Object Messaging Specification for the MODBUS/TCP Protocol

Version 1.1

November 08, 2004

Document Version

Version 0.2 – February 12, 1999
Version 0.3 – March 2, 1999
Version 1.0 – March 21, 1999 (draft)
Version 1.0 – April 6, 1999
Version 1.1—November 08, 2004

Table of Contents

Executive Summary

The message specification contains the following main elements:

- Nodes on a Modbus/TCP network must use a 2-level address structure to select a target ('server') device. The first level is the conventional 32-bit IP address. The second level is a 'Unit Identifier' field which usually has values 0-247 to select multiple targets which share a single network interface, such as use of network gateway products (note: that identifier 255 is generally used to address the gateway device itself). Broadcasting of messages is handled specifically at the application level as a point to point message service to all target devices.

- Object Messages are communicated serially in the "data" portion of the Modbus/TCP protocol of each message transaction. This allows receiving devices to begin at the start of the message frame, read the address portion, determine which device is addressed, and to know when the message frame is completed.

- On the Modbus/TCP network, the underlying transport protocol handles the framing and segmentation / re-assembly of messages including beginning and end delimiters. An additional application layer fragmentation protocol is included to maintain compatibility with existing applications; these applications generally support function data field lengths of less than or equal to 197 bytes per message (fragment). This transport protocol handles delivery to the destination device making the address field of the messaging frame redundant for the actual transmission

- An object message contains the following fields:

    Fragment Byte Count / Fragment Protocol / Object Messaging Protocol

Where the Fragment Protocol contains the following fields:

    Fragment Indicators / Fragment Sequence Number

and, where the Object Messaging Protocol contains the following fields:

    Class ID / Instance ID / Service Code / Data

Object Messaging Specification for the MODBUS/TCP Protocol

Version 1.0

Introduction

A key component of many network solution specifications, such as the SEMI sensor bus Network Communication Standard (NCS)[1], is a capability for communicating service request / response information over the network to objects in a device. Although the current Modbus I/O communication mechanism can be utilized for non-object-based communications to some devices, an object messaging protocol is needed to specify device object communications, and to handle more voluminous data transmissions that could be associated with some service transactions. The Modbus protocol can support this type of communication, however it is not specified in the protocol document [1]. This specification details an object messaging protocol for Modbus/TCP.

The requirements of the Modbus/TCP object messaging protocol are as follows:

1. Provides support for object messaging concurrently with the existing Modbus/TCP protocol [2] by specifying an object addressing scheme that supports object to object communication of service Requests and Responses.
2. Does not invalidate existing devices.
3. Supports transmission of messages (theoretically) of any length.
4. Includes an addressing scheme that supports the end-to-end communication of message transactions in a client / server fashion.

The components of the Modbus/TCP object messaging protocol are presented in the following sections. They include:

1. Data Transmission: the mechanism for transmitting object message data in Modbus/TCP networks.
2. Network Access and Addressing: the method by which a Modbus device gains access to the network for transmitting object based messages.
3. Object Message Structure: the fields / protocol within a Modbus object based message.
4. Object Addressing Protocol: the protocol within the message structure utilized for object to object communication of service Requests and Responses.

[1] Semi is an acronym for Semiconductor Equipment and Material International. The SEMI NCS is a standard for device level communications in the semiconductor industry.

This document also describes the mechanism by which message capable devices can co-exist with message in-capable devices in a Modbus/TCP network.

References

[1] PI-MBUS-300 Rev. E -- Modicon Modbus Protocol Reference Guide (March 1993).

[2] Open Modbus/TCP Specification Version 1.0, March 29, 1999 may be accessed from the Modbus/TCP Word Wide Web Home Page,
http://ww.SchneiderAutomation.com/openmbus

Glossary

Operating Cycle – A period of time during which a Modbus/TCP network device is capable of supporting the communication (transmission and/or reception) of Modbus formatted information. Note that an interruption in power will end an operating cycle. Note also that an operating cycle of a device will end if that device senses that it is no longer capable of sending or receiving Modbus information, e.g., if the device is disconnected from the network, or reset by a user.

Data Transmission

Messaging is supported in both peer-to-peer and Client/Server configured networks. In both cases object messages are communicated serially in the "data" portion of the Modbus protocol of each message transaction, as shown in Figure 1a.

The general content and format of the data portion of a Modbus message is identified by a function code preceding that data (see Figure 1a.) All Modbus devices that support and wish to utilize object messaging should implement the Object Messaging Function Code, #91 (decimal), assigned to this feature wherever possible.

All object messages are sent between two nodes to execute service transactions. All service transactions are associated with "request" services and consist of a request message and a corresponding response message. Some service transactions are associated with "notify" services and consist of a single notification message. All "notify" services require an application level acknowledge response.

An alternate transport mechanism for Modbus/TCP is defined in Appendix A. Since both clients and servers may be limited to using "standard" Modbus function codes, the alternate transport mechanism defines a standardized method for utilizing these function codes to achieve object based communication.

The Modbus protocol promotes a Client/Server communication technique and provides the internal standard that a Modbus device must use for parsing messages. Each Object Messaging Request shall be designated with an *'even'* action code. Each Object Messaging Response shall be designated with a corresponding *'odd'* response code.

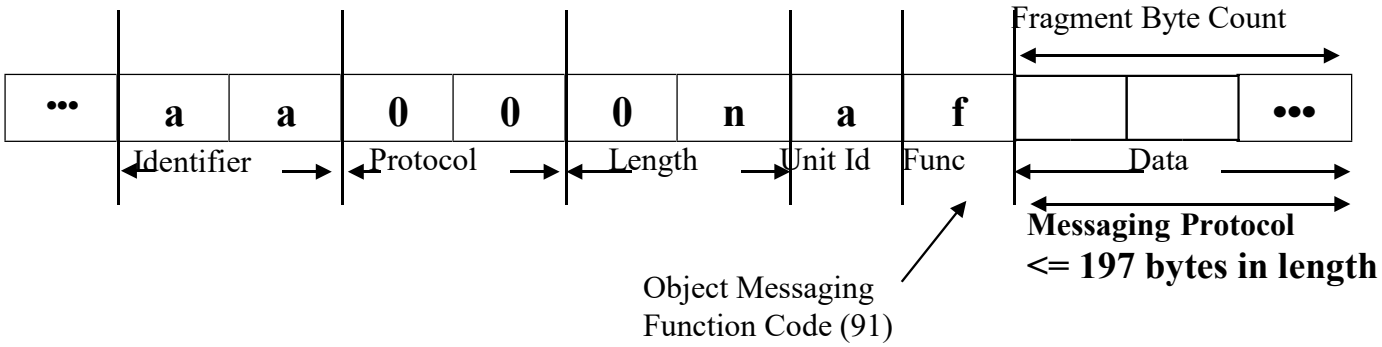
Since both clients and servers may be constructed using devices which support only the alternate transport function codes, it is necessary that client and server devices support the alternate transport mechanism in addition to the object messaging function code if so supported.

During communication on a Modbus/TCP network, the protocol determines how each device will sense its address, recognize a message addressed to it, determine the kind of action to be taken, and extract any data or other information contained in the message. Each Object Messaging request requires a corresponding Object Messaging response. The device will construct the response message and send it using the Modbus protocol. The Object Messaging protocol can also support a 'Notify' service for which a response is required from the target device. The specifics for adherence to the Modbus protocol are contained in Modicon Modbus Protocol Reference Guide [1].

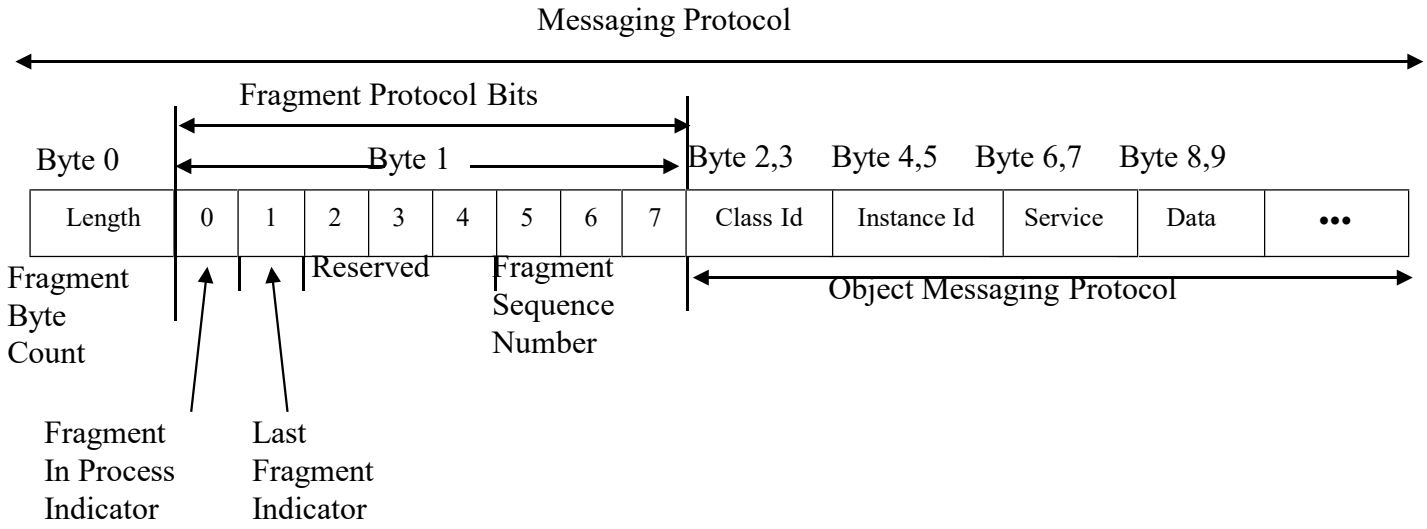
On the Modbus/TCP network, messages containing the Modbus protocol are embedded in the frame or packet structure that is used by the network. On this type of network, devices communicate using a peer-to-peer technique, in which any device can initiate a transaction with any other device. Thus a device may operate either as a server or as a client in separate transactions. At the messaging level, the Modbus protocol still applies the client/server principle even though the network communication method is peer-to-peer.

Modbus applications generally operate in a client/server fashion where a client polls its servers for information. Because of the point to point capability of the protocol, Modbus/TCP can support other methods of communication such as servers reporting asynchronously in a cyclical or change of state fashion using the 'Notify' service. Note that determinism may be compromised when utilizing these asynchronous reporting methods.

The object message that is transmitted serially in the data field comprises Modbus messages. These messages are transmitted in one or more sequential *Fragments* that together comprise the message. The components (also called "fields") of a Modbus message fragment are shown in Figure 1b. These components and the Modbus message fragmentation scheme are explained in the following sections.



(1a)



(1b)

Figure 1: Object Messaging Protocol Layout

(Each cell in Figure 1a corresponds to a single byte unless indicated as "…". Byte boundaries in Figure 1b are as indicated in the figure.)

Network Access and Addressing

Messaging is supported in a client / server fashion in the Modbus/TCP protocol whereby the underlying TCP provides Node to Node client /server communication. An addressing scheme is utilized within this messaging protocol to provide object communication within this client / server protocol.

Specifically, the Modbus/TCP messaging protocol requires that all nodes wishing to utilize messaging have a configurable IP + Unit ID identifier to uniquely determine the device address on the Modbus/TCP network. Valid Unit ID addresses range between 0 and 247 (with 255 usually reserved for communication directly with a gateway). Note that a sending device should use "0" as the Unit ID when it knows that the target IP address has exactly one device. It is the responsibility of the person configuring the network to set up device messaging addresses as necessary to guarantee uniqueness.

On a Modbus/TCP network, when a device wishes to send a message it does so by establishing a connection with another device or utilizing a device connection previously established. The Modbus/TCP network protocol handles the framing of messages with beginning and end delimiters, and with the Unit ID handles delivery of the message to the destination device.

The Modbus/TCP media access protocol utilizes various mechanisms, such as carrier sense multiple access with collision detection (CSMA-CD), to determine access to the network. This protocol is thus utilized to resolve conflicts in situations where multiple devices wish to begin transmitting messages at the same time.

Message Structure

Each Modbus object message, transmitted serially in the Data field, consists of one or more sequential message fragments. Each message fragment contains seven fields as shown in Figure 1b. Table 1 contains a description of each of these fields.

Byte Number	Field Name	Size (bits)	Description
0	Fragment Byte Count	8	Contains the byte length (not including itself) of the Messaging protocol portion of the Modbus transaction. The maximum Fragment Byte Count is 197 bytes (decimal).
1	Fragment In Process Indicator	1	Value of TRUE (1) indicates that this Messaging protocol field is one fragment of a multi-fragment message.
1	Last Fragment Indicator	1	Value of TRUE (1) indicates the last fragment in the Messaging protocol
1	Reserved	3	Not used at this time. Should be set to zero (0).
1	Fragment Sequence Number	3	Counter indicating sequential fragment number beginning with 000, 001, 010, …, 111. Counter rolls over from 111 to 000.

Table 1: Fields in a Message Fragment

The Data (i.e., Object Addressing Protocol) portion of larger messages must be broken into smaller fragments so that high priority messages can gain access to the network in a timely fashion. Specifically, a message fragment must be less than or equal to 197 bytes (including itself and the Payload Length + Fragmentation Protocol byte). When a device prepares to send a message longer than the maximum fragment length, it must break the Messaging Protocol field of the message up into sequential fragments of 195 or less bytes each, and construct data fragments (of 197 or less bytes, see Figure 2) consisting of the fields described in Table 2.

Note, a device sending a fragmented message to a second device can not begin sending another message (fragmented or unfragmented) to that second device until it has completed sending the first message. If a receiving device detects that this condition has occurred, it should reject both messages and attempt (either through a response with an error code – see Appendix B, or a standard exception code 03) to inform the sending device of the error.

Object Addressing Protocol

The *Data Fragment* portions of the message fragments, when concatenated at a device, form the *Data field* portion of the message. This data is formatted according to the *Object Addressing Protocol*, that is, the protocol used for object-to-object communication between nodes.

The objects, their groupings, structure, and behavior in Modbus devices conform to an *Object Model*. In this model, *objects* are generally regarded as entities that group structure and behavior in a logical manner. Objects usually have a physical or conceptual analog in a device application. For example, an object may be associated with a sensor in a device, or may be the collection of structure and behavior that comprises the management of the device. The Modbus Object Addressing Protocol utilizes a Class / Instance hierarchy to support inheritance to allow for the definition of "types" of objects (*Classes*) as well as specific implementations of these objects (*Instances*). For example an object class in a photodetector array device might be "photodetector" where object instances refer individually to each photodetector.

An object (class or instance) has zero or more *attributes*, which are parameters that contain information associated with the device. For example "sensor value" may be a parameter associated with a sensor object instance.

An object (class or instance) supports zero or more *services*, which are functions or capabilities that the object can provide. Most services are *request services* in which a "request" for a service is issued to an object, and the object generates a specific "response" to the requestor as part of the process of carrying out the service request. Parameters may be included with the request and / or response as necessary to carry out the service. As an example, a service supported by a sensor object may be "Get" where an associated request parameter is "attribute number" and an associated response parameter is "attribute value". Note that the first service parameter of all response service messages is an error code (see Table 2). An object may also support *notification services*. "Notifications" associated with these services are generated asynchronously by the object. Parameters may be included with the service notification as necessary to carry out the service. As an example, an Alarm Publish service may be associated with a sensor object and would reports an alarm state attribute whenever that attribute changes from zero to a non-zero value.

An object also has associated *behavior*, which identifies as appropriate how the object implements services, interacts with other objects, interacts with any outside environment, etc. For example, behavior associated with a sensor object might be: on receipt of "Get" service

request, where attribute number parameter corresponds to "value" attribute, detect environment value, store in "value" attribute, and send "Get" service response with response parameter value equal to "value" attribute.

The Object Messaging Protocol utilized in the Data field of Modbus message provides for the communication of services (requests, responses and notifications), between objects at various message capable nodes on the Modbus/TCP network, thereby supporting the Modbus object model. In order to provide this support, the Data field of a message is sub-divided into a number of sub-fields as shown in Figure 2. These sub-fields are described in Table 2. Refer to Appendix C for an example of the message format.

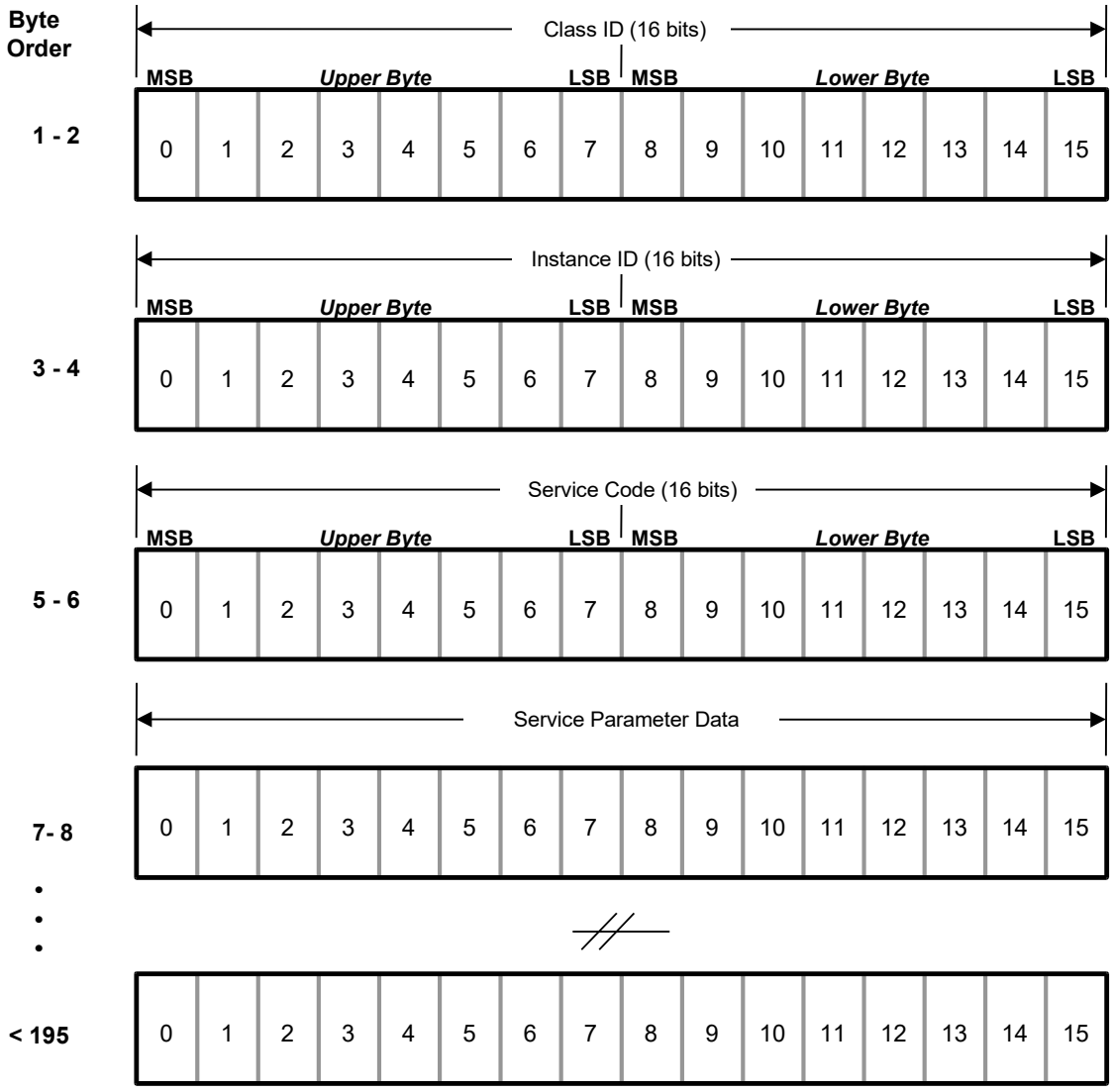


Figure 2: The Data Field in a Modbus Message
(Bit number represents transmission position)

Sub-Field Name	Size (bits)	Description
Fragment Byte Count	8	Contains the length in bytes, (not including itself) of the Messaging protocol portion of the Modbus transaction. Note that it does not include any byte that may be "stuffed" at the end of a fragment to ensure word boundaries of transmissions (see description of "Data and Stuff Byte" Sub-Fields below). Maximum Fragment Byte Count length is 197 bytes.
Fragment Protocol	8	Refer to Table 1 and Figure 1a as: Bit 0: Fragment In Process Indicator Bit 1: Last Fragment Indicator Bit 2-4: Reserved Bit 5-7: Fragment Sequence Number
Class ID	16	The object class associated with the service. In a service request this is the target class ID of the object to which the request is directed. In a service response or notification it is the class ID of the object that is sending the response.
Instance ID	16	The object instance associated with the service. In a service request this is the target instance ID of the object to which the request is directed. In a service response or notification it is the instance ID of the object that is sending the response. Note that if a service is directed at the Class object, the Instance ID is zero.
Service Code	16	A number indicating the service request / response / notification being issued. Service requests and notifications have odd numbered Service Code values. Each service response has a value equal to the corresponding Service Code (Request) + 1. A Service Code of zero is invalid.
Data	N * 16	Data associated with the service request / response / notification, i.e., service parameters. Note that the first parameter of ALL response service messages is a one-word error code. Common response service error codes are defined in Appendix B.
Stuff Byte	8 (Conditional)	If the length of the Data field is not a multiple of 16 bits, this field is included at the end of the fragment to ensure that the transmission is a multiple of 16 bits. Note that this field does not contain meaningful data and is not included in the Fragment Byte Count

Table 2: Sub-Fields of the Message Data Field

A Modbus message communicates a service request, response or notification to or from a specific object instance of an object class at a node using the Object Addressing Protocol. In this way the structure / behavior of an object can be communicated / invoked over a Modbus/TCP network using Modbus messages.

Compatibility of Messaging with Non-Message Capable Nodes and Nodes Retrofitted with Messaging Capabilities

A large number of Modbus devices do not support the object messaging capability for reasons of older devices, simplicity, memory conservation, data requirements, etc. These object message incapable devices can co-exist with message capable devices in a Modbus/TCP network. The object message incapable devices shall respond to the device "query" with an error "response" (function code 01) which would indicate "Illegal Function" error.

A number of Modbus devices are capable of encoding/decoding object messaging, but can only utilize "currently existing" Modbus function codes. For these devices an alternate set of existing function codes have been identified. Device setup and use of these function codes as an alternate method of supporting object messaging is detailed in Appendix A. Note that the default Object Messaging Function code, #91 should be used wherever possible.

Since both clients and servers may be constructed using devices which support only the alternate transport function codes, it is necessary that client and server devices support the alternate transport mechanism in addition to the object messaging function code if so supported.

Appendix A: Alternate Transport Mechanism for Modbus/TCP -- Utilizing "Existing" Register Read/Write Function Codes to Support the Modbus/TCP Object Messaging Protocol

Background

There are many situations in which facilities requiring 'new' MODBUS functions need to be retrofitted to devices which implement only the 'standard' functions. In many cases, it is impractical to replace the PLC firmware or Operating System libraries which provide MODBUS service, so an approach which allows the MODBUS functions to be 'encapsulated' or 'tunneled' is required instead.

This appendix describes such a mechanism.

Specific issues

All MODBUS devices have facilities for exchanging blocks of memory of undefined meaning. The primary functions used are

FC 3 – 'read registers'

Allows the 'read' of up to 125 16-bit words from a target device, given a start address ('reference') and length ('number of registers')

FC 16 – 'write registers'

Allows the 'write' of up to 100 16-bit words to a target device, given a start address and length as above.

Any such individual transfer may be regarded as 'atomic', meaning that the data is treated as if it were 'snapshot' at a single instant in time. A movement of data comprising more than one transfer may NOT be regarded as atomic, since it may span more than one 'scan' of the target device, and if the data were being constantly updated then the information transferred from different scans would be inconsistent.

Modbus does not maintain any true nature of a 'connection' between a client and a server, except for the duration of a single 'transaction' comprising a MODBUS request and its associated response. Therefore there is the possibility that a sequence of transactions initiated by one client to a server may be interleaved with transactions from another client. This is of particular importance when dealing with data areas such as 'mailboxes' which are designed to be updated by multiple clients. There is no guarantee that multiple attempts to update such a mailbox will not occur simultaneously.

Modern PLC's have a mechanism for accepting 'user-specified function blocks' which accept as parameters a block of registers of unknown content but discernable address and length. Such functions may be written in conventional computer language, such as C or C++, and provide a suitable way to handle algorithms or other data processing tasks for which the traditional PLC ladder or function block languages are inappropriate. However they are deliberately NOT allowed to perform non-computational tasks, such as control of I/O devices or generation of networking messages.

By combining such user function blocks with the existing MODBUS register moving primitives, it is possible to implement arbitrarily complex functions. This is exactly what is intended by this document.

Register Assignment

See text below for detail interpretation

Word 0: signature word 0 = “SE” = 0x5345

Word 1: signature word 1 = “MI” = 0x4D49

Word 2: signature checksum = 0x5F72

Word 3: number of channels supported = N (1 to 40)

Word 4: channel assignment mailbox (writable by all clients)

Word 5: channel assignment ID for channel 1

…

Word 4+N: channel assignment ID for channel N

Word 5+N: sequence/ID word of request buffer for channel 1

Word 6+N: bytes 0 and 1 of request for channel 1

…

Word 104+N: bytes 197 and 198 of request for channel 1

Word 105+N: sequence/ID word of response buffer for channel 1

Word 106+N: bytes 0 and 1 of response for channel 1

…

Word 204+N: bytes 197 and 198 of response for channel 1

Word 205+N to 404+N: as for 5+N to 204+N for channel 2

…

Word 5+200(N-1) to 204+200(N-1): request and response buffers for channel N

Note that the sequence/ID occupies a whole word. This is because the application message, starting with the byte count, is already a multiple of 16 bits in length so there is no value in using a shorter field.

Protocol Encapsulation Details

There are really three parts to this definition

1. Mailbox discovery
2. Channel assignment and release
3. Protocol message encapsulation

Mailbox Discovery

Since the mechanism is layered on top of the standard MODBUS FC 3 and FC 16 operations, it is not possible to confirm that a target device supports the protocol by interrogating it speculatively, as is true for all individual MODBUS functions. Instead, communication is arranged to use a series of data buffers, allocated from blocks of registers. In order to confirm that the registers have the purpose intended, a 'signature pattern' is deposited at a recognizable location. By arranging that the chance of accidental match of this signature pattern is very low, the chance of mis-identifying a conventional PLC is made similarly low. In this encapsulation, we use a series of 3 16-bit words as the signature pattern which must match, so that the chance of mis-identification of a random PLC is about 1 in 2^48, or one in 10^14

The reason for relying upon a signature rather than a predefined register address is to allow flexibility in retrofitting devices whose register allocation have already been made. The penalty for this technique is an increased communication burden at initial connection. However even if a register table of size 10000 words had to be scanned to find the signature, this takes only 80 read operations (125 words each), and occupies less than 20 msec of transmission time on an Ethernet network. It can therefore be readily completed in a few seconds after restart of a client machine without undue burden on the network.

The signature block chosen for this protocol consists of the following 3 words[1]:

Word 0: ASCII "SE" = 0x5345

Word 1: ASCII "MI" = 0x4D49

Word 2: Zero-sum checksum (such that words 0-2 when added would result in zero) = 0x5F72

Channel Assignment and Release

Any mechanism for supervisory control must allow for the possibility of substitution of the supervising device, either as a deliberate act or as consequence of failure. In a conventional network, this is usually achieved by allowing for multiple 'channels' at any instant, and allowing for a new 'master' to take control by issuing commands using its own channel. In the case of a MODBUS encapsulation, a slightly different mechanism is necessary, as a consequence of the inability of a server (Modbus slave) to confirm the identity of the client (MODBUS master) which is requesting any operation. Any such identification, although not provided directly by the MODBUS protocol, can be added at the encapsulation level.

The mechanism defined here uses an arbitrary number of 'channels', each channel in turn reserving:

A single register 'assignment' indicator

[1] The string "SEMI" is chosen because an initial user of this protocol was the semiconductor industry and SEMI (Semiconductor Equipment and Materials International) is the recognized standards body for that industry.

A 100-register request buffer

A 100-register response buffer

In addition, there is a single multiple-access register known as the ‘mailbox’, and which is used by all clients to request assignment of a channel.

It is arranged that the assignment registers for all channels are contiguous, so that a client may confirm the assignment status of all channels in a single READ operation. A channel is requested by writing a non-zero identification pattern into the ‘mailbox’. Details of the meaning of these identification patterns is meaningless to the mechanism, but a common approach is to allocate a unique, non-zero, 16-bit number to each device which may perform client service. That way, ID conflicts will be avoided.

To request a channel, first see if you already have one allocated. This can be done by READing the channel assignment block, to see if your chosen ID is still registered. If it is, you may continue to use it. If it has been cleared to zero, then the server has performed a unilateral close, probably due to an inactivity timeout, and the client must ‘re-bid’ for a channel using the mailbox.

The server will react to a non-zero value in the mailbox by assigning a channel if one is available, indicating this by copying the ID value from the mailbox into the appropriate ‘assignment’ register, and clearing the mailbox to zero in preparation for the next assignment request. Note that it is perfectly possible for multiple clients’ assignment requests to collide, so the client should read all registers from the mailbox to the end of the assignment block to confirm the status of his request. If his ID has been removed from the mailbox, but has not appeared in one of the assignment registers, he should assume the request has been lost and be prepared to re-submit.

A well-behaved client will release its channel voluntarily after use, this is done by writing a zero directly to the ASSIGNMENT register. The server will guarantee not to re-allocate an ‘involuntarily released’ channel for a period of at least one second, therefore a client may safely rely upon the indication that it owns the channel obtained during a preceding read, so long as the hesitation is significantly less than one second. After a voluntary close, there is no need for a time delay before re-assignment.

(This operation does not cause any collision problems because MODBUS allows the length of a write operation to have an extent of a single word, thus the clearing of the assignment word for client 1 by client 1 will not change the values recorded in the table for any other channels)

The server will perform a unilateral close on any channel which does not have a new request placed upon it for a period of one second. Supervising devices which guarantee a scan time less than one second may therefore leave the channel open and reasonably expect it to be preserved. Devices requiring intermittent service where hesitation may exceed one second should perform the voluntary close operation to avoid unexpected hesitation.

The server will maintain a ‘guard band’

Protocol Encapsulation

Once a channel has been allocated, the client is free to place requests in the 'request buffer' and to look for responses in the 'response buffer'. However the first word of both the request and response buffers has specific meaning. It is a sequence number field which will vary from one request to the next, and which allows the server to determine that the contents of the request buffer does indeed represent a NEW request, and is not simply the previous one unaltered.

The next word is the start of the Object Messaging Protocol data unit, starting with the fragment length byte and fragmentation flags byte.

(note that if a message is carried using fragmentation, the meaning of this field is taken on a fragment by fragment basis. It thus always indicates the number of significant bytes immediately following the byte count itself)

When the request has been recognized by the server, it will generate a response with the same structure. The sequence field of the response will be copied from the request. The length field of the response will indicate the length of the response (or response fragment)

The payload of the request and response buffers will be identical to the protocol definition for the OMP protocol request and response given earlier, except that the bytes up to and including the function code are discarded.

The result of this relationship is that the handling of messages either by direct transmission or by alternate transmission is largely identical. This means that testing of the alternate protocol is limited to transport encoding only, that any application functionality will be handled by common processing and therefore will generate identical results using either mechanism.

Protocol Encoding Example

(Numbers below should be assumed hexadecimal unless otherwise stated or followed directly by a period)

The examples below show MODBUS/TCP exchanges using the same typographic conventions as is used in the OPEN MODBUS/TCP SPECIFICATION. That is, an exchange written as

03 00 00 00 01 => 03 02 12 34

represents a MODBUS request of 03 00 00 00 01 (function code 03, reference 0000, count 0001)

generating a MODBUS response of 03 02 12 34 (function code 03, byte count 02, data value 1234)

When seen as a TCP data exchange, and assuming a unit identifier of 09, each of the above messages would be prepended with the 7 byte sequence consisting of transaction_id (2) protocol (2) length (2) and unit_id (1) resulting in:

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

The Object Messaging Protocol example used here is a typical enquiry to find the device type. This would be a suitable enquiry to find whether the device in question was an OMP-capable device, and if so what you could expect it to do.

The structure of the query and response over Modbus using the native encoding would be

5B 09 00 00 01 00 01 00 07 00 01 => 5B 09 00 00 01 00 01 00 08 12 34

This is broken down as follows

Request

5B Modbus function code (91. Decimal)

Number of bytes of message (fragment) following

no fragmentation

00 01 class = 0001

00 01 instance = 0001

00 07 service request = get attribute

00 01 service data = attribute number = 0001 device type

Response

5B Modbus function code

Number of bytes of message (fragment) following

no fragmentation

00 01 class = 0001

00 01 instance = 0001

00 08 service request = get attribute response

12 34 service data = attribute value = 1234

Now if the server did not support the native encoding, it would respond to the Modbus query with an ‘undefined function’ exception (Modbus exception = 01) This would come through as

5B 09 00 00 01 00 01 00 07 00 01 => DB 01

In order to use the alternate encoding, the client must first identify the ‘signature area’ found ‘somewhere’ in the register space of the target. So the client issues a series of Modbus requests to read the register space, and inspects the results looking for the signature pattern which will be the sequence (in hex)

4D49 5F72

For example, a read of the first 125. (decimal) registers would be the following exchange

03 00 00 00 7D => 03 FA (plus 250. bytes of data…)

and the next 125 registers would be

03 00 7D 00 7D => 03 FA (plus 250. bytes of data…)

At some point, the 5345 4D49 5F72 sequence will be seen, or an exception response will be generated because the register number is too large. This would normally be exception 2 (bad reference), looking like this (assume we are up to reference 7D00 by this time, a big number)

03 7D 00 00 7D => 83 02

Assume for the continuation that the signature block was found at reference 4000 (hex)

This means that if the client were now to issue a request to read 3 words at 4000, he would get the following

03 40 00 00 03 => 03 06 53 45 4D 49 5F 72

It also means that the number of channels would be found at 4003, the session mailbox at 4004, and the current assignments at 4005 on

Issue a request to read 42. (decimal) words at reference 4003. This will be enough to read the whole reservation table, since the maximum number of sessions is 40. The results will be

Word 0: number of sessions

Word 1: mailbox (ignore for now)

Word 2: reservation for channel 1

Word 3: reservation for channel 2

…

Assume the sequence was

03 40 03 00 2A => 03 54 00 08 00 00 00 00 (plus 39. other words)

The response says

function code

byte count (42. decimal words = 84. Bytes)

00 08 number of channels supported = 0008

00 00 current mailbox state = empty

00 00 channel 1 state = unreserved

It is a remote possibility on initial connection, but must still be considered, that there is already a session established between this client and this server. If that were true, then the 'client ID' number allocated to this client would be found already in the session allocation table. If not, then the client must 'bid' for a channel by putting the very same client ID pattern in the 'mailbox', and then repeating the check to see if the server is willing to allocate a channel. (It should be, because it appears there are currently unallocated channels.)

Assuming the client ID is 0xABCD, the transaction would be

10 40 04 00 01 02 AB CD => 10 40 04 00 01

which is 'write registers, reference 4004, 1 word, 2 bytes, value 0xABCD'

Now the client repeats the channel address table read request, looking for the following results

The mailbox still contains ABCD

The server has not yet processed the request, try reading again

The mailbox does not contain ABCD, but ABCD appears in one of the channel reservation words

The server has allocated the given channel for this session. Based on WHICH channel was allocated, the address of the request and receive buffers can be determined

The mailbox does not contain ABCD, and ABCD appears nowhere in the channel reservation list

Probably there was a 'collision' and 2 clients bid simultaneously, with the result that te earlier value in the mailbox was overwritten before being considered by the server. Retry the bid process from the beginning by writing the mailbox again.

Assume that the pattern ABCD was found in the first channel assignment word, for example

03 40 03 00 2A => 03 54 00 08 00 00 AB CD (plus 39. other words)

This indicates that channel 1 has been assigned to this session. That means in turn that the request buffer for this channel is found at a reference number to be calculated as follows

4005 +	(start of channel assignment table)
0008 +	(length of channel assignment table = number of channels
(0 x	(channel number minus one)
00C8)	(length of request buffer + length of response buffer = 200. Decimal)

and the response buffer is found 0064 hex words further on. In this particular case, the request buffer starts at 400D and the response buffer at 4071

Finally (!) the client gets to place the original request, which as we recall would in native encoding have been

5B 09 00 00 01 00 01 00 07 00 01 => 5B 09 00 00 01 00 01 00 08 12 34

When presented for alternate encoding the initial 5B is omitted, the rest from the byte count onwards is placed in the request buffer after the sequence word. Assuming the client's next sequence value was 2222, this would be achieved by the following sequence

10 40 0D 00 06 0C 22 22 09 00 00 01 00 01 00 07 00 01 => 10 40 0D 00 06

Now the client must check, possibly repeatedly, to see if the request has been processed by the server and a response provided. When that happens, the Modbus response will have been placed in the response buffer using the same encoding convention. The simplest way to check for this is to do a full 100 (decimal) word read of the whole response buffer, and inspect the first word to see if it contains the new sequence number (2222). Note that there is no requirement that sequence numbers be in fact contiguous, only that they be distinguishable from previous sequence numbers, so that the responses can be distinguished from a previous response still in the buffer. The value zero must not be used, since that indicates 'no message' and will be the value left in the ID field of the REQUEST buffer by the server, to allow the server to record that the request apparently in the buffer has already been processed.

The transaction to extract the response will look like this, if the response has been prepared

03 40 71 00 64 => 03 C8 22 22 09 00 00 01 00 01 00 08 12 34 (plus 94. other words)

The client extracts the response, just as if it had been handled using native encoding, where it would have been

5B 09 00 00 01 00 01 00 08 12 34

It is decoded as

byte count

not fragmented

00 01 class

00 01 instance

00 08 get attribute response

12 34 response data

If any pattern other than the ID is found in the first word, then the request has not yet been processed

The above sequence can be repeated, just change the sequence number to a different value so that the responses can be readily distinguished. Note that the server really does not care what number is used, since it just checks against the zero value in the request to see if a new request has been presented.

When the channel is to be released, which for well-behaved clients would be if the client does not intend to re-use the channel with in the next 1 second, it does so by clearing the channel reservation entry.

Recall from the example above that the allocation word for this channel was at 4005 hex (This was the word containing the client's identifier pattern of ABCD)

To clear it, issue a write register request of 1 word

10 40 05 00 01 02 00 00 => 10 40 05 00 01

Appendix B: Service Response "Error Code" Parameter Values

The first parameter of ALL service code response messages is the "Error Code" parameter. This parameter is one word in length and can have a value ranging from 0 to 65535. The following table identifies the error response type associated with value ranges of this parameter.

Error Code Value Range	Meaning
0	Success
1 – 127	Error – General
128 – 255	Error – Device Type Specific
256 – 65,535	Error – Manufacturer Specific

The following table is the enumeration of General Errors:

Error Code Value	Meaning
1	Invalid service code
2	Invalid service code parameter
3	Invalid attribute
4	Attribute out of range
5	Not valid in this state
6	Fragmentation error
7	Fragments from multiple messages
255	Unspecified error

Device Type Specific Error Code meanings would be defined through agreement of manufacturers of a specific device type within the Modbus organization. Manufacturer Specific Error Code meanings are defined and documented by the manufacturer.

Appendix C: Example of a Data Field in a Modbus Message

The following is a formatted example of the Message Data Field in a Modbus message represented in Figure 1b and Figure 2. It details a format of a service request with a service code of 00 05 directed at an object instance with a Class ID of 00 04 and an Instance ID of 00 01 (numbers are represented in hexadecimal):

The encoding of the sub-fields (see Table 2) is as follows:

Fragment Byte Count (1 byte) =	08
Fragment Protocol (1 byte):	40, (01000000 binary) where: Fragment In process Indicator = 0 (False) Last Fragment Indicator = 1 (True) Reserved = 000 Fragment Sequence Number = 000
Class ID (2 bytes) =	00 04
Instance ID (2 bytes) =	00 01
Service Code (2 bytes) =	00 05
Service Parameter (1 byte) =	08

The example below shows the Message Data Field of the message fragment as it would be transformed (most significant bit of each byte transmitted first), and viewed over the transmission line. For example, the Fragment Byte Count of 08 (00001000 binary) would be transmitted as 10 (00010000 binary).

Fragment Byte Count =	10
Fragment Protocol Bits =	02
Class ID =	00 20
Instance ID =	00 80
Service Code =	00 A0
Service Parameter =	10

The transmission would appear as:

10 02 00 20 00 80 00 A0 10 00

Notes:

1.) Since the data length is 1 byte it is not a multiple of 16 bits. Thus a byte with a value of 00 is "stuffed" at the end of the message (this is the conditional Stuff Byte field). Note that the Fragment Byte Count does not include this byte.

2.) The Fragment Protocol byte is transmitted MSB first (as with all other bytes). Thus for example, a fragment protocol with:

Fragment In process Indicator = 0 (False)
Last Fragment Indicator = 1 (True)
Reserved = 000
Fragment Sequence Number = 001

Would be represented as 41 (binary 01000001), and transmitted as 82 (binary 10000010),