OPEN MODBUS/TCP SPECIFICATION

Release 1.0, 29 March 1999

Andy Swales
Schneider Electric

aswales@modicon.com

Contents

1. Status of this specification

Initial release 3 Sept 1997

Draft for public review.

Re-release 29 March 1999 at Revision 1.0.

No technical changes, clarifications only.
Added Appendices A and B in response to common implementation questions.

This specification of MODBUS/TCP is being published by being publicly visible on the World Wide Web. It is intended for the benefit of developers wishing to use MODBUS/TCP as an interoperability standard in the field of industrial automation.

Since MODBUS and MODBUS/TCP are in reality 'de-facto' standards, in that many vendors and products implement it already, this specification primarily explains the specific encoding of MODBUS messages over the TCP communication protocol universally available on the Internet.

2. Overview

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 PLC's, I/O modules, and 'gateways' to other simple field buses or I/O networks.

The MODBUS/TCP protocol is being published as a ('de-facto') automation standard. Since MODBUS is already widely known, there should be little information in this document which 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 PLC's.

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

2.1 Connection-oriented

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, TCP allows a very large number of concurrent connections, so in most cases it is the choice of the initiator whether to reconnect as required or re-use 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 which 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 such as firewalls and proxies to be easily added.

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

2.2 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

2.3 Interpretation of reference numbers

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

input discretes	single bit, provided by an I/O system, read-only
output discretes	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, do 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 65536 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 which is dependent on the transaction function code.

There is no assumption that the data items represent a true contiguous array of data, although that is the interpretation used by most simple PLC's

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 which 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 PLC's. 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)

2.4 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 use of a CRC-32 error check code (on Ethernet) results in an infinitesimal chance of undetected corruption to a request or response message.

3. 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 which are universally implemented and totally consistent, and level 2 represents useful functions but with some idiosyncrasies. Those functions of the present set which are NOT suitable for interoperability are also identified.

It must be noted 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, and this approach will guarantee the continued interoperability of current MODBUS devices with the introduction of any such extensions. Indeed, this is the philosophy which has led to the current function code classification.

3.1 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)

3.2 Class 1

This is the additional set of functions which is commonly implemented and interoperable. As explained before, many slaves choose to treat input, output, discrete and register as equivalent.

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)

This function typically has a different meaning for each slave family

3.3 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 'unlocated' 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 'unlocated' 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)

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. Useful in certain types of event logging applications

3.4 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)

4. Protocol structure

This section describes the general form of encapsulation of a MODBUS request or response when carried on the MODBUS/TCP 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 other cases are the form 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 in sending additional requests on a single connection while a response is outstanding. Devices which wish to obtain high peak transfer rates are instead encouraged to establish multiple TCP connections to the same target However some existing client devices are known to attempt to 'pipeline' requests. Design techniques which allow a server to accommodate this behavior are described in Appendix A.

The MODBUS 'slave address' field is replaced by a single byte 'Unit Identifier' which 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 as follows

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 (since 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 on: data as needed

So an example transaction '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

See later section 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/TCP. The TCP/IP and link layer (eg. Ethernet) checksum mechanisms instead are used to verify accurate delivery of the packet.

5. Protocol reference by conformance class

Note that in the 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/TCP comprises the seven bytes

ref ref 00 00 00 len unit

The 'ref ref' above is two bytes of 'transaction reference' number which 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 like this (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 would be (assume unit identifier 09 again)

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)

5.1 Class 0 commands detail

5.1.1 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

5.1.2 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

5.2 Class 1 commands detail

5.2.1 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.

5.2.2 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.

5.2.3 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

5.2.4 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

5.2.5 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

5.2.6 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

5.3 Class 2 commands detail

5.3.1 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.

5.3.2 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.

5.3.3 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.

5.3.4 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

5.3.5 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.

5.3.6 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.

6. 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 FUNCTlON

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 DEVlCE BUSY

Specialized use in conjunction with programming commands

07 NEGATlVE 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.

Appendices

A. Client and Server Implementation Guidance

The comments in this section should not be regarded as binding upon any particular implementation of a client or server. However, if followed, these policies will minimize integration 'surprises' when implementing multi-vendor systems and gateways to installed MODBUS equipment.

The software structure below assumes familiarity with the BSD Sockets service interface, as used on for example UNIX and Windows NT.

A.1 Client design

MODBUS/TCP is designed to allow the design of a client to be as simple as possible. Examples of software are given elsewhere, but the basic process of handling a transaction is as follows

Establish a TCP connection to port 502 at the desired server using connect()

Prepare a MODBUS request, encoded as described before

Submit the MODBUS request, including its 6-byte MODBUS/TCP prefix, as a single buffer to be transmitted using send()

Wait for a response to appear on the same TCP connection. Optionally, run a timeout on this step, using select(), if you wish to be advised of communication problems faster than TCP would normally report.

Read, using recv(), the first 6 bytes of the response, which will indicate the actual length of the response message

Use recv() to read the remaining bytes of the response.

If no further communication is expected to this particular target in the immediate future, close down the TCP connection so that the resources at the server can be used in the interim to serve other clients. A time of 1 second is suggested as the maximum period to leave a connection open at the client.

In the event of a timeout waiting for a response, issue a unilateral close of the connection, open up a new one, and resubmit the request. This technique allows the client control of retry timing which is superior to that provided by default by TCP. It also allows for alternate fallback strategies, such as submitting the

request to an alternate IP address, using a totally independent communication network, in case of failure of a network infrastructure component.

A.2 Server design

A MODBUS/TCP server should always be designed to support multiple concurrent clients, even if in its intended use only a single client appears to make sense. This allows a client to close and reopen the connection in rapid sequence in order to respond quickly to non-delivery of a response.

If a conventional TCP protocol stack is used, significant memory resources can be saved by reducing the receive and transmit buffer sizes. A normal TCP service on UNIX or NT would usually allocate 8K bytes or more as a per-connection receive buffer in order to encourage 'streamed' transfer of data from for example file servers. Such buffer space has no value in MODBUS/TCP, since the maximum size of a request or response is less than 300 bytes. It is often possible to trade the storage space for additional connection resources.

Either a multithreaded or single-threaded model can be used to handle the multiple connections. Descriptions follow in the next sections.

A.2.1 Multithreaded server

Operating systems or languages which encourage the use of multiple threads, such as JAVA, can use the multithreaded strategy, described here:

Use listen() to wait for incoming connections on TCP port 502

When a new connection request is received, use accept() to accept it and spawn a new thread to handle the connection

Within the new thread, do the following in an infinite loop:

> Issue a recv(6) request for the 6 byte MODBUS/TCP header. Do not place a timeout here, but instead be willing to wait until either a request comes through or the connection is closed. Both situations will wake up the thread automatically.
>
> Analyze the header. If it appears corrupt, for example the protocol field is non-zero or the length of message is larger than 256, then UNILATERALLY CLOSE THE CONNECTION. This is the correct response as a server to a situation implying the TCP encoding is incorrect.
>
> Issue a recv() for the remaining bytes of the message, whose length is now known. Note in particular that issuing a recv() with a limit like this on the length will tolerate clients who insist on 'pipelining' requests. Any such pipelined requests would remain in the TCP buffers at either server or client, and be picked up later, when the current request has completed service.
>
> Now process the incoming MODBUS message, if necessary suspending the current thread until the correct response can be calculated. Eventually you will have either a valid MODBUS message or an EXCEPTION message to use as a response
>
> Generate the MODBUS/TCP prefix for the response, copying the 'transaction identifier' field from bytes 0 and 1 of the request, and recalculating the length field.

Submit the response, including the MODBUS/TCP prefix, as a single buffer for transmission on the connection, using send()

Go back and wait for the next 6 byte prefix record.

Eventually, when the client elects to close the connection, the recv() of the 6 – byte prefix will fail. An orderly close will usually result in a recv() with a zero return byte count. A force close may generate an error return from the recv(). In either case, close the connection and cancel the current thread.

A.2.2 Single-threaded server

Some embedded systems and older operating systems such as UNIX and MS-DOS encourage the handling of multiple connections using the 'select' call from the sockets interface. In such a system, instead of handling the processing of individual concurrent requests in their own thread, you can handle the requests as multiple state machines within a common handler. Languages such as C++ make the structure of software like this convenient.

The structure now would be as follows

Initialize multiple state machines by setting their state to 'idle'

listen() for incoming connections on TCP port 502

Now start an infinite loop checking the 'listen' port and the state machines as follows:

On the listen port, if a new connection request is received, use accept() to accept it and cause one of the state machines to change state from 'idle' to 'new request' to process the incoming connection

For each of the state machines

If state is 'new request':

Use select() to see whether a request has arrived. Normally set the timeout to zero, since you don't wish to suspend the process because of inactivity on this particular connection.

If select() indicates there is a packet, use recv(6) to read the header as in the multithreaded case. If the header is corrupt, CLOSE THE CONNECTION and set the state machine to idle.

If the read succeeded and select() indicates that more input is available, read the rest of the request.

If the request is complete, change the state of the session to 'await response'.

If the recv() returns indicating that the connection is no longer in use, close the connection and reset the state machine to 'idle'.

If state is 'await response'

See if the application response information is available, if it is, build up the response packet, and send it using send(), exactly as for the multithreaded case. Set the state to ‘new request’

It is possible to optimize performance by combining the multiple select() calls into a single call on a per-cycle basis, without affecting the functional structure of the application.

A.3 Required and expected performance

There is deliberately NO specification of required response time for a transaction over MODBUS or MODBUS/TCP.

This is because MODBUS/TCP is expected to be used in the widest possible variety of communication situations, from I/O scanners expecting sub-millisecond timing to long distance radio links with delays of several seconds.

In addition, the MODBUS family is designed to encourage automatic conversion between networks by means of ‘blind’ conversion gateways. Such devices include the Schneider ‘Ethernet to Modbus Plus Bridge’, and various devices which convert from MODBUS/TCP to MODBUS serial links. Use of such devices implies that the performance of existing MODBUS devices is consistent with use of MODBUS/TCP.

In general, devices such as PLC’s which exhibit a ‘scan’ behavior will respond to incoming requests in one scan time, which typically varies between 20 msec and 200 msec.

From a client perspective, that time must be extended by the expected transport delays across the network, to determine a ‘reasonable’ response time. Such transport delays might be milliseconds for a switched Ethernet, or hundreds of milliseconds for a wide area network connection.

In turn, any ‘timeout’ time used at a client to initiate an application retry should be larger than the expected maximum ‘reasonable’ response time. If this is not followed, there is a potential for excessive congestion at the target device or on the network, which may in turn cause further errors. This is a characteristic which should always be avoided.

So in practice, the client timeouts used in high performance applications are always likely to be somewhat dependent on network topology and expected client performance.

A timeout of say 30 msec might be reasonable when scanning 10 I/O devices across a local Ethernet and each device would normally respond in 1 msec. On the other hand, a timeout value of 1 second might be more appropriate when supervising slow PLC’s through a gateway across a serial link, where the normal scan sequence completed in 300 msec.

Applications which are not time critical can often leave timeout values to the normal TCP defaults, which will report communication failure after several seconds on most platforms.

Clients are encouraged to close and re-establish MODBUS/TCP connections which are used for data access only (not PLC programming) and where the expected time before next use is significant, for example longer than one second. If clients follow this principle, it allows a server with limited connection resources to service a larger number of potential clients, as well as facilitating error recovery strategies such as selection of alternative target IP addresses. It should be remembered that the extra communication and CPU load caused by closing and reopening a connection is comparable to that caused by a SINGLE Modbus transaction.

B. Data Encoding for non-word data

The most efficient method of transporting bulk information of any type over MODBUS us to use function codes 3 (read registers), 16 (write registers), or possibly 23 (read/write registers).

Although these functions are defined in terms of their operation on 16-bit registers, they can be used to move any type of information from one machine to another, so long as that information can be represented as a contiguous block of 16-bit words.

The original MODBUS-capable PLC's were specialized computers using a 'big-endian' architecture. Most modern PLC's are based on commodity microprocessors using a 'little-endian' architecture. The fact that MODBUS is used to exchange data potentially between these two architectures introduces some subtleties which can trap the unwary.

Almost all data types other than the primitive 'discrete bit' and '16 bit register' were introduced after the adoption of little-endian microprocessors. Therefore the representation on MODBUS of these data types follows the little-endian model, meaning

First register bits 15 - 0 = bits 15 - 0 of data item
Second register bits 15 - 0 = bits 31 - 16 of data item
Third register bits 15 - 0 = bits 47 - 32 of data item
etc. etc.

B.1 Bit numbers within a word

Modicon PLC's have predefined functions in the 984 Ladder Language which will convert a series of contiguous registers into an equivalent length block of 1-bit 'discretes'. The most common such function is BLKM (Block Move).

For consistency with the original big-endian architecture, such discretes were numbered from most significant bit to least significant bit, and to add more confusion, all number sequences started at one, not zero. (Bit numbers within this document are always referenced from zero as the least significant bit, to be consistent with modern software documentation)

So within a word (register)

Discrete 1 would be bit 15 (value 0x8000)
Discrete 2 would be bit 14 (value 0x4000)
Discrete 3 would be bit 13 (value 0x2000)
Discrete 4 would be bit 12 (value 0x1000)
Discrete 5 would be bit 11 (value 0x0800)
Discrete 6 would be bit 10 (value 0x0400)
Discrete 7 would be bit 9 (value 0x0200)
Discrete 8 would be bit 8 (value 0x0100)
Discrete 9 would be bit 7 (value 0x0080)
Discrete 10 would be bit 6 (value 0x0040)
Discrete 11 would be bit 5 (value 0x0020)
Discrete 12 would be bit 4 (value 0x0010)
Discrete 13 would be bit 3 (value 0x0008)
Discrete 14 would be bit 2 (value 0x0004)
Discrete 15 would be bit 1 (value 0x0002)
Discrete 16 would be bit 0 (value 0x0001)

When there are more than 16 bits, for example a 32 point discrete input module, discretes 1 to 16 would be in the first register, discretes 17 to 32 would be in the second register.

This numbering convention is particularly important to understand when dealing with discrete input or output devices over MODBUS/TCP, where the numbering of the discrete points has been arranged to be consistent with Modicon PLC's.

In particular, note that the IEC-1131 numbering convention for bits within a word is from 0 (least significant) to 15 (most significant), which is the opposite of the discrete numbering.

B.2 Multi-word quantities

In principle, any data structure which can be 'cast' to an array of 16-bit words can be transported, and will arrive unchanged on a machine with the same data representation.

The following PLC data types should be noted

B.2.1 984 Data Types

984 16-bit Unsigned Integer

Natural meaning: bit 15 - 0 of integer = bit 15 - 0 of register

984 16-bit Signed Integer

Natural meaning: bit 15 - 0 of integer = bit 15 - 0 of register

984 ASCII

Although PLC's had no text manipulation capabilities as such, the original ladder language editors allowed registers to be displayed as 2 ASCII characters each. The first character displayed was the UPPER byte (bits 15 - 8) and the second character displayed was the LOWER byte (bits 7- 0). Note in particular that this is the reverse of any use of a character array in C or other high level languages on modern PLC's.

984 Floating point

Intel single precision real
First register contains bits 15 - 0 of 32-bit number (bits 15 - 0 of significand)
Second register contains bits 31 - 16 of 32-bit number (exponent and bits 23 - 16 of significand)

984 Single precision unsigned decimal

Although the range of values is limited at 0 - 9999, the data representation is the same as a 16-bit unsigned integer

984 Double precision unsigned decimal

This data format is now little-used, except to drive old-style 4-digit decade displays.
The range of values is 0 to 99999999. The first register contains the MOST significant 4 digits, the second register contains the LEAST significant 4 digits, each expressed as binary values in the range 0-9999.

B.2.2 IEC-1131 data types

All IEC-1131 data types are represented on Modicon PLC's in little-endian form. Examples follow

BYTE

8-bit quantity.
Bits 7 - 0 of register = Bits 7 - 0 of BYTE

DINT

32-bit quantity.
Bits 15 - 0 of first register = bits 15 - 0 of DINT
Bits 15 - 0 of second register = bits 31 - 16 of DINT

INT

Bits 15 - 0 of register = bits 15 - 0 of INT

REAL

32-bit Intel single precision real
Bits 15 - 0 of first register = bits 15 - 0 of REAL (bits 15 - 0 of significand)
Bits 15 - 0 of second register = bits 31 - 16 of REAL (exponent + bits 23 - 16 of significand)

UDINT

32-bit quantity.
Bits 15 - 0 of first register = bits 15 - 0 of UDINT
Bits 15 - 0 of second register = bits 31 - 16 of UDINT

UINT

Bits 15 - 0 of register = bits 15 - 0 of INT

For any others, see the appropriate IEC-1131 programming manuals