# Northbound Modbus-TCP Interconnection Protocol for AC Chargers

# Change History

| Date | Version | CR ID/Defect ID | Section Number | Change Description |
|---|---|---|---|---|
| 2023-08-04 | V1.0 | | | Completed the initial draft. |

# Contents

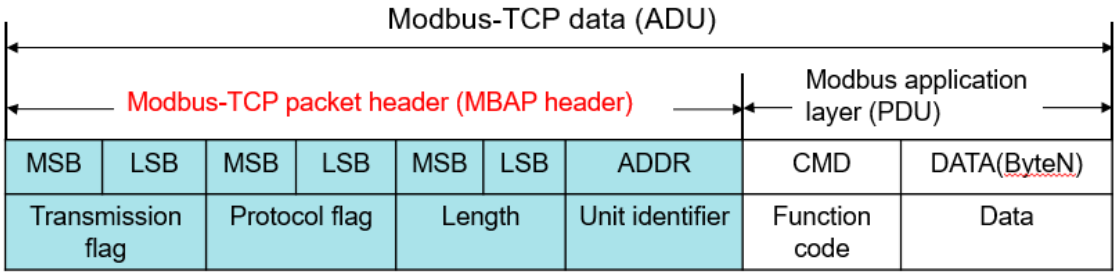
# 1 Modbus-TCP Protocol

## 1.1 About the Modbus-TCP Protocol

Modbus-TCP is a TCP/IP-based communication protocol. Therefore, separators and cyclic redundancy checks (CRCs) are not required.

## 1.2 Modbus-TCP Frame Format

The Modbus-TCP packet format is as follows:




**Format of the MBAP header (big-endian):**

| Field | Length | Description | Client | Server |
|---|---|---|---|---|
| Transmission flag | 2 bytes | Transaction ID, which is used to match requests and responses in the same transaction. | Generated by the client | Recopied by the server from the received request |
| Protocol flag | 2 bytes | 0 = Modbus protocol | Generated by the client | Recopied by the server from the received request |
| Length | 2 bytes | Number of subsequent bytes | Generated by the client | Initialized by the server (response) |
| Unit identifier | 1 byte | Identifier of a remote slave server connected to a serial line or other bus | Generated by the client | Recopied by the server from the received request |

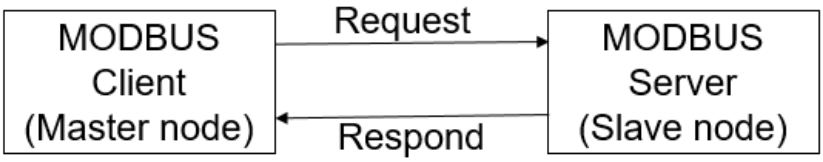
**[Note]**

1. Maximum frame length: According to the Modbus-TCP protocol, the maximum length of the application data unit (ADU) is 260 bytes, and which the length of the protocol data unit (PDU) is 253 bytes. To improve communication efficiency and be compatible with the open-source Modbus protocol, Huawei Modbus protocol extends the frame length of some customized function codes. The maximum frame length is 65,535 bytes. For details about the maximum packet length of a function code, see 4 Function Code Definition.

2. Unit ID: The Modbus-TCP protocol is based on TCP/IP. For Modbus-TCP networking communication on the same layer, the IP address in the TCP packet header can be used for addressing. In this case, the unit ID is 0 or 0xFF. This field can be used to identify the target device during cross-layer access.

### 1.3 Packet Exchange Process

Modbus-TCP uses the client/server communication mode. The client functions as the master node and initiates requests. The server functions as the slave node and receives responses. Broadcast packets cannot be sent.

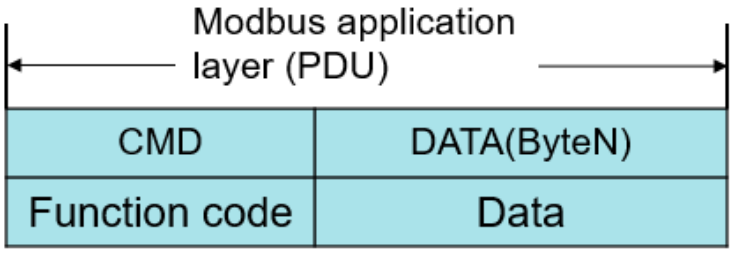



**[Note]**

1. During packet interaction of the original Modbus-TCP, the client initiates a request and the server passively responds to the request. To improve communication efficiency and be compatible with the original Modbus protocol, Huawei's Modbus protocol uses the TCP full-duplex communication for some customized function codes. For details about the communication capability of each function code, see 4 Function Code Definition.
2. Modbus-TCP does not support the sending and receiving of broadcast packets.

## 2 Application Layer Specifications

- The application layer protocol of Modbus defines PDU packets.




**[Note]** The PDU data at the Modbus application layer is encoded in **big-endian mode**.

- The Modbus function code is represented by one byte. The value range is planned as follows:

- Huawei Modbus protocol supports the following open-source Modbus function codes:

| Function Codes of Open-Source Modus | Function | Supported by Huawei Modbus |
|---|---|---|
| 01(0x01) | Reads coil registers. | No |
| 02(0x02) | Reads discrete input registers. | No |
| 03(0x03) | Reads holding registers. | Yes |
| 04(0x04) | Reads input registers. | No |
| 05(0x05) | Writes a single register. | No |
| 06(0x06) | Writes a single holding register. | Yes |
| 07(0x07) | Writes multiple coil registers. | No |
| 08(0x08) | Writes multiple holding registers. | No |
| 11(0x0B) | Obtains the communication event counter. | No |
| 12(0x0C) | Obtains communication event logs. | No |
| 15(0x0F) | Write multiple coils | No |
| 16(0x10) | Writes multiple registers. | Yes |
| 17(0x11) | Reports the server ID. | No |
| 20(0x14) | Reads file records. | No |
| 21(0x15) | Writes file records. | No |
| 22(0x16) | Writes mask registers. | No |
| 23(0x17) | Reads or writes multiple registers. | No |
| 24(0x18) | Reads the FIFO queue. | No |
| 43/13(0x2B/0x0D) | CANopen general reference requests and responses | No |
| 43/14(0x2B/0x0E) | Reads the device ID. | Yes |

- Huawei Modbus protocol supports only four function codes of the open-source Modbus: 03/06/10/0x2B0E.

## 3 Application Scenario and Interaction Process

### 3.1 Register Read/Write

| Function Category | Function Code | Function Description |
|---|---|---|
| Register read/write | 0x03 | For details, see 4.1 0x03 Reading Registers. |
| | 0x06 | For details, see 4.2 0x06 Writing a Single Register. |
| | 0x10 | For details, see 4.3 0x10 Writing Multiple Registers . |

## 4 Function Code Definition

### 4.1 0x03 Reading Registers

The register read command (function code: 0x03) is a predefined function code of the open-source Modbus protocol and is used to query information about a segment of consecutive registers.

- Frame format of a master node request

| PDU Data Field | Length (Byte) | Description |
|---|---|---|
| Function code | 1 byte | 0x03 |
| Register start address | 2 bytes | 0x0000–0xFFFF |
| Number of registers | 2 bytes | 1–125 |

- Frame format of a normal response from a slave node

| PDU Data Field | Length (Byte) | Description |
|---|---|---|
| Function code | 1 byte | 0x03 |
| Byte count | 1 byte | 2 x *N* |
| Register value | 2 x *N* bytes | |

Note: *N* is the number of registers.

- Frame format of an abnormal response from a slave node

| PDU Data Field | Length (Byte) | Description |
|---|---|---|
| Error code | 1 byte | 0x83 |
| Exception code | 1 byte | For details, see 6.1 Exception Code List. |

## 4.2 0x06 Writing a Single Register

This command (function code: 0x06) is used to write a single register. It is a predefined function code of the open-source Modbus protocol and is used to set the value of a single register.

- Frame format of a request from a master node

| PDU Data Field | Length (Byte) | Description |
|---|---|---|
| Function code | 1 byte | 0x06 |
| Register address | 2 bytes | 0x0000–0xFFFF |
| Register value | 2 bytes | 0x0000–0xFFFF |

- Frame format of a normal response from a slave node

| PDU Data Field | Length (Byte) | Description |
|---|---|---|
| Function code | 1 byte | 0x06 |
| Register address | 2 bytes | 0x0000–0xFFFF |
| Register value | 2 bytes | 0x0000–0xFFFF |

- Frame format of an abnormal response from a slave node

| PDU Data Field | Length (Byte) | Description |
|---|---|---|
| Error code | 1 byte | 0x86 |

| PDU Data Field | Length (Byte) | Description |
|---|---|---|
| Exception code | 1 byte | For details, see 6.1 Exception Code List. |

## 4.3 0x10 Writing Multiple Registers

This command (function code: 0x10) is used to write multiple registers. It is a predefined function code of the open-source Modbus protocol and is used to set the value of multiple registers.

- Frame format of a request from a master node

| PDU Data Field | Length (Byte) | Description |
|---|---|---|
| Function code | 1 byte | 0x10 |
| Register start address | 2 bytes | 0x0000–0xFFFF |
| Number of registers | 2 bytes | 0x0001–0x007b |
| Byte count | 1 byte | 2 x *N* |
| Register value | 2 x *N* bytes | Value |

Note: *N* is the number of registers.

- Frame format of a normal response from a slave node

| PDU Data Field | Length (Byte) | Description |
|---|---|---|
| Function code | 1 byte | 0x10 |
| Register address | 2 bytes | 0x0000–0xFFFF |
| Number of registers | 2 bytes | 0x0000–0x007b |

- Frame format of an abnormal response from a slave node

| PDU Data Field | Length (Byte) | Description |
|---|---|---|
| Error code | 1 byte | 0x90 |
| Exception code | 1 byte | For details, see 6.1 Exception Code List. |

## 4.4 Signal List

### 4.4.1 Collected Signals

| Data Name | Register Start Address | Register End Address | Register Attribute | Data Type | Data Length | Unit | Precision | Remarks |
|---|---|---|---|---|---|---|---|---|
| Phase L1 output voltage | 4096 | 4097 | Read-only | UINT32 | 4 | V | 1 | The reported value is the actual value multiplied by 10. |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Phase L2 output voltage | 4098 | 4099 | Read-only | UINT32 | 4 | V | 1 | The reported value is the actual value multiplied by 10. |
| Phase L3 output voltage | 4100 | 4101 | Read-only | UINT32 | 4 | V | 1 | The reported value is the actual value multiplied by 10. |
| Phase L1 output current | 4102 | 4103 | Read-only | UINT32 | 4 | A | 1 | The reported value is the actual value multiplied by 10. |
| Phase L2 output current | 4104 | 4105 | Read-only | UINT32 | 4 | A | 1 | The reported value is the actual value multiplied by 10. |
| Phase L3 output current | 4106 | 4107 | Read-only | UINT32 | 4 | A | 1 | The reported value is the actual value multiplied by 10. |
| Total output power | 4108 | 4109 | Read-only | UINT32 | 4 | KW | 1 | The reported value is the actual value multiplied by 10. |

#### 4.4.2 Setting Signals

| Data Name | Register Start Address | Register End Address | Register Attribute | Data Type | Data Length | Unit | Default Value | Min. | Max. | Precision | Remarks |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Maximum charge power | 8192 | 8193 | Read and write | UINT 32 | 4 | K W | 22 | 0 | 22 | 1 | The set and reported value is the actual value multiplied by 10. |
| Charging control | 8198 | 8198 | Read and write | UINT 16 | 2 | / | 0 | 0 | 2 | 0 | [0 = On Standby] [1 = Charge Paused] [2 = Charge Resumed] |

## 5 Reference Documents

- MODBUS over Serial Line Specification and Implementation Guide V1.02
- MODBUS Messaging on TCP/IP implementation guide V1.0b
- MODBUS application protocol specification V1.1b3

## 6 Appendix

### 6.1 Exception Code List

When sending a request to the slave node, the master node expects a normal response. When the slave node responses, the following situations may occur:

- If the slave node receives the request and is able to process the request properly, the slave node returns a normal response.

- If the slave node does not receive the request due to abnormal communication, it does not return a response.
- If the slave node receives the request but detects a communication error (during parity check, LRC, CRC, ...), it does not return a response.
- If the slave receives a request without a communication error but cannot process it (e.g., if the request is to read a non-existent output or register), the master returns an exception response informing the client of the nature of the error.

An abnormal response has two fields, which are used to distinguish it from a normal response.

- Function code: indicates the function code of a request in a normal response. In an exception response, the value of this field is (function code + 0x80).
- Exception code: In an abnormal response, the server returns an exception code in the data field. The code defines the server conditions that cause the exception.

Examples of client requests and abnormal server responses

Exception codes must be unique for each network element (NE) type. The names and descriptions should be provided in the NE interface document. Different versions of the same NE type must be backward compatible. Exception codes in use cannot be assigned to other exceptions.

The following table lists the exception codes returned by NEs (0x00–0x8F are for common exception codes).

| Code | Name | Description | NMS Action |
|---|---|---|---|
| 0x01 | Invalid function | The function code received in the query is not allowable for the server (or slave node). This may be because the function code is only applicable to newer devices, and cannot be implemented in the unit selected. It also indicates that the server (or slave node) is in the wrong state to process a request of this type, for example because it is not configured and is being asked to return register values. | |
| 0x02 | Invalid data address | The data address received in the query is not an allowable address for the server (or slave node). More specifically, the combination of the reference number and transfer length is invalid. For a controller with 100 registers, a request with an offset of 96 and a length of 4 is successfully executed, and a request with an offset of 96 and a length of 5 is responded with the error code 02. | |
| 0x03 | Invalid data value | The value contained in the query data field is not an allowable value for the server (or slave node). The value indicates a fault in the structure of the remainder of a complex | |

| Code | Name | Description | NMS Action |
|---|---|---|---|
| | | request, such as an incorrectly implied length. It does not mean that a data item submitted for storage in a register has a value outside the expectation of the application program since the Modbus protocol is unaware of the significance of any particular value of any particular register. | |
| 0x04 | Slave device failure | An error occurs while the server attempts to perform the requested action. | |
| 0x05 | Confirm ation | The server has accepted the request and is processing it, but a long duration of time will be required to do so. This response is returned to confirm the acceptance of the request. | |
| 0x06 | Slave device busy | The server cannot accept a Modbus request PDU. The client application determines whether and when to retransmit the request. | (1) File upload<br>Upon receiving this exception code, the NMS resends the command once every 10 seconds for a maximum of six consecutive times.<br>(2) File load startup<br>Upon receiving this exception code, the NMS resends the request once every 20s for three consecutive times. If the exception code persists, the upgrade process is regarded as a failure.<br>(3) File load data:<br>Upon receiving this exception code, the NMS resends the data once every 20s for three consecutive times. |
| 0x08 | Memory parity error | Used in conjunction with function codes 20 and 21 and reference type 6 to indicate that the extended file area failed to pass a consistency check. The server (or slave node) attempted to read a record file, but detected a parity error in the memory. The client (master node) can retry the request, but a service may be required on the server (or slave node). | Upon receiving this exception code, the NMS resends a command for a maximum of three consecutive times. |
| 0x0A | Gateway path unavaila | Applies to the TCP/IP protocol. | |

| Code | Name | Description | NMS Action |
| --- | --- | --- | --- |
| | ble | | |
| 0x0B | Gateway target device failed to respond | Applies to the TCP/IP protocol. | |
| 0x80 | No permission | An operation is not allowed because of a permission authentication failure or permission expiration. | The NMS needs to re-authenticate the device. |

[Note] The preceding exception code definitions are mainly used for the predefined function codes (0x03, 0x06, and 0x10) of the open-source Modbus. For the function codes defined in Huawei Modbus2.0 protocol, the response packets of the command can better indicate the cause of the exception.

## 6.2 Modbus CRC

The CRC code consists of 16 bits and applies to all bytes in front of it. The reference code is as follows:

static unsigned char auchCRCHi[] = {

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 values for the low-order byte*/
static char auchCRCLo[] = {
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
};
unsigned short CRC16 ( puchMsg, usDataLen ) /* The function returns the CRC as a unsigned short
type */
unsigned char *puchMsg ; /* message to calculate CRC upon */
unsigned short usDataLen ; /* quantity of bytes in message */
{
unsigned char uchCRCHi = 0xFF ; /* high byte of CRC initialized */
unsigned char uchCRCLo = 0xFF ; /* low byte of CRC initialized */
unsigned uIndex ; /* will index into CRC lookup table */
while (usDataLen--) /* pass through message buffer */
{
uIndex = uchCRCLo ^ *puchMsg++ ; /* calculate the CRC */
uchCRCLo = uchCRCHi ^ auchCRCHi[uIndex] ;
uchCRCHi = auchCRCLo[uIndex] ;
}
return (uchCRCHi << 8 | uchCRCLo) ;
}
```

Code source: *MODBUS over Serial Line Specification and Implementation Guide V1.02*

## 6.3 Endian Definition

- Big Endian: The most significant bytes are stored first at the lowest storage addresses. For example, the big-endian bytes of the integer 0x12345678 are expressed as follows:

| 0x12 | 0x34 | 0x56 | 0x78 |
|---|---|---|---|

[Description] The network byte order is big-endian.

- Little Endian: The most significant bytes are stored at the highest storage addresses. (The least significant bytes are stored first.)

For example, the little-endian bytes of the integer 0x12345678 are expressed as follows:

| 0x78 | 0x56 | 0x34 | 0x12 |
|---|---|---|---|