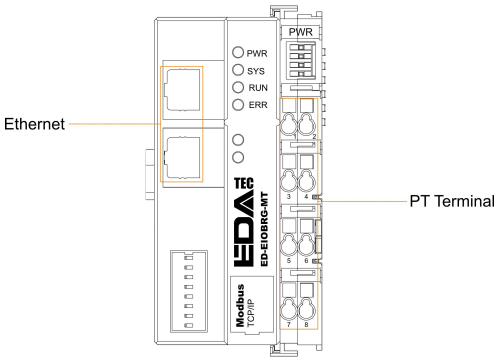
The ED-EIOBRG-MT is a Modbus TCP coupler primarily used to seamlessly integrate slave devices employing different fieldbus protocols (such as Profibus DP, CANopen, DeviceNet, etc.) into a standard Modbus TCP Ethernet network. It acts as a protocol conversion gateway, enabling a host computer (PLC, SCADA, HMI) to access data from heterogeneous bus devices on the lower layer through a unified Modbus TCP interface.

The maximum transmission distance of the ED-EIOBRG-MT is 100m, and it supports expanding up to 32 I/O modules. As a connection node, it can connect multiple Modbus TCP slave devices to a Modbus TCP network, meeting the requirements of various application scenarios.



1.1.1 Interface

The ED-EIOBRG-MT features 2 100M Ethernet ports and 8 PT terminals, as shown in the figure below.



- The 2 100M Ethernet ports use standard RJ45 connectors, designated as IN and OUT ports, for communication with the upper-level control system and system networking.

Ethernet Ports	
IN	Input port, used to connect to a Modbus TCP master or the OUT port of another Modbus TCP slave on the same subnet.

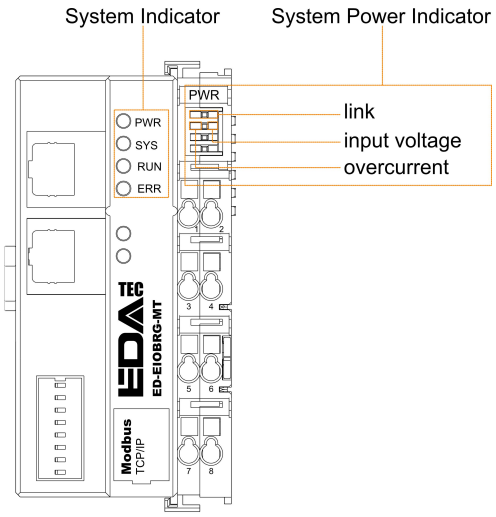
Ethernet Ports	
OUT	Output port, used to connect to the IN port of another Modbus TCP slave on the same subnet.

- The 8 PT terminals are push-in spring-type terminals used for connecting the system-side and field-side power supplies.

PT Terminals			
Pin No.	Description	Pin No.	Description
1	24V SYS	2	0V SYS
3	24V Field	4	24V Field
5	0V Field	6	0V Field
7	PE	8	PE

1.1.2 Indicator

The ED-EIOBRG-MT includes 4 system indicators and 3 power supply indicators, as shown in the figure below.



System Indicator	
PWR	Coupler power indicator, green, indicates the connection status of the system power supply. • Steady ON: System power connected normally; • OFF: System not powered or power failure.
SYS	System status indicator, green, indicates the operating status of the coupler. • Slow flash (1s interval): Normal operation; • Fast flash (3-5 times/s): Abnormal operation.
RUN	Run indicator, green, indicates the running status of the coupler. • Steady ON: System running normally; • OFF: System not running.

System Indicator	
ERR	Fault indicator, red, indicates whether an I/O module has gone offline. • OFF: All I/O modules are communicating normally, no disconnection; • Steady ON: At least one I/O module is offline.

System Power Indicators	
Link Indicator	Green, indicates whether the common terminal wiring is correct. • Steady ON: System power wiring normal (24V SYS connected to DC+, 0V SYS connected to DC-); • OFF: System power wiring abnormal.
Input Voltage Indicator	Green, indicates whether the input voltage is normal. • Steady ON: Input voltage normal; • OFF: Input voltage abnormal.
Overcurrent Indicator	Red, indicates whether the coupler exceeds its maximum operating current. • OFF: Operating current does not exceed the maximum; • Steady ON: Operating current has exceeded the maximum.

1.2 Networking and Connection

The ED-EIOBRG-MT supports connecting I/O modules and must be used with a Modbus TCP master. It also supports cascading with other Modbus TCP slaves. The specific system networking diagram is shown below.

- The ED-EIOBRG-MT coupler connects to the Modbus TCP master via the Ethernet.

- Multiple ED-EIOBRG-MT couplers support cascading.
- The IP addresses of the Modbus TCP master, windows PC, and ED-EIOBRG-MT must be on the same subnet.
- A Windows PC (with configuration software installed) can be used to configure the coupler parameters.
- The I/O modules (digital input/output, analog input/output, etc.) mounted on each ED-EIOBRG-MT communicate with the coupler via the internal backplane bus. The coupler automatically scans and identifies the model and order of each I/O module. A single ED-EIOBRG-MT supports mounting up to 32 I/O modules.
- Each ED-EIOBRG-MT requires a DC 24V power supply. The wiring steps are as follows:
  1. Strip approximately 8-10 mm of insulation from the wire end;
  2. Use a flat-blade screwdriver (recommended size 2×75mm) to vertically insert into the square hole above the terminal and press down to open the spring;
  3. Insert the stripped wire into the round wiring hole, then remove the screwdriver; the spring will automatically reset and clamp the wire.

> TIP
>
> When connecting the power cable, strictly follow the wiring instructions shown in the diagram above. Be sure to distinguish between the positive and negative terminals of the power supply and avoid reverse connection; otherwise, the module may malfunction, operate abnormally, or even become damaged.

1.3 Configurting Parameters

1.3.1 Configuring ED-EIOBRG-MT Parameters

The ED-EIOBRG-MT supports parameter configuration via the host computer software. The following sections detail how to use the IO Controller software to complete the configuration.

1.3.1.1 Configuring Network

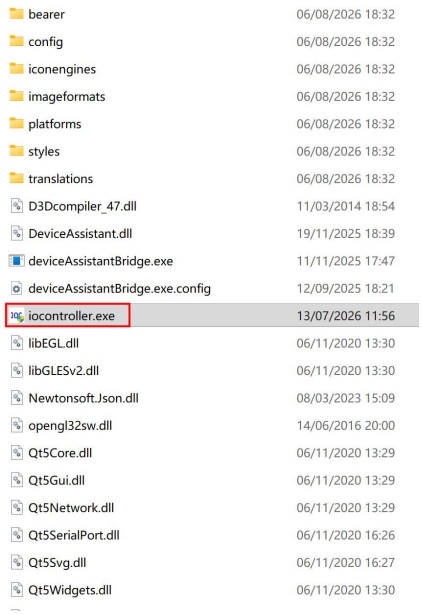
The default IP address of the ED-EIOBRG-MT at the factory is `192.168.0.50`. In actual use, its IP address must be configured to be on the same subnet as the Modbus TCP master.

Preparation:

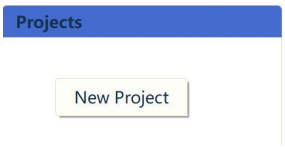
- The IO Controller software has been downloaded onto a Windows PC. Download link: IO Controller V1.2.3.zip (https://1826505135.cdn.123clouddisk.com/1826505135/41441095) .
- The physical connections for the ED-EIOBRG-MT have been completed, ensuring that the IP addresses of the Modbus TCP master, Windows PC, and ED-EIOBRG-MT are on the same subnet.

Steps:

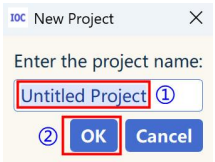
1. Unzip the downloaded `IO Controller V1.2.3.zip`.
2. In the unzipped `IO Controller V1.2.3` folder, double-click `iocontroller.exe` to run the IO Controller software.



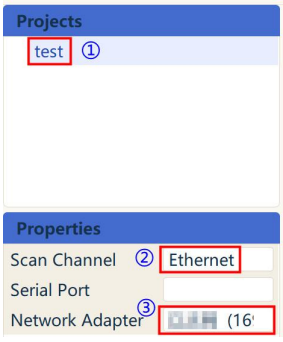
3. In the opened interface, right-click on the blank area under the "Projects" region and select "New Project".



4. Customize the project name, then click "OK".

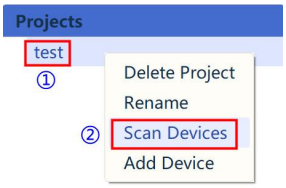


5. Left-click on the newly created project "test". Configure the project properties in the "Properties" area.

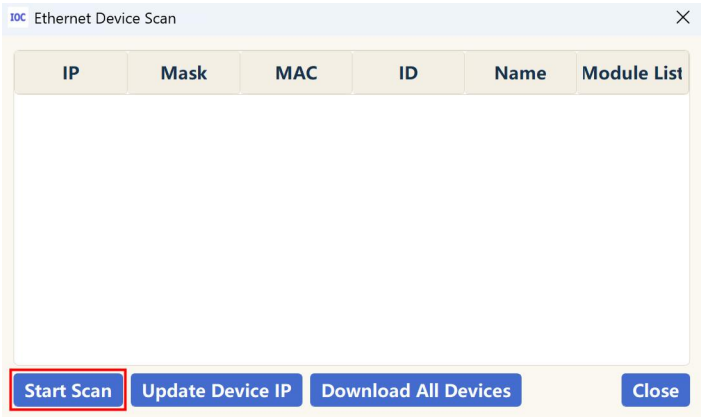


- Scan Channel: Select "Ethernet"
- Serial Port: Not required
- Network Adapter: Select the IP address of the Windows PC

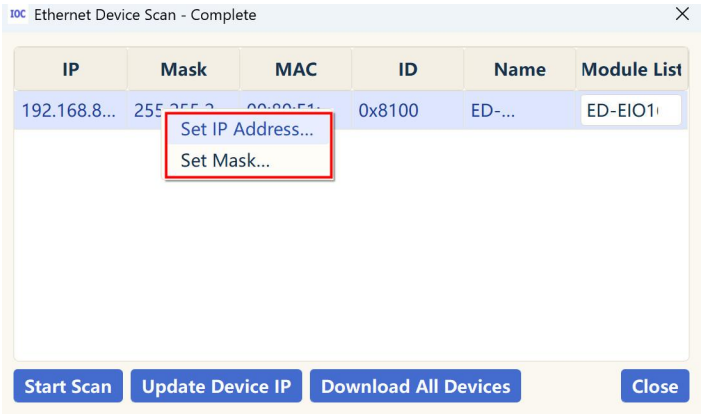
6. Right-click on the newly created project "test" and select "Scan Devices".



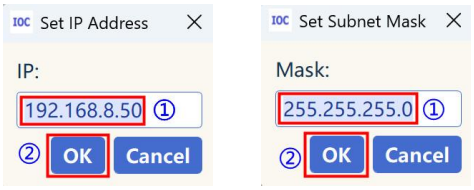
7. In the "Ethernet Device Scan" window, click "Start Scan" at the bottom to obtain the IP address information of the ED-EIOBRG-MT.



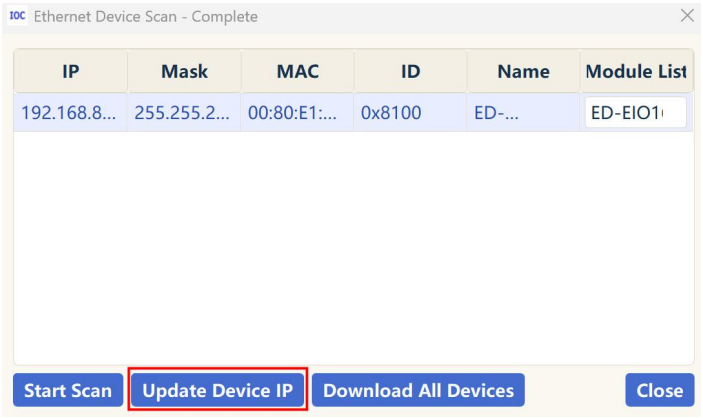
8. Right-click on the scanned IP address entry and select "Set IP Address..." or "Set Mask..." from the context menu.



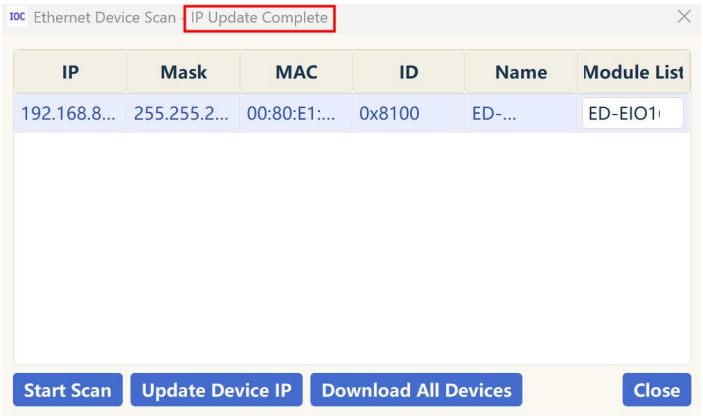
9. Set the IP address and subnet mask as needed.



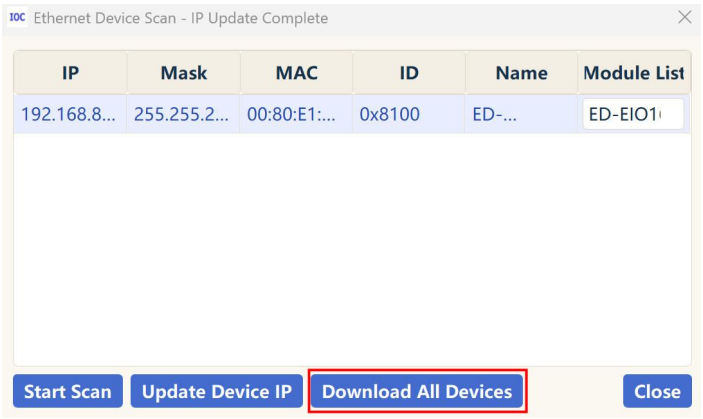
10. After setting the IP address, click "Update Device IP".



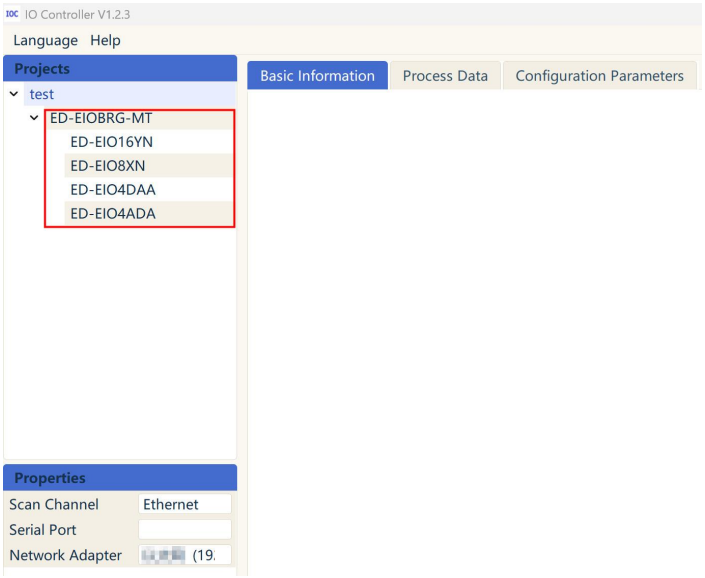
11. Once the IP address update is complete, the title bar at the top of the window will display "IP Update Complete".



12. Click "Download All Devices" at the bottom of the window to download the data for the ED-EIOBRG-MT and its expanded I/O modules.



13. After the download is complete, the information for the ED-EIOBRG-MT and its expanded I/O modules will be displayed in the interface.



1.3.1.2 Setting Other Parameters

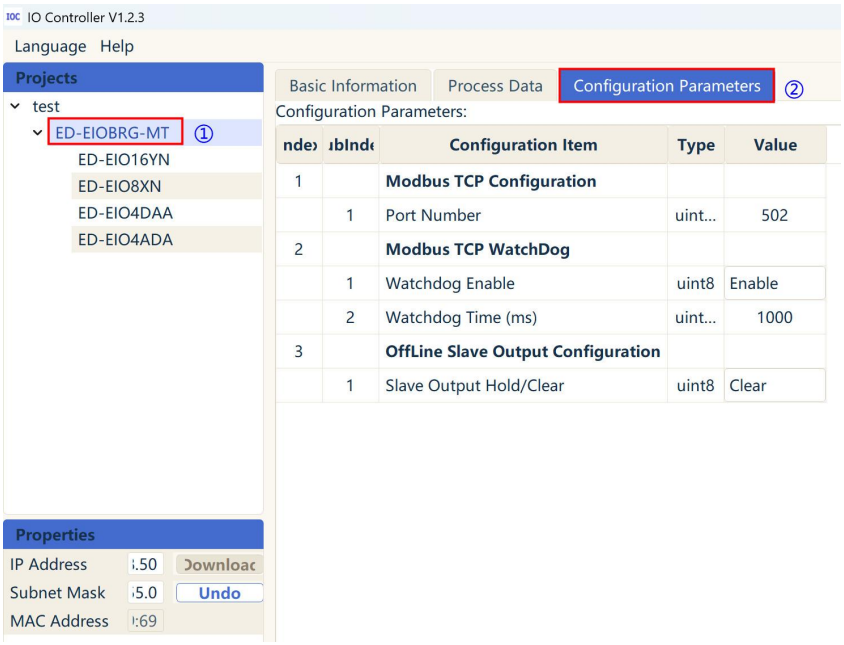
The port number, Watchdog, and Slave Output Hold/Clear mode of the ED-EIOBRG-MT can all be set via the IO Controller software. The specific operations are detailed below.

Preparation:

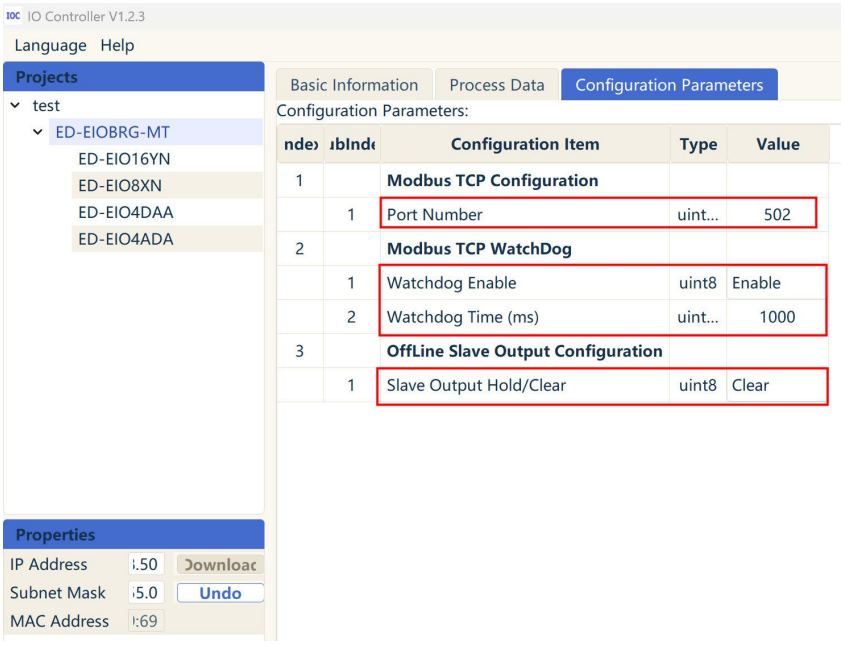
- The IP address of the ED-EIOBRG-MT has been set.
- The device download for the ED-EIOBRG-MT and I/O modules has been completed.

Steps:

1. In the left panel of the interface, under the "test" project, left-click on "ED-EIOBRG-MT". Then, select the "Configuration Parameters" tab on the right side.



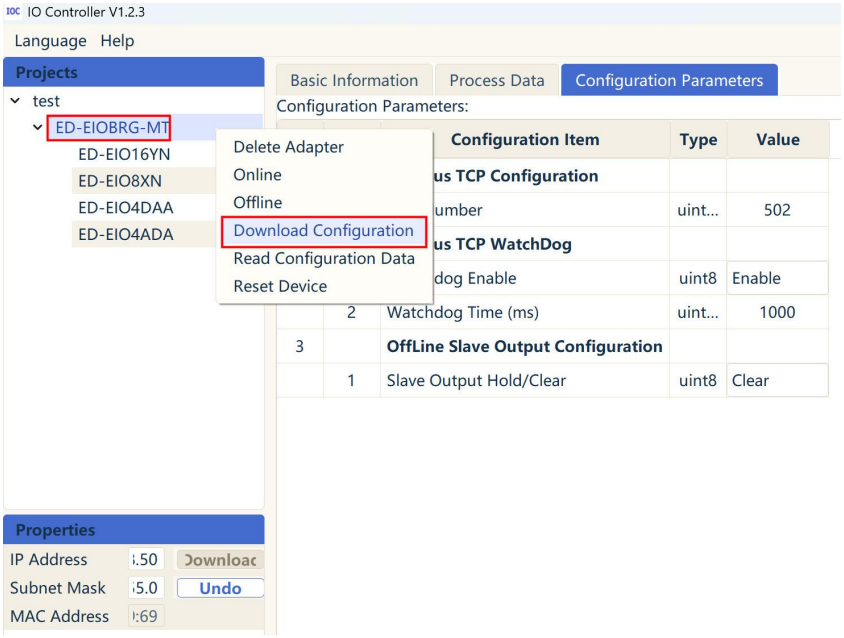
1. Double-click the value in the "Value" column of the "Configuration Parameters" table to modify parameters such as "Port Number", "Watchdog Enable", "Watchdog Time (ms)", and Slave Output Hold/Clear as needed.



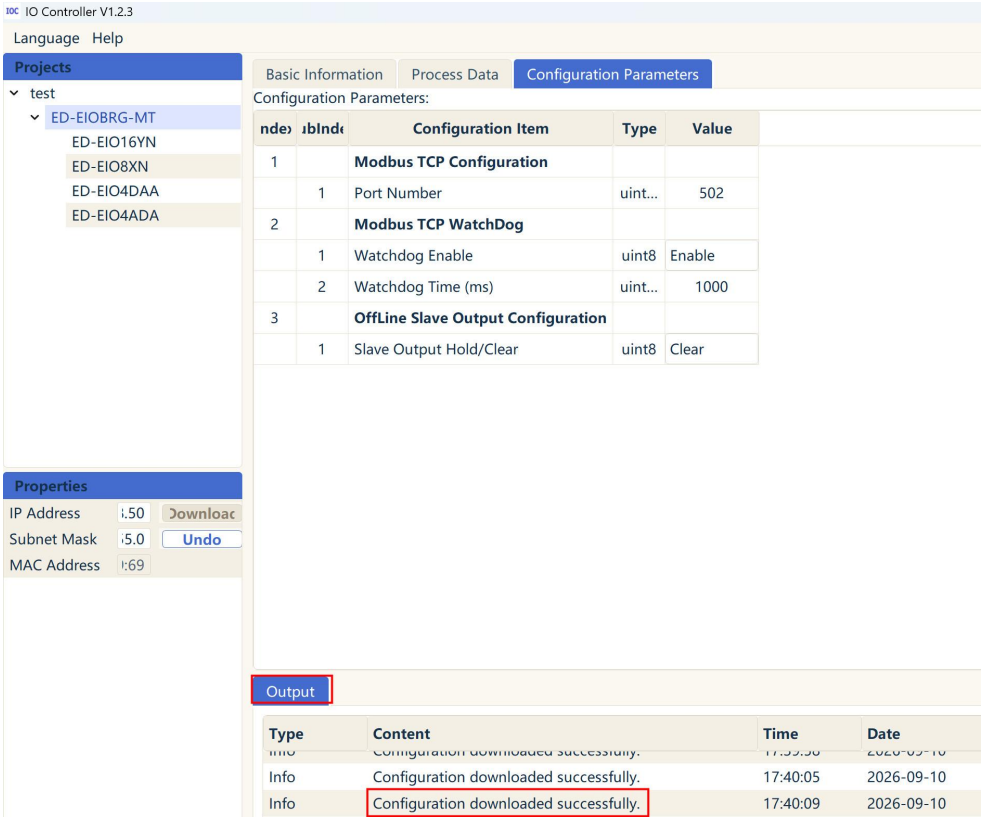
- The description of the configurable parameters is as follows:

Parameter	Description
Port Number	Default value is 502. Range is 1~65535. Set as needed.
Watchdog Enable	Default value is 1. Values include 0 and 1. • 1: Enabled • 0: Disabled
Watchdog Time (ms)	Default value is 180000. Range is 1000~4294967295. Set as needed.
Slave Output Hold/ Clear	Default value is 0. Values include 0 and 1. • 0: Clear outputs when disconnected • 1: Hold last output state when disconnected

3. After setting the parameters, right-click on "ED-EIOBRG-MT" under the "test" project in the left panel, and select "Download Configuration" from the menu.



4. Once the download is complete, the "Output" area at the bottom right of the interface will display the message "Configuration downloaded successfully".



1.3.2 Configuring I/O Module Parameters

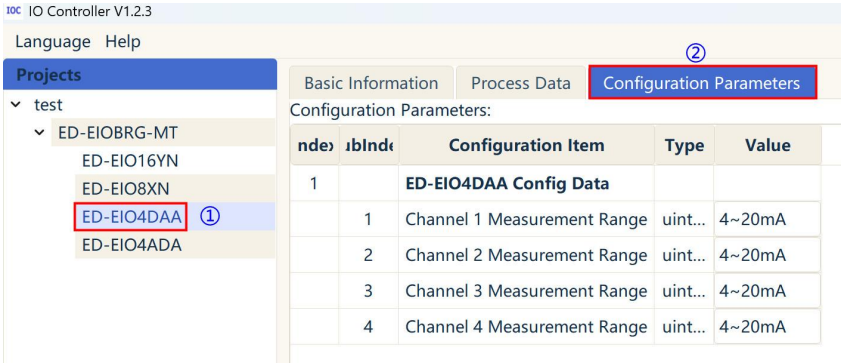
The I/O modules mounted on the ED-EIOBRG-MT are classified into Digital Input (DI), Digital Output (DO), Analog Input (AI), and Analog Output (AO). DI and DO modules do not require configuration. AI and AO modules support setting the voltage and current ranges via the IO Controller software. The specific operations are detailed below.

Preparation:

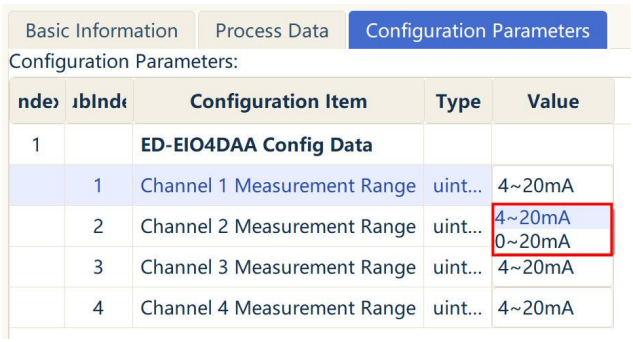
- The IP address of the ED-EIOBRG-MT has been set.
- The device download for the ED-EIOBRG-MT and I/O modules has been completed.

Steps:

1. In the left panel of the interface, under the "test" project, left-click on the I/O module (e.g., ED-EIO4DAA). Then, select the "Configuration Parameters" tab on the right side.



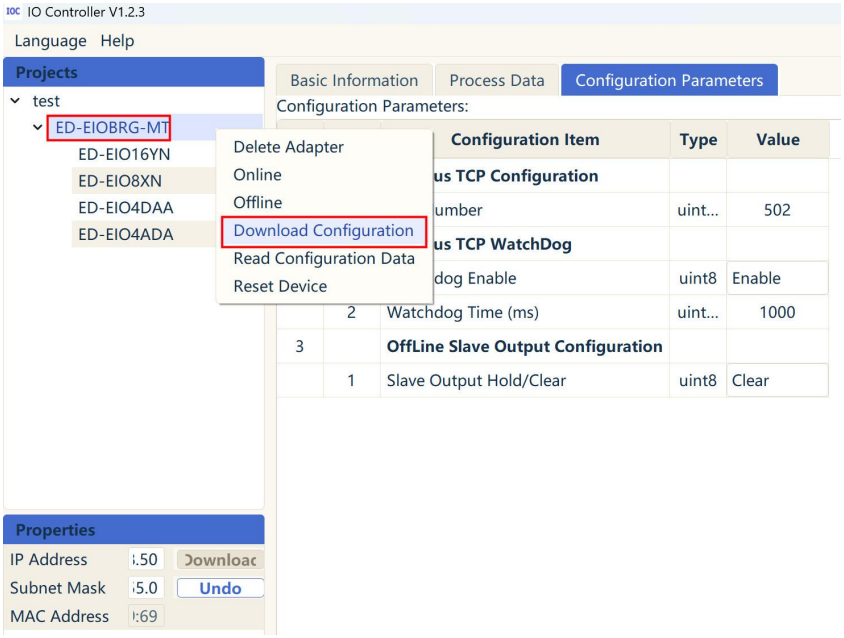
2. Left-click on the voltage range value in the "Value" column of the "Configuration Parameters" table to view the list of selectable ranges. Choose the appropriate voltage range based on the actual application scenario.



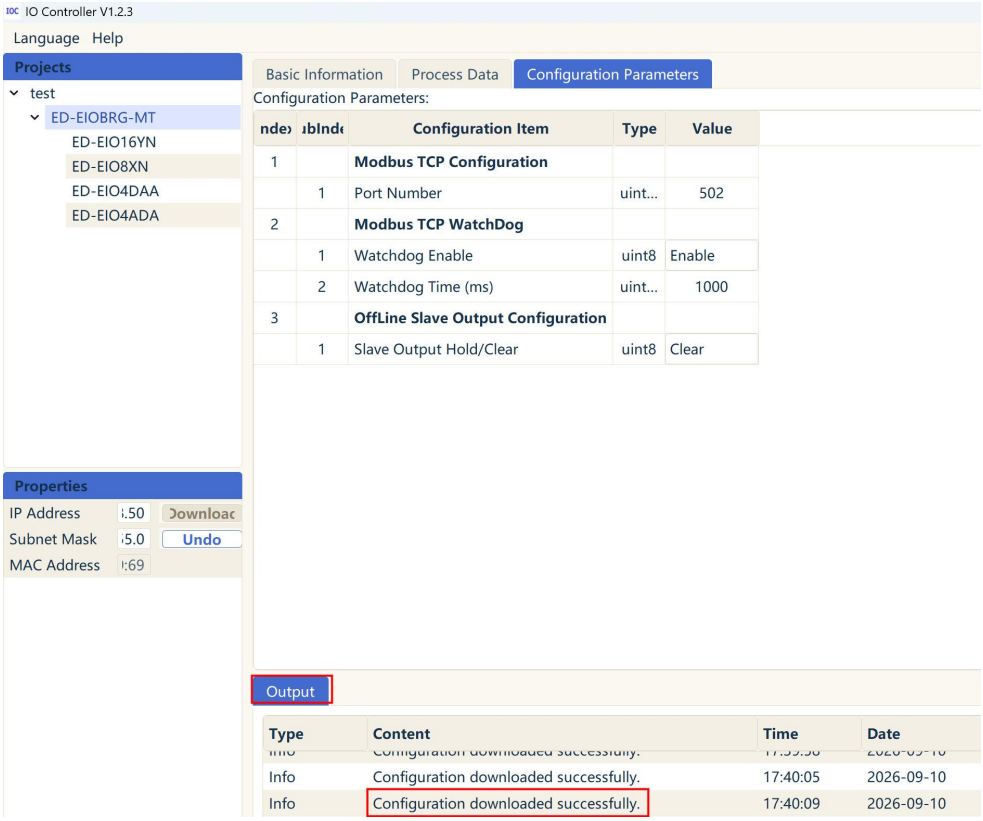
TIP

- Each channel of the I/O module can be configured independently.
- This example uses one current output module for demonstration only. Operations for other module types are similar.

3. After setting the parameters, right-click on "ED-EIOBRG-MT" under the "test" project in the left panel, and select "Download Configuration" from the menu.



4. Once the download is complete, the "Output" area at the bottom right of the interface will display the message "Configuration downloaded successfully".



2 I/O Module Register Read/Write Examples

In an actual Modbus TCP network, it is necessary to read and write the internal registers of the I/O modules on the ED-EIOBRG-MT via software to control their I/O channels. The following sections detail the I/O address mapping rules and register read/write examples for this module.

2.1 I/O Module Address Mapping Rules

DI and DO modules support both Bit mapping and Word mapping register access methods. AI and AO modules only support Word mapping register access.

- Description of Bit Mapping Method

Module Type	Function Code	Offset Start Address	Bit Address Range	Data Length Range	Offset Addr + Length
DI (Input Bit)	0x02	0x00	0~1023	1~1024	≤1024(R)
DO (Input Bit)	0x05 0x15 0x01(R)	0x00(R/W)	0~1023	1~1024	≤1024(R/W)

- Description of Word Mapping Method

Module Type	Function Code	Offset Start Address	Register Address Range	Data Length Range	Offset Addr + Length
DI (Input Word)	0x03	Hex: 0x5000 Decimal: 20480	0x5000~0x507F 20480~20607	1~128	≤20608(R)
DO (Output Word)	0x16 0x03(R)	Hex: 0x3000(W) Decimal: 12288(W) Hex: 0x4000(R) Decimal: 16384(R)	0x3000~0x307F(W) 12288~12415(W) 0x4000~0x407F(R) 16384~16511(R)	1~128	≤12416(W) ≤16512(R)
AI (Input Word)	0x03 0x04	0x00	0~511	1~512	≤512(R)
AO (Output Word)	0x06 0x16 0x03(R)	Hex: 0x00(W) Decimal: 0(W) Hex: 0x2000(R) Decimal: 8192(R)	0x00~0x1FF(W) 0~511(W) 0x2000~0x21FF(R) 8192~8703(R)	1~512	≤512(W) ≤8704(R)

TIP

- DI and AI modules only support read functionality.
- DO and AO modules support both read and write functionality.

2.2 Example Code for Reading and Writing I/O Module Registers

Example code is provided in both Python and C++. They are introduced separately below.

2.2.1 Python Example Code

Introduces Python example code for DI, DO, AI, and AO modules.

2.2.1.1 DI Module (Bit Mapping Method)

text

```
from pymodbus.client import ModbusTcpClient
import time

# Modbus coupler communication parameter configuration
COUPLER_IP = "192.168.0.50"  # Modbus coupler IP address
COUPLER_PORT = 502           # Default communication port of Modbus TCP
UNIT_ID = 1                  # Modbus slave address (slave id)
DI_ADDRESS = 0               # Starting offset address of DI (decimal)
DI_COUNT = 8                 # Number of DI points to be read

def read_modbus_di_bit(client: ModbusTcpClient, unit_id: int, address: int, count: int):
    """
    Modbus TCP read DI module by bit (Function Code 02)
    :param client: Pre-established ModbusTcpClient instance
    :param unit_id: Slave ID
    :param address: Starting offset address of DI
    :param count: Number of DI bits to read
    :return: DI bits list on success, None on failure
    """
    # Read DI module status
    res = client.read_discrete_inputs(address=address, count=count, slave=unit_id)
    if res.isError():
        print(f"Failed to read DI module: {res}")
        return None
    return res.bits[:count]

if __name__ == "__main__":
    # Establish persistent TCP connection
    client = ModbusTcpClient(COUPLER_IP, port=COUPLER_PORT)
    if not client.connect():
        print("Modbus connection failed")
```

```
        exit()
    print("Persistent connection established, cyclically reading DI, press Ctrl+C to stop")
    try:
        while True:
            # Call function to read DI by bit
            di_data = read_modbus_di_bit(client, UNIT_ID, DI_ADDRESS, DI_COUNT)
            if di_data is not None:
                print("DI Module Status List: ", di_data)
            time.sleep(1)   # Refresh every 1 second
    except KeyboardInterrupt:
        print("\nStop signal received by program")
    finally:
        # Close connection when program exits
        client.close()
        print("Modbus connection closed")
```

2.2.1.2 DI Module (Word Mapping Method)

text

```
from pymodbus.client import ModbusTcpClient
import time
# Modbus coupler communication parameter configuration
COUPLER_IP = "192.168.0.50"  # Modbus coupler IP address
COUPLER_PORT = 502           # Default communication port of Modbus TCP
UNIT_ID = 1                  # Modbus slave address (slave id)
DI_ADDRESS = 20480           # Starting offset address of DI (decimal)
DI_COUNT = 1                 # Number of DI registers to be read

# Read DI
def read_modbus_di(client: ModbusTcpClient, unit_id: int, address: int, count: int):
    """
    Modbus TCP read DI module (Function Code 03)
    :param client: Pre-established ModbusTcpClient instance
    :param unit_id: Slave ID
    :param address: Starting offset address of DI
    :param count: Number of DI registers to read
    :return: DI register list on success, None on failure
    """
    # Read DI module status
    res = client.read_holding_registers(address=address, count=count, slave=unit_id)
    if res.isError():
        print(f"Failed to read DI module: {res}")
        return None
    return res.registers

if __name__ == "__main__":
    # Establish persistent TCP connection
    client = ModbusTcpClient(COUPLER_IP, port=COUPLER_PORT)
    if not client.connect():
```

```
        print("Modbus connection failed!")
        exit()
    try:
        while True:
            # Call function to read DI by word
            ret = read_modbus_di(client, UNIT_ID, DI_ADDRESS, DI_COUNT)
            if ret:
               print("DI Module Values: ",ret)   # Convert output value to binary for correspo
            time.sleep(1)  # Refresh every 1 second
    except KeyboardInterrupt:
        print("Program terminated")
    finally:
        # Close connection when program exits
        client.close()
        print("Modbus connection closed")
```

2.2.1.3 DO Module (Bit Mapping Method)

text

```
from pymodbus.client import ModbusTcpClient
import time

# Modbus coupler communication parameter configuration
COUPLER_IP = "192.168.0.50"  # Modbus coupler IP address
COUPLER_PORT = 502           # Default communication port of Modbus TCP
UNIT_ID = 1                  # Modbus slave address (slave id)
DO_ADDRESS = 0               # Starting offset address of DO (decimal)
DO_COUNT = 16                # Number of DO points to be read
DO_DATA = [0, 1, 1, 0, 1, 0, 1, 1]  # DO point values to be written in batch

def read_modbus_do_bit(client: ModbusTcpClient, unit_id: int, address: int, count: int):
    """
    Modbus TCP read DO module by bit (Function Code 01 Read Coils)
    :param client: Pre-established ModbusTcpClient instance
    :param unit_id: Slave ID
    :param address: Starting offset address of DO
    :param count: Number of DO bits to read
    :return: DO bits list on success, None on failure
    """
    # Read DO output status
    res = client.read_coils(address=address, count=count, slave=unit_id)
    if res.isError():
        print(f"Failed to read DO module: {res}")
        return None
    return res.bits[:count]

# Write single DO point
def write_modbus_do_Single(client: ModbusTcpClient, unit_id: int, address: int, value: bool):
    """
    Modbus TCP write DO module (Function Code 05)
```

```
    :param client: Established ModbusTcpClient instance
    :param unit_id: Slave ID
    :param address: DO starting offset address
    :param value: 1=ON, 0=OFF
    :return: True on success, None on failure
    """
    # Write single DO point
    res = client.write_coil(address=address, value=value, slave=unit_id)
    if res.isError():
        print(f"Failed to write DO module: {res}")
        return None
    return True

# Batch write DO points
def write_modbus_do_Batch(client: ModbusTcpClient, unit_id: int, address: int, coil_list: list
    """
    Modbus TCP write DO module (Function Code 15)
    :param client: Established ModbusTcpClient instance
    :param unit_id: Slave ID
    :param address: DO starting offset address
    :param coil_list: Boolean status list
    :return: True on success, None on failure
    """
    # Batch write DO points
    res = client.write_coils(address=address, values=coil_list, slave=unit_id)
    if res.isError():
        print(f"Failed to write DO module: {res}")
        return None
    return True

if __name__ == "__main__":
    # Establish persistent TCP connection
    client = ModbusTcpClient(COUPLER_IP, port=COUPLER_PORT)
    if not client.connect():
        print("Modbus connection failed")
        exit()
    print("Persistent connection established,  press Ctrl+C to stop program")
    try:
        while True:
            # Call function to read DO by bit
            # do_data = read_modbus_do_bit(client, UNIT_ID, DO_ADDRESS, DO_COUNT)
            # if do_data is not None:
            #     print("DO Module Status List: ", do_data)

            # Batch write
            # ret = write_modbus_do_Batch(client, UNIT_ID, DO_ADDRESS, DO_DATA)
            # if ret:
            #    print("Batch DO point write operation completed")

            # Single write
            # ret = write_modbus_do_Single(client, UNIT_ID, DO_ADDRESS, value=1)
```

```
            # if ret:
            #     print("Single DO point write operation completed")

            time.sleep(1)   # Refresh every 1 second
    except KeyboardInterrupt:
        print("\nStop signal received by program")
    finally:
        # Close connection when program exits
        client.close()
        print("Modbus connection closed")
```

2.2.1.4 DO Module (Word Mapping Method)

text

```
from pymodbus.client import ModbusTcpClient
from pymodbus.pdu.register_message import WriteSingleRegisterResponse
import struct
import time
# Modbus coupler communication parameter configuration
COUPLER_IP = "192.168.0.50"  # Modbus coupler IP address
COUPLER_PORT = 502           # Default communication port of Modbus TCP
UNIT_ID = 1                  # Modbus slave address (slave id)
DO_ADDRESS_READ = 16384      # Read the starting offset address of DO (decimal)
DO_COUNT = 1                 # Number of DO registers to be read
DO_ADDRESS_WRITE = 12288     # Write the starting offset address of DO (decimal)
DO_DATA = [0xFFFF]           # DO values written by word (hexadecimal)

# Write DO by word
def write_modbus_do_word(client: ModbusTcpClient, unit_id: int, address: int, reg_list: list[i
    """
    Modbus TCP write DO module (Function Code 16)
    :param client: Established ModbusTcpClient instance
    :param unit_id: Slave ID
    :param address: DO starting offset address
    :param reg_list: Register value list to write
    :return: True on success, None on failure
    """
    # Write DO by word
    res = client.write_registers(address=address, values=reg_list, slave=unit_id)
    if res.isError():
        print(f"Failed to write DO module: {res}")
        return None
    return True

# Read DO by word
def read_modbus_do_word(client: ModbusTcpClient, unit_id: int, address: int, count: int):
    """
    Modbus TCP read DO module (Function Code 03)
    :param client: Pre-established ModbusTcpClient instance
```

```
    :param unit_id: Slave ID
    :param address: Starting offset address of DO
    :param count: Number of DO registers to read
    :return: DO register list on success, None on failure
    """
    # Read DO module status
    res = client.read_holding_registers(address=address, count=count, slave=unit_id)
    if res.isError():
        print(f"Failed to read DO module: {res}")
        return None
    return res.registers

if __name__ == "__main__":
    # Establish persistent TCP connection
    client = ModbusTcpClient(COUPLER_IP, port=COUPLER_PORT)
    if not client.connect():
        print("Modbus connection failed!")
        exit()
    try:
        while True:
            # Call function to write DO by word
             ret = write_modbus_do_word(client, UNIT_ID, DO_ADDRESS_WRITE, DO_DATA)
             if ret:
                print("DO write operation by word completed")
            # Call function to read DO by word
            # ret = read_modbus_do_word(client, UNIT_ID, DO_ADDRESS_READ, DO_COUNT)
            # if ret:
            #    print("DO Module Values: ",ret)       # Convert output value to binary for co

                time.sleep(1)  # Refresh every 1 second to maintain DO status
    except KeyboardInterrupt:
        print("Program terminated")
    finally:
        # Close connection when program exits
        client.close()
        print("Modbus connection closed")
```

2.2.1.5 AI Module (Word Mapping Method)

```text
from pymodbus.client import ModbusTcpClient
import time
# Modbus coupler communication parameter configuration
COUPLER_IP = "192.168.0.50"  # Modbus coupler IP address
COUPLER_PORT = 502           # The default communication port of Modbus TCP
UNIT_ID = 1                  # Modbus slave station address (slave ID)
AI_ADDRESS = 0               # Starting offset address of the AI module (decimal)
AI_COUNT = 4                 # The number of AI points that need to be read

# Function code 03 to read AI points
```

```
def read_modbus_ai_03(client: ModbusTcpClient, unit_id: int, address: int, count: int):
    """
    Modbus TCP read AI module (Function Code 03)
    :param client: Pre-established ModbusTcpClient instance
    :param unit_id: Slave ID
    :param address: Starting offset address of AI
    :param count: Number of AI points to read
    :return: AI register list on success, None on failure
    """
    # Read AI module data
    res = client.read_holding_registers(address=address, count=count, slave=unit_id)
    if res.isError():
        print(f"Failed to read AI module: {res}")
        return None
    return res.registers[:count]

# Function code 04 to read AI points
def read_modbus_ai_04(client: ModbusTcpClient, unit_id: int, address: int, count: int):
    """
    Modbus TCP read AI module (Function Code 04)
    :param client: Pre-established ModbusTcpClient instance
    :param unit_id: Slave ID
    :param address: Starting offset address of AI
    :param count: Number of AI points to read
    :return: AI register list on success, None on failure
    """
    # Read AI module data
    res = client.read_input_registers(address=address, count=count, slave=unit_id)
    if res.isError():
        print(f"Failed to read AI module: {res}")
        return None
    return res.registers

if __name__ == "__main__":
    # Establish persistent TCP connection
    client = ModbusTcpClient(COUPLER_IP, port=COUPLER_PORT)
    if not client.connect():
        print("Modbus connection failed!")
        exit()
    try:
        while True:
            # Call function using function code 03
            ret = read_modbus_ai_03(client, UNIT_ID, AI_ADDRESS, AI_COUNT)
            if ret:
               print("AI Module Values: ",ret)
            # Call function using function code 04
            # ret = read_modbus_ai_04(client, UNIT_ID, AI_ADDRESS, AI_COUNT)
            # if ret:
            #  print("AI Module Values: ",ret)
            time.sleep(1)  # Refresh every 1 second
    except KeyboardInterrupt:
```

```
        print("Program terminated")
    finally:
        # Close connection when program exits
        client.close()
        print("Modbus connection closed")
```

2.2.1.6 AO Module (Word Mapping Method)

text

```
from pymodbus.client import ModbusTcpClient
from pymodbus.pdu.register_message import WriteSingleRegisterResponse
import struct
import time
# Modbus coupler communication parameter configuration
COUPLER_IP = "192.168.0.50"  # Modbus coupler IP address
COUPLER_PORT = 502           # Default communication port of Modbus TCP
UNIT_ID = 1                  # Modbus slave address (slave id)
AO_ADDRESS_WRITE = 0         # Write the starting offset address of AO (in decimal)
AO_ADDRESS_READ = 8192       # Read the starting offset address of AO (decimal)
AO_COUNT = 4                 # Number of AO points to be read
AO_DATA = [1000, 2000, 3000, 4000] # AO point values to be written in batch

# Override decode method
def new_decode(self, data):
    data = data[:4]
    self.address, self.registers = struct.unpack(">HH", data)
WriteSingleRegisterResponse.decode = new_decode

# Read AO points
def read_modbus_ao(client: ModbusTcpClient, unit_id: int, address: int, count: int):
    """
    Modbus TCP read AO module (Function Code 03)
    :param client: Pre-established ModbusTcpClient instance
    :param unit_id: Slave ID
    :param address: Starting offset address of AO
    :param count: Number of AO points to read
    :return: AO register list on success, None on failure
    """
    # Read AO module data
    res = client.read_holding_registers(address=address, count=count, slave=unit_id)
    if res.isError():
        print(f"Failed to read AO module: {res}")
        return None
    return res.registers[:count]

# Write single AO point
def write_modbus_ao_Single(client: ModbusTcpClient, unit_id: int, address: int, value: int):
    """
    Modbus TCP write AO module (Function Code 06)
```

```
    :param client: Established ModbusTcpClient instance
    :param unit_id: Slave ID
    :param address: AO starting offset address
    :param value: Value to be written
    :return: True on success, None on failure
    """
    # Write single AO point
    res = client.write_register(address=address, value=value, slave=unit_id)
    if res.isError():
        print(f"Failed to write AO module: {res}")
        return None
    return True

# Batch write AO points
def write_modbus_ao_Batch(client: ModbusTcpClient, unit_id: int, address: int, reg_list: list[
    """
    Modbus TCP write AO module (Function Code 16)
    :param client: Established ModbusTcpClient instance
    :param unit_id: Slave ID
    :param address: AO starting offset address
    :param reg_list: Register value list to write
    :return: True on success, None on failure
    """
    # Batch write AO points
    res = client.write_registers(address=address, values=reg_list, slave=unit_id)
    if res.isError():
        print(f"Failed to write AO module: {res}")
        return None
    return True

if __name__ == "__main__":
    # Establish persistent TCP connection
    client = ModbusTcpClient(COUPLER_IP, port=COUPLER_PORT)
    if not client.connect():
        print("Modbus connection failed!")
        exit()
    try:
        while True:
            # Call batch write function
            ret = write_modbus_ao_Batch(client, UNIT_ID, AO_ADDRESS_WRITE, AO_DATA)
            if ret:
              print("Batch AO point write operation completed")

            # Call single write function
            # ret = write_modbus_ao_Single(client, UNIT_ID, AO_ADDRESS_WRITE, value=2000)
            # if ret:
            #   print("Single AO point write operation completed")

            # Call AO reading function
            # ret = read_modbus_ao(client, UNIT_ID, AO_ADDRESS_READ, AO_COUNT)
            # if ret:
```

```
            #    print("AO Module Values: ",ret)

            time.sleep(1)  # Refresh every 1 second to maintain AO status
    except KeyboardInterrupt:
        print("Program terminated")
    finally:
        # Close connection when program exits
        client.close()
        print("Modbus connection closed")
```

2.2.2 C++ Example Code

Introduces C++ example code for DI, DO, AI, and AO modules.

2.2.2.1 DI Module (Bit Mapping Method)

text

```
#include <iostream>
#include <thread>
#include <chrono>
#include <csignal>
#include <cstdlib>
#include <modbus/modbus.h>

using namespace std;
// Modbus coupler communication configuration
static const char* COUPLER_IP = "192.168.0.50";    // Modbus coupler IP address
static const int COUPLER_PORT = 502;               // Default Modbus TCP port
static const int UNIT_ID = 1;                      // Modbus slave ID
static const int DI_ADDRESS = 0;                   // DI start offset address (decimal)
static const int DI_COUNT = 8;                     // Number of DI points to read

static volatile bool g_running = true; // Control main loop

// Signal handler for Ctrl+C exit
void signal_handler(int sig) {
    if (sig == SIGINT) {
        cout << "\nInterrupt signal received, exiting..." << endl;
        g_running = false;
    }
}

/**
 * @brief Modbus FC02, read discrete inputs (DI)
 * @param ctx        modbus context handle
 * @param unit_id    Modbus slave ID
 * @param address    DI start offset address
 * @param count      Number of DI points to read
 * @param dest       Bit receive buffer, uint8 array, allocated externally
 * @return true:success  false:failed to set slave or read data
```

```
 */
bool read_modbus_di_bit(modbus_t* ctx, int unit_id, int address, int count, uint8_t* dest) {
    // Set slave ID
    if (modbus_set_slave(ctx, unit_id) == -1) {
        cerr << "Failed to set slave ID: " << modbus_strerror(errno) << endl;
        return false;
    }
    // Read DI module status
    int rc = modbus_read_input_bits(ctx, address, count, dest);
    if (rc == -1) {
        cerr << "Failed to read discrete inputs (FC02): " << modbus_strerror(errno) << endl;
        return false;
    }
    return true;
}

int main() {
    // Register Ctrl+C signal handler
    signal(SIGINT, signal_handler);

    // Create Modbus TCP context
    modbus_t* ctx = modbus_new_tcp(COUPLER_IP, COUPLER_PORT);
    if (ctx == nullptr) {
        cerr << "Failed to create Modbus TCP context" << endl;
        return EXIT_FAILURE;
    }
    // Establish persistent TCP connection
    if (modbus_connect(ctx) == -1) {
        cerr << "Modbus connection failed: " << modbus_strerror(errno) << endl;
        modbus_free(ctx);
        return EXIT_FAILURE;
    }
    cout << "Modbus connected successfully" << endl;

    // DI bit data buffer
    uint8_t di_bits[DI_COUNT];
    while (g_running) {
        if (read_modbus_di_bit(ctx, UNIT_ID, DI_ADDRESS, DI_COUNT, di_bits)) {
            cout << "DI Channel Status: ";
            for (int i = 0; i < DI_COUNT; ++i) {
                cout << (int)di_bits[i] << " ";
            }
            cout << endl;
        }
        // Sleep for 1 second
        this_thread::sleep_for(chrono::seconds(1));
    }

    // Close connection and release resources
    modbus_close(ctx);
    modbus_free(ctx);
```

```
    cout << "Modbus connection closed" << endl;

    return EXIT_SUCCESS;
}
```

2.2.2.2 DI Module (Word Mapping Method)

text

```
#include <iostream>
#include <thread>
#include <chrono>
#include <csignal>
#include <cstdlib>
#include <modbus/modbus.h>

using namespace std;
// Modbus coupler communication configuration
static const char* COUPLER_IP = "192.168.0.50";    // Modbus coupler IP address
static const int COUPLER_PORT = 502;               // Default Modbus TCP port
static const int UNIT_ID = 1;                      // Modbus slave ID
static const int DI_ADDRESS = 20480;               // DI start offset address (decimal)
static const int DI_COUNT = 1;                     // Number of DI registers to read

static volatile bool g_running = true; // Control main loop

// Signal handler for Ctrl+C exit
void signal_handler(int sig) {
    if (sig == SIGINT) {
        cout << "\nInterrupt signal received, exiting..." << endl;
        g_running = false;
    }
}

/**
 * @brief Modbus FC03, read DI data stored in holding registers
 * @param ctx        modbus context handle
 * @param unit_id    Modbus slave ID
 * @param address    DI start offset address
 * @param count      Number of registers to read
 * @param dest       Register receive buffer, uint16 array, allocated externally
 * @return true:success  false:failed to set slave or read data
 */
bool read_modbus_di_reg(modbus_t* ctx, int unit_id, int address, int count, uint16_t* dest) {
    // Set slave ID
    if (modbus_set_slave(ctx, unit_id) == -1) {
        cerr << "Failed to set slave ID: " << modbus_strerror(errno) << endl;
        return false;
    }
    // Read holding registers (FC03)
    int rc = modbus_read_registers(ctx, address, count, dest);
```

```
    if (rc == -1) {
        cerr << "Failed to read DI holding registers (FC03): " << modbus_strerror(errno) << en
        return false;
    }
    return true;
}

int main() {
    // Register Ctrl+C signal handler
    signal(SIGINT, signal_handler);

    // Create Modbus TCP context
    modbus_t* ctx = modbus_new_tcp(COUPLER_IP, COUPLER_PORT);
    if (ctx == nullptr) {
        cerr << "Failed to create Modbus TCP context" << endl;
        return EXIT_FAILURE;
    }
    // Establish persistent TCP connection
    if (modbus_connect(ctx) == -1) {
        cerr << "Modbus connection failed: " << modbus_strerror(errno) << endl;
        modbus_free(ctx);
        return EXIT_FAILURE;
    }
    cout << "Modbus connected successfully" << endl;

    // DI register buffer
    uint16_t di_regs[DI_COUNT];
    while (g_running) {
        if (read_modbus_di_reg(ctx, UNIT_ID, DI_ADDRESS, DI_COUNT, di_regs)) {
            cout << "DI Register Value: "; // Converting the output value to binary shows the
            for (int i = 0; i < DI_COUNT; ++i) {
                cout << di_regs[i] << " ";
            }
            cout << endl;
        }
        // Sleep for 1 second
        this_thread::sleep_for(chrono::seconds(1));
    }

    // Close connection and release resources
    modbus_close(ctx);
    modbus_free(ctx);
    cout << "Modbus connection closed" << endl;

    return EXIT_SUCCESS;
}
```

2.2.2.3 DO Module (Bit Mapping Method)

```text
#include <iostream>
#include <thread>
#include <chrono>
#include <csignal>
#include <cstdlib>
#include <modbus/modbus.h>

using namespace std;
// Modbus coupler communication parameters configuration
static const char* COUPLER_IP = "192.168.0.50";             // Modbus coupler IP address
static const int COUPLER_PORT = 502;                        // Default Modbus TCP communicatio
static const int UNIT_ID = 1;                               // Modbus slave ID
static const int DO_ADDRESS = 0;                            // DO starting offset address (dec
static const int DO_COUNT = 16;                             // Number of DO points to read
static const uint8_t DO_DATA[] = {1, 1, 1, 1, 0, 0, 0, 0};  // Batch DO point values to be wri
static const int DO_WRITE_COUNT = sizeof(DO_DATA) / sizeof(DO_DATA[0]);  // Number of points t

static volatile bool g_running = true; // Control main loop

// Signal handler for Ctrl+C exit
void signal_handler(int sig) {
    if (sig == SIGINT) {
        cout << "\nInterrupt signal received, exiting..." << endl;
        g_running = false;
    }
}

/**
 * @brief Modbus FC05, write single coil (single DO channel)
 * @param ctx        modbus context handle
 * @param unit_id    Modbus slave ID
 * @param address    DO starting offset address
 * @param value      Target value (0=OFF, 1=ON)
 * @return true:success  false:failed
 */
bool write_modbus_do_single(modbus_t* ctx, int unit_id, int address, uint8_t value)
{
    // Set slave ID
    if (modbus_set_slave(ctx, unit_id) == -1)
    {
        cerr << "Failed to set slave ID: " << modbus_strerror(errno) << endl;
        return false;
    }
    // Write single DO coil
    int rc = modbus_write_bit(ctx, address, value ? 1 : 0);
    if (rc == -1)
    {
        cerr << "Single DO write failed (FC05): " << modbus_strerror(errno) << endl;
```

```
        return false;
    }
    return true;
}

/**
 * @brief Modbus FC05, write multiple coils (multiple DO channels)
 * @param ctx        modbus context handle
 * @param unit_id    Modbus slave ID
 * @param address    DO starting coil address
 * @param count      Number of coils to write
 * @param bits       Array of target coil status
 * @return true:success  false:failed
 */
bool write_modbus_do_batch(modbus_t* ctx, int unit_id, int address, int count, const uint8_t*
{
    // Set slave ID
    if (modbus_set_slave(ctx, unit_id) == -1)
    {
        cerr << "Failed to set slave ID: " << modbus_strerror(errno) << endl;
        return false;
    }
    // Batch write DO coils
    int rc = modbus_write_bits(ctx, address, count, bits);
    if (rc == -1)
    {
        cerr << "Batch DO write failed (FC15): " << modbus_strerror(errno) << endl;
        return false;
    }
    return true;
}

/**
 * @brief Modbus FC01, read DO coil status
 * @param ctx        modbus context handle
 * @param unit_id    Modbus slave ID
 * @param address    DO start offset address
 * @param count      Number of DO points to read
 * @param dest       Bit receive buffer, uint8 array, allocated externally
 * @return true:success  false:failed to set slave or read data
 */
bool read_modbus_do_bit(modbus_t* ctx, int unit_id, int address, int count, uint8_t* dest) {
    // Set slave ID
    if (modbus_set_slave(ctx, unit_id) == -1) {
        cerr << "Failed to set slave ID: " << modbus_strerror(errno) << endl;
        return false;
    }
    // FC01 Read coils (DO)
    int rc = modbus_read_bits(ctx, address, count, dest);
    if (rc == -1) {
        cerr << "Failed to read DO coils (FC01): " << modbus_strerror(errno) << endl;
```

```
        return false;
    }
    return true;
}

int main()
{
    // Register Ctrl+C signal handler
    signal(SIGINT, signal_handler);

    // Create Modbus TCP context
    modbus_t* ctx = modbus_new_tcp(COUPLER_IP, COUPLER_PORT);
    if (ctx == nullptr)
    {
        cerr << "Failed to create Modbus TCP context" << endl;
        return EXIT_FAILURE;
    }
    // Establish persistent TCP connection
    if (modbus_connect(ctx) == -1)
    {
        cerr << "Modbus connection failed: " << modbus_strerror(errno) << endl;
        modbus_free(ctx);
        return EXIT_FAILURE;
    }
    cout << "Modbus connected successfully" << endl;

    // DO bit data buffer
    uint8_t do_bits[DO_COUNT];
    while (g_running)
    {
        // Batch write DO channels
        if (write_modbus_do_batch(ctx, UNIT_ID, DO_ADDRESS, DO_WRITE_COUNT, DO_DATA))
        {
            cout << "Batch DO point write completed" << endl;
        }
        // Single DO write
        /* if (write_modbus_do_single(ctx, UNIT_ID, DO_ADDRESS, 1))
        {
            cout << "Single DO point write completed" << endl;
        } */
        // read DO
        /* if (read_modbus_do_bit(ctx, UNIT_ID, DO_ADDRESS, DO_COUNT, do_bits))
        {
        cout << "DO Channel Status: ";
        for (int i = 0; i < DO_COUNT; ++i) {
            cout << (int)do_bits[i] << " ";
        }
        cout << endl;
        } */
        this_thread::sleep_for(chrono::seconds(1)); // Refresh every 1 second
    }
```

```
    // Close connection and release resources
    modbus_close(ctx);
    modbus_free(ctx);
    cout << "Modbus connection closed" << endl;

    return EXIT_SUCCESS;
}
```

2.2.2.4 DO Module (Word Mapping Method)

text

```
#include <iostream>
#include <thread>
#include <chrono>
#include <csignal>
#include <cstdlib>
#include <modbus/modbus.h>

using namespace std;
// Modbus coupler communication parameters configuration
static const char* COUPLER_IP = "192.168.0.50";                          // Modbus coupler IP
static const int COUPLER_PORT = 502;                                     // Default Modbus TCP
static const int UNIT_ID = 1;                                            // Modbus slave ID
static const int DO_ADDRESS_READ = 16384;                                // Read the starting
static const int DO_COUNT = 1;                                           // Number of DO regis
static const int DO_ADDRESS_WRITE = 12288;                               // Write the starting
static const uint16_t DO_DATA[] = {0xFFFF};                              // DO values written
static const int DO_WRITE_COUNT = sizeof(DO_DATA) / sizeof(DO_DATA[0]);  // Number of register

static volatile bool g_running = true; // Control main loop

// Signal handler for Ctrl+C exit
void signal_handler(int sig) {
    if (sig == SIGINT) {
        cout << "\nInterrupt signal received, exiting..." << endl;
        g_running = false;
    }
}

/**
 * @brief Modbus FC16, write DO registers
 * @param ctx        modbus context handle
 * @param unit_id    Modbus slave ID
 * @param address    Starting register address
 * @param count      Number of registers to write
 * @param regs       Target register value array
 * @return true:success  false:failed
 */
bool write_modbus_do_word_batch(modbus_t* ctx, int unit_id, int address, int count, const uint
```

```
{
    // Set slave ID
    if (modbus_set_slave(ctx, unit_id) == -1)
    {
        cerr << "Failed to set slave ID: " << modbus_strerror(errno) << endl;
        return false;
    }
    // Write registers (FC16)
    int rc = modbus_write_registers(ctx, address, count, regs);
    if (rc == -1)
    {
        cerr << "DO word-based write failed (FC16): " << modbus_strerror(errno) << endl;
        return false;
    }
    return true;
}

/**
 * @brief Modbus FC03, read DO status mapped to holding registers
 * @param ctx        modbus context handle
 * @param unit_id    Modbus slave ID
 * @param address    DO register start offset address
 * @param count      Number of registers to read
 * @param dest       Register receive buffer, uint16 array, allocated externally
 * @return true:success  false:failed to set slave or read data
 */
bool read_modbus_do_reg(modbus_t* ctx, int unit_id, int address, int count, uint16_t* dest) {
    // Set slave ID
    if (modbus_set_slave(ctx, unit_id) == -1) {
        cerr << "Failed to set slave ID: " << modbus_strerror(errno) << endl;
        return false;
    }
    // Read DO module status (FC03)
    int rc = modbus_read_registers(ctx, address, count, dest);
    if (rc == -1) {
        cerr << "Failed to read DO holding registers (FC03): " << modbus_strerror(errno) << en
        return false;
    }
    return true;
}

int main()
{
    // Register Ctrl+C signal handler
    signal(SIGINT, signal_handler);

    // Create Modbus TCP context
    modbus_t* ctx = modbus_new_tcp(COUPLER_IP, COUPLER_PORT);
    if (ctx == nullptr)
    {
        cerr << "Failed to create Modbus TCP context" << endl;
```

```
        return EXIT_FAILURE;
    }
    // Establish persistent TCP connection
    if (modbus_connect(ctx) == -1)
    {
        cerr << "Modbus connection failed: " << modbus_strerror(errno) << endl;
        modbus_free(ctx);
        return EXIT_FAILURE;
    }
    cout << "Modbus connected successfully" << endl;
    uint16_t do_regs[DO_COUNT];
    while (g_running)
    {
        // Write an example of DO module call
        /* if (write_modbus_do_word_batch(ctx, UNIT_ID, DO_ADDRESS_WRITE, DO_WRITE_COUNT, DO_D
        {
            cout << "Word-based DO write completed" << endl;
        } */
        // Reading the example of DO module call
        if (read_modbus_do_reg(ctx, UNIT_ID, DO_ADDRESS_READ, DO_COUNT, do_regs)) {
            cout << "DO Register Value: "; // Converting the output value to binary shows the
            for (int i = 0; i < DO_COUNT; ++i) {
                cout << do_regs[i] << " ";
            }
            cout << endl;
        }

        this_thread::sleep_for(chrono::seconds(1)); // Refresh every 1 second
    }

    // Close connection and release resources
    modbus_close(ctx);
    modbus_free(ctx);
    cout << "Modbus connection closed" << endl;

    return EXIT_SUCCESS;
}
```

2.2.2.5 AI Module (Word Mapping Method)

text

```
#include <iostream>
#include <thread>
#include <chrono>
#include <csignal>
#include <cstdlib>
#include <modbus/modbus.h>

using namespace std;
// Modbus coupler communication configuration
```

```
static const char* COUPLER_IP = "192.168.0.50";    // Modbus coupler IP address
static const int COUPLER_PORT = 502;               // Default Modbus TCP port
static const int UNIT_ID = 1;                      // Modbus slave ID
static const int AI_ADDRESS = 0;                   // Start offset address of AI module (decim
static const int AI_COUNT = 4;                     // Number of AI points to read

static volatile bool g_running = true; // Control main loop

// Signal handler for Ctrl+C exit
void signal_handler(int sig) {
    if (sig == SIGINT) {
        cout << "\nInterrupt signal received, exiting..." << endl;
        g_running = false;
    }
}

/**
 * @brief Modbus function code 03, read holding registers
 * @param ctx        modbus context handle
 * @param unit_id    Modbus slave ID
 * @param address    Start offset address of AI module
 * @param count      Number of AI points to read
 * @param dest       Receive buffer, provided externally, length >= count
 * @return true:success  false:failed to set slave or read registers
 */
bool read_modbus_ai_03(modbus_t* ctx, int unit_id, int address, int count, uint16_t* dest) {
    // Set slave ID
    if (modbus_set_slave(ctx, unit_id) == -1) {
        cerr << "Failed to set slave ID: " << modbus_strerror(errno) << endl;
        return false;
    }
    // Read AI module data
    int rc = modbus_read_registers(ctx, address, count, dest);
    if (rc == -1) {
        cerr << "Failed to read AI holding registers (FC03): " << modbus_strerror(errno) << en
        return false;
    }
    return true;
}

/**
 * @brief Modbus function code 04, read input registers
 * @param ctx        modbus context handle
 * @param unit_id    Modbus slave ID
 * @param address    Start offset address of AI module
 * @param count      Number of AI points to read
 * @param dest       Receive buffer, provided externally, length >= count
 * @return true:success  false:failed to set slave or read registers
 */
bool read_modbus_ai_04(modbus_t* ctx, int unit_id, int address, int count, uint16_t* dest) {
    // Set slave ID
```

```
    if (modbus_set_slave(ctx, unit_id) == -1) {
        cerr << "Failed to set slave ID: " << modbus_strerror(errno) << endl;
        return false;
    }
    // Read AI module data
    int rc = modbus_read_input_registers(ctx, address, count, dest);
    if (rc == -1) {
        cerr << "Failed to read AI input registers (FC04): " << modbus_strerror(errno) << endl
        return false;
    }
    return true;
}

int main() {
    // Register Ctrl+C signal handler
    signal(SIGINT, signal_handler);

    // Create Modbus TCP context
    modbus_t* ctx = modbus_new_tcp(COUPLER_IP, COUPLER_PORT);
    if (ctx == nullptr) {
        cerr << "Failed to create Modbus TCP context" << endl;
        return EXIT_FAILURE;
    }

    // Establish persistent TCP connection
    if (modbus_connect(ctx) == -1) {
        cerr << "Modbus connection failed: " << modbus_strerror(errno) << endl;
        modbus_free(ctx);
        return EXIT_FAILURE;
    }
    cout << "Modbus connected successfully" << endl;

    // Buffer for received register data
    uint16_t regs[AI_COUNT];
    while (g_running) {
        // Read data via function code 03
        if (read_modbus_ai_03(ctx, UNIT_ID, AI_ADDRESS, AI_COUNT, regs)) {
            cout << "AI Values (FC03): ";
            for (int i = 0; i < AI_COUNT; ++i) {
                cout << regs[i] << " ";
            }
            cout << endl;
        }
        // Uncomment block below and comment FC03 call if using function code 04
        /* if (read_modbus_ai_04(ctx, UNIT_ID, AI_ADDRESS, AI_COUNT, regs)) {
            cout << "AI Values (FC04): ";
            for (int i = 0; i < AI_COUNT; ++i) {
                cout << regs[i] << " ";
            }
            cout << endl;
        } */
```

```
        // Sleep for 1 second
        this_thread::sleep_for(chrono::seconds(1));
    }
    // Close connection and release resources
    modbus_close(ctx);
    modbus_free(ctx);
    cout << "Modbus connection closed" << endl;

    return EXIT_SUCCESS;
}
```

2.2.2.6 AO Module (Word Mapping Method)

text

```
#include <iostream>
#include <thread>
#include <chrono>
#include <csignal>
#include <cstdlib>
#include <cstdint>
#include <unistd.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <modbus/modbus.h>

using namespace std;

// Modbus coupler communication parameters
static const char* COUPLER_IP = "192.168.0.50";       // Modbus coupler IP address
static const int COUPLER_PORT = 502;                  // Default Modbus TCP communication port
static const int UNIT_ID = 1;                         // Modbus slave ID
static const int AO_ADDRESS_WRITE = 0;                // Write the starting offset address of
static const int AO_ADDRESS_READ = 8192;              // Read the starting offset address of A
static const int AO_COUNT = 4;                        // Number of AO points to read
static uint16_t AO_DATA[] = {1000, 2000, 3000, 4000}; // AO channel values to be written in ba
static const int AO_WRITE_COUNT = sizeof(AO_DATA)/sizeof(AO_DATA[0]);  // Number of registers

static volatile bool g_running = true; // Main loop control flag

// Signal handler for Ctrl+C exit
void signal_handler(int sig) {
    if (sig == SIGINT) {
        cout << "\nInterrupt signal received, exiting..." << endl;
        g_running = false;
    }
}

// Flush TCP socket receive buffer
bool flush_tcp_buffer(int fd)
{
```

```
    if(fd < 0)
        return false;
    // Save original blocking flag
    int flags = fcntl(fd, F_GETFL, 0);
    if(flags == -1)
        return false;
    // Set non-blocking mode
    fcntl(fd, F_SETFL, flags | O_NONBLOCK);
    uint8_t tmp[64];
    while (true)
    {
        ssize_t r = recv(fd, tmp, sizeof(tmp), MSG_DONTWAIT);
        if(r <= 0)
            break;
    }
    // Restore blocking mode
    fcntl(fd, F_SETFL, flags);
    return true;
}

/**
 * @brief Modbus Function Code 06, write single holding register (single AO channel)
 * @param ctx        modbus context handle
 * @param unit_id    Modbus slave ID
 * @param address    register address
 * @param value      value to write into register
 * @return true:success  false:failure
 */
bool write_modbus_ao_single(modbus_t* ctx, int unit_id, int address, uint16_t value)
{
    // Set slave ID
    if (modbus_set_slave(ctx, unit_id) == -1)
    {
        cerr << "Failed to set slave ID: " << modbus_strerror(errno) << endl;
        return false;
    }
    // Write single AO register
    int rc = modbus_write_register(ctx, address, value);

    int sockfd = modbus_get_socket(ctx);
    flush_tcp_buffer(sockfd);

    if (rc == -1)
    {
        cerr << "Single AO write failed (FC06): " << modbus_strerror(errno) << endl;
        return false;
    }
    return true;
}

/**
```

```
 * @brief Modbus Function Code 16, batch write holding registers (multiple AO channels)
 * @param ctx        modbus context handle
 * @param unit_id    Modbus slave ID
 * @param address    starting register address
 * @param count      number of registers to write
 * @param regs       array of register values to write
 * @return true:success  false:failure
 */
bool write_modbus_ao_batch(modbus_t* ctx, int unit_id, int address, int count, uint16_t* regs)
{
    // Set slave ID
    if (modbus_set_slave(ctx, unit_id) == -1)
    {
        cerr << "Failed to set slave ID: " << modbus_strerror(errno) << endl;
        return false;
    }
    // Batch write AO registers
    int rc = modbus_write_registers(ctx, address, count, regs);
    if (rc == -1)
    {
        cerr << "Batch AO write failed (FC16): " << modbus_strerror(errno) << endl;
        return false;
    }
    return true;
}

/**
 * @brief Modbus FC03, read holding registers (AO registers)
 * @param ctx        modbus context handle
 * @param unit_id    Modbus slave ID
 * @param address    AO module start offset address
 * @param count      Number of points to read
 * @param dest       Receive buffer, provided externally, length >= count
 * @return true:success  false:failed to set slave or read registers
 */
bool read_modbus_ao(modbus_t* ctx, int unit_id, int address, int count, uint16_t* dest) {
    // Set slave ID
    if (modbus_set_slave(ctx, unit_id) == -1) {
        cerr << "Failed to set slave ID: " << modbus_strerror(errno) << endl;
        return false;
    }
    // Read holding registers (FC03)
    int rc = modbus_read_registers(ctx, address, count, dest);
    if (rc == -1) {
        cerr << "Failed to read AO holding registers (FC03): " << modbus_strerror(errno) << en
        return false;
    }
    return true;
}

uint16_t regs[AO_COUNT];
```

```
int main()
{
    // Register Ctrl+C signal handler
    signal(SIGINT, signal_handler);

    // Create Modbus TCP context
    modbus_t* ctx = modbus_new_tcp(COUPLER_IP, COUPLER_PORT);
    if (ctx == nullptr)
    {
        cerr << "Failed to create Modbus TCP context" << endl;
        return EXIT_FAILURE;
    }
    // Establish persistent TCP connection
    if (modbus_connect(ctx) == -1)
    {
        cerr << "Modbus connection failed: " << modbus_strerror(errno) << endl;
        modbus_free(ctx);
        return EXIT_FAILURE;
    }
    cout << "Modbus connected successfully" << endl;

    while (g_running)
    {
        // Batch write AO (FC16)
        if (write_modbus_ao_batch(ctx, UNIT_ID, AO_ADDRESS_WRITE, AO_WRITE_COUNT, AO_DATA))
        {
            cout << "Batch AO write completed" << endl;
        }
        // Single channel write (FC06)
        /* if (write_modbus_ao_single(ctx, UNIT_ID, AO_ADDRESS_WRITE, 1000))
        {
            cout << "Single AO write completed" << endl;
        } */
        // Read AO(FC03)
       /*  if (read_modbus_ao(ctx, UNIT_ID, AO_ADDRESS_READ, AO_COUNT, regs)) {
            cout << "AO Values: ";
            for (int i = 0; i < AO_COUNT; ++i) {
                cout << regs[i] << " ";
            }
            cout << endl;
        } */
        this_thread::sleep_for(chrono::seconds(1));
    }

    // Close connection and release resources
    modbus_close(ctx);
    modbus_free(ctx);
    cout << "Modbus connection closed" << endl;
```

```
    return EXIT_SUCCESS;
}
```