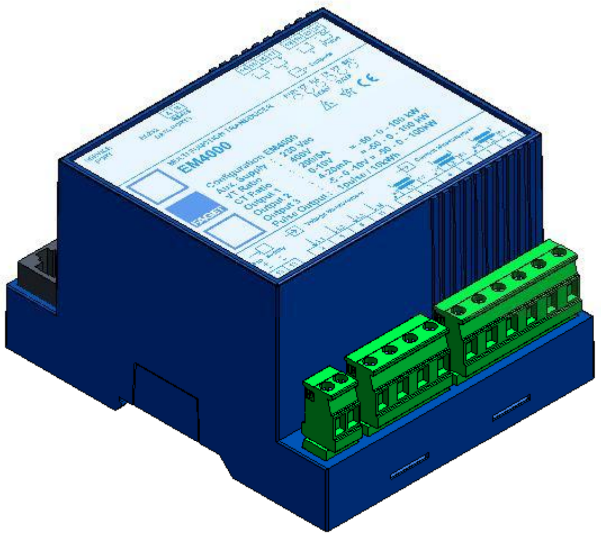
Metering
Protection
Lighting

MODbus® Protocol Reference Guide for FAGET EM4000 Software Release V2.80



Rev. 2.8, January 2009

ELEQ Steenwijk b.v.
P.O. Box 12, 8330 AA Steenwijk, The Netherlands
Tukseweg 130, 8331 LH Steenwijk, The Netherlands

T +31 (0) 521 533 333
F +31 (0) 521 533 398

E-mail: info@eleq.com
K.v.K. Zwolle: 05029362

ABN AMRO 48.26.14.900
Swift: ABNANL2A
IBAN: NL67ABNA0482614900

Metering
Protection
Lighting

Chapter 1 MODbus® Protocol

Introducing MODbus® Protocol

Transactions on MODbus® Networks

Standard MODbus® ports on Modicon controllers use an RS–232C compatible serial interface that defines connector pin outs, cabling, signal levels, transmission baud rates, and parity checking. Controllers can be networked directly or via modems.
Controllers communicate using a master–slave technique, in which only one device (the master) can initiate transactions (called 'queries'). The other devices (the slaves) respond by supplying the requested data to the master, or by taking the action requested in the query. Typical master devices include host processors and programming panels. Typical slaves include programmable controllers.
The master can address individual slaves, or can initiate a broadcast message to all slaves. Slaves return a message (called a 'response') to queries that are addressed to them individually. Responses are not returned to broadcast queries from the master.
The MODbus® protocol establishes the format for the master's query by placing into it the device (or broadcast) address, a function code defining the requested action, any data to be sent, and an error–checking field. The slave's response message is also constructed using MODbus® protocol. It contains fields confirming the action taken, any data to be returned, and an error–checking field. If an error occurred in receipt of the message, or if the slave is unable to perform the requested action, the slave will construct an error message and send it as its response.

The Query–Response Cycle

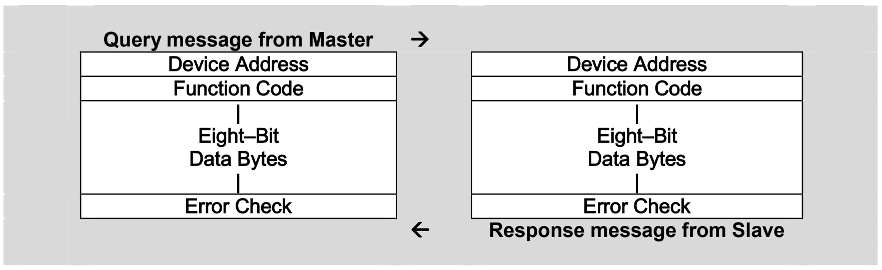


Figure 1: Master–Slave Query–Response Cycle

The Query: The function code in the query tells the addressed slave device what kind of action to perform. The data bytes contain any additional information that the slave will need to perform the function. For example, function code 03 will query the slave to read holding registers and respond with their contents. The data field must contain the information telling the slave which register to start at and how many registers to read. The error check field provides a method for the slave to validate the integrity of the message contents.

The Response: If the slave makes a normal response, the function code in the response is an echo of the function code in the query. The data bytes contain the data collected by the slave, such as register values or status. If an error occurs, the function code is modified to indicate that the response is an error response, and the data bytes contain a code that describes the error. The error check field allows the master to confirm that the message contents are valid.

The Two Serial Transmission Modes

Controllers can be setup to communicate on standard MODbus® networks using either of two transmission modes: ASCII or RTU. **The EM4000 serviceport is uses the ASCII mode, the dataport uses the RTU mode (RS232 and RS485).** Users can select the other serial port communication parameters (baud rate, parity mode, etc), during configuration of each controller. The mode and serial parameters must be the same for all devices on a MODbus® network.
The selection of ASCII or RTU mode pertains only to standard MODbus® networks.
It defines the bit contents of message fields transmitted serially on those networks.
It determines how information will be packed into the message fields and decoded.
On other networks like MAP and MODbus® Plus, MODbus® messages are placed into frames that are not related to serial transmission. For example, a request to read holding registers can be handled between two controllers on MODbus® Plus without regard to the current setup of either controller's serial MODbus® port.

ASCII Mode

When controllers are setup to communicate on a MODbus® network using ASCII (American Standard Code for Information Interchange) mode, each 8–bit byte in a message is sent as two ASCII characters. The main advantage of this mode is that it allows time intervals of up to one second to occur between characters without causing an error.

The format for each byte in ASCII mode is:

Coding System:	Hexadecimal, ASCII characters 0–9, A–F One hexadecimal character contained in each ASCII character of the message
Bits per Byte:	1 start bit 7 data bits, least significant bit sent first 1 bit for even/odd parity; no bit for no parity 1 stop bit if parity is used; 2 bits if no parity
Error Check Field:	Longitudinal Redundancy Check (LRC)

RTU Mode

When controllers are setup to communicate on a MODbus® network using RTU (Remote Terminal Unit) mode, each 8–bit byte in a message contains two 4–bit hexadecimal characters. The main advantage of this mode is that its greater character density allows better data throughput than ASCII for the same baud rate.
Each message must be transmitted in a continuous stream.

The format for each byte in RTU mode is:

Coding System:	8–bit binary, hexadecimal 0–9, A–F Two hexadecimal characters contained in each 8–bit field of the message
Bits per Byte:	1 start bit 8 data bits, least significant bit sent first 1 bit for even/odd parity; no bit for no parity 1 stop bit if parity is used; 2 bits if no parity
Error Check Field:	Cyclical Redundancy Check (CRC)

MODbus® Message Framing

In either of the two serial transmission modes (ASCII or RTU), a MODbus® message is placed by the transmitting device into a frame that has a known beginning and ending point. This allows receiving devices to begin at the start of the message, read the address portion and determine which device is addressed (or all devices, if the message is broadcast), and to know when the message is completed. Partial messages can be detected and errors can be set as a result.
On networks like MAP or MODbus® Plus, the network protocol handles the framing of messages with beginning and end delimiters that are specific to the network.
Those protocols also handle delivery to the destination device, making the MODbus® address field imbedded in the message unnecessary for the actual transmission. (The MODbus® address is converted to a network node address and routing path by the originating controller or its network adapter.)

ASCII Framing

In ASCII mode, messages start with a 'colon' (:) character (ASCII 3A hex), and end with a 'carriage return – line feed' (CRLF) pair (ASCII 0D and 0A hex).
The allowable characters transmitted for all other fields are hexadecimal 0–9, A–F.
Networked devices monitor the network bus continuously for the 'colon' character.
When one is received, each device decodes the next field (the address field) to find out if it is the addressed device.
Intervals of up to one second can elapse between characters within the message.
If a greater interval occurs, the receiving device assumes an error has occurred.
A typical message frame is shown below.

START	ADDRESS	FUNCTION	DATA	LRC CHECK	END
1 CHAR :	2 CHARS	2 CHARS	n CHARS	2 CHARS	2 CHARS CRLF

Figure 2: ASCII Message Frame

RTU Framing

In RTU mode, messages start with a silent interval of at least 3.5 character times.
This is most easily implemented as a multiple of character times at the baud rate that is being used on the network.
The first field then transmitted is the device address.
The allowable characters transmitted for all fields are hexadecimal 0–9, A–F.
Networked devices monitor the network bus continuously, including during the 'silent' intervals. When the first field (the address field) is received, each device decodes it to find out if it is the addressed device.
Following the last transmitted character, a similar interval of at least 3.5 character times marks the end of the message. A new message can begin after this interval.
The entire message frame must be transmitted as a continuous stream. If a silent interval of more than 1.5 character times occurs before completion of the frame, the receiving device flushes the incomplete message and assumes that the next byte will be the address field of a new message.
Similarly, if a new message begins earlier than 3.5 character times following a previous message, the receiving device will consider it a continuation of the previous message. This will set an error, as the value in the final CRC field will not be valid for the combined messages. A typical message frame is shown below.
A typical message frame is shown below.

START	ADDRESS	FUNCTION	DATA	CRC CHECK	END
T1+T2+T3+T4	8 BITS	8 BITS	n x 8 BITS	16 BITS	T1+T2+T3+T4

Figure 3: RTU Message Frame

How the Address Field is handled

The address field of a message frame contains two characters (ASCII) or eight bits (RTU). Valid slave device addresses are in the range of 0 – 247 decimal.
The individual slave devices are assigned addresses in the range of 1 – 247. A master addresses a slave by placing the slave address in the address field of the message. When the slave sends its response, it places its own address in this address field of the response to let the master know which slave is responding.
Address 0 is used for the broadcast address, which all slave devices recognize.
When MODbus® protocol is used on higher level networks, broadcasts may not be allowed or may be replaced by other methods. For example, MODbus® Plus uses a shared global database that can be updated with each token rotation.

How the Function Field is handled

The function code field of a message frame contains two characters (ASCII) or eight bits (RTU). Valid codes are in the range of 1 – 255 decimal. Of these, some codes are applicable to all Modicon controllers, while some codes apply only to certain models, and others are reserved for future use. Current codes are described in Chapter 2.
When a message is sent from a master to a slave device the function code field tells the slave what kind of action to perform. Examples are to read the ON/OFF states of a group of discrete coils or inputs; to read the data contents of a group of registers; to read the diagnostic status of the slave; to write to designated coils or registers; or to allow loading, recording, or verifying the program within the slave.
When the slave responds to the master, it uses the function code field to indicate either a normal (error–free) response or that some kind of error occurred (called an exception response). For a normal response, the slave simply echoes the original function code. For an exception response, the slave returns a code that is equivalent to the original function code with its most–significant bit set to logic 1.
For example, a message from master to slave to read a group of holding registers would have the following function code:

    0000 0011 (Hexadecimal 03)

If the slave device takes the requested action without error, it returns the same code in its response. If an exception occurs, it returns:

    1000 0011 (Hexadecimal 83)

In addition to its modification of the function code for an exception response, the slave places a unique code into the data field of the response message. This tells the master what kind of error occurred, or the reason for the exception.
The master device's application program has the responsibility of handling exception responses. Typical processes are to post subsequent retries of the message, to try diagnostic messages to the slave, and to notify operators.

Contents of the Data Field

The data field is constructed using sets of two hexadecimal digits, in the range of 00 to FF hexadecimal. These can be made from a pair of ASCII characters, or from one RTU character, according to the network's serial transmission mode.
The data field of messages sent from a master to slave devices contains additional information which the slave must use to take the action defined by the function code. This can include items like discrete and register addresses, the quantity of items to be handled, and the count of actual data bytes in the field.
For example, if the master requests a slave to read a group of holding registers (function code 03), the data field specifies the starting register and how many registers are to be read. If the master writes to a group of registers in the slave (function code 10 hexadecimal), the data field specifies the starting register, how many registers to write, the count of data bytes to follow in the data field, and the data to be written into the registers.
If no error occurs, the data field of a response from a slave to a master contains the data requested. If an error occurs, the field contains an exception code that the master application can use to determine the next action to be taken. The data field can be nonexistent (of zero length) in certain kinds of messages. For example, in a request from a master device for a slave to respond with its communications event log (function code 0B hexadecimal), the slave does not require any additional information. The function code alone specifies the action.

Contents of the Error Checking Field

Two kinds of error–checking methods are used for standard MODbus® networks.
The error checking field contents depend upon the method that is being used.

ASCII

When ASCII mode is used for character framing, the error checking field contains two ASCII characters. The error check characters are the result of a Longitudinal Redundancy Check (LRC) calculation that is performed on the message contents, exclusive of the beginning 'colon' and terminating CRLF characters.
The LRC characters are appended to the message as the last field preceding the CRLF characters.

RTU

When RTU mode is used for character framing, the error checking field contains a 16–bit value implemented as two 8–bit bytes. The error check value is the result of a Cyclical Redundancy Check calculation performed on the message contents.
The CRC field is appended to the message as the last field in the message.
When this is done, the low–order byte of the field is appended first, followed by the high–order byte.
The CRC high–order byte is the last byte to be sent in the message.
Additional information about error checking is contained later in this chapter.
Detailed steps for generating LRC and CRC fields can be found in Appendix C.

How Characters are transmitted serially

When messages are transmitted on standard MODbus® serial networks, each character or byte is sent in this order (left to right):

Least Significant Bit (LSB) . . . Most Significant Bit (MSB)

With ASCII character framing, the bit sequence is:

With Parity Checking

Start	1	2	3	4	5	6	7	Par	Stop

Without Parity Checking

Start	1	2	3	4	5	6	7	Stop	Stop

Figure 4: Bit Order (ASCII)

With RTU character framing, the bit sequence is:

With Parity Checking

Start	1	2	3	4	5	6	7	8	Par	Stop

Without Parity Checking

Start	1	2	3	4	5	6	7	8	Stop	Stop

Figure 5: Bit Order (RTU)

Error Checking Methods

Standard MODbus® serial networks use two kinds of error checking. Parity checking (even or odd) can be optionally applied to each character. Frame checking (LRC or CRC) is applied to the entire message. Both the character check and message frame check are generated in the master device and applied to the message contents before transmission. The slave device checks each character and the entire message frame during receipt.

The master is configured by the user to wait for a predetermined timeout interval before aborting the transaction. This interval is set to be long enough for any slave to respond normally. If the slave detects a transmission error, the message will not be acted upon. The slave will not construct a response to the master.
Thus the timeout will expire and allow the master's program to handle the error.
Note that a message addressed to a nonexistent slave device will also cause a timeout.

Other networks such as MAP or MODbus® Plus use frame checking at a level above the MODbus® contents of the message. On those networks, the MODbus® message LRC or CRC check field does not apply. In the case of a transmission error, the communication protocols specific to those networks notify the originating device that an error has occurred, and allow it to retry or abort according to how it has been setup. If the message is delivered, but the slave device cannot respond, a timeout error can occur which can be detected by the master's program.

Parity Checking

Users can configure controllers for Even or Odd Parity checking, or for No Parity checking. This will determine how the parity bit will be set in each character.

If either Even or Odd Parity is specified, the quantity of 1 bits will be counted in the data portion of each character (seven data bits for ASCII mode, or eight for RTU).
The parity bit will then be set to a 0 or 1 to result in an Even or Odd total of 1 bit.

For example, these eight data bits are contained in an RTU character frame:

1100 0101

The total quantity of 1 bits in the frame is four. If Even Parity is used, the frame's parity bit will be a 0, making the total quantity of 1 bits still an even number (four).
If Odd Parity is used, the parity bit will be a 1, making an odd quantity (five).

When the message is transmitted, the parity bit is calculated and applied to the frame of each character. The receiving device counts the quantity of 1 bits and sets an error if they are not the same as configured for that device (all devices on the MODbus® network must be configured to use the same parity check method).

Note that parity checking can only detect an error if an odd number of bits are picked up or dropped in a character frame during transmission. For example, if Odd Parity checking is employed, and two 1 bits are dropped from a character containing three 1 bits, the result is still an odd count of 1 bits.

If No Parity checking is specified, no parity bit is transmitted and no parity check can be made. An additional stop bit is transmitted to fill out the character frame.

LRC Checking

In ASCII mode, messages include an error–checking field that is based on a Longitudinal Redundancy Check (LRC) method. The LRC field checks the contents of the message, exclusive of the beginning 'colon' and ending CRLF pair.
It is applied regardless of any parity check method used for the individual characters of the message.
The LRC field is one byte, containing an 8–bit binary value. The LRC value is calculated by the transmitting device, which appends the LRC to the message. The receiving device calculates an LRC during receipt of the message, and compares the calculated value to the actual value it received in the LRC field. If the two values are not equal, an error results.
The LRC is calculated by adding together successive 8–bit bytes of the message, discarding any carries, and then two's complementing the result. It is performed on the ASCII message field contents excluding the 'colon' character that begins the message, and excluding the CRLF pair at the end of the message.
For applications using host computers, a detailed example of LRC generation is contained in Appendix C.

CRC Checking

In RTU mode, messages include an error–checking field that is based on a Cyclical Redundancy Check (CRC) method. The CRC field checks the contents of the entire message. It is applied regardless of any parity check method used for the individual characters of the message.
The CRC field is two bytes, containing a 16–bit binary value. The CRC value is calculated by the transmitting device, which appends the CRC to the message.
The receiving device recalculates a CRC during receipt of the message, and compares the calculated value to the actual value it received in the CRC field.
If the two values are not equal, an error results.
The CRC is started by first preloading a 16–bit register to all 1's. Then a process begins of applying successive 8–bit bytes of the message to the current contents of the register. Only the eight bits of data in each character are used for generating the CRC. Start and stop bits, and the parity bit, do not apply to the CRC.
During generation of the CRC, each 8–bit character is exclusive ORed with the register contents. Then the result is shifted in the direction of the least significant bit (LSB), with a zero filled into the most significant bit (MSB) position. The LSB is extracted and examined. If the LSB was a 1, the register is then exclusive Ored with a preset, fixed value. If the LSB was a 0, no exclusive OR takes place.
This process is repeated until eight shifts have been performed. After the last (eighth) shift, the next 8–bit byte is exclusive ORed with the register's current value, and the process repeats for eight more shifts as described above. The final contents of the register, after all the bytes of the message have been applied, is the CRC value.
When the CRC is appended to the message, the low-order byte is appended first, followed by the high-order byte.
For applications using host computers, a detailed example of CRC generation is contained in Appendix C.

Chapter 2 Data and Control Functions

MODbus® Function Formats

How Numerical Values are expressed

Unless specified otherwise, numerical values (such as addresses, codes, or data) are expressed as decimal values in the text of this section. They are expressed as hexadecimal values in the message fields of the figures,

Data Addresses in MODbus® Messages

All data addresses in MODbus® messages are referenced to zero. The first occurrence of a data item is addressed as item number zero. For example:

- The coil known as 'coil 1' in a programmable controller is addressed as coil 0000 in the data address field of a MODbus® message.
- Coil 127 decimal is addressed as coil 007E hex (126 decimal).
- Holding register 40001 is addressed as register 0000 in the data address field of the message. The function code field already specifies a 'holding register' operation. Therefore the '4XXXX' reference is implicit.
- Holding register 40108 is addressed as register 006B hex (107 decimal).

Field Contents in MODbus® Messages

Figure 6 shows an example of a MODbus® query message. Figure 7 is an example of a normal response. Both examples show the field contents in hexadecimal, and also show how a message could be framed in ASCII or in RTU mode.
The master query is a Read Holding Registers request to slave device address 06.
The message requests data from three holding registers, 40108 through 40110.
Note that the message specifies the starting register address as 0107 (006B hex).
The slave response echoes the function code, indicating this is a normal response. The 'Byte Count' field specifies how many 8–bit data items are being returned.
It shows the count of 8–bit bytes to follow in the data, for either ASCII or RTU.
With ASCII, this value is one–half the actual count of ASCII characters in the data.
In ASCII, each 4–bit hexadecimal value requires one ASCII character, therefore two ASCII characters must follow in the message to contain each 8–bit data item.

For example, the value 63 hex is sent as one 8–bit byte in RTU mode (01100011).
The same value sent in ASCII mode requires two bytes, for ASCII '6' (0110110) and '3' (0110011). The 'Byte Count' field counts this data as one 8–bit item, regardless of the character framing method (ASCII or RTU).

How to Use the Byte Count Field: When you construct responses in buffers, use a Byte Count value that equals the count of 8–bit bytes in your message data.
The value is exclusive of all other field contents, including the Byte Count field.
Figure 7 shows how the byte count field is implemented in a typical response.

QUERY

Field Name	Example (Hex)	ASCII Characters	RTU 8–Bit Field
Header		: (colon)	None
Slave Address	06	0 6	0000 0110
Function	03	0 3	0000 0011
Starting Address Hi	00	0 0	0000 0000
Starting Address Lo	6B	6 B	0110 1011
No. of Registers Hi	00	0 0	0000 0000
No. of Registers Lo	03	0 3	0000 0011
Error Check		LRC (2 chars.)	CRC (16 bits)
Trailer		CR LF	None
	Total Bytes:	17	8

Figure 6: Master Query with ASCII/RTU Framing

RESPONSE

Field Name	Example (Hex)	ASCII Characters	RTU 8–Bit Field
Header		: (colon)	None
Slave Address	06	0 6	0000 0110
Function	03	0 3	0000 0011
Byte Count	06	0 6	0000 0110
Data Hi	02	0 2	0000 0010
Data Lo	2B	2 B	0010 1011
Data Hi	00	0 0	0000 0000
Data Lo	00	0 0	0000 0000
Data Hi	00	0 0	0000 0000
Data Lo	63	6 3	0110 0011
Error Check		LRC (2 chars.)	CRC (16 bits)
Trailer		CR LF	None
	Total Bytes:	23	11

Figure 7: Slave Response with ASCII/RTU Framing

Function Codes Supported by EM4000

The listing below shows the function codes supported by EM4000.
Codes are listed in decimal.

Function	Description
03	Read Holding Registers
04	Read Input Registers
06	Preset Single Register
16	Preset Multiple Registers

03 Read Holding Registers

Description

Reads the binary contents of holding registers (4XXXX references) in the slave.
Broadcast is not supported.
Appendix B lists the maximum parameters supported by EM4000.

Query

The query message specifies the starting register and quantity of registers to be read. Registers are addressed starting at zero: registers 1–16 are addressed as 0–15.
Here is an example of a request to read registers 40108–40110 from slave device 17:

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	03
Starting Address Hi	00
Starting Address Lo	6B
No. of Points Hi	00
No. of Points Lo	03
Error Check (LRC or CRC)	—

Response

The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register, the first byte contains the high order bits and the second contains the low order bits.
The response is returned when the data is completely assembled.
EM4000 uses a 16–bit binary value.
Here is an example of a response to the query:

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	03
Byte Count	06
Data Hi (Register 40108)	02
Data Lo (Register 40108)	2B
Data Hi (Register 40109)	00
Data Lo (Register 40109)	00
Data Hi (Register 40110)	00
Data Lo (Register 40110)	64
Error Check (LRC or CRC)	—

The contents of register 40108 are shown as the two byte values of 02 2B hex or 555 decimal. The contents of registers 40109–40110 are 00 00 and 00 64 hex, or 0 and 100 decimal.

04 Read Input Registers

Description

Reads the binary contents of input registers (3XXXX references) in the slave.
Broadcast is not supported.
Appendix B lists the maximum parameters supported by EM4000.

Query

The query message specifies the starting register and quantity of registers to be read. Registers are addressed starting at zero: registers 1–16 are addressed as 0–15.
Here is an example of a request to read register 30009 from slave device 17:

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	04
Starting Address Hi	00
Starting Address Lo	08
No. of Points Hi	00
No. of Points Lo	01
Error Check (LRC or CRC)	—

Response

The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register, the first byte contains the high order bits and the second contains the low order bits.
The response is returned when the data is completely assembled.
EM4000 uses a 16–bit binary value.
Here is an example of a response to the query:

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	04
Byte Count	02
Data Hi (Register 30009)	00
Data Lo (Register 30009)	0A
Error Check (LRC or CRC)	—

The contents of register 30009 are shown as the two byte values of 00 0A hex or 10 decimal.

06 Preset Single Register

Description

Presets a value into a single holding register (4XXXX reference).
Appendix B lists the maximum parameters supported by EM4000.

Query

The query message specifies the register reference to be preset. Registers are addressed starting at zero: register 1 is addressed as 0.
The requested preset value is specified in the query data field.
EM4000 uses a 16–bit binary value.
Here is an example of a request to preset register 40002 to 00 03 hexes in slave device 17:

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	06
Register Address Hi	00
Register Address Lo	01
Preset Data Hi	00
Preset Data Lo	03
Error Check (LRC or CRC)	—

Response

The normal response is an echo of the query, returned after the register contents have been preset.
Here is an example of a response to the query:

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	06
Register Address Hi	00
Register Address Lo	01
Preset Data Hi	00
Preset Data Lo	03
Error Check (LRC or CRC)	—

16 Preset Multiple Register

Description

Presets values into a sequence of holding registers (4XXXX references).
Appendix B lists the maximum parameters supported by EM4000.

Query

The query message specifies the register reference to be preset. Registers are addressed starting at zero: register 1 is addressed as 0.
EM4000 uses a 16–bit binary value.
Here is an example of a request to preset register 40002 to 00 03 hex in slave device 17:
The query message specifies the register references to be preset. Registers are addressed starting at zero: register 1 is addressed as 0.
The requested preset values are specified in the query data field.
Here is an example of a request to preset two registers starting at 40002 to 00 0A and 01 02 hex, in slave device 17:

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	10
Starting Address Hi	00
Starting Address Lo	01
No. of Registers Hi	00
No. of Registers Lo	02
Byte Count	04
Data Hi	00
Data Lo	0A
Data Hi	01
Data Lo	02
Error Check (LRC or CRC)	––

Response

The normal response returns the slave address, function code, starting address, and quantity of registers preset.
Here is an example of a response to the query.

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	10
Starting Address	Hi 00
Starting Address Lo	01
No. of Registers Hi	00
No. of Registers Lo	02
Error Check (LRC or CRC)	––

Appendix A

Exception Responses

Exception Responses
Except for broadcast messages, when a master device sends a query to a slave device it expects a normal response. One of four possible events can occur from the master's query:

- If the slave device receives the query without a communication error, and can handle the query normally, it returns a normal response.
- If the slave does not receive the query due to a communication error, no response is returned. The master program will eventually process a timeout condition for the query.
- If the slave receives the query, but detects a communication error (parity, LRC, or CRC), no response is returned. The master program will eventually process a timeout condition for the query.
- If the slave receives the query without a communication error, but cannot handle it (for example, if the request is to read a non–existent coil or register), the slave will return an exception response informing the master of the nature of the error.

The exception response message has two fields that differentiate it from a normal response:

Function Code Field: In a normal response, the slave echoes the function code of the original query in the function code field of the response. All function codes have a most–significant bit (MSB) of 0 (their values are all below 80 hexadecimal).
In an exception response, the slave sets the MSB of the function code to 1. This makes the function code value in an exception response exactly 80 hexadecimal higher than the value would be for a normal response.
With the function code's MSB set, the master's application program can recognize the exception response and can examine the data field for the exception code.

Data Field: In a normal response, the slave may return data or statistics in the data field (any information that was requested in the query). In an exception response, the slave returns an exception code in the data field. This defines the slave condition that caused the exception.

Here is an example of a master query and slave exception response.
The field examples are shown in hexadecimal.

QUERY

Byte	Contents	Example
1	Slave Address	0A
2	Function	01
3	Starting Address Hi	04
4	Starting Address Lo	A1
5	No. of Coils Hi	00
6	No. of Coils Lo	01
7	LRC	4F

EXCEPTION RESPONSE

Byte	Contents	Example
1	Slave Address	0A
2	Function	81
3	Exception Code	06
4	LRC	73

In this example, the master addresses a query to slave device 10 (0A hex). The function code (01) is for a Read Coil Status operation. It requests the status of the coil at address 1245 (04A1 hex).
A listing of exception codes begins on the next page.

Exception Codes generated by EM4000

Code	Name	Meaning
01	ILLEGAL FUNCTION	The function code received in the query is not an allowable action for the slave. If a Poll Program Complete command was issued, this code indicates that no program function preceded it.
02	ILLEGAL DATA ADDRESS	The data address received in the query is not an allowable address for the slave.
03	ILLEGAL DATA VALUE	A value contained in the query data field is not an allowable value for the slave.

Metering
Protection
Lighting

Appendix B
Application Notes

Maximum Query / Response Parameters
The listing in this section show the maximum amount of data the EM4000 can return in a slave response. All function codes and quantities are in decimal.

EM4000

Function	**Description**	**Query**	**Response**
1	Read Coil Status	Not supported	Not supported
2	Read Input Status	Not supported	Not supported
3	Read Holding Registers	60 registers	60 registers
4	Read Input Registers	60 registers	60 registers
5	Force Single Coil	Not supported	Not supported
6	Preset Single Register	1 register	1 register
7	Read Exception Status	Not supported	Not supported
8	Diagnostics	Not supported	Not supported
9	Program	Not supported	Not supported
10	Poll	Not supported	Not supported
11	Fetch Comm. Event	Not supported	Not supported
12	Fetch Comm. Event Log	Not supported	Not supported
13	Program Controller	Not supported	Not supported
14	Poll Controller	Not supported	Not supported
15	Force Multiple Coils	Not supported	Not supported
16	Preset Multiple Registers	60 registers	60 registers
17	Report Slave ID	Not supported	Not supported
18	Program	Not supported	Not supported
19	Reset Comm. Link	Not supported	Not supported
20	Read General Reference	Not supported	Not supported
21	Write General Reference	Not supported	Not supported

Appendix C

LRC/CRC Generation

LRC Generation

The Longitudinal Redundancy Check (LRC) field is one byte, containing an 8–bit binary value. The LRC value is calculated by the transmitting device, which appends the LRC to the message. The receiving device recalculates an LRC during receipt of the message, and compares the calculated value to the actual value it received in the LRC field. If the two values are not equal, an error results.

The LRC is calculated by adding together successive 8–bit bytes in the message, discarding any carries, and then two's complementing the result. The LRC is an 8–bit field, therefore each new addition of a character that would result in a value higher than 255 decimal simply 'rolls over' the fields value through zero. Because there is no ninth bit, the carry is discarded automatically.
A procedure for generating an LRC is:

1. Add all bytes in the message, excluding the starting 'colon' and ending CRLF. Add them into an 8–bit field, so that carries will be discarded.

2. Subtract the final field value from FF hex (all 1's), to produce the ones–complement.

3. Add 1 to produce the twos–complement.

Placing the LRC into the Message

When the 8–bit LRC (2 ASCII characters) is transmitted in the message, the high–order character will be transmitted first, followed by the low–order character.
For example, if the LRC value is 61 hex (0110 0001):

Colon	Addr	Func	Data Count	Data	Data	Data	Data	LRC Hi	LRC Lo	CR	LF
								6	1		

LRC Character Sequence

CRC Generation

The Cyclical Redundancy Check (CRC) field is two bytes, containing a 16–bit binary value. The CRC value is calculated by the transmitting device, which appends the CRC to the message. The receiving device recalculates a CRC during receipt of the message, and compares the calculated value to the actual value it received in the CRC field. If the two values are not equal, an error results.

The CRC is started by first preloading a 16–bit register to all 1's. Then a process begins of applying successive 8–bit bytes of the message to the current contents of the register. Only the eight bits of data in each character are used for generating the CRC. Start and stop bits, and the parity bit, do not apply to the CRC.

During generation of the CRC, each 8–bit character is exclusive ORed with the register contents. Then the result is shifted in the direction of the least significant bit (LSB), with a zero filled into the most significant bit (MSB) position. The LSB is extracted and examined. If the LSB was a 1, the register is then exclusive Ored with a preset, fixed value. If the LSB was a 0, no exclusive OR takes place.

This process is repeated until eight shifts have been performed. After the last (eighth) shift, the next 8–bit character is exclusive ORed with the register's current value, and the process repeats for eight more shifts as described above. The final contents of the register, after all the characters of the message have been applied, is the CRC value.

A procedure for generating a CRC is:

1. Load a 16–bit register with FFFF hex (all 1's). Call this the CRC register.
2. Exclusive OR the first 8–bit byte of the message with the low–order byte of the 16–bit CRC register, putting the result in the CRC register.
3. Shift the CRC register one bit to the right (toward the LSB), zero–filling the MSB. Extract and examine the LSB.
4. (If the LSB was 0): Repeat Step 3 (another shift).
   (If the LSB was 1): Exclusive OR the CRC register with the polynomial value A001 hex (1010 0000 0000 0001).
5. Repeat Steps 3 and 4 until 8 shifts have been performed. When this is done, a complete 8–bit byte will have been processed.
6. Repeat Steps 2 through 5 for the next 8–bit byte of the message. Continue doing this until all bytes have been processed.
7. The final content of the CRC register is the CRC value.
8. When the CRC is placed into the message, its upper and lower bytes must be swapped as described below.

Placing the CRC into the Message

When the 16–bit CRC (two 8–bit bytes) is transmitted in the message, the low-order byte will be transmitted first, followed by the high-order byte.
For example, if the CRC value is 1241 hex (0001 0010 0100 0001):

Addr	Func	Data Count	Data	Data	Data	Data	CR Clo	CR CHi
							41	12

CRC Byte Sequence

Appendix D

Examples registers calculation EM4000

All input and holding registers of the EM4000 are 16-bit unsigned values. We call these raw values. Depending of the measurand this raw value has to be rescaled. This is explained in the examples below.

Rescaling the raw value into an engineering value is indicated by the following type definitions.
The kind of type for every register value is given in the input and holding register table.

Register Type	Engineering span	Rescaling
Signed integer	-32768 … 32767	**rescaling is necessary**
Unsigned integer	0 … 65535	**no rescaling is necessary**

To rescale a signed integer, which is stored as a raw value in the registers, apply the following formula:

Engineering value	=	raw value + minimum value engineering span
	=	raw value + (-32768)

Converting the engineering value back into the measuring value is done with the following formula:

Measuring value	=	Engineering value x $10^{\text{power of ten}}$

The power of ten is determined by rescaling the register value (from the holding register), into an engineering value, which then is used as a power of ten.

Note:
No extra calculations are necessary when using current and or voltage transformers. The power of ten is determined by the EM4000 firmware. The raw value is already scaled with the primary values of the current and / or voltage transformers.

Some devices do not support the different MODbus® reading functions for reading holding or input registers. Therefore the register structure of the EM4000 is re-mapped for these 2 different functions.

Function 3:	Normally reads input registers	register 30000 and further
Function 4	Normally reads holding register	register 40000 and further

In addition the functions are extended as follows:

Function 3:	register 35000 and further	reads register 40000 and further
Function 4:	register 45000 and further	reads register 30000 and further

Example 1: Current and average current (Address 30001 to 30004)

The raw value is presented by the addresses mentioned above.
The register value (raw value) range is 0 to 65535. (unsigned int)
The engineering span is also 0 to 65535 (unsigned int), so no rescaling is necessary.

Power of ten current range is presented by address 40044.
Register value (raw value) 0 to 65535
Engineering span is -32768 to 32767. (signed int), so rescaling is necessary.

For example:

I_{L1}	Raw value input register	30001	= 5000
	Raw value holding register	40044	= 32765

	Input Register Measured value	Holding Register Power of ten	Result
I_{L1}	30001	40044	
Raw value	5000	32765	
Type	unsigned int	signed int	
Engineering span	0 …65535	-32768 …32767	
Engineering value	5000	32765 + (-32768) = -3	
			5000 x 10^{-3} = **5.000 A**

Example 2: Phase voltage and average, line voltage and average (Address 30005 to 30012)

The raw value is presented by the addresses mentioned above.
The register value (raw value) range is 0 to 65535. (unsigned int)
The engineering span is also 0 to 65535. (unsigned int), so no rescaling is necessary.

Power of ten voltage range is presented by address 40047.
Register value (raw value) is 0 to 65535
Engineering span is -32768 to 32767. (signed int), so rescalling is necessary.

For example:

U_{L1-N}	Raw value input register	30005	= 23000
	Raw value holding register	40047	= 32766

	Input Register Measured value	Holding Register Power of ten	Result
U_{L1-N}	30005	40047	
Raw value	23000	32766	
Type	unsigned int	signed int	
Engineering span	0 ... 65535	-32768 ... 32767	
Engineering value	23000	32766 + (-32768) = -2	
			23000 x 10^{-2} = **230.00 V**

Example 3: **Active power, Reactive power**
(Address 30014 to 30017, 30022 to 30025)

The raw value is presented by the addresses mentioned above.
The register value (raw value) range is 0 to 65535. (unsigned int)
The engineering span is -32768 to 32767. (signed int), so rescaling is necessary.

Power of ten address 40048 (unsigned int).
Register value (raw value) is 0 to 65535
Engineering span is -32768 to 32767. (signed int), so rescaling is necessary.

For example:
Raw value is 33800 on address 30014
Raw value is 32770 on address 40048

$P_{W\,L1}$	Raw value input register	30014	= 33767
	Raw value holding register	40048	= 32770

	Input Register Measured value	Holding Register Power of ten	Result
$P_{W\,L1}$	30014	40048	
Raw value	33800	32770	
Type	signed int	signed int	
Engineering span	-32768 … 32767	-32768 ... 32767	
Engineering value	33800 + (-32768) = 1032	32770 + (-32768) = 2	
			1032 x 10^2 = 103.2 kW

Example 4: **Apparent power**
(Address 30018 to 30021)

The raw value is presented by the addresses mentioned above.
The register value (raw value) range is 0 to 65535. (unsigned int)
The engineering span is also 0 to 65535. (unsigned int), so no rescaling is necessary.

Power of ten address 40048.
Register value (raw value) 0 to 65535
Engineering span is -32768 to 32767. (signed int), so rescaling is necessary.

For example:
Raw value is 52768 on address 30018
Raw value is 32770 on address 40048

$P_{Q\,L1}$	Raw value input register	30018	= 20000
	Raw value holding register	40048	= 32770

	Input Register Measured value	Holding Register Power of ten	Result
$P_{Q\,L1}$	30018	40048	
Raw value	20000	32770	
Type	unsigned int	signed int	
Engineering span	0 … 65535	-32768 ... 32767	
Engineering value	20000	32770 + (-32768) = 2	
			20000 x 10^2 = 2000 kVa

Example 5: Cos Phi
(Address 30026 to 30029)

The raw value is presented by the addresses mentioned above.
Register value (raw value) 0 to 65535.
This value is represented as 4 significant numbers at the most. The power factor span is 2000.
The engineering span is 31767 to 33768.
Values below 31767 or higher then 33768 are not possible (this will be controlled by the firmware)
To get the measuring value back, the raw value has to be rescaled.

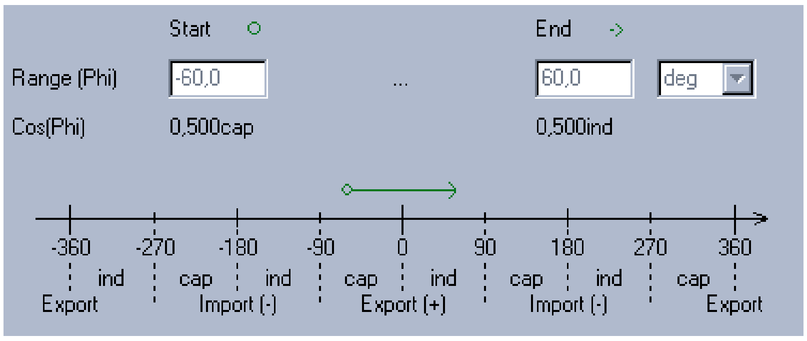
The power of ten is a constant value, -3.

For example:
Raw value is 33768 on address 30026

Cos Phi $_{L1}$ Raw value input register 30026 = 33768

	Input Register Measured value	Holding Register Power of ten	Result
Cos Phi $_{L1}$	30026		
Raw value	33768		
Type	signed int		
Engineering span	-32768 … 32767		
Engineering value	33768 + (-32768) = 1000	-3	
			1000 x 10^{-3} = 1.000

NOTE:
The value of the cos Phi, does not indicate capacitive or inductive behavior. This can be detected through the phase angle value (Phi => register 30034 .. 30037).
In the diagram below the relation between phase angle (Phi) and inductive and capacitive behavior is displayed. The span of Phi is 0..360 degrees, so -90 degrees corresponds with 270 degrees in the register of the EM4000.



Example 6: **Watt Hours and var hours (import and export)**
(Address 30038 to 30045 import Wh[*1],
Address 30047 to 30054 import varh,
Address 30055 to 30062 export Wh,
Address 30063 to 30070 export varh)

Important Note *1:

From software version 2.5 and higher, the registers are scaled with a power of ten.
The earlier software versions have only a Watt Hour counter without the power of ten scaling. The scaling factor is 1. The counter runs up or down, according to export or import operation.
In software version V2.50 the import and export functions are divided in separate registers.
Therefore earlier products are not compatible if these registers are used and an application software chance is necessary.

The Wh and varh registers consist of 2 registers, a most significant word (MSW) and a least significant word (LSW). The computation of the Wh and varh value is:

$$(\text{MSW} \times 2^{16} + \text{LSW}) \times \text{Power of ten}$$

The raw value is presented by the addresses mentioned above.
The register value (raw value) range is 0 to 65535. (unsigned int)
The engineering span is 0 to 65535. (unsigned int), so no rescaling is necessary.

Power of ten address 40049 (signed int).
Register value (raw value) is 0 to 65535
Engineering span is -32768 to 32767. (signed int), so rescaling is necessary.

For example:
Raw value is 150 on address 30038 (MSW)
Raw value is 2500 on address 30039 (LSW)
Raw value is 32771 on address 40049

Wh $_{L1}$	Raw value input register	30038	= 150
	Raw value input register	30039	= 2500
	Raw value holding register	40049	= 32771

	Input Register Measured value	Holding Register Power of ten	Result
Wh $_{L1}$	30038 | 30039	40049	
Raw value	150 | 2500	32771	
Type	unsigned int	signed int	
Engineering span	0 … 65535	-32768 ... 32767	
Engineering value	$150 \times 2^{16} + 2500 =$ 9 832 900	32771 + (-32768) = 3	
			9 832 900 x 10^3 = 9 832.9 MWh

Input registers EM4000

Address	Description	Type	Unit	Power of ten	Comments
30001	Current L1	unsigned int	A	-5, -4, -3, -2, -1, 0	Primairy value
30002	Current L2	unsigned int	A	-5, -4, -3, -2, -1, 0	Primairy value
30003	Current L3	unsigned int	A	-5, -4, -3, -2, -1, 0	Primairy value
30004	Average current	unsigned int	A	-5, -4, -3, -2, -1, 1	Primairy value
30005	Phase voltage L1 - N	unsigned int	V	-3, -2, -1, 0	Primairy value
30006	Phase voltage L2 - N	unsigned int	V	-3, -2, -1, 0	Primairy value
30007	Phase voltage L3 - N	unsigned int	V	-3, -2, -1, 0	Primairy value
30008	Average of all phase voltages	unsigned int	V	-3, -2, -1, 1	Primairy value
30009	Line voltage L1 - L2	unsigned int	V	-3, -2, -1, 0	Primairy value
30010	Line voltage L2 - L3	unsigned int	V	-3, -2, -1, 0	Primairy value
30011	Line voltage L3 - L1	unsigned int	V	-3, -2, -1, 0	Primairy value
30012	Average of all line voltages	unsigned int	V	-3, -2, -1, 1	Primairy value
30013	Frequency	unsigned int	Hz	-2	
30014	Active power L1	signed int	W	-3, -2, -1, 0, 1, 2, 3, 4, 5	
30015	Active power L2	signed int	W	-3, -2, -1, 0, 1, 2, 3, 4, 5	
30016	Active power L3	signed int	W	-3, -2, -1, 0, 1, 2, 3, 4, 5	
30017	Active power sum	signed int	W	-3, -2, -1, 0, 1, 2, 3, 4, 5	
30018	Apparent power L1	unsigned int	VA	-3, -2, -1, 0, 1, 2, 3, 4, 5	
30019	Apparent power L2	unsigned int	VA	-3, -2, -1, 0, 1, 2, 3, 4, 5	
30020	Apparent power L3	unsigned int	VA	-3, -2, -1, 0, 1, 2, 3, 4, 5	
30021	Apparent power sum	unsigned int	VA	-3, -2, -1, 0, 1, 2, 3, 4, 5	
30022	Reactive power L1	signed int	var	-3, -2, -1, 0, 1, 2, 3, 4, 5	
30023	Reactive power L2	signed int	var	-3, -2, -1, 0, 1, 2, 3, 4, 5	
30024	Reactive power L3	signed int	var	-3, -2, -1, 0, 1, 2, 3, 4, 5	
30025	Reactive power sum	signed int	var	-3, -2, -1, 0, 1, 2, 3, 4, 5	
30026	Cos phi L1	signed int		-3	
30027	Cos phi L2	signed int		-3	
30028	Cos phi L3	signed int		-3	
30029	Cos phi average sum	signed int		-3	
30030	Sin phi L1	signed int		-3	
30031	Sin phi L2	signed int		-3	
30032	Sin phi L3	signed int		-3	
30033	Sin phi average sum	signed int		-3	
30034	phi L1	unsigned int		-2	
30035	phi L2	unsigned int		-2	
30036	phi L3	unsigned int		-2	
30037	phi sum average	unsigned int		-2	
30038 30039	Wh counter phase 1 MSW Wh counter phase 1 LSW	unsigned int unsigned int	Wh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Import
30040 30041	Wh counter phase 2 MSW Wh counter phase 2 LSW	unsigned int unsigned int	Wh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Import
30042 30043	Wh counter phase 3 MSW Wh counter phase 3 LSW	unsigned int unsigned int	Wh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Import
30044 30045	Wh counter sum MSW Wh counter sum LSW	unsigned int unsigned int	Wh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Import
30046	Temperature	signed int	°C	-1	
30047 30048	varh counter phase 1 MSW varh counter phase 1 LSW	unsigned int unsigned int	varh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Import
30049 30050	varh counter phase 2 MSW varh counter phase 2 LSW	unsigned int unsigned int	varh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Import
30051 30052	varh counter phase 3 MSW varh counter phase 3 LSW	unsigned int unsigned int	varh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Import
30053 30054	varh counter sum MSW varh counter sum LSW	unsigned int unsigned int	varh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Import

Address	Description	Type	Unit	Power of ten	Comments
30055 30056	Wh counter phase 1 MSW Wh counter phase 1 LSW	unsigned int unsigned int	Wh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Export
30057 30058	Wh counter phase 2 MSW Wh counter phase 2 LSW	unsigned int unsigned int	Wh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Export
30059 30060	Wh counter phase 3 MSW Wh counter phase 3 LSW	unsigned int unsigned int	Wh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Export
30061 30062	Wh counter sum MSW Wh counter sum LSW	unsigned int unsigned int	Wh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Export
30063 30064	varh counter phase 1 MSW varh counter phase 1 LSW	unsigned int unsigned int	varh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Export
30065 30066	varh counter phase 2 MSW varh counter phase 2 LSW	unsigned int unsigned int	varh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Export
30067 30068	varh counter phase 3 MSW varh counter phase 3 LSW	unsigned int unsigned int	varh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Export
30069 30070	varh counter sum MSW varh counter sum LSW	unsigned int unsigned int	varh	-3, -2, -1, 0, 1, 2, 3, 4, 5	Export

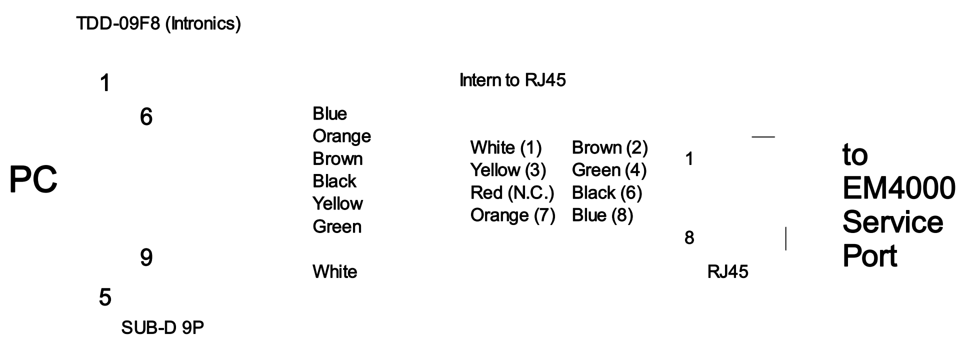
Holding registers EM4000

Address	Description	Setting range	Type	Unit	Presetting	Register value	Remarks
0001	EM4000 device address	1 … 247	unsigned int		1	1_h	valid data 1 to 247
40002	UART1 setting register service port Bit 0 ..2 Bit 3 Bit 4, 5 Bit 6 Bit 7 … 15		unsigned int		20	14_h	19k2, even parity, 1 stopbit Baudrate 0=1200, 1=2400, 2=4800, 3=9600 4=19200, 5=38400, 6=57600 0= 8 databits; 1= 7 databits parity 0= none, 1= even, 2 = odd 0= 1 stopbit, 1= 2 stopbits Spare
40003	UART2 setting register RS232 & RS485 Bit 0 ..2 Bit 3 Bit 4, 5 Bit 6 Bit 7,8 Bit 9 … 15		unsigned int		148	94_h	19k2, even parity, 1 stopbit Baudrate 0=1200, 1=2400, 2=4800, 3=9600 4=19200, 5=38400, 6=57600 0= 8 databits; 1= 7 databits parity 0= none, 1= even, 2 = odd 0= 1 stopbit, 1= 2 stopbits 0 = No handshake, 1 = RS485 2 = Half Duplex Modem, 3 = Dialup Modem Spare
40010	Current transformer primary	1 … 60000	unsigned int	A	1	1_h	
40011	Current transformer secondary	1 … 5	unsigned int	A	1	1_h	valid data 1 or 5
40012	Current value nominal	4000 … 60000	unsigned int	A	50000	$C350_h$	nominal value x 10000 (HEX value corresponds to the span)
40013	Voltage transformer primary	100 … 60000	unsigned int	V	400	190_h	HEX value corresponds to the span
40014	Voltage transformer secondary	5774 … 44000	unsigned int	V	40000	$9C40_h$	*100 (HEX value corresponds to the span)
40015	Phase voltage value nominal	5774 … 44000	unsigned int	V	23094	$5A36_h$	nominal value * 100 (HEX value corresponds to the span)
40016	Power grid	0 … max	unsigned int		19	13_h	
40017	Frequency	1000 … 50000	unsigned int	Hz	5000	1388h	nominal value * 100
40031	Software release	Read only	unsigned int		###		
40032	Hardware expansion	0 … 65535	Write once		0		
40033	Wh counter phase 1 MSW	Read only	unsigned int	Wh	0		Import
40034	Wh counter phase 1 LSW	Read only	unsigned int	Wh	0		Import
40035	Wh counter phase 2 MSW	Read only	unsigned int	Wh	0		Import
40036	Wh counter phase 2 LSW	Read only	unsigned int	Wh	0		Import
40037	Wh counter phase 3 MSW	Read only	unsigned int	Wh	0		Import
40038	Wh counter phase 3 LSW	Read only	unsigned int	Wh	0		Import
40039	Wh counter phase sum MSW	Read only	unsigned int	Wh	0		Import
40040	Wh counter phase sum LSW	Read only	unsigned int	Wh	0		Import
40041	Output card Identification	Read only	unsigned int		0		
40042	Min range current	Read only	unsigned int		0		Value corresponds with chosen range
40043	Max range current	Read only	unsigned int		0		
40044	Power of ten current	Read only	signed int		0		10^n n = -5, -4, -3, -2, -1, 0, 1, 2
40045	Min range voltage	Read only	unsigned int		0		
40046	Max range voltage	Read only	unsigned int		0		
40047	Power of ten voltage	Read only	signed int		0		10^n n =-3, -2, -1, 0, 1, 2
40048	Power of ten power	Read only	signed int		0		10^n n =-3, -2, -1, 0, 1, 2
40049	Power of ten Wh and varh	Read only	signed int		0		10^n n =-3, -2, -1, 0, 1, 2
40051	Serial number upper	0 … 65535	Write once		0	0	If 0 (zero), write only once, e.g. 123456-1
40052	Serial number lower	0 … 65535	Write once		0	0	If 0 (zero), write only once, e.g. 123456-1

Address	Description	Setting range	Type	Unit	Presetting	Register value	Remarks
40060	Force reset Wh counters	65535	signed int		0	0	Reset is 0xFFFF
40061	Force reset varh counters	65535	signed int		0	0	Reset is 0xFFFF
40063 40064	varh counter phase 1 MSW varh counter phase 1 LSW	Read only Read only	signed int signed int	varh	0		Import
40065 40066	varh counter phase 2 MSW varh counter phase 2 LSW	Read only Read only	signed int signed int	varh	0		Import
40067 40068	varh counter phase 3 MSW varh counter phase 3 LSW	Read only Read only	signed int signed int	varh	0		Import
40069 40070	varh counter phase sum MSW varh counter phase sum LSW	Read only Read only	signed int signed int	varh	0		Import
40131 40132	Wh counter phase 1 MSW Wh counter phase 1 LSW	Read only Read only	signed int signed int	Wh	0		Export
40133 40134	Wh counter phase 2 MSW Wh counter phase 2 LSW	Read only Read only	signed int signed int	Wh	0		Export
40135 40136	Wh counter phase 3 MSW Wh counter phase 3 LSW	Read only Read only	signed int signed int	Wh	0		Export
40137 40138	Wh counter phase sum MSW Wh counter phase sum LSW	Read only Read only	signed int signed int	Wh	0		Export
40139 40140	varh counter phase 1 MSW varh counter phase 1 LSW	Read only Read only	signed int signed int	varh	0		Export
40141 40142	varh counter phase 2 MSW varh counter phase 2 LSW	Read only Read only	signed int signed int	varh	0		Export
40143 40144	varh counter phase 3 MSW varh counter phase 3 LSW	Read only Read only	signed int signed int	varh	0		Export
40145 40146	varh counter phase sum MSW varh counter phase sum LSW	Read only Read only	signed int signed int	varh	0		Export

Connecting the EM4000 to a PC

Wiring diagram SUBD-9P to RJ45 Adapter



In combination with 1 : 1 RJ45 cable

Connect the 1 : 1 serial interface cable to the RJ45 Service Port on the EM4000.
Connect the Sub-D connector to COM1 on your PC.

Data port settings (RTU)

Address 40003	14h is RS232 19k2 even parity 8-databits 1-stopbit
Address 40003	94h is RS485 19k2 even parity 8-databits 1-stopbit

For further details see Holding Register Table.