REAL TIME AUTOMATION

# Modbus TCP Unplugged

## An Introduction to Modbus TCP Networking

PREPARED BY REAL TIME AUTOMATION
WWW.RTAAUTOMATION.COM

# MODBUS TCP Unplugged

## An Introduction to Modbus TCP Networking

## Introduction

MODBUS TCP is a variant of the MODBUS family of simple, vendor-neutral communication protocols intended for supervision and control of automation equipment. Specifically, it covers the use of MODBUS messaging in an 'Intranet' or 'Internet' environment using the TCP/IP protocols. The most common use of the protocols at this time are for Ethernet attachment of PLCs, I/O modules, and "gateways" to other simple field buses or I/O networks.

The MODBUS TCP protocol was published as a de-facto automation standard. Since MODBUS is already widely known, there should be little information in this document that could not be obtained elsewhere. However, an attempt has been made to clarify which functions within MODBUS have value for interoperability of general automation equipment, and which parts are "baggage" from the alternate use of MODBUS as a programming protocol for PLCs.

This is done by grouping supported message types into conformance classes, which differentiate between those messages that are universally implemented and those that are optional, particularly those specific to devices such as PLCs.

### Connection-Oriented

**Rocco wonders:**

**Why TCP over UDP?**

In MODBUS, data transactions are traditionally stateless, making them highly resistant to disruption from noise and yet requiring minimal recovery information to be maintained at either end.

Programming operations, on the other hand, expect a connection-oriented approach. This was achieved on the simpler variants by an exclusive 'login' token, and on the MODBUS Plus variant by explicit 'Program Path' capabilities which maintained a duplex association until explicitly broken down.

MODBUS TCP handles both situations. A connection is easily recognized at the protocol level, and a single connection may carry multiple independent transactions. In addition, MODBUS TCP allows a very large number of concurrent connections, so in most cases it is the initiator's choice whether to reconnect as required or reuse a long-lived connection.

Developers familiar with MODBUS may wonder why the connection-oriented TCP protocol is used rather than the datagram-oriented UDP. The main reason is to keep control of an individual transaction by enclosing it in a connection that can be identified, supervised, and canceled without requiring specific action on the part of the client and server applications. This gives the mechanism a wide tolerance to network performance changes and allows security features like firewalls and proxies to be easily added.

The original developers of the World Wide Web used similar reasoning when they chose to implement a minimal Web query as a single transaction using TCP on well-known port 80.

## Data Encoding

MODBUS uses a 'big-endian' representation for addresses and data items. This means that when a numerical quantity larger than a single byte is transmitted, the MOST significant byte is sent first. So for example:

- **16 - bits 0x1234 would be 0x12 0x34**
- **32 - bits 0x12345678L would be 0x12 0x34 0x56 0x78**

## Interpretation of Reference Numbers

MODBUS bases its data model on a series of tables which have distinguishing characteristics. The four primary tables are:

- **input discrete — single bit, provided by an I/O system, read-only**
- **output discrete — single bit, alterable by an application program, read-write**
- **input registers — 16-bit quantity, provided by an I/O system, read-only**
- **output registers — 16-bit quantity, alterable by an application program, read-write**

The distinction between inputs and outputs, and between bit-addressable and word-addressable data items, does not imply any application behavior. It is perfectly acceptable, and very common, to regard all four tables as overlaying one another if this is the most natural interpretation on the target machine in question.

For each of the primary tables, the protocol allows individual selection of 65,536 data items, and the operations of read or write of those items are designed to span multiple consecutive data items up to a data size limit that is dependent on the transaction function code.

There is no assumption that the data items represent a contiguous array of data, although that is the interpretation used by most simple PLCs.

The 'read and write general reference' function codes are defined to carry a 32 bit reference number and could be used to allow direct access to data items within a very large space. Today, there are no PLC devices that take advantage of that.

One potential source of confusion is the relationship between the reference numbers used in MODBUS functions and the register numbers used in Modicon PLCs. For historical reasons, user reference numbers were expressed as decimal numbers with a starting offset of 1. However, MODBUS uses the more natural software interpretation of an unsigned integer index starting at zero.

So a MODBUS message requesting the read of a register at offset 0 would return the value known to the application programmer as found in register 4:00001. (memory type 4 = output register, reference 00001)

## Implied Length Philosophy

All MODBUS requests and responses are designed in such a way that the recipient can verify that a message is complete. For function codes where the request and response are of fixed length, the function code alone is sufficient. For function codes carrying a variable amount of data in the request or response, the data portion will be preceded by a byte count.

When MODBUS is carried over TCP, additional length information is carried in the prefix to allow the recipient to recognize message boundaries even if the message had to be split into multiple packets for transmission. The existence of explicit and implicit length rules, and the use of a CRC-32 error check code (on Ethernet), results in an infinitesimal chance of undetected corruption to a request or response message.

# Conformance Class Summary

When defining a new protocol from scratch, it is possible to enforce consistency of numbering and interpretation. MODBUS, by its nature, is implemented already in many places, and disruption to existing implementations must be avoided.

Therefore, the existing set of transaction types have been classified into conformance classes where level 0 represents functions that are universally implemented and totally consistent and level 2 represents useful functions with some idiosyncrasies. Those functions of the present set that are not suitable for interoperability are also identified.

Note that future extensions to this standard may define additional function codes to handle situations where the existing de-facto standard is deficient. However, it would be misleading for details of such proposed extensions to appear in this document. It will always be possible to determine if a particular target device supports a particular function code by sending it speculatively and checking for the type of exception response, if any. This approach will guarantee the continued interoperability of current MODBUS devices with the introduction of any such extensions. Indeed, this is the philosophy that has led to the current function code classification.

## Class 0

This is the minimum useful set of functions for both a MASTER and a SLAVE.

- **read multiple registers (fc 3)**
- **write multiple registers (fc 16)**

## Class 1

This function typically has a different meaning for each slave family.

- **read coils (fc 1)**
- **read input discretes (fc 2)**
- **read input registers (fc 4)**
- **write coil (fc 5)**
- **write single register (fc 6)**
- **read exception status (fc 7)**

## Class 2

These are the data transfer functions needed for routine operations such as HMI and supervision.

- **force multiple coils (fc 15)**
- **read general reference (fc 20)**

This function has the ability to handle multiple simultaneous requests, and can accept a reference number of 32 bits. Current 584 and 984 PLC's only use this function to accept references of type 6 (extended register files). This function would be the most appropriate to extend to handle large register spaces and data items which currently lack reference numbers such as un-located variables.

- **write general reference (fc 21)**

This function has the ability to handle multiple simultaneous requests, and can accept a reference number of 32 bits. Current 584 and 984 PLC's only use this function to accept references of type 6 (extended register files). This function would be the most appropriate to extend to handle large register spaces and data items which currently lack reference numbers such as un-located variables.

- **mask write register (fc 22)**
- **read/write registers (fc 23)**

This function allows the input of a range of registers and the output of a range of registers as a single transaction. It is the most efficient way, using MODBUS, to perform a regular exchange of a state image, such as with an I/O module. Thus, a high performance but versatile data collection device might choose to implement functions 3, 16 and 23 to combine rapid regular exchange of data (23) with the ability to perform on-demand interrogations or updates of particular data items (3 and 16)

- **read FIFO queue (fc 24)**

This function is a somewhat specialized function intended to allow the transfer of data from a table structured as a FIFO (for use with the FIN and FOUT function blocks on the 584/984) to a host computer. It is useful in certain types of event-logging applications.

## Machine/Vendor/Network-Specific Functions

All of the following functions, although mentioned in the MODBUS protocol manuals, are not appropriate for interoperability purposes because they are too machine-dependent.

- **diagnostics (fc 8)**
- **program (484) (fc 9)**
- **poll (484) (fc 10)**
- **get comm event counters (MODBUS) (fc 11)**
- **get comm event log (MODBUS) (fc 12)**
- **program (584/984) (fc 13)**
- **poll (584/984) (fc 14)**
- **report slave ID (fc 17)**
- **program (884/u84) (fc 18)**
- **reset comm link (884/u84) (fc 19)**
- **program (ConCept) (fc 40)**
- **firmware replacement (fc 125)**
- **program (584/984) (fc 126)**
- **report local address (MODBUS) (fc 127)**

# Protocol Structure

This section describes the general form of encapsulation of a MODBUS request or response when carried on the MODBUS network. It is important to note that the structure of the request and response body, from the function code to the end of the data portion, have exactly the same layout and meaning as in the other MODBUS variants, such as:

- **MODBUS serial port - ASCII encoding**
- **MODBUS serial port - RTU (binary) encoding**
- **MODBUS PLUS network - data path**

The only differences in these cases are the forms of any framing sequence, error check pattern, and address interpretation. All requests are sent via TCP on registered port 502.

Requests are normally sent in half-duplex fashion on a given connection. That is, there is no benefit to sending additional requests on a single connection while a response is outstanding. Devices that wish to obtain high peak transfer rates are instead encouraged to establish multiple TCP connections to the same target. Some existing client devices are known to attempt to 'pipeline' requests.

The MODBUS slave address field is replaced by a single byte Unit Identifier that may be used to communicate via devices such as bridges and gateways, which use a single IP address to support multiple independent end units.

The request and response are prefixed by six bytes.

- **byte 0: transaction identifier — copied by server, usually 0**
- **byte 1: transaction identifier — copied by server, usually 0**
- **byte 2: protocol identifier — 0**
- **byte 3: protocol identifier — 0**
- **byte 4: length field (upper byte) — 0 (all messages are smaller than 256)**
- **byte 5: length field (lower byte) — number of bytes following**
- **byte 6: unit identifier — previously slave address**
- **byte 7: MODBUS function code**
- **byte 8+: data as needed**

So an example transaction to read 1 register at offset 4 from UI 9 returning a value of 5 would be:

- **request: 00 00 00 00 00 06 09 03 00 04 00 01**
- **response: 00 00 00 00 00 05 09 03 02 00 05**

Keep reading for examples of the use of each of the function codes in conformance classes 0-2.

Designers familiar with MODBUS should note that the CRC-16 or LRC check fields are NOT needed in MODBUS. The TCP/IP and link layer (e.g.. Ethernet) checksum mechanisms are used to verify accurate delivery of the packet.

# Protocol Reference by Conformance Class

Note that in these examples, the request and response are listed from the function code byte onwards. As said before, there will be a transport-dependent prefix which, in the case of MODBUS, comprises the seven bytes

- **ref ref 00 00 00 len unit**

The “ref ref” above is two bytes of transaction reference number that have no value at the server but are copied verbatim from request to response for the convenience of the client. Simple clients usually choose to leave the values at zero.

In the examples, the format for a request and response is given as follows (the example is for a ‘read register’ request, see detail in later section):

- **03 00 00 00 01 => 03 02 12 34**

This represents a hexadecimal series of bytes to be appended to the prefix, so the full message on the TCP connection, assuming unit identifier 09 again, would be:

- **request: 00 00 00 00 00 06 09 03 00 00 00 01**
- **response: 00 00 00 00 00 05 09 03 02 12 34**

All of these requests and responses were verified by using an automatic tool querying a current specification Modicon Quantum PLC.

## Class 0 Commands Detail

### Read multiple registers (FC 3)

**Request**

Byte 0: FC = 03

Byte 1-2: Reference number

Byte 3-4: Word count (1-125)

**Response**

Byte 0: FC = 03

Byte 1: Byte count of response (B=2 x word count)

Byte 2-(B+1): Register values

**Exceptions**

Byte 0: FC = 83 (hex)

Byte 1: exception code = 01 or 02

**Example**

Read 1 register at reference 0 (40001 in Modicon 984) resulting in value 1234 hex

03 00 00 00 01 => 03 02 12 34

### Write multiple registers (FC 16)

**Request**

Byte 0: FC = 10 (hex)

Byte 1-2: Reference number

Byte 3-4: Word count (1-100)

Byte 5: Byte count (B=2 x word count)

Byte 6-(B+5): Register values

**Response**

Byte 0: FC = 10 (hex)

Byte 1-2: Reference number

Byte 3-4: Word count

**Exceptions**

Byte 0: FC = 90 (hex)

Byte 1: exception code = 01 or 02

**Example**

Write 1 register at reference 0 (40001 in Modicon 984) of value 1234 hex

10 00 00 00 01 02 12 34 => 10 00 00 00 01

## Class 1 Commands Detail

### Read coils (FC 1)

**Request**

Byte 0: FC = 01

Byte 1-2: Reference number

Byte 3-4: Bit count (1-2000)

**Response**

Byte 0: FC = 01

Byte 1: Byte count of response (B=(bit count+7)/8)

Byte 2-(B+1): Bit values (least significant bit is first coil!)

**Exceptions**

Byte 0: FC = 81 (hex)

Byte 1: exception code = 01 or 02

**Example**

Read 1 coil at reference 0 (00001 in Modicon 984) resulting in value 1

01 00 00 00 01 => 01 01 01

Note that the format of the return data is not consistent with a big-endian architecture. Note also that this request can be very computation-intensive on the slave if the request calls for multiple words and they are not aligned on 16-bit boundaries.

## Read input discretes (FC 2)

**Request**

Byte 0: FC = 02

Byte 1-2: Reference number

Byte 3-4: Bit count (1-2000)

**Response**

Byte 0: FC = 02

Byte 1: Byte count of response (B=(bit count+7)/8)

Byte 2-(B+1): Bit values (least significant bit is first coil!)

**Exceptions**

Byte 0: FC = 82 (hex)

Byte 1: exception code = 01 or 02

**Example**

Read 1 discrete input at reference 0 (10001 in Modicon 984) resulting in value 1

02 00 00 00 01 => 02 01 01

Note that the format of the return data is not consistent with a big-endian architecture. Note also that this request can be very computation-intensive on the slave if the request calls for multiple words and they are not aligned on 16-bit boundaries.

## Read input registers (FC 4)

**Request**

Byte 0: FC = 04

Byte 1-2: Reference number

Byte 3-4: Word count (1-125)

**Response**

Byte 0: FC = 04

Byte 1: Byte count of response (B=2 x word count)

Byte 2-(B+1): Register values

**Exceptions**

Byte 0: FC = 84 (hex)

Byte 1: exception code = 01 or 02

**Example**

Read 1 input register at reference 0 (30001 in Modicon 984) resulting in value 1234 hex

04 00 00 00 01 => 04 02 12 34

### Write coil (FC 5)

**Request**

Byte 0: FC = 05

Byte 1-2: Reference number

Byte 3: = FF to turn coil ON, =00 to turn coil OFF

Byte 4: = 00

**Response**

Byte 0: FC = 05

Byte 1-2: Reference number

Byte 3: = FF to turn coil ON, =00 to turn coil OFF (echoed)

Byte 4: = 00

**Exceptions**

Byte 0: FC = 85 (hex)

Byte 1: exception code = 01 or 02

**Example**

Write 1 coil at reference 0 (00001 in Modicon 984) to the

value 1

05 00 00 FF 00 => 05 00 00 FF 00

### Write single register (FC 6)

**Request**

Byte 0: FC = 06

Byte 1-2: Reference number

Byte 3-4: Register value

**Response**

Byte 0: FC = 06

Byte 1-2: Reference number

Byte 3-4: Register value

**Exceptions**

Byte 0: FC = 86 (hex)

Byte 1: exception code = 01 or 02

**Example**

Write 1 register at reference 0 (40001 in Modicon 984) of value 1234 hex

06 00 00 12 34 => 06 00 00 12 34

### Read exception status (FC 7)

Note that 'exception status' has nothing to do with 'exception response'. The 'read exception status' message was intended to allow maximum responsiveness in the early MODBUS polled multidrop networks using slow baud rates. PLC's would typically map a range of 8 coils (output discretes) which would be interrogated using this message.

**Request**

Byte 0: FC = 07

**Response**

Byte 0: FC = 07

Byte 1: Exception status (usually a predefined range of 8 coils)

**Exceptions**

Byte 0: FC = 87 (hex)

Byte 1: exception code = 01 or 02

**Example**

Read exception status resulting in value 34 hex

07 => 07 34

## Class 2 Commands Detail

### Force multiple coils (FC 15)

**Request**

Byte 0: FC = 0F (hex)

Byte 1-2: Reference number

Byte 3-4: Bit count (1-800)

Byte 5: Byte count (B = (bit count + 7)/8)

Byte 6-(B+5): Data to be written (least significant bit = first coil)

**Response**

Byte 0: FC = 0F (hex)

Byte 1-2: Reference number

Byte 3-4: Bit count

**Exceptions**

Byte 0: FC = 8F (hex)

Byte 1: exception code = 01 or 02

**Example**

Write 3 coils at reference 0 (00001 in Modicon 984) to values 0,0,1

0F 00 00 00 03 01 04 => 0F 00 00 00 03

Note that the format of the input data is not consistent with a big-endian architecture. Note also that this request can be very computation-intensive on the slave if the request calls for multiple words and they are not aligned on 16-bit boundaries.

## Read general reference (FC 20)

**Request**

Byte 0: FC = 14 (hex)

Byte 1: Byte count for remainder of request (=7 x number of groups)

Byte 2: Reference type for first group = 06 for 6xxxx extended register files

Byte 3-6: Reference number for first group

= file number:offset for 6xxxx files

= 32 bit reference number for 4xxxx registers

Byte 7-8: Word count for first group

Bytes 9-15: (as for bytes 2-8, for 2nd group)

**Response**

Byte 0: FC = 14 (hex)

Byte 1: Overall byte count of response

(=number of groups + sum of byte counts for groups)

Byte 2: Byte count for first group (B1=1 + (2 x word count))

Byte 3: Reference type for first group

Byte 4-(B1+2): Register values for first group

Byte (B1+3): Byte count for second group (B2=1 + (2 x word count))

Byte (B1+4): Reference type for second group

Byte (B1+5)-(B1+B2+2): Register values for second group

**Exceptions**

Byte 0: FC = 94 (hex)

Byte 1: exception code = 01 or 02 or 03 or 04

**Example**

Read 1 extended register at reference 1:2 (File 1 offset 2 in Modicon 984) resulting in value 1234 hex

14 07 06 00 01 00 02 00 01 => 14 04 03 06 12 34

(future)

Read 1 register at reference 0 returning 1234 hex, and 2 registers at reference 5 returning 5678 and 9abc hex

14 0E 04 00 00 00 00 00 01 04 00 00 00 05 00 02 => 14 0A 03 04 12 34 05 04 56 78 9A BC

Note that the transfer size limits are difficult to define in a mathematical formula. Broadly, the message sizes for request and response are each limited to 256 bytes for buffer size reasons, and the aggregate size of the individual request and response data frames must be considered. Exception type 04 will be generated if the slave is unwilling to process the message because the response would be too large.

### Write general reference (FC 21)

**Request**

Byte 0: FC = 15 (hex)

Byte 1: Byte count for remainder of request

Byte 2: Reference type for first group = 06 for 6xxxx extended register files

Byte 3-6: Reference number for first group

= file number:offset for 6xxxx files

= 32 bit reference number for 4xxxx registers

Byte 7-8: Word count for first group (W1)

Byte 9-(8 + 2 x W1): Register data for first group

(copy group data frame from byte 2 on for any other groups)

**Response**

Response is a direct echo of the query

Byte 0: FC = 15 (hex)

Byte 1: Byte count for remainder of request

Byte 2: Reference type for first group = 06 for 6xxxx extended register files

Byte 3-6: Reference number for first group

= file number:offset for 6xxxx files

= 32 bit reference number for 4xxxx registers

Byte 7-8: Word count for first group (W1)

Byte 9-(8 + 2 x W1): Register data for first group

(copy group data frame from byte 2 on for any other groups)

**Exceptions**

Byte 0: FC = 95 (hex)

Byte 1: exception code = 01 or 02 or 03 or 04

**Example**

Write 1 extended register at reference 1:2 (File 1 offset 2 in Modicon 984) to value 1234 hex

15 09 06 00 01 00 02 00 01 12 34 => 15 09 06 00 01 00 02 00 01 12 34

(future)

Write 1 register at reference 0 to value 1234 hex, and 2 registers at reference 5 to values 5678 and 9abc hex

15 14 04 00 00 00 00 00 01 12 34 04 00 00 00 05 00 02 56 78 9A BC

=>15 14 04 00 00 00 00 00 01 12 34 04 00 00 00 05 00 02 56 78 9A BC

Note that the transfer size limits are difficult to define in a mathematical formula. Broadly, the message sizes for request and response are each limited to 256 bytes for buffer size reasons, and the aggregate size of the individual request and response data frames must be considered. Exception type 04 will be generated if the slave is unwilling to process the message because the response would be too large.

### Mask write register (FC 22)

**Request**

Byte 0: FC = 16 (hex)

Byte 1-2: Reference number

Byte 3-4: AND mask to be applied to register

Byte 5-6: OR mask to be applied to register

**Response**

Byte 0: FC = 16 (hex)

Byte 1-2: Reference number

Byte 3-4: AND mask to be applied to register

Byte 5-6: OR mask to be applied to register

**Exceptions**

Byte 0: FC = 96 (hex)

Byte 1: exception code = 01 or 02

**Example**

Change the field in bits 0-3 of register at reference 0 (40001 in Modicon 984) to value 4 hex

(AND with 000F, OR with 0004)

16 00 00 00 0F 00 04 => 16 00 00 00 0F 00 04

### Read/write registers (FC 23)

**Request**

Byte 0: FC = 17 (hex)

Byte 1-2: Reference number for read

Byte 3-4: Word count for read (1-125)

Byte 5-6: Reference number for write

Byte 7-8: Word count for write (1-100)

Byte 9: Byte count (B = 2 x word count for write)

Byte 10-(B+9): Register values

**Response**

Byte 0: FC = 17 (hex)

Byte 1: Byte count(B = 2 x word count for read)

Byte 2-(B+1) Register values

**Exceptions**

Byte 0: FC = 97 (hex)

Byte 1: exception code = 01 or 02

**Example**

Write 1 register at reference 3 (40004 in Modicon 984) of value 0123 hex and read 2 registers at reference 0 returning values 0004 and 5678 hex

17 00 00 00 02 00 03 00 01 02 01 23 => 17 04 00 04 56 78

Note that if the register ranges for writing and reading overlap, the results are undefined. Some devices implement the write before the read, but others implement the read before the write.

### Read FIFO queue (FC 24)

**Request**

Byte 0: FC = 18 (hex)

Byte 1-2: Reference number

**Response**

Byte 0: FC = 18 (hex)

Byte 1-2: Byte count (B = 2 + word count) (maximum 64)

Byte 3-4: Word count (number of words accumulated in FIFO) (maximum 31)

Byte 5-(B+2): Register data from front of FIFO

**Exceptions**

Byte 0: FC = 98 (hex)

Byte 1: exception code = 01 or 02 or 03

**Example**

Read contents of FIFO block starting at reference 0005 (40006 in Modicon 984) which contains 2 words of value 1234 and 5678 hex outstanding

18 00 05 => 18 00 06 00 02 12 34 56 78

Note that this function as implemented on the 984 is very limited in versatility - the block of registers is assumed to consist of a count which can have values from 0 to 31, followed by up to 31 words of data. When the function completes, the count word is NOT reset to zero, as might have been expected from a FIFO operation.

All in all, this should be considered a limited subset of fn 16 - read multiple registers, since the latter can be used to perform all of the required functionality.

## Exception Codes

There is a defined set of exception codes to be returned by slaves in the event of problems. Note that masters may send out commands ‘speculatively’, and use the success or exception codes received to determine which MODBUS commands the device is willing to respond to and to determine the size of the various data regions available on the slave.

All exceptions are signaled by adding 0x80 to the function code of the request, and following this byte by a single reason byte for example as follows:

03 12 34 00 01 => 83 02

request read 1 register at index 0x1234 response exception type 2 - ‘illegal data address’

The list of exceptions follows:

**01 ILLEGAL FUNCTION**

The function code received in the query is not an allowable action for the slave. This may be because the function code is only applicable to newer controllers, and was not implemented in the unit selected. It could also indicate that the slave is in the wrong state to process a request of this type, for example because it is unconfigured and is being asked to return register values.

**02 ILLEGAL DATA ADDRESS**

The data address received in the query is not an allowable address for the slave. More specifically, the combination of reference number and transfer length is invalid. For a controller with 100 registers, a request with offset 96 and length 4 would succeed, a request with offset 96 and length 5 will generate exception 02.

**03 ILLEGAL DATA VALUE**

A value contained in the query data field is not an allowable value for the slave. This indicates a fault in the structure of the remainder of a complex request, such as that the implied length is incorrect. It specifically 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.

**04 ILLEGAL RESPONSE LENGTH**

Indicates that the request as framed would generate a response whose size exceeds the available MODBUS data size. Used only by functions generating a multi-part response, such as functions 20 and 21.

**05 ACKNOWLEDGE**

Specialized use in conjunction with programming commands

**06 SLAVE DEVICE BUSY**

Specialized use in conjunction with programming commands

**07 NEGATIVE ACKNOWLEDGE**

Specialized use in conjunction with programming commands

**08 MEMORY PARITY ERROR**

Specialized use in conjunction with function codes 20 and 21, to indicate that the extended file area failed to pass a consistency check.

**0A GATEWAY PATH UNAVAILABLE**

Specialized use in conjunction with MODBUS Plus gateways, indicates that the gateway was unable to allocate a MODBUS Plus PATH to use to process the request. Usually means that the gateway is misconfigured.

**0B GATEWAY TARGET DEVICE FAILED TO RESPOND**

Specialized use in conjunction with MODBUS Plus gateways, indicates that no response was obtained from the target device. Usually means that the device is not present on the network.