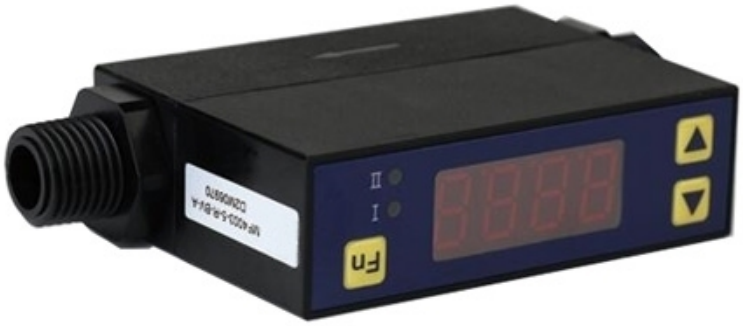
ATO

# Small Gas Mass Flow Meter

# Modbus Communication Protocol




（VerA.2）

ATO

# Table of Contents

ATO

# 1 Scope of Application

This agreement applies to MF4003/MF4008 gas mass flow meters equipped with an RS-485 communication interface and running software version V4.2 or later.

# 2 **Modbus** Protocols

## 2.1 Overview of the Protocol

This protocol defines the format of communication messages between the Modbus bus master and slave devices. For the master device, the Modbus protocol serves as an interface to connect with host computers (such as PCs, PLCs, HMIs, etc.), and all communication is “transparent.” Controller communication uses master-slave technology, meaning that only one device (the master) can initiate a transmission (query), while other devices (slaves) respond based on the data provided by the master’s query.

The master device can communicate with a single slave device or broadcast a message to all slave devices (limited to point-to-point communication). In the case of a one-to-one communication, the slave device returns a frame in response; in the case of a broadcast query, no response is sent. The Modbus protocol defines the format of a master device query: the device (or broadcast) address, the function code, all data to be sent, and the error detection field.The message frames sent by the slave device are also structured according to the Modbus protocol, including fields for confirming the action to be performed, any data to be returned, and error detection. If an error occurs during message reception, or if the slave device cannot execute the command, the slave device will construct an error message frame and send it as a response.

The Modbus protocol is the most widely used PLC communication protocol in industry. It includes two modes: RTU (Remote Terminal Unit) and ASCII (American Standard Code for Information Interchange). The interpretation of message fields is identical in both modes; the primary difference lies in their error checking methods and character formatting.

The controller can be configured to communicate over a standard Modbus network using either of the two transmission modes (ASCII or RTU). The user selects the desired mode, including serial communication parameters (baud rate, parity, etc.). When configuring each controller, all devices on a Modbus network must use the same transmission mode and serial parameters. The selected ASCII or RTU mode applies only to standard Modbus networks; it defines each bit of the message segments transmitted over these networks, as well as how information is packaged into message fields and how it is decoded.

## 2.2 Communication Parameters

Modbus uses RS-232, RS-485, or RS-422 interfaces as its hardware medium. The MF series of industrial gas mass flow meters uses the following communication parameters:

Table 2.1 Modbus Communication Parameters

| Communication Parameters | Protocol Format |
|---|---|
| | R T U |
| Communication Rate | 4800，9600bps，19200，38400 |
| Start Position | 1-bit |
| Data Bit | 8-bit |
| Stop Bit | 1-bit |
| Parity Check | None |
| Max Buffer Length (Data) | 20 |
| Max Number of Nodes | 247 |

The format for sending and receiving each character is as follows (with the least significant bit, D0, leading):

RTU mode (10-bit):

| ST | D0 | D1 | D2 | D3 | D4 | D5 | D6 | D7 | SP |
|---|---|---|---|---|---|---|---|---|---|
| Start position | 8-bit data | | | | | | | | Stop bit |

## 2.3 Modbus Message Frame

Transmission devices can convert Modbus messages into frames with a start and end point, allowing the receiving device to begin processing at the start of the message, read the address allocation information, determine which device has been selected (or, in broadcast mode, which devices have been selected), and recognize when the message has been completed. The receiving device can also detect errors in parts of the message and, if errors are found, return the error information to the master controller via a response message frame.

### 2.3.1 RTU Mode

When the controller is set to communicate in RTU mode on a Modbus network, each 8-bit byte in a message (consisting of two 4-bit hexadecimal characters) is transmitted as a single 8-bit character, and CRC (Cyclic Redundancy Check) is used for data verification. This maximizes the utilization of each data bit and improves communication efficiency. The main advantage of RTU mode is that, at the same baud rate, it can transmit more data than the ASCII encoding method.

When using RTU mode, message transmission must begin with a pause interval of at least 3.5 character times (here, "character time" refers to the values listed in Table 1, not simply the transmission time of a character consisting of 7-bit or 8-bit data; the same applies below). Given the varying character times at network baud rates, this is the easiest method to implement (as shown by T1-T2-T3-T4 in the figure below, equivalent to the transmission delay of four bytes).

The first field transmitted is the device address. The transmission characters that can be used are hexadecimal 0–9 and A–F. Network devices continuously monitor the network bus, including the pause intervals. Upon receiving the first field (the address field), each device decodes it to determine if the message is intended for itself. After the last transmitted character, a pause of at least 3.5 character times marks the end of the message.

A new message may begin after this pause. During Modbus RTU communication, the entire message frame must be transmitted as a continuous data stream. If there is a pause of more than 1.5 character times before the frame is complete, the receiving device will discard the incomplete message and assume that the next byte is the address field of a new message (indicating that the transmission of the current message frame has been interrupted). Similarly, if a new message begins within less than 3.5 character times after the previous message, the receiving device will treat it as a continuation of the previous message. This will result in an error because the value in the final CRC field cannot be correct. A typical message frame is shown below:

| Start position | Device address | Function code | Data | CRC check | End tag |
|---|---|---|---|---|---|
| T1-T2-T3-T4 | 8-bit | 8-bit | n 8-bit (20≥n≥0) | 16-bit | T1-T2-T3-T4 |

### 2.3.2 Device Address Space

The device address field of a message frame contains 8 bits (RTU). Possible slave device addresses range from 0 to 247 (decimal), with individual device addresses ranging from 1 to 247. Address 0 is used as the broadcast address so that all slave devices can recognize it. The master device selects a slave device by placing the address of the slave device to be contacted in the address field of the message. When the slave device sends a response message, it places its own address in the address field of the response so that the master device knows which device responded.

### 2.3.3 Function Code Area

The function code field in a message frame contains 8 bits (RTU), with a possible code range of 1 to 255 in decimal. Of course, some codes are applicable to all controllers, some are specific to certain controllers, and others are reserved for future use.

When a message is sent from the master device to the slave device, the function code field instructs the slave device on the actions it must perform. For example, reading the status of an input switch, reading the contents of a set of registers, reading the diagnostic status of the slave device, or allowing the loading, logging, and verification of programs within the slave device.

When the slave device responds, it uses the function code field to indicate whether the response is normal (no error) or whether an error has occurred (known as a non-conforming response). For a normal response, the slave device simply returns the corresponding function code. For a non-conforming response, the slave device returns a code equivalent to the normal code, but with the most significant bit set to a logical 1 (normal function code + 0x80).

For example: A message sent from the master device to the slave device requesting a read of a set of hold registers will generate the following functional code:

0 0 0 0 0 0 1 1(Hexadecimal 0x03)

For a normal response, the device returns only the same function code. For an error response, it returns:

1 0 0 0 0 0 1 1(Hexadecimal 0x83)

In addition to modifying the function code due to an error caused by a conflict, the slave device places a unique error code in the data field of the response message, which informs the master device of the error that occurred.

When the master device application receives a response indicating an error, the typical procedure is to resend the message or diagnose the message sent to the slave device and report the issue to the operator.

The function codes used by the MF4003 gas mass flow meter are a subset of those specified by the Modbus standard communication protocol, primarily consisting of the codes listed in Table 2.2 (function codes are expressed as decimal numbers; see the next chapter for detailed descriptions).

**Table 2.2 Modbus Function Codes Used by the MF Series Industrial Gas Mass Flow Meters**

| Function code | Name | Data types | Function |
|---|---|---|---|
| **03** | Read the hold register | Integer, character, status, floating-point | Read the values of one or more consecutive hold registers |
| **06** | Pre-set single-register | Integer, character, status, floating-point | Load a specific binary value into a hold register |
| **08** | Error diagnosis query | Integer | Check whether communication between the main unit and the flow meter is working properly |
| **16** | Multiple registers predefined | Integer, character, status, floating-point | Load specific binary values into multiple consecutive hold registers |

### 2.3.4 Data Area

The data field consists of a set of two hexadecimal numbers ranging from 0x00 to 0xFF. Depending on the network transmission mode, this may consist of a pair of ASCII characters or a single RTU character. The data field in messages sent from the master device to the slave device contains additional information that the slave device must use to perform the behavior defined by the function code.

This includes details such as non-contiguous register addresses, the number of items to be processed, and the actual number of data bytes in the field. For example, if the master device needs to read a set of holding registers from the slave device (function code 0x03), the data field specifies the starting register address and the number of registers to be read.

If the master device writes a set of the slave device's registers (function code 0x10), the data field specifies the starting register address, the number of registers to be written, the number of data bytes in the data field, and the data to be written to the registers. If no error occurs, the data field returned by the slave device contains the requested data. If an error occurs, this field contains an error code that the master device application can use to determine the next course of action.

In certain messages, the data field may be absent (zero-length). For example, when the master device requests that the slave device acknowledge a communication event record (function code 0x0B), the slave device does not require any additional information.

### 2.3.5 Error Detection Domain

When the RTU mode is selected for character frames, the error detection field contains a 16-bit value (implemented using two 8-bit characters). The contents of the error detection field are derived by applying the CRC (Cyclic Redundant Check) method to the message content. The CRC field is appended to the end of the message, with the low byte added first, followed by the high byte. Therefore, the high-order byte of the CRC is the last byte of the transmitted message frame.

For an introduction to the CRC algorithm, please refer to the discussion on error detection methods in the next section.

## 2.4 Error Detection Methods

The standard Modbus network employs two error detection methods: parity checking and frame checking (i.e., error detection fields). Parity checking is available for each character, while frame checking (LRC or CRC) is applied to the entire message. Both are generated by the master device before the message is sent, and the slave device checks each character and the entire message frame during reception. The user must configure the master device with a predefined timeout interval that is long enough to allow any slave device to respond normally. If a slave device detects a transmission error, it will not receive the message and will not respond to the master device. This triggers a timeout event, prompting the master device to handle the error. Sending a message to a non-existent slave device address will also result in a timeout.

### 2.4.1 Parity Check

Users can configure the controller to use odd parity, even parity, or no parity. This determines how the parity bit and stop bits are set for each character. If odd or even parity is specified, the "1" bit is counted as part of the total number of bits per character (7 data bits in ASCII mode, 8 data bits in RTU mode). For example, an RTU character frame contains the following 8 data bits:

1 1 0 0 0 1 0 1

The total number of "1"s is 4. If even parity is used, the parity bit in the frame will be 0, so the total number of "1"s remains 4 (an even number). If odd parity is used, the parity bit in the frame will be 1, so the total number of "1"s is 5 (an odd number). If no parity bit is specified, no parity bit is transmitted, and no parity check is performed. Instead, an additional stop bit is inserted into the character frame to be transmitted.

### 2.4.2 CRC Check

In RTU mode, the message frame includes an error detection field based on the CRC method. The CRC field is two bytes long and contains a 16-bit binary value that checks the contents of the entire message frame (excluding the CRC field itself). It is calculated by the transmitting device and added to the message. The receiving device recalculates the CRC of the received message and compares it with the value in the received CRC field; if the two values differ, an error has occurred. To calculate the CRC, a 16-bit register is first initialized with all "1"s, and then a procedure is called to process the consecutive bytes in the message frame. Only the 8-bit data bytes of each character are included in the CRC calculation; the start bit, stop bit, and parity bit are excluded. During the CRC generation process, each 8-bit character is individually XORed with the contents of the 16-bit register. The result is shifted toward the least significant bit (LSB), and the most significant bit (MSB) is filled with zeros. The LSB is extracted for testing,

If the LSB is 1, the register is ORed with a preset fixed value; otherwise, no operation is performed. This process is repeated 8 times. After the last bit (the 8th bit) is processed, the next 8-bit byte is individually XORed with the current value in the register. The final value in the register is the CRC value calculated after all bytes in the message (excluding the CRC field itself) have been processed.

The value of the CRC error detection field can be calculated using the following method:

**a)** Initialize a 16-bit CRC register to 0xFFFF (all 1s);
**b)** Perform a bitwise OR operation between the lower 8 bits of the CRC register and each 8-bit byte in the message frame, and store the result back in the CRC register;
**c)** Shift the CRC register one bit to the right, pad the most significant bit (MSB) with a zero, and prepare to test the least significant bit (LSB) that was shifted out;
**d)** If the LSB is 0, repeat step c); if the LSB is 1, perform a bitwise OR operation on the CRC register with the value 0xA001 of a polynomial;
**e)** Repeat steps c) and d) until 8 shift operations have been completed. At this point, a complete 8-bit byte has been processed;
**f)** Repeat steps b) through e) to process the next byte in the message frame until all bytes in the message frame have been processed;
**g)** The value in the final CRC register is the value of the CRC error detection field.

The following are examples of CRC generation written in C (Example 2 uses the direct calculation method, suitable for microprocessors or microcontrollers with strong computational capabilities and limited memory; Example 3 uses the lookup table method, suitable for microcontrollers with weaker computational capabilities and ample memory):

**Example 2: C Language implementation of CRC using the direct calculation method**

```
// For simplified calculations, define a composite structure
typedef union
{
  UINT16 val;
  struct
  {
    UINT16 bit0 : 1;
    UINT16 bit1 : 1;
    UINT16 bit2 : 1;
    UINT16 bit3 : 1;
    UINT16 bit4 : 1;
    UINT16 bit5 : 1;
    UINT16 bit6 : 1;
    UINT16 bit7 : 1;
    UINT16 bit8 : 1;
    UINT16 bit9 : 1;
    UINT16 bit10: 1;
    UINT16 bit11: 1;
```

```
    UINT16 bit12: 1;
    UINT16 bit13: 1;
    UINT16 bit14: 1;
    UINT16 bit15: 1;
  } bits;
} TCRCRegs;
TCRCRegs regs;

// pBuf：Message buffers to be included in the calculation
// nDataLen：The number of bytes to be processed by the CRC (message buffer length)
UINT16 CRC1(UINT8 *pBuf, UINT16 nDataLen)
{
  int      i;
  UINT8 j,nTest;

  regs.val = 0xFFFF;
  for (i = 0; i < nDataLen; i++)
  {
      regs.val ^= *pBuf++;
      for (j = 0; j < 8; j++)
      {
         nTest = (regs.bits.bit0) ? 0x01:0x00;
         regs.val >>= 1;
         if (nTest == 1)
            regs.val ^= 0xA001;
      }
  }
  return regs.val;
}
```

**Example 3: C Implementation of CRC using Lookup Tables**

```
//  Define a combinatorial structure to simplify the calculation
typedef union
{
  UINT16 nCRC16;
  struct
  {
    UINT8 nCRCHi;
    UINT8 nCRCLo;
  }
} TCRCRegs;

// pBuf：Message buffers to be included in the calculation
// nDataLen：The number of bytes to be processed by the CRC (message buffer length)
UINT16 CRC2(UINT8 *pBuf, UINT16 nDataLen)
```

```
{
   TCRCRegs nCRC;
   UINT16 nIndex;                              //CRC Lookup Table Index Definitions

   nCRC.nCRC16 = 0xFFFF;                       // CRC Initialization
   while (uDataLen—)                           // Iterate through the message buffer
   {
      nIndex = nCRC.nCRCHi ^ *pBuf++;     // calculation CRC
      nCRC.nCRCHi = nCRC.nCRCLo ^ CRCHi[nIndex];
      nCRC.nCRCLo = CRCLo[nIndex];
   }
   return (nCRC.nCRC16);
}

/* CRC's 8-bit lookup table  */
static UINT8 CRCHi[] =  {

    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
    0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
    0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
    0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81,
    0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
    0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
    0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
    0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
    0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
    0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
    0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
    0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
    0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
    0x40
};

/* CRC low 8-bit lookup table  */
static INT8 CRCLo[] =
{
    0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06, 0x07, 0xC7, 0x05, 0xC5, 0xC4,
    0x04, 0xCC, 0x0C, 0x0D, 0xCD, 0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09,
    0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A, 0x1E, 0xDE, 0xDF, 0x1F, 0xDD,
    0x1D, 0x1C, 0xDC, 0x14, 0xD4, 0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13, 0xD3,
    0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3, 0xF2, 0x32, 0x36, 0xF6, 0xF7,
    0x37, 0xF5, 0x35, 0x34, 0xF4, 0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A,
    0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29, 0xEB, 0x2B, 0x2A, 0xEA, 0xEE,
    0x2E, 0x2F, 0xEF, 0x2D, 0xED, 0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26,
    0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60, 0x61, 0xA1, 0x63, 0xA3, 0xA2,
    0x62, 0x66, 0xA6, 0xA7, 0x67, 0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC, 0xAD, 0x6D, 0xAF, 0x6F,
    0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68, 0x78, 0xB8, 0xB9, 0x79, 0xBB,
    0x7B, 0x7A, 0xBA, 0xBE, 0x7E, 0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C, 0xB4, 0x74, 0x75, 0xB5,
    0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71, 0x70, 0xB0, 0x50, 0x90, 0x91,
    0x51, 0x93, 0x53, 0x52, 0x92, 0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54, 0x9C, 0x5C,
    0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B, 0x99, 0x59, 0x58, 0x98, 0x88,
    0x48, 0x49, 0x89, 0x4B, 0x8B, 0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C, 0x8C,
    0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42, 0x43, 0x83, 0x41, 0x81, 0x80,
    0x40
};
```

# 3 Function Description

## 3.1 Modbus Function Format

### 3.1.1 Representation of Numerical Values

Unless otherwise specified, the numerical values described in this section (such as addresses, function codes, or data) are expressed in decimal form, but are represented in hexadecimal form in the message frames to be sent or received.

### 3.1.2 Data Address in a Message Frame

Data addresses in all Modbus message frames are relative to 0, and the first data element in the data sequence always starts at 0. For example:

a）In the device, the address of "Coil 1" in the data address field of the Modbus message frame is 0x0000, and the address of "Coil 127" is 0x007E (i.e., 126);

b）The address of "Hold Register 40001" in the data address field of a Modbus message frame is 0x0000. Since the function code specifies that the operation is on a hold register, "4xxxx" implicitly refers to a hold register. For example, the address of hold register 40108 is 0x006B (i.e., 107).

### 3.1.3 Field Content in a Message Frame

Table 3.1 shows an example of a Modbus query message, and Table 3.2 shows an example of a normal response. Both examples display the field contents in hexadecimal format and illustrate how the messages are organized into message frames in ASCII or RTU mode.

**Table 3.1 Query Message Frame Format**

| Domain Name | Example | RTU |
|---|---|---|
| Frame Header | | 3.5T Timeout |
| Source Address | 6 | 0000 0110 |
| Function Code | 0x03 | 0000 0011 |
| Start Address (High Byte) | 0x00 | 0000 0000 |
| Start Address (Low Byte) | 0x6b | 0110 1011 |
| Number of Registers (High Byte) | 0x00 | 0000 0000 |
| Number of Registers (Low Byte) | 0x03 | 0000 0011 |
| Checksum | | crc |
| Frame Tail | | 3.5T Timeout |

Table 3.2 Response Message Frame Format

| Domain Name | Example | RTU |
|---|---|---|
| Frame Header | | None |
| From Device Address | 0x06 | 0000 0110 |
| Function Code | 0x03 | 0000 0011 |

| | | |
|---|---|---|
| Number of Bytes | 0x06 | 0000 0110 |
| Data High (Bytes) | 0x02 | 0000 0010 |
| Data Low (Bytes) | 0x2B | 0010 1011 |
| Data High (Bytes) | 0x00 | 0000 0000 |
| Data Low (Bytes) | 0x00 | 0000 0000 |
| Data High (Bytes) | 0x00 | 0000 0000 |
| Data Low (Bytes) | 0x63 | 0110 0011 |
| Checksum | | CRC (16-bit) |
| Frame End | | Nnoe |

The master device query is a message frame that reads the hold registers of slave device 06. This message requests data from hold registers 40108 through 40110, with the starting register address set to 0x006B (i.e., 107). The slave device responds with the same function code, indicating that this is a normal response. The number of bytes field specifies how many 8-bit data bytes will be returned to the master device in the response message frame.

### 3.1.4 Using the "Number of Bytes" Field

When constructing a response buffer, the byte count field must be used to specify the number of bytes in the data field of a message frame. This value indicates only the number of bytes in the data field and does not include other fields (including this field itself).

### 3.1.5 Exception Response

When the master device sends a query message to the slave device, the master device expects to receive a normal response message; however, an exception may occur at this point for some reason. When the master device executes the query, one of the following four events may occur:

a) If the slave device receives the query message and no communication error occurs, it will process the query normally and return a valid response message;

b) If the slave device does not receive te query message due to a communication error, no response message will be returned. The master device will ultimately handle this situation by triggering a timeout event;

c) If the slave device receives the query message but detects a communication error (such as a parity error, LRC, or CRC error), the slave device will not return a response message. The master device will ultimately handle this situation by triggering a timeout event;

d) If the slave device correctly receives the query message (without any communication errors) but cannot process the message normally (for example, the output relay coil or latch register to be read does not exist), the slave device will return an exception response message to notify the master device of the type of error.

An error response message differs from a normal response in two fields:

a) Function Code Field

In a normal response message, the slave device echoes the function code from the original query message. The most significant bit (M B) of all function codes is 0 (i.e., their values are less than 0x80). In an error response message, however, the slave device sets the most significant bit (MSB) of the function code to 1, resulting in function code values greater than 0x80.

By setting the most significant bit of the function code to 1, the application on the master device can detect that an exception has occurred in the response message.

b）Data Field

In a normal response message, the device may return data or statistics in its data field. In an error response message, the device returns an error code in the data field that identifies the cause of the error.

The table below lists the error codes that may be generated by the flow meter during Modbus communication.

Table 3.3 Error codes returned when responding to device anomalies

| Error code | Name | Meaning |
|---|---|---|
| 1 | ILLEGAL_FUNCTION | The function code in the received query message is invalid (not permitted) for the current slave device. |
| 2 | ILLEGAL_DATA_ADDRESS | The data address in the received query message is invalid (not permitted) for the current slave device. |
| 3 | ILLEGAL_DATA_VALUE | The value in the data field of the received query message is invalid (not permitted) for the current slave device. |
| 4 | SLAVE DEVICE FAILURE | An unrecoverable error occurred while the device was attempting to perform the requested action. |
| 5 | ACKNOWLEDGE | The device has received a request and is currently processing it, but the operation is taking a long time. To prevent a timeout error on the master device, you can send an ACKNOWLEDGE response to the master device. |
| 6 | SLAVE_DEVICE_BUSY | Since the slave device is currently processing a time-consuming program command, the master device should resend the query message when the slave device is not busy. |

The table below shows an example of a query message frame and an error response frame resulting from writing to a read-only register address.

Table 3.4 Example of an exception response frame format

| Query message frame | | Exception response message frame | |
|---|---|---|---|
| Message frame compositi | Data | Message frame composition | Data |
| From the device address | 0x11 | From the device address | 0x11 |
| Function code | 0x06 | Function code | 0x86 |
| High register address | 0x00 | Error code | 0x02 |
| Lower register address | 0x00 | Checksum | 0xC2 |
| High preset data | 0x01 | | |
| Low preset data | 0x03 | | |
| Checksum | 0xCA | | 0x64 |
| | 0xCB | | |
| Instructions | In this example, an attempt was made to modify the value in read-only hold register 40000 inside flow meter No. 17, resulting in an exception (error code = ILLEGAL_DATA_ADDRESS = 0x02). | | |

## 3.2 Read the Hold Register (Function Code = 03)

### 3.2.1 Function Description

Reads the values of the internal hold registers (starting with 4) in the flowmeter; the result is one or more 16-bit integers or unsigned integers.

For the starting address of the data and the number of registers that can be read at a time, please refer to Appendix B.

### 3.2.2 Message Frame Format

The query message frame (sent by the master device) consists of five parts: the device address field, the function code, the register address field, the data field, and the checksum. The function code is fixed at 0x03, representing the read-retain-register function; The register address field specifies the starting address of the hold registers to be read in this operation (register addresses start at 0x0000, with registers 1 through 16 having addresses ranging from 0x00000 to 0x000F) and consists of two bytes; the data field specifies the number of hold registers to be read in this operation and consists of two bytes.

The response message frame (sent by the flow meter) consists of five parts: the device address field, function code, number of bytes, data field, and checksum. The function code is fixed at 0x03, indicating that the response is for the read-hold register function; the number of bytes specifies the number of data bytes contained in the data field; The data field consists of a series of contiguous double-byte data entries representing the data from the hold registers to be read. Each register's data is represented by two bytes, and the register addresses are listed in ascending order from least significant to most significant.

An example (in RTU format) is as follows:

Table 3.5 Example of Frame Format

| **Query message frame** | | **Response message frame** | |
|---|---|---|---|
| **Message frame composition** | **Data** | **Message frame composition** | **Data** |
| From the device address | 0x11 | From the device address | 0x11 |
| Function code | 0x03 | Function code | 0x03 |
| High starting address | 0x00 | Number of bytes | 0x06 |
| Low starting address | 0x6B | Data high (register 40108) | 0x02 |
| High read count | 0x00 | Data low (register 40108) | 0x2B |
| Low read count | 0x03 | Data high (register 40109) | 0x00 |
| Checksum | 0x76 | Data low (register40109） | 0x00 |
| | | Data high (register40110） | 0x00 |
| | 0x87 | Data low (register40110） | 0x64 |
| | | Checksum | 0xC8 |
| | | | 0xBA |
| **Instructions** | Example: Read the values in registers 40108–40110 from flowmeter No. 17 | | |

The value of register 40108 is represented by the two bytes 0x02 and 0x2B, which is 555 in decimal. The values of registers 40109 through 40110 are 0x00, 0x00, and 0x00, 0x65, which are 0 and 100 in decimal.

## 3.3 Preset Single Register (Function Code = 06)

### 3.3.1 Function Description

Sets a value in the specified hold register (starting with 4) of the flow meter. In broadcast mode, this command sets the same value in the corresponding hold registers of all connected flow meters simultaneously.

The value in the flowmeter's hold register may also be modified by the flowmeter itself based on its internal control logic.

For the starting address and length of the data, please refer to Appendix B.

### 3.3.2 Message Frame Format

The formats of the query message frame (sent by the master device) and the response message frame are identical; both consist of five parts: the device address field, the function code, the register address field, the data field, and the checksum. The function code is fixed at 0x06, representing the single-register preset function; the register address field specifies the address of the register to be preset (register addresses start at 0x0000, meaning register 1 has the address 0x0000) and consists of two bytes; The data field specifies the data to be preset at the designated register address and consists of two bytes. In non-broadcast mode, the normal response is an echo of the query message frame (i.e., identical in content to the query message frame), which is returned after the specified register has been preset; in broadcast mode, the flow meter does not return a response message frame.

For an example (RTU format), see the table below.

Table 3.6 Example of a Frame Format

| **Query message frame** | | **Response message frame** | |
|---|---|---|---|
| **Message frame composition** | **Data** | **Message frame composition** | **Data** |
| From the device address | 0x11 | From the device address | 0x11 |
| Function code | 0x06 | Function code | 0x06 |
| High register address | 0x00 | High register address | 0x00 |
| Low register address | 0x01 | Low register address | 0x01 |
| High preset data | 0x00 | High preset data | 0x00 |
| Low preset data | 0x03 | Low preset data | 0x03 |
| Checksum | 0x9A | Checksum | 0x9A |
| | 0x9B | | 0x9B |
| **Instructions** | Example: The value of register 40002 for flow meter No. 17 is 0x0003 | | |

## 3.4 Error Diagnosis Inquiry (Function Code = 08)

### 3.4.1 Function Description

Using diagnostic function 08 provided by the Modbus protocol, you can test the communication between the host computer and the flowmeter to verify that communication is functioning properly. This function does not support broadcast mode. The error diagnosis query function uses a two-byte subfunction code field in the query message frame to define the type of test operation to be performed. This version of the protocol supports only one of the standard diagnostic subfunction codes, namely Return Query Data (Subfunction Code: 00). All other subfunction codes are reserved for future use; if a user attempts to use them, an error response will be returned. The data in the data field consists of arbitrary two-byte data. If communication is normal, the flowmeter (slave device) will return the master device's query message frame to the master device as a response frame without modification.

### 3.4.2 Message Frame Format

The formats of the query message frame (sent by the master device) and the response message frame are identical; both consist of five parts: slave device address, function code, sub-function code, diagnostic data, and checksum. Specifically, the function code is fixed at 0x08, representing the device diagnostic function; the sub-function code is fixed at 0x0000, consisting of two bytes, representing the echo query data sub-function; and the diagnostic data consists of two bytes and can be any value.Here is an example of using the diagnostic feature:

Table 3.7 Example of Frame Format

| Query message frame | | Response message frame | |
|---|---|---|---|
| **Message frame structure** | **Date** | **Message frame structure** | **Date** |
| From the device address | 0x11 | From the device address | 0x11 |
| Function code | 0x08 | Function code | 0x08 |
| High sub-function | 0x00 | High sub-function | 0x00 |
| Low sub-function | 0x00 | Low sub-function | 0x00 |
| High data | 0xA5 | High data | 0xA5 |
| Low data | 0x37 | Low data | 0x37 |
| Checksum | 0x36 | Checksum | 0x36 |
| | 0xF9 | | 0xF9 |
| **Instructions** | Example: Check whether communication between Flow Meter No. 17 and the host computer is normal (a response is displayed when normal). | | |

## 3.5 Pre-set Multi-Register (Function Code = 16)

### 3.5.1 Function Description

Set multiple values in multiple hold registers (starting with 4) of the flow meter. In broadcast mode, this command sets the same value in the corresponding hold registers of all connected flow meters simultaneously.
The value in the flowmeter's hold register may also be modified by the flowmeter itself based on its internal control logic.

For the starting address and length of the data, please refer to Appendix B.

### 3.5.2 Query Message Frame

The query message frame (sent by the master device) specifies the registers to be initialized and the data to be initialized. Register addresses start at 0; that is, register 1 has an address of 0.

The requested default values are specified in the data field of the query message frame. The value of each register is represented by 2 bytes (16 bits in total).

### 3.5.3 Response Message Frame

A normal response returns the device address, function code, start address, and the number of preset registers; it is returned after all registers have been initialized.

For an example (RTU format), see the table below.

Table 3.8 Example of Frame Format

| Query message frame | | Response message frame | |
|---|---|---|---|
| **Message frame structure** | **Date** | **Message frame structure** | **Date** |
| From the device address | 0x11 | From the device address | 0x11 |
| Function code | 0x10 | Function code | 0x10 |
| Starting address is high | 0x00 | High starting address | 0x00 |
| Starting address is low | 0x01 | Low starting address | 0x01 |
| High number of registers | 0x00 | High number of registers | 0x00 |
| Low number of registers | 0x02 | Low number of registers | 0x02 |
| Number of bytes | 0x04 | Checksum | 0x12 |
| High preset data | 0x00 | | |
| Low preset data | 0x0A | | |
| High preset data | 0x01 | | 0x98 |
| Low preset data | 0x02 | | |
| Checksum | 0xC6 | | |
| | 0xF0 | | |
| **Instructions** | In the example preset, the values of registers 17 and 18 (starting at address 40002) are 0x000A and 0x0102, respectively. | | |

ATO

# Appendix A Parameter Descriptions

## A.1 Parameter Limits

Table A.1 lists the maximum number of data items that the master device may request or send in a master query, or that the slave device (flow meter) may return in a response message; all data values are expressed in decimal.

**Table A.1 Maximum Number of Query/Response Parameters**

| Function code | Function Description | Query message frame | Response message frame |
|---|---|---|---|
| **03** | Read the hold register | 8 registers | 8 registers |
| **06** | Pre-set single-register | 1 registers | 1 registers |
| **08** | Error diagnosis query | N/A | N/A |
| **16** | Predefined multi-register | 8 registers | 8 registers |

## A.2 Parameter Overview

See the notes in section A.3.2.

## A.3 Parameter Details

### A.3.1 Agreement

Data type:

| Data type | Description | Length (in bytes) |
|---|---|---|
| UINT8 | unsigned char，8 bits | 1 |
| UINT16 | unsgined int，16 bits | 2 |
| UINT32 | unsigned long int，32 bits | 4 |
| STRING | ASCII-encoded string | |
| Combination type | Composed of a combination of UINT8, UINT16, or UINT32 | |

Others：

1) The read/write attributes of a parameter indicate whether the parameter is readable or writable. Specifically, R stands for read-only, W stands for write-only, and RW stands for read-write;

2) (H) retrieves the upper 8 bits of a word variable at the specified address, while (L) retrieves the lower 8 bits of a word variable at the specified address. For example, if the variable value stored at address 0x0000 is 0x1234, then 0x0000(H) = 0x12, and 0x0000(L) = 0x34.

## A.3.2 Detailed Parameter Description

| | | | |
|---|---|---|---|
| **Local address** | 0x0001 | **Edit** | Allow |
| | | **Read** | Allow |
| **Description** | Local device address | | |
| **Data type** | UINT16 | | |
| **Data represent** | Any value from 1 to 247. 0 is the broadcast address; the local address cannot be set to 0. | | |

| | | | |
|---|---|---|---|
| **Real-time flow** | 0x0002～0x0003 | **Edit** | Not allowed |
| | | **Read** | Allow |
| **Description** | Current gas flow rate. | | |
| **Data type** | UINT32 | | |
| **Data represent** | 0x0002–0x0003 form a UINT16 unsigned integer representing the current gas flow rate;<br>Flow rate V = value(0x0002) * 65536 + value(0x0003)<br>The value of V is the flow rate (in base units) * 1000<br>Example: If the current unit is L/min and the flow rate value is 20.34, the value obtained via Modbus is 20.34 * 1000 = 20340 | | |

| | | | |
|---|---|---|---|
| **Current total flow** | 0x0004～0x0006 | **Edit** | Allow |
| | | **Read** | Allow |
| **Description** | The cumulative total flow recorded by the meter from the last reset to the present | | |
| **Data type** | UINT32 + UINT16 | | |
| **Data represent** | V1 = value (0x0004) * 65536 + value (0x0005);<br>V2 = value (0x0006)<br>Total V = V1 * 1000 + V2;<br>Here, V1 is a 32-bit integer representing the integer part of the current total flow; V2 is a 16-bit integer representing the fractional part of the current total flow.<br>Example: If the current total volume is 3452.245 cubic meters, the value obtained via Modbus is 3452 * 1000 + 245 = 3452245 | | |

| | | | |
|---|---|---|---|
| **Baud rate** | 0x0015 | **Edit** | Allow |
| | | **Read** | Allow |
| **Description** | The index corresponding to the instrument's current baud rate | | |
| **Data type** | UINT16 | | |
| **Data represent** | Index mapping for the instrument's current baud rate: 0: 4800, 1: 9600, 2: 19200, 3: 38400<br>Example: When the instrument's current baud rate is 38400, the value retrieved via the Modbus protocol is 3 | | |

| | | | |
|---|---|---|---|
| **GDCF** | 0x0016 | **Edit** | Allow* |
| | | **Read** | Allow |
| **Description** | A correction factor for flow rate | | |

| | | | |
|---|---|---|---|
| **Data type** | UINT16 | | |
| **Represent** | UINT16: If the correction factor is 1000, the value obtained via the Modbus protocol will be 1000. | | |
| **Response time** | 0x0017 | **Edit** | Allow* |
| | | **Read** | Allow |
| **Description** | Index of sensor response times | | |
| **Data type** | UINT16 | | |
| **Represent** | 0:10ms, 1:20ms, 2:50ms, 3:100ms, 4:200ms, 5:500ms,6:1000ms<br>Example: If the response time is 10 ms, the value obtained via the Modbus protocol is 0. | | |
| **Filter depth** | 0x0018 | **aEdit** | Allow* |
| | | **Read** | Allow |
| **Description** | Depth of the moving average filter (filter buffer length) | | |
| **Data type** | UINT16 | | |
| **Represent** | Example: If the filter depth is 8, the value retrieved via the Modbus protocol is 8. | | |
| **Instrument version number** | 0x0019 | **Edit** | Not allowed |
| | | **Read** | Allow |
| **Description** | Software version number | | |
| **Data type** | HEX | | |
| **Represent** | Example: If the product software version is V4201, the value retrieved via the Modbus protocol is 0x4001. | | |
| **Instrument serial number** | 0x001A~0x001F | **Edit** | Not allowed |
| | | **Read** | Allow |
| **Description** | Product serial number (length = 12 bytes) | | |
| **Data type** | STRING | | |
| **Represent** | Example: If the product serial number is **C2E01023**, the value retrieved via the Modbus protocol will be **C2E01023**. | | |
| **Set the internal code value for zero** | 0x0020 | **Edit** | Not allowed |
| | | **Read** | Allow |
| **Description** | Internal code value when traffic is zero | | |
| **Data type** | UINT16 | | |
| **at t rypeysyn** | Example: If the internal code value at zero is 32765, the value obtained via the Modbus protocol is 32765. | | |
| **Mini flow rate of the meter** | 0x0021~0x0022 | **Edit** | Not allowed |
| | | **Read** | Allow |
| **Description** | Mini output flow rate of the meter (amplified by a factor of 1,000) | | |
| **Data type** | UINT32 | | |
| **Data represent** | Example: If the instrument's mini output flow rate is 0 SLPM, the value obtained via the Modbus protocol will be 0. | | |
| **Max flow rate of the meter** | 0x0023~0x0024 | **Edit** | Not allowed |
| | | **Read** | Allow |

| | | | |
|---|---|---|---|
| Description | Max output flow rate of the meter (amplified by a factor of 1,000) | | |
| **Data type** | UINT32 | | |
| Data represent | Example: If the instrument's max output flow rate is 5 SLPM, the value obtained via the Modbus protocol is 5000. | | |
| **Mini output voltage of the instrument (mV)** | 0x0025 | Edit | Not allowed |
| | | **Read** | Allow |
| Description | Mini output voltage of the instrument (mV) | | |
| **Data type** | UINT16 | | |
| Data represent | Example: If the mini output voltage is 500 mV, the value obtained via the Modbus protocol is 500. | | |
| **Max output voltage of the meter (mV)** | 0x0026 | Edit | Not allowed |
| | | **Read** | Allow |
| Description | Max output voltage of the meter (mV) | | |
| **Data type** | UINT16 | | |
| Data represent | Example: If the max output voltage is 500 mV, the value obtained via the Modbus protocol is 500. | | |
| **Auto-zero** | 0x0027 | Edit | Allow |
| | | **Read** | Not allowed |
| Description | This command forces the instrument to perform an automatic zero calibration. Please note that you must ensure the gas flow in the sensor line has come to a complete stop before performing this operation. | | |
| **Data type** | Specified data 0xAA55 | | |
| Data represent | Example: Simply write the specified value 0xAA55 to the register to perform an automatic zero-calibration. | | |
| **Write-protect register** | 0x0014 | Edit | Allow |
| | | **Read** | Not allowed |
| Description | Writing to some registers is protected to prevent accidental modifications that could cause errors in instrument parameters. To modify these parameters, you must first set the write-protect register. This setting is valid for a single operation; you must set the write-protect register again before making subsequent modifications. For registers marked with an asterisk, you must set the write-protect register. | | |
| **Data type** | Specified data 0xAA55 | | |
| **Data represent** | Example: To modify the GDCF, you must first write 0xAA55 to the write-protect register for the modification to succeed. | | |