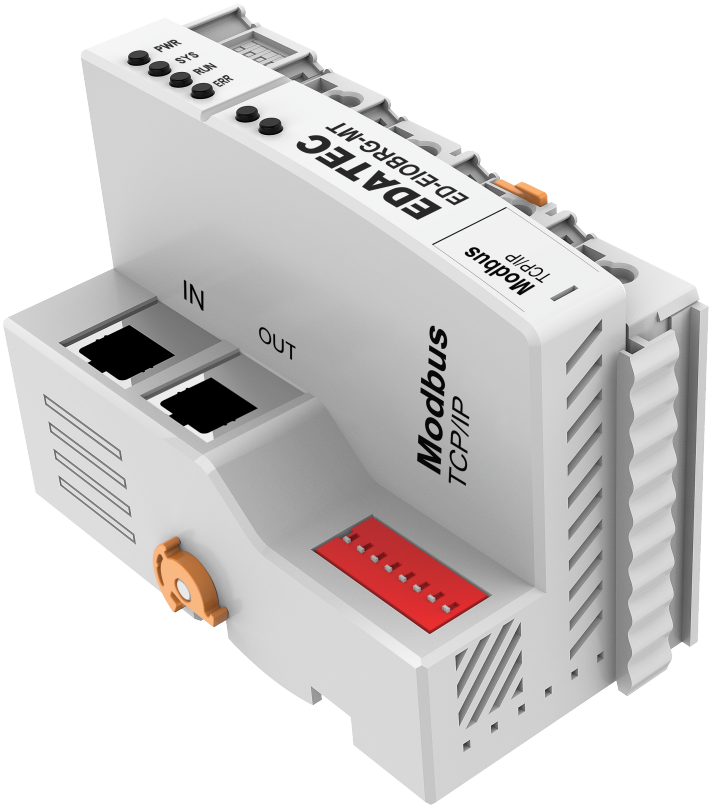


Modbus TCP耦合器

用户手册

by EDA Technology Co., Ltd

built: 2026-09-11

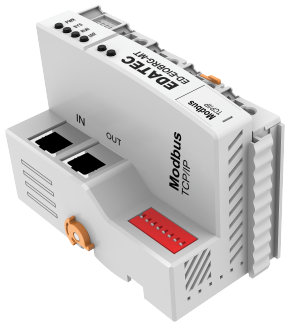
1 快速入门指南

介绍Modbus TCP/IP耦合器的概述、组网、接线、网络配置、其他参数设置和I/O模块设置。

1.1 产品概述

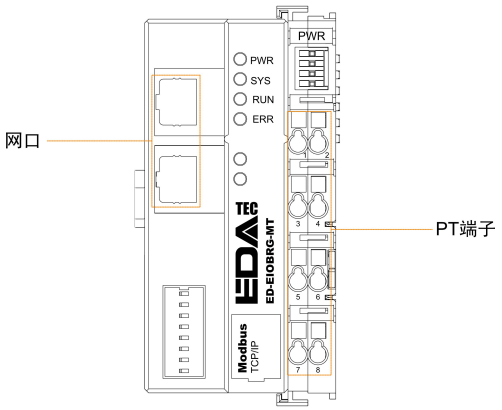
ED-EIOBRG-MT是一款Modbus TCP耦合器，主要用于将采用不同现场总线协议的从站设备（如Profibus DP、CANopen、DeviceNet等）无缝接入标准Modbus TCP以太网网络。它充当协议转换网关的角色，使上位机（PLC、SCADA、HMI）能够通过统一的Modbus TCP接口访问下层异构总线设备的数据。

ED-EIOBRG-MT的传输距离最长为100m，且最多支持扩展连接32个I/O模块。作为连接节点，可将多个Modbus TCP 从站设备连接到Modbus TCP网络中，满足不同应用场景的需求。



1.1.1 接口

ED-EIOBRG-MT包含2个100M以太网口和8个PT端子，如下图所示。



- 2个100M以太网口采用标准RJ45连接器，分为IN和OUT端口，用于实现与上位控制系统的通信连接及系统组网。

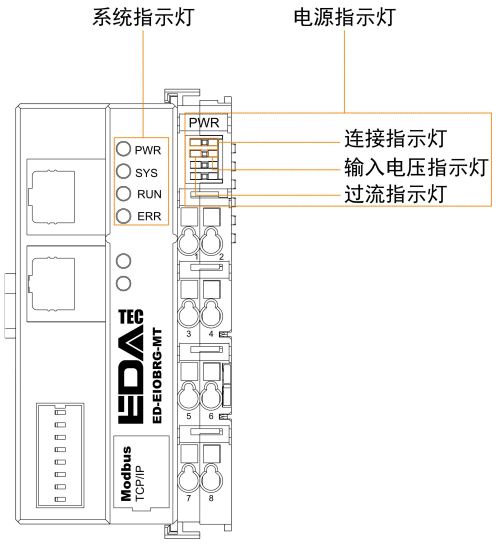
以太网口	
IN	输入端口，用于连接同一网段的Modbus TCP主站或其他Modbus TCP从站的OUT端口
OUT	输出端口，用于连接同一网段的其他Modbus TCP从站的IN端口

- 8个PT端子采用直插式弹簧接线端子，用于接入系统侧和现场侧的电源。

PT端子			
Pin编号	定义	Pin编号	定义
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 指示灯

ED-EIOBRG-MT包含4个系统指示灯和3个电源指示灯，如下图所示。



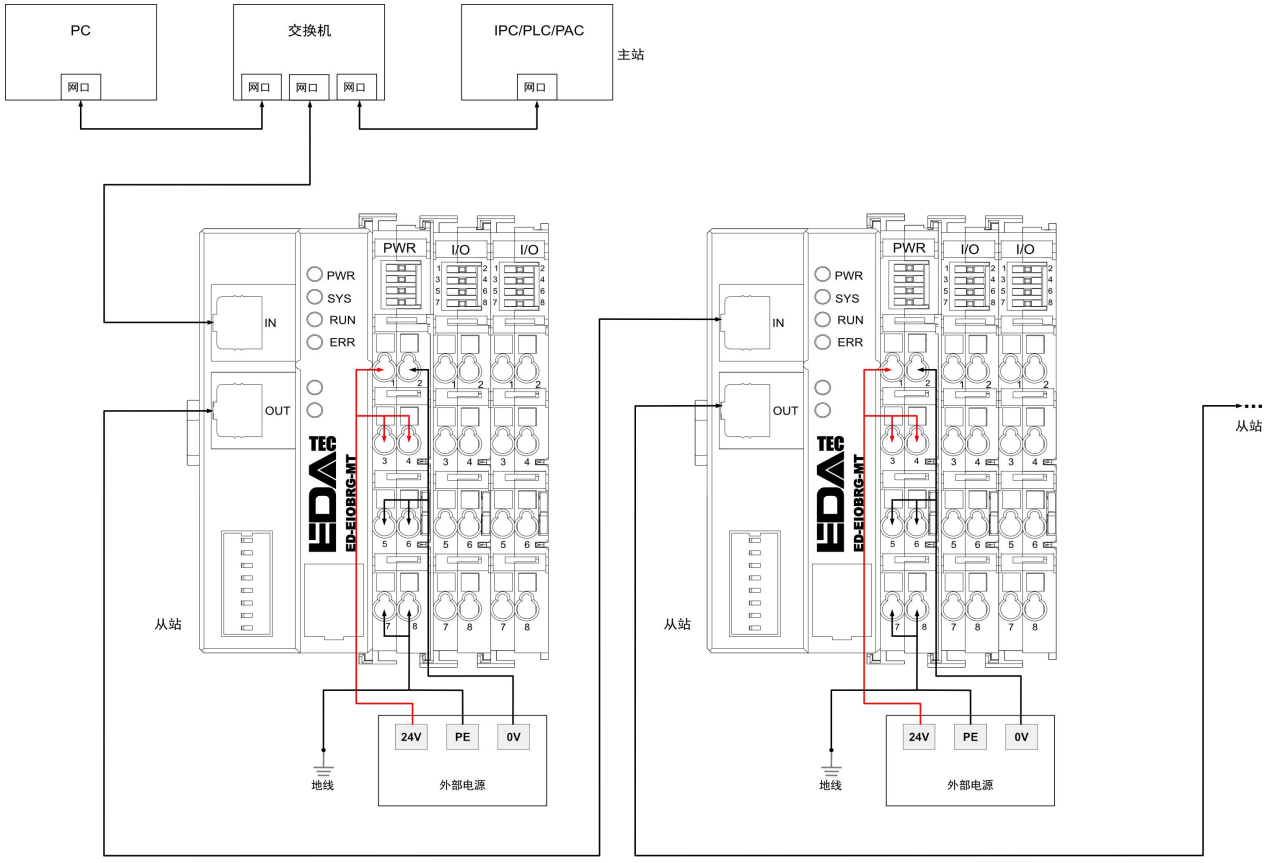
系统指示灯	
PWR	耦合器电源指示灯，绿色，用于查看耦合器系统电源的连接状态。 • 常亮：已正常连接系统电源； • 熄灭：系统未上电或电源故障。
SYS	系统提示灯，绿色，用于查看耦合器的操作状态。 • 慢闪（1s/次）：系统操作正常； • 快闪（1s/3次-5次）：系统操作异常。
RUN	运行指示灯，绿色，用于查看耦合器的运行状态。 • 常亮：系统正常运行； • 熄灭：系统未运行。
ERR	故障指示灯，红色，用于查看I/O模块是否发生掉线故障。 • 熄灭：所有I/O模块通信正常，无掉线情况； • 常亮：存在至少一个I/O模块掉线。

电源指示灯	
连接指示灯	

电源指示灯	
	绿色，用于查看公共端的连线是否正常。 • 常亮：系统电源接线正常（24V SYS接入DC+，0V SYS接入DC-）； • 熄灭：系统电源接线异常。
输入电压指示灯	绿色，用于查看输入电压是否正常。 • 常亮：电压输入正常 • 熄灭：电压输入异常
过流指示灯	红色，用于查看耦合器是否超过最大工作电流。 • 熄灭：耦合器的工作电流未超过最大工作电流 • 常亮：耦合器的工作电流已超过最大工作电流

1.2 组网和连接

ED-EIOBRG-MT支持连接I/O模块，需要搭配Modbus TCP的主站使用，同时支持级联其他的Modbus TCP的从站，具体的系统组网如下图。



- ED-EIOBRG-MT耦合器通过网络与Modbus TCP主站进行连接。
- 多个ED-EIOBRG-MT耦合器支持级联。
- Modbus TCP主站、PC和ED-EIOBRG-MT的IP地址需要在同一个网段。
- 支持通过windows PC（安装上位机软件）来配置耦合器的相关参数。
- 每个ED-EIOBRG-MT挂载的I/O模块（数字量输入/输出、模拟量输入/输出等）通过内部背板总线与耦合器通信，耦合器自动扫描并识别各I/O模块的型号及顺序，单个ED-EIOBRG-MT最多可挂载32个I/O模块。

- 每个ED-EIOBRG-MT需接入DC 24V电源为其供电，接线步骤如下：
  1. 先将导线端部剥去8~10mm绝缘层；
  2. 使用一字型螺丝刀（推荐型号2×75mm）垂直插入端子上方孔内并向下按压以开启弹簧；
  3. 将剥好的导线插入圆形接线孔，再拔出螺丝刀，弹簧将自动复位压紧导线；

> 提示
>
> 连接电源线时，请严格参照上图所示的接线指引进行接线，务必注意区分电源正负极，切勿接反；否则可能导致模块无法正常工作、运行异常，甚至造成模块损坏。

1.3 配置参数

1.3.1 配置ED-EIOBRG-MT参数

ED-EIOBRG-MT支持通过上位机软件进行参数配置。以下将详细介绍如何使用上位机软件（IO Controller）完成配置操作。

1.3.1.1 配置网络

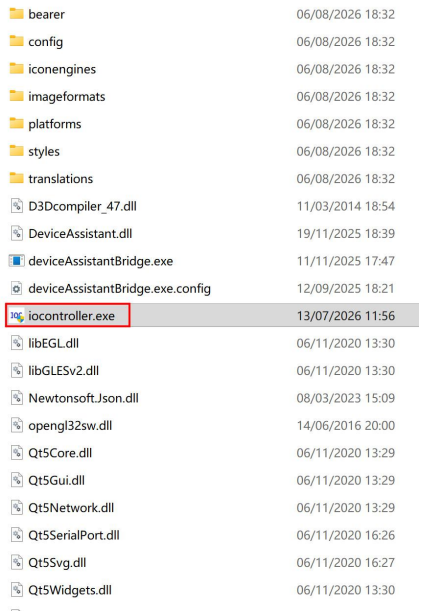
ED-EIOBRG-MT出厂时的默认IP地址为 `192.168.0.50` 。在实际使用时，需将其IP地址配置为与Modbus TCP主站处于同一网段。

前提条件：

- 已在Windows PC上下载IO Controller软件，下载地址为：IO Controller V1.2.1.zip (https://1826505135.cdn.123clouddisk.com/1826505135/39382825) 。
- 已完成ED-EIOBRG-MT的物理连接，确保Modbus TCP主站、Windows PC和ED-EIOBRG-MT的IP地址在同一个网段。

操作步骤：

1. 解压已下载的 `IO Controller V1.2.1.zip` 。
2. 在解压后的 `IO Controller V1.2.1` 文件夹中，鼠标左键双击 `iocontroller.exe` ，运行IO Controller上位机软件。



3. 在打开的界面中，右键单击“Projects”区域下方的空白处，并选择“新建工程”。

4. 自定义修改工程名称（如下图中定义工程名称为“test”），再单击“OK”。

5. 左键单击新建的工程test，在“属性”区域配置工程的属性。

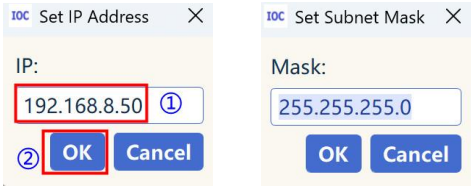
- 扫描通道：选择“以太网”
- 串口：无需选择
- 网卡：选择Windows PC的IP地址

6. 右键单击新建的工程test，在菜单中选择“扫描设备”。

7. 在打开的“以太网设备扫描”界面中，单击下方的“开始扫描”，获取ED-EIOBRG-MT的IP地址信息。

8. 右键单击扫描出的IP地址条目，在右键菜单中选择“Set IP Address...”或者“Set Mask...”。

9. 按需设置IP地址和子网掩码。



10. IP地址设置完成后，单击界面下方的“更新设备IP”。

11. IP地址更新完成后，界面上方的标题栏会显示“IP更新完成”。

12. 单击界面下方的“下载所有设备”，下载ED-EIOBRG-MT以及扩展的I/O模块的数据。

13. 下载完成后，ED-EIOBRG-MT以及扩展的I/O模块的信息将会显示在界面中。

1.3.1.2 设置其他参数

ED-EIOBRG-MT的端口号、Watchdog和断线输出模式均支持通过IO Controller上位机软件来设置，下文详细介绍具体的操作。

前提条件：

- 已完成ED-EIOBRG-MT的IP地址设置。
- 已完成ED-EIOBRG-MT和I/O模块的设备下载。

操作步骤：

1. 在界面左侧的test工程下，左键单击“ED-EIOBRG-MT”，再选择界面右侧的“配置参数”。

2. 鼠标左键双击“配置参数”表中的“值”列对应的数值，按需设置“端口号”、Watchdog Enable、Watchdog时间 (ms)和Slave Output Hold/Clear等参数。

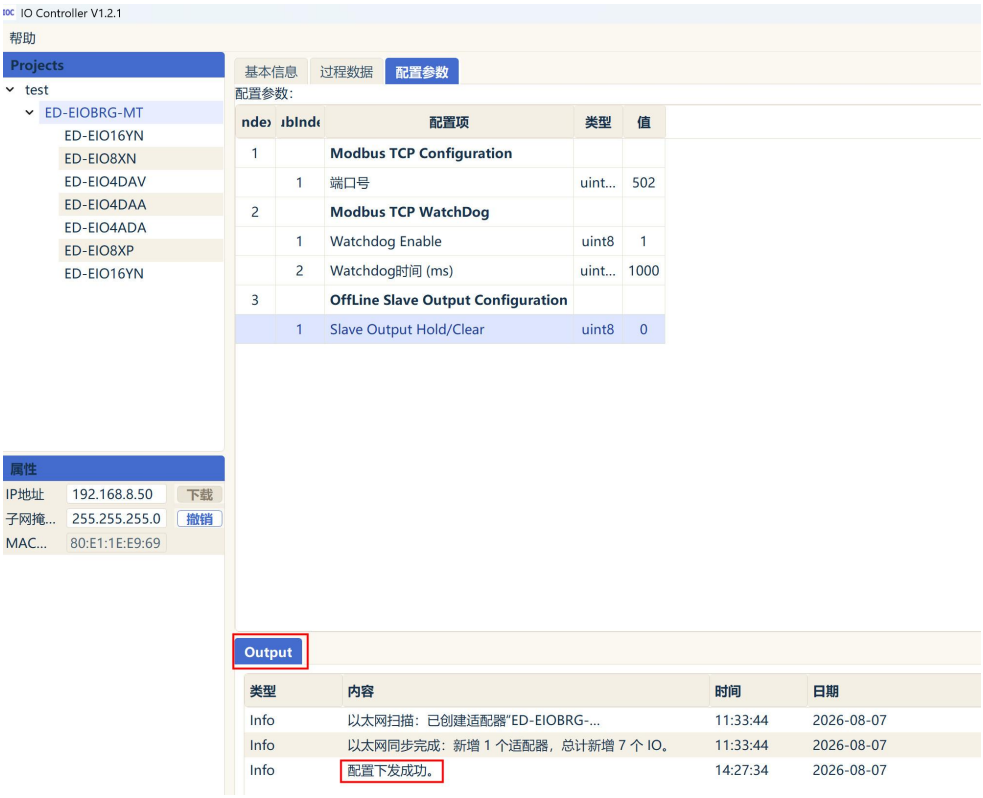
- 具体的参数配置说明如下表：

参数	配置说明
端口号	默认值为502，取值范围为1~65535，按需设置
Watchdog Enable	

参数	配置说明
	默认值为1，取值包含0和1 • 1：使能 • 0：禁用
Watchdog时间 (ms)	默认值为180000，取值范围为1000~4294967295，按需设置
Slave Output Hold/ Clear（输出模块断线时的输出模式）	默认值为0，取值包含0和1 • 0：输出模块断线时，输出清零 • 1：输出模块断线时，输出保持

3. 参数设置完成后，在界面左侧的test工程下，右键单击“ED-EIOBRG-MT”，再选择菜单中的“下载配置”。

4. 下载完成后，界面右下方的“Output”区域会显示“配置下发成功”的信息。



1.3.2 配置I/O模块参数

安装在ED-EIOBRG-MT上的I/O模块类型分为数字量输入（DI）、数字量输出（DO）、模拟量输入（AI）和模拟量输出（AO），其中DI和DO模块无需设置，AI和AO模块支持通过IO Controller上位机软件来配置电压和电流的范围，下文详细介绍具体的操作。

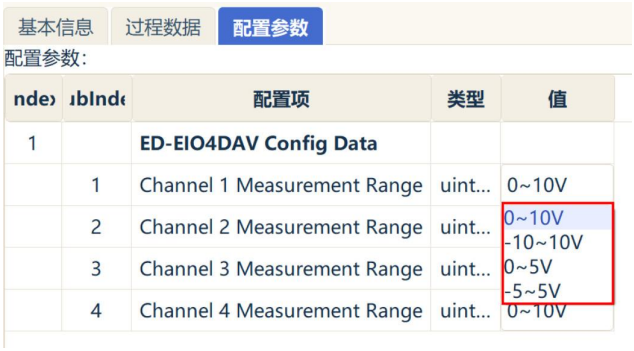
前提条件：

- 已完成ED-EIOBRG-MT的IP地址设置。
- 已完成ED-EIOBRG-MT和I/O模块的设备下载。

操作步骤：

1. 在界面左侧的test工程下，左键单击I/O模块（例如ED-EIO4DAV），再选择界面右侧的“配置参数”。

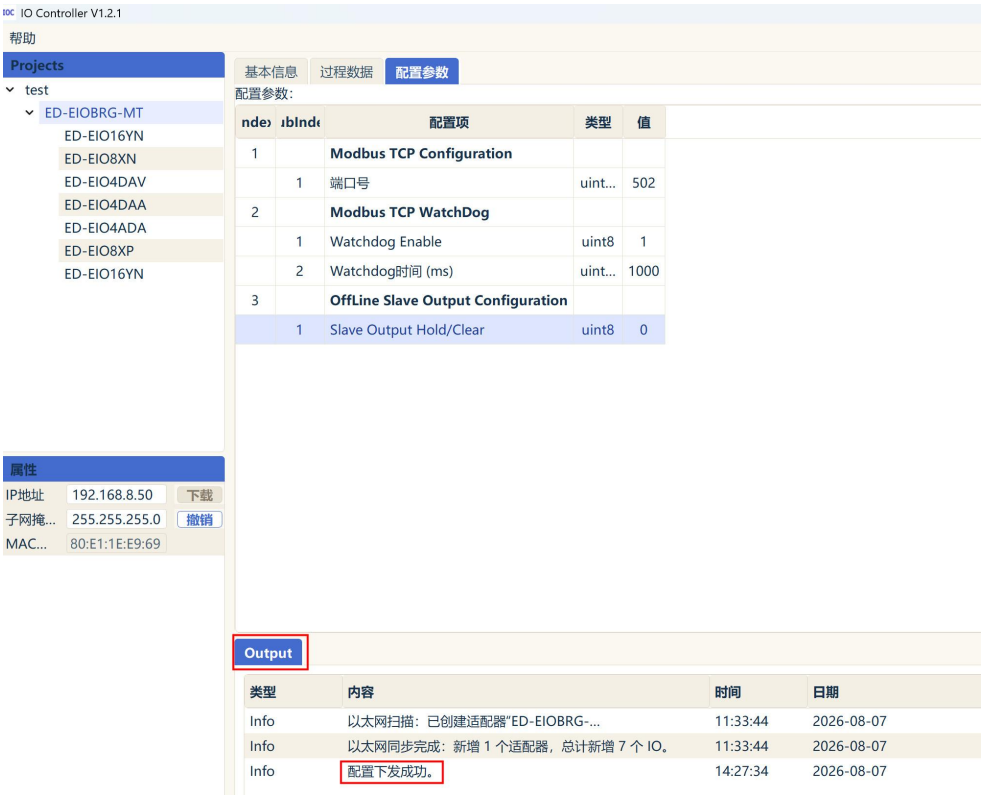
2. 鼠标左键单击“配置参数”表中的“值”列对应的电压范围，查看支持设置的范围列表，再根据实际的场景选择合适的电压范围。



> 提示
>
> - I/O模块的每一个通道均支持配置。
> - 此处仅以一个电压输出模块为例进行操作配置，其他模块的操作均类似。

3. 参数设置完成后，在界面左侧的test工程下，右键单击“ED-EIOBRG-MT”，再选择菜单中的“下载配置”。

4. 下载完成后，界面右下方的“Output”区域会显示“配置下发成功”的信息。



2 I/O模块寄存器读写示例

在实际的Modbus TCP组网中，需通过软件读写ED-EIOBRG-MT上I/O模块的内部寄存器，以实现对其I/O通道的控制。下文将详细介绍该模块的I/O地址映射规则及寄存器读写示例。

2.1 I/O模块地址映射规则

DI和DO模块支持Bit映射和Word映射两种寄存器访问方式，AI和AO仅支持Word映射寄存器访问方式。

- Bit映射方式的说明

模块类型	功能码	偏移起始地址	位地址范围	数据长度范围	偏移地址+长度
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)

- Word映射方式的说明

模块类型	功能码	偏移起始地址	寄存器地址范围	数据长度范围	偏移地址+长度
DI (Input Word)	0x03	十六进制： 0x5000 十进制：20480	0x5000~0x507F 20480~20607	1~128	≤20608(R)
DO (Output Word)	0x16 0x03(R)	十六进制： 0x3000(W) 十进制： 12288(W) 十六进制： 0x4000(R) 十进制： 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)	十六进制： 0x00(W) 十进制：0(W) 十六进制： 0x2000(R)	0x00~0x1FF(W) 0~511(W) 0x2000~0x21FF(R) 8192~8703(R)	1~512	≤512(W) ≤8704(R)

模块类型	功能码	偏移起始地址	寄存器地址范围	数据长度范围	偏移地址+长度
		Decimal: 8192(R)			

> 提示
>
> - DI和AI模块仅支持读取功能。
> - DO和AO支持读取和写入功能。

2.2 I/O模块读写寄存器的示例代码

示例代码包含Python和C++两种语言的，下文分别进行介绍。

2.2.1 Python示例代码

介绍DI、DO、AI和AO模块的Python示例代码。

2.2.1.1 DI模块（Bit映射方式）

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模块（Word映射方式）

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模块（Bit映射方式）

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模块（Word映射方式）

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 = 16384           # Starting offset address of DO (decimal)
DO_COUNT = 1                 # Number of DO registers to be read

# 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 read DO by word
            ret = read_modbus_do_word(client, UNIT_ID, DO_ADDRESS, DO_COUNT)
            if ret:
               print("DO Module Values: ",ret)       # Convert output value to binary for corr
            time.sleep(1)  # Refresh every 1 second
    except KeyboardInterrupt:
        print("Program terminated")
    finally:
        # Close connection when program exits
        client.close()
        print("Modbus connection closed")
```

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 = 12288           # 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

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, DO_DATA)
            if ret:
               print("DO write operation by word completed")

            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模块（Word映射方式）

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模块（Word映射方式）

```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++示例代码

介绍DI、DO、AI和AO模块的C++示例代码。

2.2.2.1 DI模块（Bit映射方式）

```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模块（Word映射方式）

```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模块（Bit映射方式）

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模块（Word映射方式）

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 DO_ADDRESS = 16384;               // DO start offset address (decimal)
```

```
static const int DO_COUNT = 1;                    // Number of DO 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 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;

    // DO register buffer
    uint16_t do_regs[DO_COUNT];
    while (g_running) {
        if (read_modbus_do_reg(ctx, UNIT_ID, DO_ADDRESS, 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;
        }
        // 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;
}
```

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 = 12288;                                     // DO starting offset
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;
}

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)
    {
```

```
        if (write_modbus_do_word_batch(ctx, UNIT_ID, DO_ADDRESS, DO_WRITE_COUNT, DO_DATA))
        {
            cout << "Word-based DO write completed" << 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模块（Word映射方式）

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模块（Word映射方式）

```
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;
}
```