# Modbus TCP/IP Protocol Specification for PowerGo System V1.0

## Index



## 1. Overview

This document defines the Modbus TCP/IP protocol used for communication between a HEMS (home Energy Management System) and the PowerGo Series products.

- IP Adress: 192.168.1.200
- Port: 5002
- Slave ID: 81
- Byte Order: Big-Endian

## 2. Function Codes

| Function Code | Description |
|---|---|
| 0x03 | Read Holding Registers |
| 0x04 | Read Input Registers |
| 0x06 | Write Single Register |
| 0x10 | Write Multiple Registers |

## 3. Register Mapping

| Name | Add DEC | Register Type | Data Type | Len | R/W | Scaling Factor | Unit | Format | Initial Value | Remarks |
|---|---|---|---|---|---|---|---|---|---|---|
| **Power Grid Information** | | | | | | | | | | |
| Grid-Side Voltage | 573 | Holding_Register | UINT16 | 1 | R | 0.1 | V | %.1V | - | The device samples the current grid voltage from the power grid. |
| Grid-Side Power | 530 | Holding_Register | INT16 | 1 | R | 1 | W | %dW | - | AC input/output port power, where:<br>• Positive values indicate power flow from grid to device<br>• Negative values indicate power flow from device to grid |

| **PowerGo Information** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| IP Adress | 220 | Holding_Register | UINT32 | 2 | R/W | - | - | %d | | PowerGo's static IP defaults to 192.168.1.200. It is a 32-bit unsigned integer formatted in byte order: big-endian, with word order: little-endian.<br><br>Note: After modifying the network configuration parameters, you must write the value 9029 to Holding Register 265 to activate the changes. |
| Ethernet Port | 231 | Holding_Register | UINT16 | 1 | R/W | - | - | %d | | Default port: 5002<br>Note: After modifying network configuration parameters, you must write the value 9029 to Holding Register 265 to activate the changes. |
| Serial Number | 12 | Holding_Register | UINT32 | 2 | R | - | - | %d | - | Product serial identifier (Byte order: big-endian, Word order: little-endian) |
| Main Software Version | 1 | Holding_Register | UINT16 | 1 | R | - | - | %d | 40985 | Main Software Version Number |
| Device Current Operating Status | 306 | Holding_Register | UINT16 | 1 | R | - | - | %d | 0 | Charge/Discharge Status Flag (1-byte):<br>• Bit 0: Charge enable<br>- 0: Charging allowed<br>- 1: Charging prohibited<br>• Bit 1: Discharge enable<br>- 0: Discharging allowed<br>- 1: Discharging prohibited |
| Device Current Operating Time | 548 | Holding_Register | UINT16 | 1 | RW | 1 | - | %d | - | Year(from 2000): High 8, Month: Low 8 |
| | 549 | Holding_Register | UINT16 | 1 | RW | 1 | - | %d | - | Day: High 8, Hour: Low 8 |
| | 550 | Holding_Register | UINT16 | 1 | RW | 1 | - | %d | - | Hour: High 8, Second Low 8 |
| State of Charge (SOC) | 529 | Holding_Register | UINT16 | 1 | R | 1 | - | %d | - | Remaining Battery Capacity |
| Fault Information | 508 | Holding_Register | UINT16 | 1 | R | - | - | %d | 0 | For details and examples, refer to the 'Fault Information Table'. |
| | 509 | Holding_Register | UINT16 | 1 | R | - | - | %d | 0 | |
| | 510 | Holding_Register | UINT16 | 1 | R | - | - | %d | 0 | |
| | 521 | Holding_Register | UINT16 | 1 | R | - | - | %d | 0 | |
| | 522 | Holding_Register | UINT16 | 1 | R | - | - | %d | 0 | |
| | 523 | Holding_Register | UINT16 | 1 | R | - | - | %d | 0 | |
| | 524 | Holding_Register | UINT16 | 1 | R | - | - | %d | 0 | |
| | 525 | Holding_Register | UINT16 | 1 | R | - | - | %d | 0 | |
| | 526 | Holding_Register | UINT16 | 1 | R | - | - | %d | 0 | |
| Daily Cumulative Discharge Energy | 540 | Holding_Register | UINT16 | 1 | R | 0.1 | kWh | %.1fkWh | - | Daily Power Generation |
| 7-Days Cumulative Discharge Energy | 533 | Holding_Register | UINT16 | 1 | R | 0.1 | kWh | %.1fkWh | - | Registers 533-539 store the device's energy generation data for the most recent 1-7 days respectively. |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 534 | Holding_Register | UINT16 | 1 | R | 0.1 | kWh | %.1fkWh | - | |
| | 535 | Holding_Register | UINT16 | 1 | R | 0.1 | kWh | %.1fkWh | - | |
| | 536 | Holding_Register | UINT16 | 1 | R | 0.1 | kWh | %.1fkWh | - | |
| | 537 | Holding_Register | UINT16 | 1 | R | 0.1 | kWh | %.1fkWh | - | |
| | 538 | Holding_Register | UINT16 | 1 | R | 0.1 | kWh | %.1fkWh | - | |
| | 539 | Holding_Register | UINT16 | 1 | R | 0.1 | kWh | %.1fkWh | - | |
| Total Discharge Energy | 541 | Holding_Register | UINT16 | 1 | R | 0.1 | kWh | %.1fkWh | - | Cumulative Total Power Generation of the Equipment (Low Register) |
| | 542 | Holding_Register | UINT16 | 1 | R | 0.1 | kWh | %.1fkWh | - | Cumulative Total Power Generation of the Equipment (Lo Register) |
| **Operation** | | | | | | | | | | |
| Charging Power | 280 | Holding_Register | UINT16 | 1 | RW | 1 | W | %dW | 0 | Maximum Charging Power<br>Note: Adjusting this value enables dynamic control of device charging power. |
| Charge Enable | 551 | Holding_Register | UINT16 | 1 | RW | 1 | - | %d | 0 | Charge Enable Flag:<br>0: Charging disabled<br>1: Charging enabled<br><br>Priority Rule:<br>Charging function has higher priority than discharging<br>When both charge and discharge are enabled (charge_en=1 AND discharge_en=1), the system will always execute charging |
| Period 1 Charging Start Time | 552 | Holding_Register | UINT16 | 1 | RW | 1 | h/m | %d | 0 | **Time Encoding Format (HH:MM):**<br>High Byte (Bits 15-8): Hour (0-23), Low Byte (Bits 7-0): Minute (0-59)<br>Example: 23:59 = 23 * 256 + 59 = 5947 (Dec) / 0x173B (Hex),<br>**Discharge Time Windows:**<br>Period 1: Start=HH1:MM1, End=HH2:MM2,<br>Period 2: Start=HH3:MM3, End=HH4:MM4<br>PowerGo System will compare current time with these windows to enable/disable discharge.,<br>**Default Charging Periods:**<br>Period 1: 00:00-23:59 (All-day charging allowed), Period 2: 00:00-00:00 (Disabled by default) |
| Period 1 Charging End Time | 553 | Holding_Register | UINT16 | 1 | RW | 1 | h/m | %d | 5947 | |
| Period 2 Charging Start Time | 554 | Holding_Register | UINT16 | 1 | RW | 1 | h/m | %d | 0 | |
| Period 2 Charging End Time | 555 | Holding_Register | UINT16 | 1 | RW | 1 | h/m | %d | 0 | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Discharging Power | 543 | Holding_Register | UINT16 | 1 | RW | 1 | W | %dW | 0 | Maximum Discharge Power<br>Note: Adjusting this value for dynamic control of device discharging power. |
| Discharge Enbale | 558 | Holding_Register | UINT16 | 1 | RW | 1 | - | %d | 0 | Discharge Enable Flag:<br>0：Discharging disabled<br>1：Discharging Enabled<br>Please note that the charge enable signal has higher priority over the discharge enable signal. When both charge enable and discharge enable are active, the system will initiate charging. To initiate discharging, set charge enable to 0 and discharge enable to 1. |
| Period 1 Discharging Start Time | 559 | Holding_Register | UINT16 | 1 | RW | 1 | h/m | %d | 0 | **Time is encoded in 16-bit format:**<br>High byte (Bits 15-8): Hour (0-23), Low byte (Bits 7-0): Minute (0-59)<br>Example: 23:59 = 23×256 + 59 = 5947 (0x173B in hex),<br>**Two configurable discharge periods:**<br>Period 1: Start=HH1:MM1, End=HH2:MM2,<br>Period 2: Start=HH3:MM3, End=HH4:MM4<br>The PowerGo system automatically enables discharge when current time falls within either active period.,<br>**Pre-set charging windows:**<br>Period 1: 00:00-23:59 (always enabled by default), Period 2: 00:00-00:00 (disabled by default, requires manual configuration) |
| Period 1 Discharging End Time | 560 | Holding_Register | UINT16 | 1 | RW | 1 | h/m | %d | 5947 | |
| Period 2 Discharging Start Time | 561 | Holding_Register | UINT16 | 1 | RW | 1 | h/m | %d | 0 | |
| Period 2 Discharging End Time | 562 | Holding_Register | UINT16 | 1 | RW | 1 | h/m | %d | 0 | |

## 4. Exception Codes

| Code | Meaning |
|---|---|
| 0x01 | Illegal Function |
| 0x02 | Illegal Data Address |
| 0x03 | Illegal Data Value |
| 0x04 | Slave Device Failure |

## 5. Fault Information

| **Bit0-15** | **0x508-Word0** | **0x509-Word1** | **0x510-Word2** |
|---|---|---|---|
| Bit0 | | | |
| Bit1 | | | |
| Bit2 | | | |

| | | | |
|---|---|---|---|
| Bit3 | | Low State of Charge(SOC) Protection | |
| Bit4 | | | Cell Failure Alarm |
| Bit5 | | | |
| Bit6 | | | |
| Bit7 | | | |
| Bit8 | | | |
| Bit9 | Cell Overvoltage Protection | | |
| Bit10 | | | |
| Bit11 | Cell Undervoltage Protection | | |
| Bit12 | | | |
| Bit13 | Total Overvoltage Protection | | |
| Bit14 | | | |
| Bit15 | Total Undervoltage Protection | | |

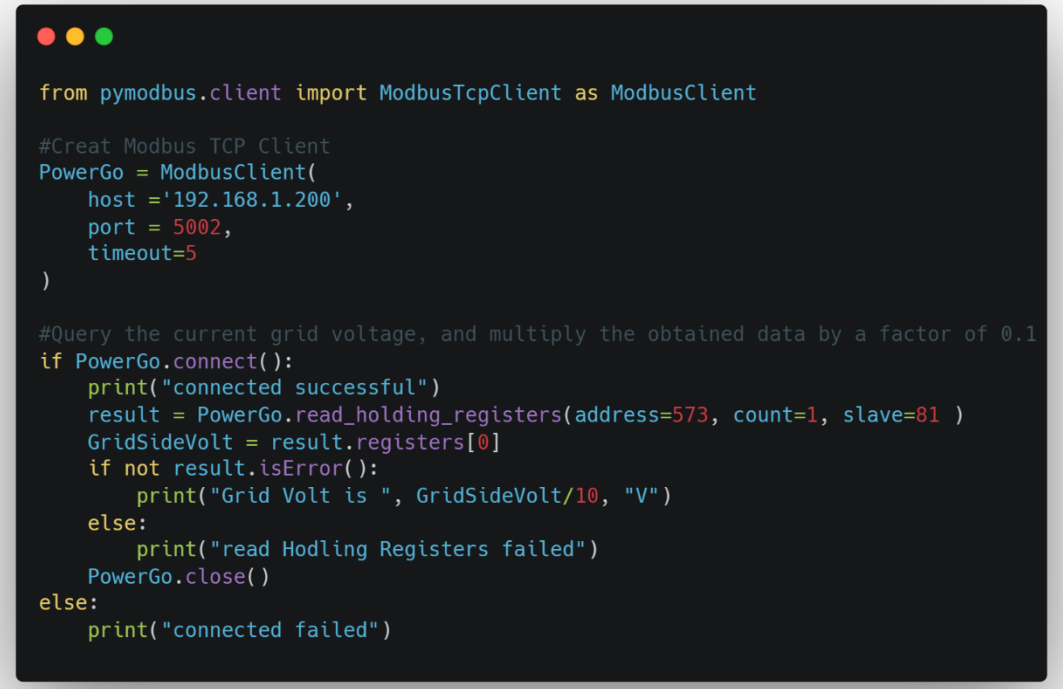
| **Bit0-15** | **0x521-Word3** | **0x522-Word4** | **0x523-Word5** |
|---|---|---|---|
| Bit0 | Battery Lowvoltage Alarm | Busbar Short Circuit | Control CAN Communication Fault |
| Bit1 | Battery EOD | PV Input Overvoltage | Communication CAN Bus Fault |
| Bit2 | Battery Current Software Overcurrent Alarm | PV Current Software Overcurrent | Parallel Mode Configuration Error |
| Bit3 | Battery Hardware Overcurrent Alarm | PV Hardware Overcurrent | Parallel Current Sharing Fault |
| Bit4 | Battery Open Circuit | PV Insulation Resistance Low | Parallel ID Conflict |
| Bit5 | Battery Overvoltage Alarm | PV Heat Sink Overtemperature | Parallel Battery Inconsistency Alarm |
| Bit6 | Low Battery Capacity Rate | Inverter Heat Sink Overtemperature | Parallel Grid Input Mismatch |
| Bit7 | Battery Low Capacity Shutdown | Transformer Overtemperature | Parallel Communication Signal Anomaly |
| Bit8 | Bypass Output Overload | Grid Input Relay Short Circuit | Parallel Firmware Incompatibility |
| Bit9 | Inverter Output Overload | Output Relay Short Circuit | Parallel Cable Connection Fault |
| Bit10 | Inverter AC Output Short Circuit | Fan Failure | Invalid Product Serial Number |
| Bit11 | Inverter Hardware Overcurrent Alarm | EEPROM Fault | System Low Battery Voltage Shutdown |
| Bit12 | Inverter DC Component High Alarm | SPI Communication Fault | Slave IC Abnormal Shutdown |
| Bit13 | Busbar Hardware Overvoltage | Model Configuration Error | DC Bus Voltage Imbalance |
| Bit14 | Busbar Software Overvoltage | Busbar Soft Start Failure | External CT Master Configuration Error |
| Bit15 | Busbar Undervoltage Alarm | Leakage Current Abnormal | Grid Input Phase Abnormality |

| **Bit0-15** | **0x524-Word6** | **0x525-Word7** | **0x526-Word8** |
|---|---|---|---|
| Bit0 | BMS Communication Fault | Grid Overvoltage | Reserve |
| Bit1 | BMS Abnormal Alarm | Grid Undervoltage | Reserve |
| Bit2 | BMS Battery Overtemperature | Grid Overfrequency | Reserve |
| Bit3 | BMS Battery Overcurrent | Grid Underfrequency | Reserve |
| Bit4 | BMS Battery Overvoltage | Grid 10-Minute Average Overvoltage | Reserve |
| Bit5 | BMS Battery Undervoltage | LVRT (Low Voltage Ride-Through) Fault | Reserve |
| Bit6 | BMS Battery Low Temperature | HVRT (High Voltage Ride-Through) Fault | Reserve |
| Bit7 | | System Grounding Fault | Reserve |
| Bit8 | | DC Arc Fault Detection Error | Reserve |

| | | | |
|---|---|---|---|
| Bit9 | | Anti-Islanding Protection | Reserve |
| Bit10 | | Reserve | Reserve |
| Bit11 | BMS System Fault | Reserve | Reserve |
| Bit12 | | Reserve | Reserve |
| Bit13 | | Reserve | Reserve |
| Bit14 | | Reserve | Reserve |
| Bit15 | | Reserve | Reserve |

## 6. Example

The following content uses Python as an example and requires the use of the pymodbus library.

| Operation | **Read Holding Registers 573-Grid-Side Voltage** | |
|---|---|---|
| Code: |  | |
| Console： |  | Read Holding Register 573 to obtain the raw value 2334.<br>· Scaling Factor: Multiply by 0.1 to derive the actual mains voltage:<br>2334 × 0.1 = 233.4V<br>· Data Source: Sampled and provided by PowerGo. |

```python
from pymodbus.client import ModbusTcpClient as ModbusClient

#Creat Modbus TCP Client
PowerGo = ModbusClient(
    host ='192.168.1.200',
    port = 5002,
    timeout=5
)

#Query the current grid voltage, and multiply the obtained data by a factor of 0.1
if PowerGo.connect():
    print("connected successful")
    result = PowerGo.read_holding_registers(address=573, count=1, slave=81 )
    GridSideVolt = result.registers[0]
    if not result.isError():
        print("Grid Volt is ", GridSideVolt/10, "V")
    else:
        print("read Hodling Registers failed")
    PowerGo.close()
else:
    print("connected failed")
```

```
connected successful
Grid Volt is  233.4 V
```

<table>
<tr><td>Operation</td><td colspan="2">Read Holding Registers 530-Gride Side Power</td></tr>
<tr><td>Code:</td><td colspan="2">

```python
from pymodbus.client import ModbusTcpClient as ModbusClient

#Creat Modbus TCP Client
PowerGo = ModbusClient(
    host ='192.168.1.200',
    port = 5002,
    timeout=5
)

#Query the current grid side Power
if PowerGo.connect():
    print("connected successful")
    result = PowerGo.read_holding_registers(address=530, count=1, slave=81 )
    GridSidePower = result.registers[0]
    if not result.isError():
        print("Grid Side Power is ", GridSidePower, "W")
    else:
        print("read Hodling Registers failed")
    PowerGo.close()
else:
    print("connected failed")
```

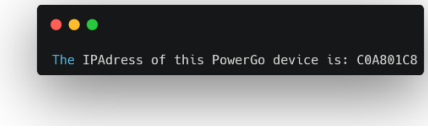
</td></tr>
<tr><td>Console ：</td><td>

```
connected successful
Grid Side Power is  593 W
```

</td><td>Read Holding Register 530 to obtain the power value 593.<br>· Positive Value (+593W):<br>Indicates PowerGo is charging at a rate of 593W (drawing power from the grid).<br>· Negative Value (-593W):<br>Indicates PowerGo is discharging to the grid at 593W (feeding power back).</td></tr>
</table>

| Operation | **Read /Write Holding Registers 220-IP Adress** |
|---|---|
| Code: | |

```python
from pymodbus.client import ModbusTcpClient as ModbusClient
from struct import pack, unpack

#Creat Modbus TCP Client
PowerGo = ModbusClient(
    host ='192.168.1.200',
    port = 5002,
    timeout=5
)

def read_two_registers_big_endian_bitwise(client, address, slave_id=81):
    #read two holding registers
    Response = client.read_holding_registers(address=address, count=2,
slave=slave_id)
    if Response.isError():
        raise Exception("read failed")

    #get two holding registers
    reg1, reg2 = Response.registers

    #Handling big-endian and little-endian byte orders:
    #1.Convert each register into a byte string in big-endian order.
    #Arrange th two words in little-endian word order.
    byte_data = pack('>HH', reg2, reg1)
    CombinedValue = unpack('>I', byte_data)[0]


    return CombinedValue
if PowerGo.connect():

    value = read_two_registers_big_endian_bitwise(client=PowerGo, address=220)
    #covert to Hex()
    HexValue = format(value, 'X')
    print("The IPAdress of this PowerGo device is:", HexValue)
    PowerGo.close()
else:
    print("connected failed")
```

| Console ： |  | Read Holding Register 220 to obtain PowerGo's IP address (default: 192.168.1.200).<br>Example: The returned value is C0 A8 01 C8, which converts to decimal as:<br>· C0 → 192<br>· A8 → 168<br>· 01 → 1<br>· C8 → 200<br>**Note: After modifying network settings, write the value 9029 to Register 265 to activate the updated network configuration parameters.** |
|---|---|---|

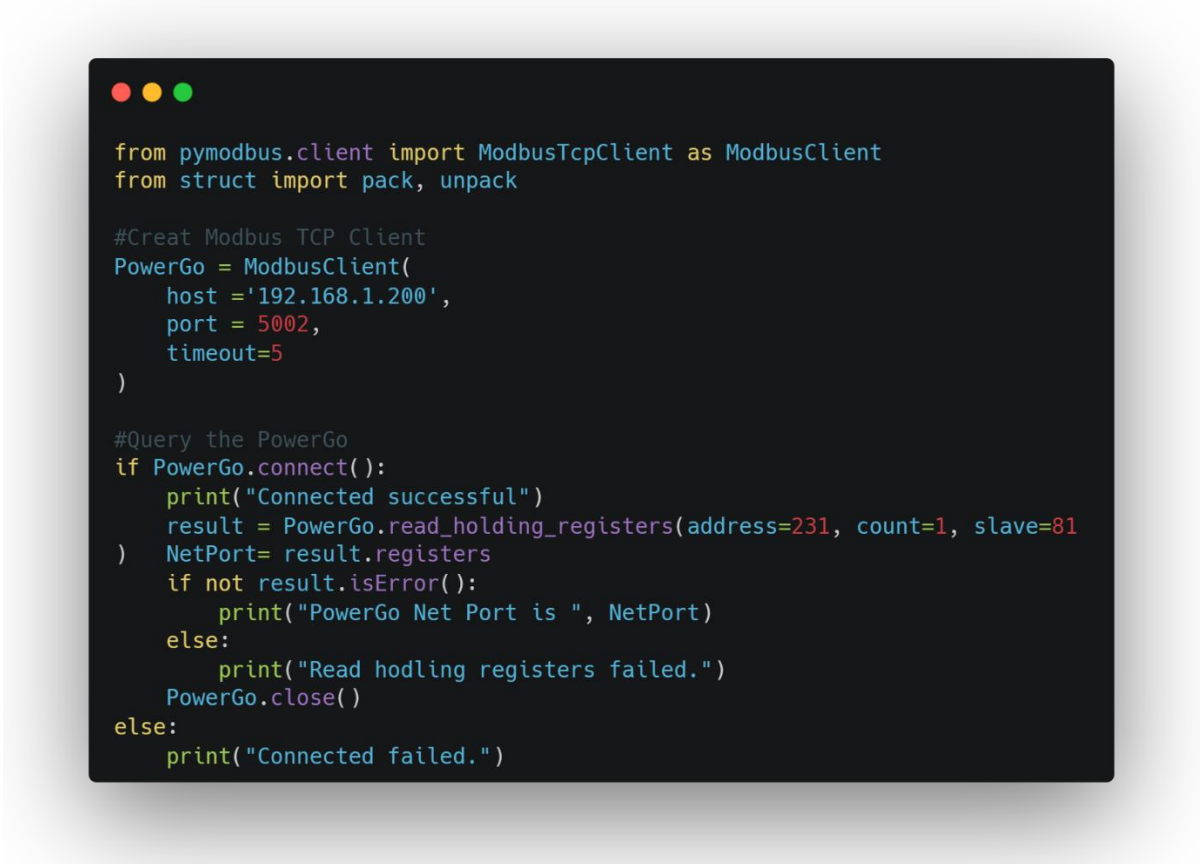
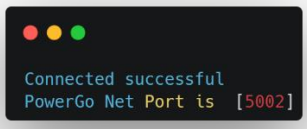
<table>
<tr><td>Operation</td><td colspan="2">Read /Write Holding Registers 231-Ethernet Port</td></tr>
<tr><td>Code:</td><td colspan="2">

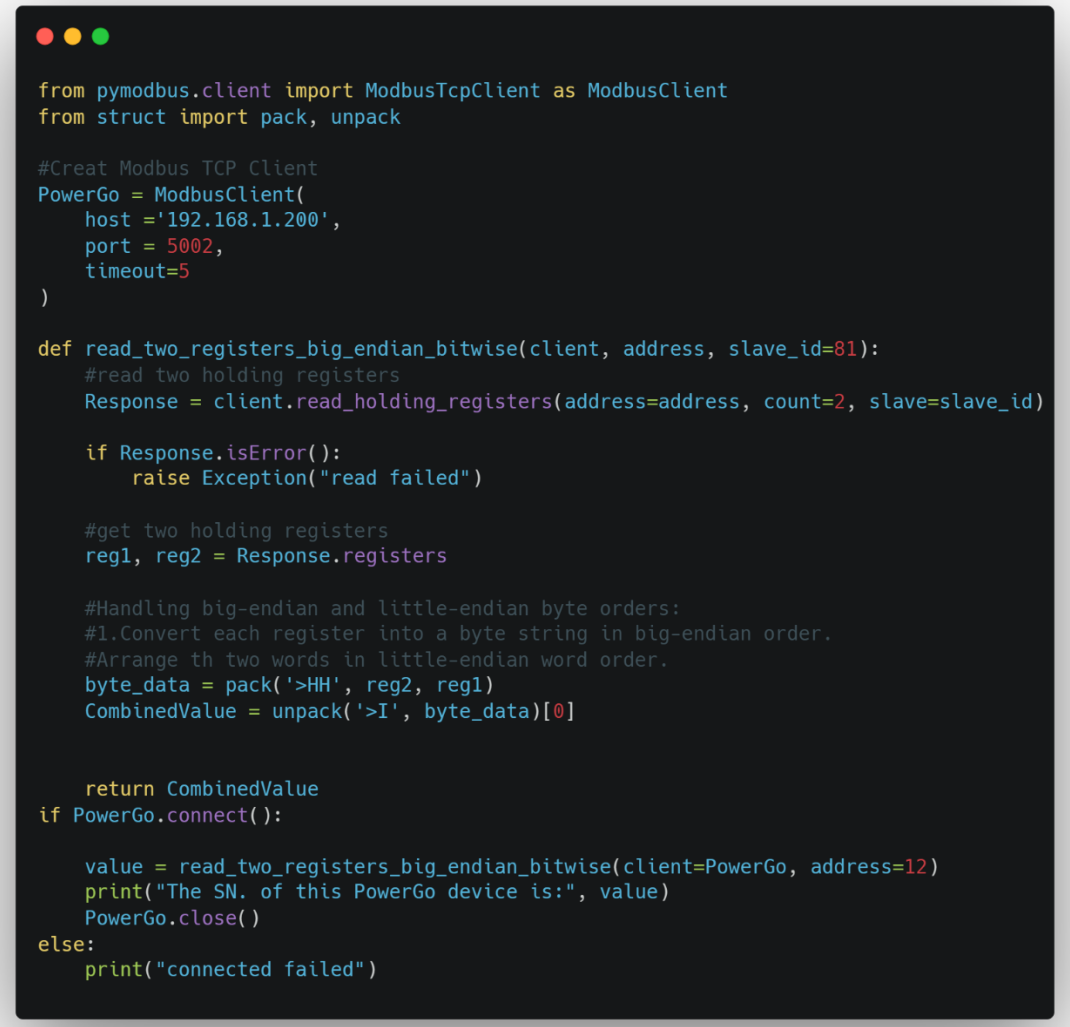
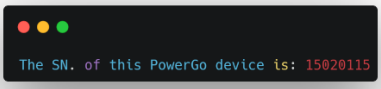
```python
#write holding registers
if PowerGo.connect():
    PowerGoAddress = 231
    PowerGoValue = 5002

    print("Connected successful.")
    ReadResult = PowerGo.read_holding_registers(address=PowerGoAddress, count=1, slave=81)
    PowerGoPort= ReadResult.registers[0]
    if not ReadResult.isError():
        print("PowerGo Ethernet Port is:",PowerGoPort)
    else:
        print("Read holding registers failed.")

    #write single holding register
    WriteResult = PowerGo.write_register(address=PowerGoAddress, value = PowerGoValue, slave=81)
    if not WriteResult.isError():
        print("Write single holding register successfully")
    else:
        print("Write single holding register failed.")
    #Write 9029 to register 265, then refresh NET parameters.
    Response=PowerGo.write_register(address=265, value=9029, slave=81)


    PowerGo.close()

else:
    print("Connected failed")
```

</td></tr>
<tr><td>Console ：</td><td>

```
Connected successful.
PowerGo Ethernet Port is: 5003
Write single holding register successfully
```

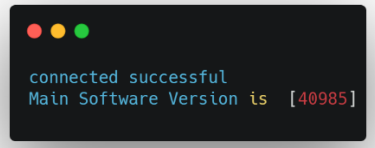
</td><td>Read/Write Holding Register 231 to configure PowerGo's Ethernet port number (default: 5002).<br>Example:<br>To change the port number from 5003 to 5002:<br>1. Write 5002 (0x138A in hex) to Register 231<br>2. Write 9029 to Register 265 to activate the updated network configuration.</td></tr>
</table>

| | | |
|---|---|---|
| Code: |  <br>```<br>from pymodbus.client import ModbusTcpClient as ModbusClient<br>from struct import pack, unpack<br><br>#Creat Modbus TCP Client<br>PowerGo = ModbusClient(<br>    host ='192.168.1.200',<br>    port = 5002,<br>    timeout=5<br>)<br><br>#Query the PowerGo<br>if PowerGo.connect():<br>    print("Connected successful")<br>    result = PowerGo.read_holding_registers(address=231, count=1, slave=81<br>)   NetPort= result.registers<br>    if not result.isError():<br>        print("PowerGo Net Port is ", NetPort)<br>    else:<br>        print("Read hodling registers failed.")<br>    PowerGo.close()<br>else:<br>    print("Connected failed.")<br>``` | |
| Console： |  <br>```<br>Connected successful<br>PowerGo Net Port is  [5002]<br>``` | Read Holding Register 231 again to verify the new port number (updated value: 5002). |

| Operation | **Read Holding Registers 12-Serial Number** | |
|---|---|---|
| Code: |  | |
| Console ： |  | Read Holding Register 12 to obtain the unique serial number (SN) of this PowerGo device.<br>Data Specifications:<br>· **Data Type: uint32**<br>· **Byte Order: Big-endian**<br>· **Word Order: Little-endian** |

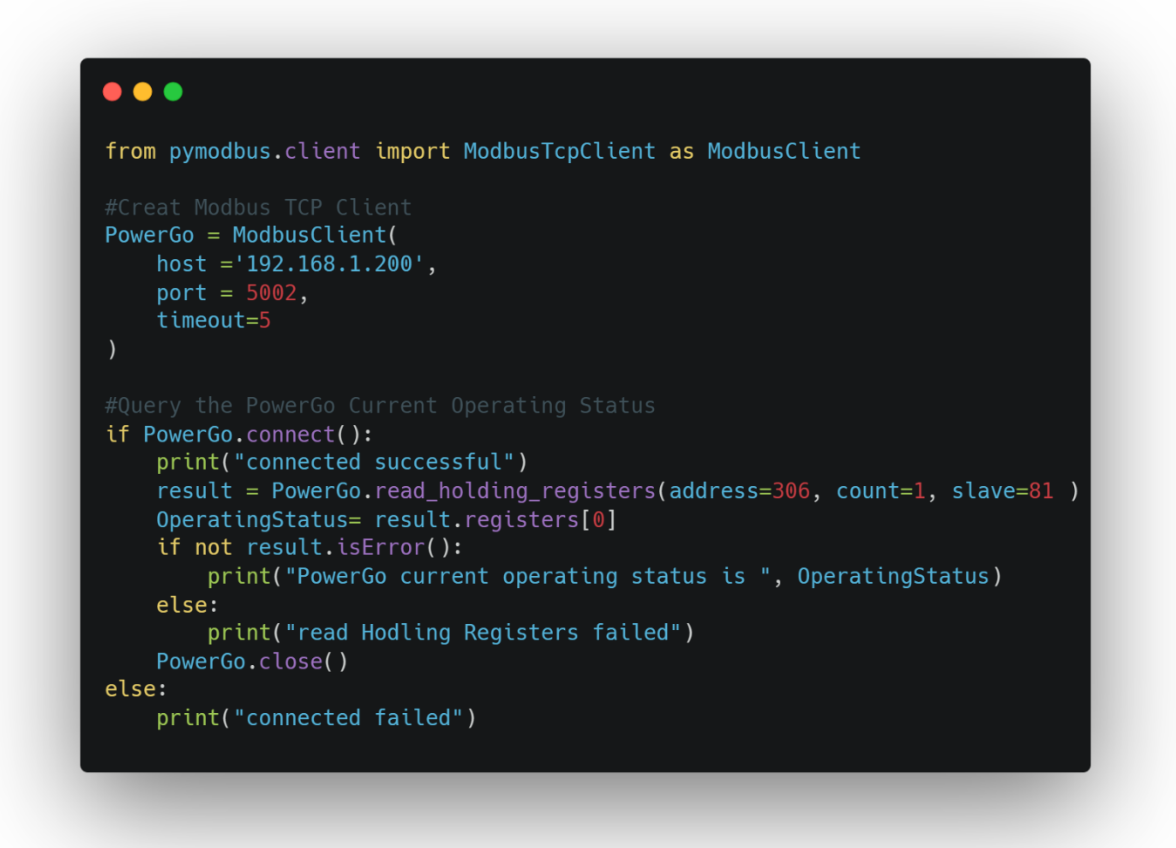
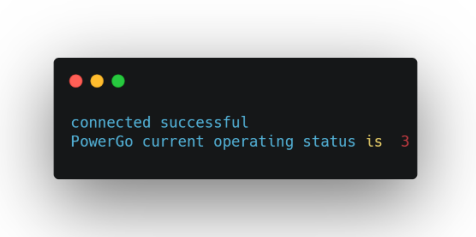
```python
from pymodbus.client import ModbusTcpClient as ModbusClient
from struct import pack, unpack

#Creat Modbus TCP Client
PowerGo = ModbusClient(
    host ='192.168.1.200',
    port = 5002,
    timeout=5
)

def read_two_registers_big_endian_bitwise(client, address, slave_id=81):
    #read two holding registers
    Response = client.read_holding_registers(address=address, count=2, slave=slave_id)

    if Response.isError():
        raise Exception("read failed")

    #get two holding registers
    reg1, reg2 = Response.registers

    #Handling big-endian and little-endian byte orders:
    #1.Convert each register into a byte string in big-endian order.
    #Arrange th two words in little-endian word order.
    byte_data = pack('>HH', reg2, reg1)
    CombinedValue = unpack('>I', byte_data)[0]


    return CombinedValue
if PowerGo.connect():

    value = read_two_registers_big_endian_bitwise(client=PowerGo, address=12)
    print("The SN. of this PowerGo device is:", value)
    PowerGo.close()
else:
    print("connected failed")
```

```
The SN. of this PowerGo device is: 15020115
```

| Operation | **Read Holding Registers 1-Main Software Version** | |
|---|---|---|
| Code: |  | |
| Console ： |  | Holding Register 1 stores the main firmware version (for technical support purposes). |

```python
from pymodbus.client import ModbusTcpClient as ModbusClient

#Creat Modbus TCP Client
PowerGo = ModbusClient(
    host ='192.168.1.200',
    port = 5002,
    timeout=5
)

#Query the PowerGo Main Software Version
if PowerGo.connect():
    print("connected successful")
    result = PowerGo.read_holding_registers(address=1, count=1, slave=81 )
    MSoftwareVersion= result.registers
    if not result.isError():
        print("Main Software Version is ", MSoftwareVersion)
    else:
        print("read Hodling Registers failed")
    PowerGo.close()
else:
    print("connected failed")
```

```
connected successful
Main Software Version is  [40985]
```

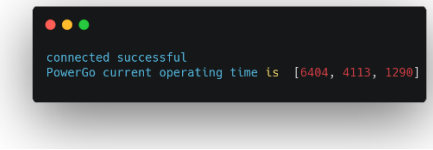
| Operation | **Read Holding Registers 306-Device Current Operating Status** | |
|---|---|---|
| Code: |  | |
| Console ： |  | Read Holding Register 306 to obtain the device's current operational state (for HEMS command reference).<br>Status Codes and Meanings:<br>0: Charge/Discharge Allowed - Normal operation with both charging and discharging permitted.<br>1: Discharge-Only Mode - Charging prohibited while discharging remains allowed. (Typical scenarios: Battery at full capacity)<br>2: Charge-Only Mode - Discharging prohibited while charging remains allowed. (Common situations:Low battery state )<br>3: Charge/Discharge Prohibited - Complete operational lockdown indicating: |

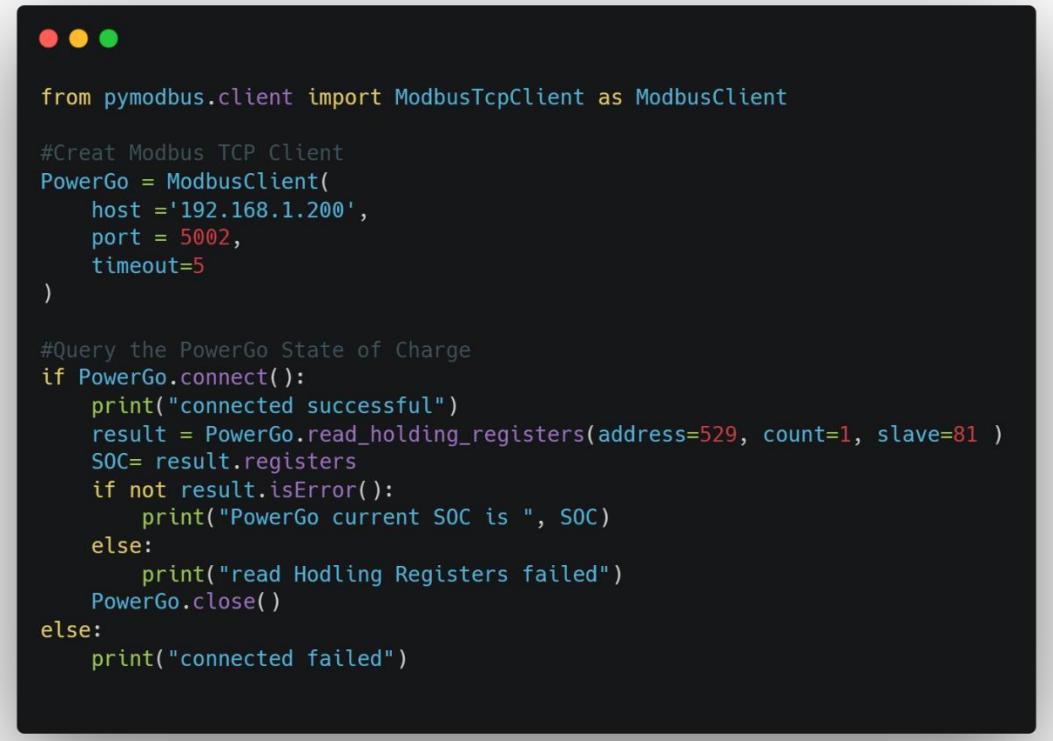
```
from pymodbus.client import ModbusTcpClient as ModbusClient

#Creat Modbus TCP Client
PowerGo = ModbusClient(
    host ='192.168.1.200',
    port = 5002,
    timeout=5
)

#Query the PowerGo Current Operating Status
if PowerGo.connect():
    print("connected successful")
    result = PowerGo.read_holding_registers(address=306, count=1, slave=81 )
    OperatingStatus= result.registers[0]
    if not result.isError():
        print("PowerGo current operating status is ", OperatingStatus)
    else:
        print("read Hodling Registers failed")
    PowerGo.close()
else:
    print("connected failed")
```

```
connected successful
PowerGo current operating status is  3
```

| Operation | **Read /Write Holding Registers 548-550-Device Current Operating Time** |
|---|---|
| Code: | |

```python
from pymodbus.client import ModbusTcpClient as ModbusClient

#Creat Modbus TCP Client
PowerGo = ModbusClient(
    host ='192.168.1.200',
    port = 5002,
    timeout=5
)

#Query the PowerGo Current Operating Time
if PowerGo.connect():
    print("connected successful")
    result = PowerGo.read_holding_registers(address=548, count=3, slave=81 )
    CurrentTime= result.registers
    if not result.isError():
        print("PowerGo current operating time is ", CurrentTime)
    else:
        print("read Hodling Registers failed")
    PowerGo.close()
else:
```

| | | |
|---|---|---|
| Console ： |  | Holding Register 548-550 records the device's current time.<br>Since the device cannot update the time automatically, it is recommended that HEMS periodically refresh the device time via Register 306.<br>Time Data Decoding Method<br>(Example data: 6404, 4133, 1290):<br>1. Year & Month Decoding (Register value 6404):<br>· Calculation: 6404 ÷ 256 = 25 ..... 4<br>· Result: April 2025<br>2. Day & Hour Decoding (Register value 4133):<br>· Calculation: 4133 ÷ 256 = 16 ..... 17<br>· Result: 17:00 hour<br>3. Minute & Second Decoding (Register value 1290):<br>· Calculation: 1290 ÷ 256 = 5 ..... 10<br>· Result: 05 minutes 10 seconds<br>Final Decoded Time: 17:05:10, April 16, 2025 |

| Operation | **Read Holding Registers 529-State of Charge(SOC)** | |
|---|---|---|
| Code: |  | |
| Console ： |  | Read Holding Register 529 to obtain the device's current State of Charge (SOC).<br>Example:<br>Register value: 94 → Current SOC = 94% |

```python
from pymodbus.client import ModbusTcpClient as ModbusClient

#Creat Modbus TCP Client
PowerGo = ModbusClient(
    host ='192.168.1.200',
    port = 5002,
    timeout=5
)

#Query the PowerGo State of Charge
if PowerGo.connect():
    print("connected successful")
    result = PowerGo.read_holding_registers(address=529, count=1, slave=81 )
    SOC= result.registers
    if not result.isError():
        print("PowerGo current SOC is ", SOC)
    else:
        print("read Hodling Registers failed")
    PowerGo.close()
else:
    print("connected failed")
```

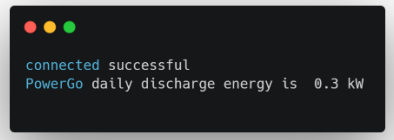
```
connected successful
PowerGo current SOC is  [94]
```

| Operation | **Read Holding Registers 540-Holding_Register** |
|---|---|

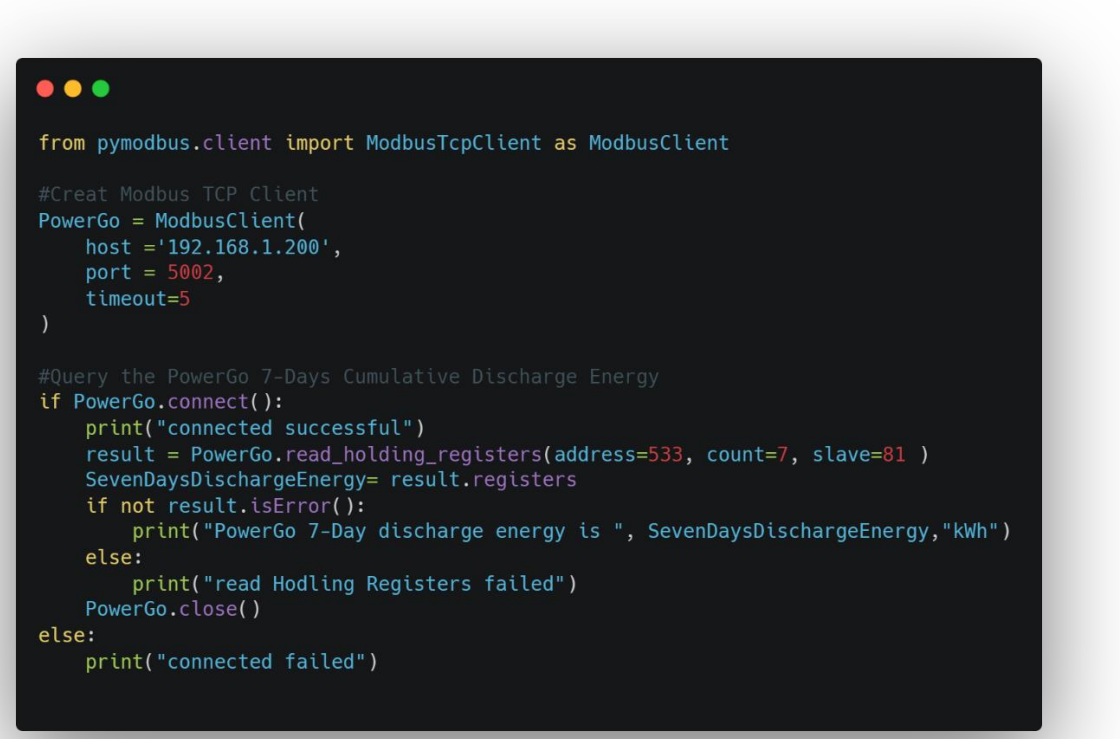
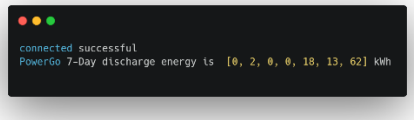
Code:

```python
from pymodbus.client import ModbusTcpClient as ModbusClient

#Creat Modbus TCP Client
PowerGo = ModbusClient(
    host ='192.168.1.200',
    port = 5002,
    timeout=5
)

#Query the PowerGo Daily Cumulative Discharge Energy
if PowerGo.connect():
    print("connected successful")
    result = PowerGo.read_holding_registers(address=540, count=1, slave=81 )
    DailyDischargeEnergy= result.registers[0]
    if not result.isError():
        print("PowerGo daily discharge energy is ", DailyDischargeEnergy/10,"kWh")
    else:
        print("read Hodling Registers failed")
    PowerGo.close()
else:
    print("connected failed")
```

Console ：




Read Holding Register 540 to obtain the device's daily discharge energy data.
Raw Value: Integer read from the register (e.g., 3)
Scaling Factor: Multiply by 0.1 → 0.3 kWh

| Operation | **Read Holding Registers 533-539- 7 Days Cumulative Discharge Energy** | |
|---|---|---|
| Code: |  | |
| Console ： |  | To read holding register 533 (length 7) and obtain the device's 7-day discharge energy data<br>Example Data Decoding (Raw values: [0, 2, 0, 0, 10, 13, 62]):<br>Assuming today is Monday, the discharge energy is:<br>· Sunday (latest): 0 × 0.1 = 0.0 kWh**<br>· Saturday: 2 × 0.1 = 0.2 kWh**<br>· Friday: 0 × 0.1 = 0.0 kWh**<br>· Thursday: 0 × 0.1 = 0.0 kWh**<br>· Wednesday: 10 × 0.1 = 1.0 kWh**<br>· Tuesday: 13 × 0.1 = 1.3 kWh**<br>· Monday (oldest): 62 × 0.1 = 6.2 kWh** |

```python
from pymodbus.client import ModbusTcpClient as ModbusClient

#Creat Modbus TCP Client
PowerGo = ModbusClient(
    host ='192.168.1.200',
    port = 5002,
    timeout=5
)

#Query the PowerGo 7-Days Cumulative Discharge Energy
if PowerGo.connect():
    print("connected successful")
    result = PowerGo.read_holding_registers(address=533, count=7, slave=81 )
    SevenDaysDischargeEnergy= result.registers
    if not result.isError():
        print("PowerGo 7-Day discharge energy is ", SevenDaysDischargeEnergy,"kWh")
    else:
        print("read Hodling Registers failed")
    PowerGo.close()
else:
    print("connected failed")
```

```
connected successful
PowerGo 7-Day discharge energy is  [0, 2, 0, 0, 18, 13, 62] kWh
```

<table>
<tr><td>Operation</td><td colspan="2">Read Holding Registers 541-542-Total Discharge Energy</td></tr>
<tr><td>Code:</td><td colspan="2">

```python
from pymodbus.client import ModbusTcpClient as ModbusClient

#Creat Modbus TCP Client
PowerGo = ModbusClient(
    host ='192.168.1.200',
    port = 5002,
    timeout=5
)

#Query the PowerGo Total Discharge Energy
if PowerGo.connect():
    print("connected successful")
    result = PowerGo.read_holding_registers(address=541, count=2, slave=81 )
    TotalDischargeEnergy= result.registers
    if not result.isError():
        print("PowerGo total discharge energy is ", TotalDischargeEnergy,"kWh")
    else:
        print("read Hodling Registers failed")
    PowerGo.close()
else:
    print("connected failed")
```

</td></tr>
<tr><td>Console ：</td><td>

```
connected successful
PowerGo total discharge energy is  [1029, 0] kWh
```

</td><td>Read Holding Register 541 to obtain PowerGo's cumulative discharge energy. The value must be multiplied by a 0.1 scaling factor, and the unit is kWh. This value cannot be modified.<br>Example: The current cumulative discharge energy of the device is 102.9 kWh.<br>Register 542 stores the high-order value of the total cumulative power generation.</td></tr>
</table>

| Operation | **Read /Write Holding Registers 280-Charging Power** |
|---|---|
| Code: | |

```python
#write holding registers
if PowerGo.connect():
    PowerGoAddress = 280
    PowerGoValue = 1200
    print("Connected successful.")
    ReadResult = PowerGo.read_holding_registers(address=PowerGoAddress, count=1, slave=81)
    ChargingPower = ReadResult.registers[0]
    if not ReadResult.isError():
        print("PowerGo charging power is:",ChargingPower)
    else:
        print("Read holding registers failed.")

    #write single holding register
    WriteResult = PowerGo.write_register(address=PowerGoAddress, value = PowerGoValue, slave=81)
    if not WriteResult.isError():
        print("Write single holding register successfully")
    else:
        print("Write single holding register failed.")

    #read again.
    ReadResult = PowerGo.read_holding_registers(address=PowerGoAddress, count=1, slave=81)
    ChargingPower = ReadResult.registers[0]
    if not ReadResult.isError():
        print("The updated status of PowerGo is:", ChargingPower)
    else:

        print("Update failed.")
    PowerGo.close()

else:
    print("Connected failed")
```

| Console： | | Read or write to Holding Register 280 to control the charging power of PowerGo.<br>Example: Modify the charging power of PowerGo from 800W to 1200W.<br>**Note: HEMS can achieve real-time charge/discharge control by continuously adjusting the parameter in Register 280, but the maximum charging power should not exceed 1200W.** |
|---|---|---|

```
Connected successful.
PowerGo charging power is: 800
Write single holding register successfully
The updated status of PowerGo is: 1200
```

<table>
<tr><td>Operation</td><td colspan="2">Read /Write Holding Registers 551-Charge Enable</td></tr>
<tr><td>Code:</td><td colspan="2">

```python
#write holding registers
if PowerGo.connect():
    PowerGoAddress = 551
    PowerGoValue = 0
    print("Connected successful.")
    ReadResult = PowerGo.read_holding_registers(address=PowerGoAddress, count=1, slave=81)
    ChargingEnable = ReadResult.registers[0]
    if not ReadResult.isError():
        print("PowerGo charging enable status is:",ChargingEnable)
    else:
        print("Read holding registers failed.")

    #write single holding register
    WriteResult = PowerGo.write_register(address=PowerGoAddress, value = PowerGoValue, slave=81)
    if not WriteResult.isError():
        print("Write single holding register successfully")
    else:
        print("Write single holding register failed.")

    #read again.
    ReadResult = PowerGo.read_holding_registers(address=PowerGoAddress, count=1, slave=81)
    ChargingEnable = ReadResult.registers[0]
    if not ReadResult.isError():
        print("The updated status of PowerGo is:", ChargingEnable)
    else:

        print("Update failed.")
    PowerGo.close()

else:
    print("Connected failed")
```

</td></tr>
<tr><td>Console ：</td><td>

```
Connected successful.
PowerGo charging enable status is: 1
Write single holding register successfully
The updated status of PowerGo is: 0
```

</td><td>Read or write Holding Register 551 to control PowerGo charging:<br>· 0: Charging disabled<br>· 1: Charging enabled<br>Example: This demonstrates disabling PowerGo's charging function. If Register 551 is changed from 0 to 1, PowerGo will charge at the power set in Holding Register 280.<br>Note:<br>· Holding Register 551 has higher priority than Holding Register 558.<br>For charging operation, set:<br>· 551 = 1 (enable charging)<br>· 558 = 0 (disable discharging)<br>For discharging operation, set:<br>· 551 = 0 (disable charging)<br>· 558 = 1 (enable discharging)</td></tr>
</table>

<table>
<tr><td>Operation</td><td colspan="2">Read /Write Holding Registers 553-Period1 Charging End Time</td></tr>
<tr><td>Code:</td><td colspan="2">

```python
#write holding registers
if PowerGo.connect():
    PowerGoAddress = 553
    PowerGoValue = 5918
    print("Connected successful.")
    ReadResult = PowerGo.read_holding_registers(address=PowerGoAddress, count=1, slave=81)
    ChargingEndTime1 = ReadResult.registers[0]
    if not ReadResult.isError():
        print("PowerGo period 1 charging End time is:",ChargingEndTime1)
    else:
        print("Read holding registers failed.")

    #write single holding register
    WriteResult = PowerGo.write_register(address=PowerGoAddress, value = PowerGoValue, slave=81)
    if not WriteResult.isError():
        print("Write single holding register successfully")
    else:
        print("Write single holding register failed.")

    #read again.
    ReadResult = PowerGo.read_holding_registers(address=PowerGoAddress, count=1, slave=81)
    ChargingEndTime1 = ReadResult.registers[0]
    if not ReadResult.isError():
        print("The updated status of PowerGo is:", ChargingEndTime1)
    else:
        print("Update failed.")
    PowerGo.close()

else:
    print("Connected failed")
```

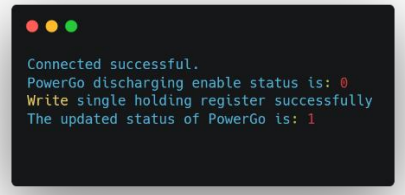
</td></tr>
<tr><td>Console ：</td><td>

```
Connected successful.
PowerGo period 1 charging End time is: 0
Write single holding register successfully
The updated status of PowerGo is: 5918
```

</td><td>
Holding Registers 552-555 & 559-562 Function Description:<br>
PowerGo supports configuring 2 separate charge/discharge time periods per day. When charge/discharge enable is activated, PowerGo will execute operations according to the configured time windows.<br>
Configuration Recommendation:<br>
For real-time power adjustment:<br>
1. Set both Phase 1 charge/discharge periods to 00:00-23:59<br>
2. Dynamically modify power values in:<br>
· Register 280 (charging power)<br>
· Register 543 (discharging power)<br>
Time Decoding Example:<br>
Register 553 value: 5918<br>
Calculation:<br>
5918 ÷ 256 = 23 (hour) with remainder 30 (minutes)<br>
→ Phase 1 charge end time = 23:30
</td></tr>
</table>

| Operation | **Read /Write Holding Registers 558-Discharge Enable** |
|---|---|
| Code: | |

```python
#write holding registers
if PowerGo.connect():
    PowerGoAddress = 558
    PowerGoValue = 1
    print("Connected successful.")
    ReadResult = PowerGo.read_holding_registers(address=PowerGoAddress, count=1, slave=81)
    DisChargingEnable = ReadResult.registers[0]
    if not ReadResult.isError():
        print("PowerGo discharging enable status is:",DisChargingEnable)
    else:
        print("Read holding registers failed.")

    #write single holding register
    WriteResult = PowerGo.write_register(address=PowerGoAddress, value = PowerGoValue, slave=81)
    if not WriteResult.isError():
        print("Write single holding register successfully")
    else:
        print("Write single holding register failed.")

    #read again.
    ReadResult = PowerGo.read_holding_registers(address=PowerGoAddress, count=1, slave=81)
    DisChargingEnable = ReadResult.registers[0]
    if not ReadResult.isError():
        print("The updated status of PowerGo is:", DisChargingEnable)
    else:

        print("Update failed.")
    PowerGo.close()

else:
    print("Connected failed")
```

| | | |
|---|---|---|
| Console： |  | Read/Write Holding Register 558 to Control PowerGo Discharge Function:<br>· 0: Discharge disabled<br>· 1: Discharge enabled<br>Example:<br>When enabled (set to 1), PowerGo will discharge at the power level configured in Holding Register 543.<br><br>**Important Notes:**<br>1. Priority Logic: Holding Register 551 (Charge Control) has higher priority than Register 558<br>2. Operation Mode Configuration:<br>▪ **For charging mode:**<br>▪ Set Register 551 = 1 (enable charge)<br>▪ Set Register 558 = 0 (disable discharge)<br>▪ **For discharging mode:**<br>▪ Set Register 551 = 0 (disable charge)<br>▪ Set Register 558 = 1 (enable discharge) |