Modbus Protocol

Table of Contents

Chapter 1
Modbus Protocol

Introducing Modbus Protocol

Two Serial Transmission Modes

Modbus Message Framing

Error Checking Methods

1.1 Introducing Modbus Protocol

Modicon programmable controllers can communicate with each other and with other devices over a variety of networks. Supported networks include the Modicon Modbus and Modbus Plus industrial networks, and standard networks such as MAP and Ethernet. Networks are accessed by built-in ports in the controllers or by network adapters, option modules, and gateways that are available from Schneider Automation. For original equipment manufacturers, Schneider Automation ModConnect partner programs are available for closely integrating networks like Modbus Plus into proprietary product designs.

The common language used by all Modicon controllers is the Modbus protocol. This protocol defines a message structure that controllers will recognize and use, regardless of the type of networks over which they communicate. It describes the process a controller uses to request access to another device, how it will respond to requests from the other devices, and how errors will be detected and reported. It establishes a common format for the layout and contents of message fields.

The Modbus protocol provides the internal standard that the Modicon controllers use for parsing messages. During communications on a Modbus network, the protocol determines how each controller will know its device 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. If a reply is required, the controller will construct the reply message and send it using Modbus protocol.

On other networks, messages containing Modbus protocol are imbedded into the frame or packet structure that is used on the network. For example, Modicon network controllers for Modbus Plus or MAP, with associated application software libraries and drivers, provide conversion between the imbedded Modbus message protocol and the specific framing protocols those networks use to communicate between their node devices.

This conversion also extends to resolving node addresses, routing paths, and error-checking methods specific to each kind of network. For example, Modbus device addresses contained in the Modbus protocol will be converted into node addresses prior to transmission of the messages. Error-checking fields will also be applied to message packets, consistent with each network's protocol. At the final point of delivery, however-for example, a controller-the contents of the imbedded message, written using Modbus protocol, define the action to be taken.

Figure 1 shows how devices might be interconnected in a hierarchy of networks that employ widely differing communication techniques. In message transactions, the Modbus protocol imbedded into each network's packet structure provides the common language by which the devices can exchange data.

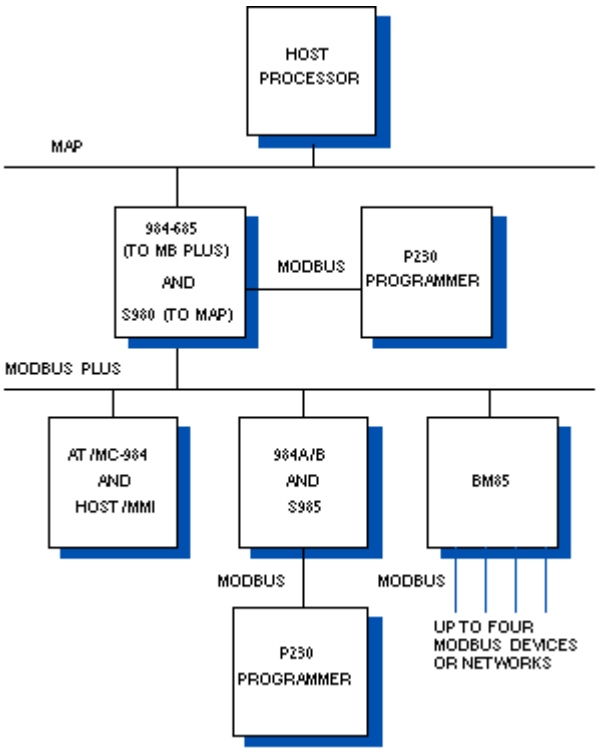

Figure 1 Overview of Modbus Protocol Application

1.1.1 Transactions on Modbus Networks

Standard Modbus ports on Modicon controllers use an RS-232C compatible serial interface that defines connector pinouts, 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 (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 (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.

1.1.2 Transactions on Other Kinds of Networks

In addition to their standard Modbus capabilities, some Modicon controller models can communicate over Modbus Plus using built-in ports or network adapters, and over MAP, using network adapters.

On these networks, the controllers communicate using a peer-to-peer technique, in which any controller can initiate transactions with the other controllers. Thus a controller may operate either as a slave or as a master in separate transactions. Multiple internal paths are frequently provided to allow concurrent processing of master and slave transactions.

At the message level, the Modbus protocol still applies the master-slave principle even though the network communication method is peer-to-peer. If a controller originates a message, it does so as a master device, and expects a response from a slave device. Similarly, when a controller receives a message it constructs a slave response and returns it to the originating controller.

1.1.3 The Query-Response Cycle

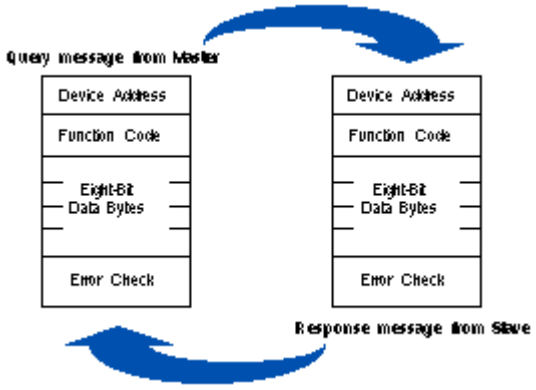

Figure 2 Master-Slave Query-Response Cycle

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.

1.2 Two Serial Transmission Modes

Controllers can be setup to communicate on standard Modbus networks using either of two transmission modes: ASCII or RTU. Users select the desired mode, along with the 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 tranasmission. 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.

1.2.1 ASCII Mode

When controllers are setup to communicate on a Modbus network using ASCII (American Standard Code for Information Interchange) mode, each eight-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.

Coding System

V Hexadecimal, ASCII characters 0 ... 9, A ... F

V One hexadecimal character contained in each ASCII character of the message

Bits per Byte

V 1 start bit

V 7 data bits, least significant bit sent first

V 1 bit for even / odd parity-no bit for no parity

V 1 stop bit if parity is used-2 bits if no parity

Error Check Field

V Longitudinal Redundancy Check (LRC)

1.2.2 RTU Mode

When controllers are setup to communicate on a Modbus network using RTU (Remote Terminal Unit) mode, each eight-bit byte in a message contains two four-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.

Coding System

V Eight-bit binary, hexadecimal 0 ... 9, A ... F

V Two hexadecimal characters contained in each eight-bit field of the message

Bits per Byte

V 1 start bit

V 8 data bits, least significant bit sent first

V 1 bit for even / odd parity-no bit for no parity

V 1 stop bit if parity is used-2 bits if no parity

Error Check Field

V Cyclical Redundancy Check (CRC)

1.3 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.)

1.3.1 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 3 ASCII Message Frame

Exception

With the 584 and 984A/B/X controllers, an ASCII message can normally terminate after the LRC field without the CRLF characters being sent. An interval of at least one second must then occur. If this happens, the controller will assume that the message terminated normally.

1.3.2 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 (shown as T1-T2-T3-T4 in the figure below). 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.

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 4 RTU Message Frame

1.3.3 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.

1.3.4 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.

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 a 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.

1.3.5 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.

1.3.6 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 eight-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 Chapter .

1.3.7 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:

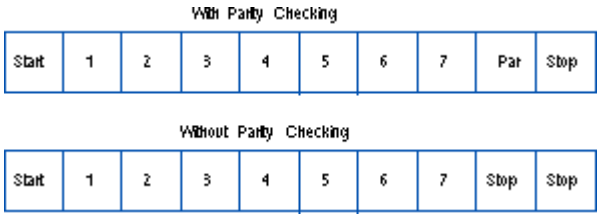

Figure 5 Bit Order (ASCII)

With RTU character framing, the bit sequence is:

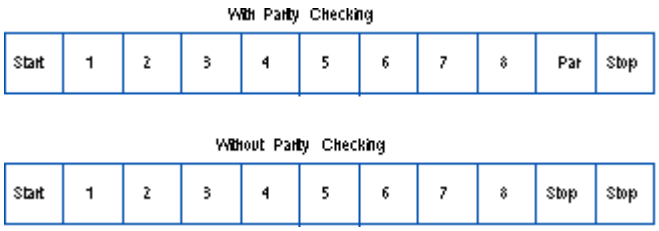

Figure 6 Bit Order (RTU)

1.4 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: 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.

1.4.1 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 bits. 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.

1.4.2 LRC Checking

In ASCII mode, messages include an error-checking field that is based on a 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 eight-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 eight-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.

In ladder logic, the CKSM function calculates a LRC from the message contents. For applications using host computers, a detailed example of LRC generation is contained in Appendix C.

1.4.3 CRC Checking

In RTU mode, messages include an error-checking field that is based on a 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 eight-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 eight-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 eight-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.

In ladder logic, the CKSM function calculates a CRC from the message contents. For applications using host computers, a detailed example of CRC generation is given on page .

Chapter 2
Data and Control Functions

Modbus Function Formats

Function Codes

Read Holding Registers

Force Single Coil

Preset Single Register

2.1 Modbus Function Formats

☞
Note: 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.

2.1.1 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:

V Coil 1 in a programmable controller is addressed as coil 0000 in the data address field of a Modbus message

V Coil 127 decimal is addressed as coil 007E hex (126 decimal)

V 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 4*x* reference is implicit.

V Holding register 40108 is addressed as register 006B hex (107 decimal)

2.1.2 Field Contents in Modbus Messages

The following tables show examples of a Modbus query and 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.

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

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

Note: 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 eight-bit data items are being returned. It shows the count of eight-bit bytes to follow in the data, for either ASCII or RTU. With ASCII, this value is half the actual count of ASCII characters in the data. In ASCII, each four-bit hexadecimal value requires one ASCII character, therefore two ASCII characters must follow in the message to contain each eight-bit data item.

For example, the value 63 hex is sent as one eight-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 eight-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 eight-bit bytes in your message data. The value is exclusive of all other field contents, including the Byte Count field.

listed in decimal; Y indicates that the function is supported, and N indicates that it is not supported.

Code	Name	384	484	584	884	M84	984
01	Read Coil Status	Y	Y	Y	Y	Y	Y
02	Read Input Status	Y	Y	Y	Y	Y	Y
03	Read Holding Registers	Y	Y	Y	Y	Y	Y
04	Read Input Registers	Y	Y	Y	Y	Y	Y
05	Force Single Coil	Y	Y	Y	Y	Y	Y
06	Preset Single Register	Y	Y	Y	Y	Y	Y
07	Read Exception Status	Y	Y	Y	Y	Y	Y
08	Diagnostics	see page NO TAG)					
09	Program 484	N	Y	N	N	N	N
10	Poll 484	N	Y	N	N	N	N
11	Fetch Comm Event Counter	Y	N	Y	N	N	Y

2.2.3 03 Read Holding Registers

Reads the binary contents of holding registers (4*x* references) in the slave. Broadcast is not supported. The maximum parameters supported by various controller models are listed on page .

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:

Field Name	Example (Hex)
Slave Address	11
Function	03
Starting Address Hi	00
Starting Address Lo	6B
Number of Points Hi	00
Number 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.

Data is scanned in the slave at the rate of 125 registers per scan for 984-X8X controllers (984-685, etc), and at the rate of 32 registers per scan for all other controllers. The response is returned when the data is completely assembled.

Here is an example of a response to the query:

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.

2.2.5 05 Force Single Coil

Forces a single coil (0*x* reference) to either ON or OFF. When broadcast, the function forces the same coil reference in all attached slaves. The maximum parameters supported by various controller models are listed on page .

Note: The function will override the controller's memory protect state and the coil's disable state. The forced state will remain valid until the controller's logic next solves the coil. The coil will remain forced if it is not programmed in the controller's logic.

Query

The query message specifies the coil reference to be forced. Coils are addressed starting at zero-coil 1 is addressed as 0.

The reguested ON / OFF state is specified by a constant in the query data field. A value of FF 00 hex requests the coil to be ON. A value of 00 00 requests it to be OFF. All other values are illegal and will not affect the coil.

Here is an example of a request to force coil 173 ON in slave device 17:

Field Name	Example (Hex)
Slave Address	11
Function	05
Coil Address Hi	00
Coil Address Lo	AC
Force Data Hi	FF
Force Data Lo	00
Error Check (LRC or CRC)	--

Response

The normal response is an echo of the query, returned after the coil state has been forced.

Here is an example of a response to the query:

Field Name	Example (Hex)
Slave Address	11
Function	05
Coil Address Hi	00
Coil Address Lo	AC
Force Data Hi	FF
Force Data Lo	00
Error Check (LRC or CRC)	--

2.2.6 06 Preset Single Register

Presets a value into a single holding register (4*x* reference). When broadcast, the function presets the same register reference in all attached slaves. The maximum parameters supported by various controller models are listed on page .

Note: The function will override the controller's memory protect state. The preset value will remain valid in the register until the controller's logic next solves the register contents. The register's value will remain if it is not programmed in the controller's logic.

Query

The query message specifies the register reference to be preset. Registers are addressed starting at zero-register 1 is addressed as 0.

The reguested preset value is specified in the query data field. M84 and 484 controllers use a 10-bit binary value, with the six high order bits set to zeros. All other controllers use 16-bit values.

Here is an example of a request to preset register 40002 to 00 03 hex in slave device 17:

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:

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)	--

Chapter 4
Exception Responses

Exception Responses

Exception Codes

4.1 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:

V If the slave device receives the query without a communication error, and can handle the query normally, it returns a normal response.

V 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.

V 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.

V If the slave receives the query without a communication error, but cannot handle it (for example, if the request is to read a nonexistent 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	Number of Coils Hi	00
6	Number of Coils Lo	01
7	LRC	4F

Exception Response

Byte	Contents	Example
1	Slave Address	0A
2	Function	81
3	Exception Code	02
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).

Note: Only one coil is to be read, as specified by the number of coils field (0001).

If the coil address is nonexistent in the slave device, the slave will return the exception response with the exception code shown (02). This specifies an illegal data address for the slave. For example, if the slave is a 984-385 with 512 coils, this code would be returned.

4.2 Exception Codes

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
04	SLAVE DEVICE FAILURE	An unrecoverable error occurred while the slave was attempting to perform the requested action.
05	ACKNOWLEDGE	The slave has accepted a request and is processing it, but a long duration of time is required. This response is returned to prevent a timeout error from occurring in the master. The master can next issue a Poll Program Complete message to determine if processing is completed.
06	SLAVE DEVICE BUSY	The slave is processing a long-duration program command. The master should retransmit the message later when the slave is free.
07	NEGATIVE ACKNOWLEDGE	The slave cannot perform the program function received in the query. This code is returned for an unsuccessful programming request using function code 13 or 14 decimal. The master should request diagnostic or error information from the slave.
08	MEMORY PARITY ERROR	The slave attempted to read extended memory, but detected a parity error in the memory. The master can retry the request, but service may be required on the slave device.

Chapter 6
LRC / CRC Generation

LRC Generation

CRC Generation

6.1 LRC Generation

The Longitudinal Redundancy Check (LRC) field is one byte, containing an eight-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 eight-bit bytes in the message, discarding any carries, then two's complementing the result. The LRC is an eight-bit field, therefore each new addition of a character that would result in a value higher than 255 decimal simply rolls over the field's value through zero. Because there is no ninth bit, the carry is discarded automatically.

Generating an LRC

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

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

Step 3 Add 1 to produce the two's-complement.

Placing the LRC into the Message

When the the eight-bit LRC (two ASCII characters) is transmitted in the message, the high order character will be transmitted first, followed by the low order character-e.g., if the LRC value is 61 hex (0110 0001):

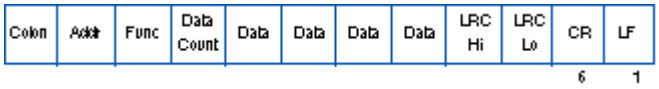

Figure 8 LRC Character Sequence

Example

An example of a C language function performing LRC generation is shown below. The function takes two arguments:

```
unsigned char *auchMsg ;        A pointer to the message buffer containing binary
                                data to be used for generating the LRC
unsigned short usDataLen ;      The quantity of bytes in the message buffer.
```

The function returns the LRC as a type unsigned char.

LRC Generation Function

```
static unsigned char LRC(auchMsg, usDataLen)

unsigned char *auchMsg ;                /* message to calculate  */
unsigned short usDataLen ;              /* LRC upon quantity of  */
/* bytes in message      */

{
        unsigned char uchLRC = 0 ;      /* LRC char initialized   */

        while (usDataLen--)             /* pass through message  */
                uchLRC += *auchMsg++ ;  /* buffer add buffer byte*/
        /* without carry         */

        return ((unsigned char)(-((char_uchLRC))) ;
        /* return twos complemen */
}
```

6.2 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 eight-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 eight-bit character is exclusive ORed with the register contents. 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 eight-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.

Generating a CRC

Step 1 Load a 16-bit register with FFFF hex (all 1's). Call this the CRC register.

Step 2 Exclusive OR the first eight-bit byte of the message with the low order byte of the 16-bit CRC register, putting the result in the CRC register.

Step 3 Shift the CRC register one bit to the right (toward the LSB), zerofilling the MSB. Extract and examine the LSB.

Step 4 If the LSB is 0, repeat Step 3 (another shift). If the LSB is 1, Exclusive OR the CRC register with the polynomial value A001 hex (1010 0000 0000 0001).

Step 5 Repeat Steps 3 and 4 until eight shifts have been performed. When this is done, a complete eight-bit byte will have been processed.

Step 6 Repeat Steps 2 ... 5 for the next eight-bit byte of the message. Continue doing this until all bytes have been processed.

Result The final contents of the CRC register is the CRC value.

Step 7 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 eight-bit bytes) is transmitted in the message, the low order byte will be transmitted first, followed by the high order byte-e.g., if the CRC value is 1241 hex (0001 0010 0100 0001):

Figure 9 CRC Byte Sequence

Example

An example of a C language function performing CRC generation is shown on the following pages. All of the possible CRC values are preloaded into two arrays, which are simply indexed as the function increments through the message buffer. One array contains all of the 256 possible CRC values for the high byte of the 16-bit CRC field, and the other array contains all of the values for the low byte.

Indexing the CRC in this way provides faster execution than would be achieved by calculating a new CRC value with each new character from the message buffer.

☞
Note: This function performs the swapping of the high/low CRC bytes internally. The bytes are already swapped in the CRC value that is returned from the function. Therefore the CRC value returned from the function can be directly placed into the message for transmission.

The function takes two arguments:

```
unsigned char *puchMsg ;        A pointer to the message buffer containing binary
                                data to be used for generating the CRC
unsigned short usDataLen ;      The quantity of bytes in the message buffer.
```

The function returns the CRC as a type unsigned short.

CRC Generation Function

```
unsigned short CRC16(puchMsg, usDataLen)

unsigned char *puchMsg ;                /* message to calculate CRC upon */
unsigned short usDataLen ;              /* quantity of bytes in message  */

{
        unsigned char uchCRCHi = 0xFF ; /* high CRC byte initialized */
        unsigned char uchCRCLo = 0xFF ; /* low CRC byte initialized  */
        unsigned uIndex ;               /* will index into CRC lookup*/


        /* table   */

        while (usDataLen--)             /* pass through message buffer */
                {
                uIndex = uchCRCHi ^ *puchMsgg++ ;       /* calculate the CRC */
                uchCRCHi = uchCRCLo ^ auchCRCHi[uIndex} ;
                uchCRCLo = auchCRCLo[uIndex] ;
                }

        return (uchCRCHi << 8 | uchCRCLo) ;
}
```

High Order Byte Table

```
/* Table of CRC values for high-order byte */

static unsigned char auchCRCHi[] = {
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,
0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,
0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,
0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40
} ;
```

Low Order Byte Table

```
/* Table of CRC values for low-order byte */

static char auchCRCLo[] = {
0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06,
0x07, 0xC7, 0x05, 0xC5, 0xC4, 0x04, 0xCC, 0x0C, 0x0D, 0xCD,
0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09,
0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A,
0x1E, 0xDE, 0xDF, 0x1F, 0xDD, 0x1D, 0x1C, 0xDC, 0x14, 0xD4,
0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13, 0xD3,
0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3,
0xF2, 0x32, 0x36, 0xF6, 0xF7, 0x37, 0xF5, 0x35, 0x34, 0xF4,
0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A,
0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29,
0xEB, 0x2B, 0x2A, 0xEA, 0xEE, 0x2E, 0x2F, 0xEF, 0x2D, 0xED,
0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26,
0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60,
0x61, 0xA1, 0x63, 0xA3, 0xA2, 0x62, 0x66, 0xA6, 0xA7, 0x67,
0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC, 0xAD, 0x6D, 0xAF, 0x6F,
0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68,
0x78, 0xB8, 0xB9, 0x79, 0xBB, 0x7B, 0x7A, 0xBA, 0xBE, 0x7E,
0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C, 0xB4, 0x74, 0x75, 0xB5,
0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71,
0x70, 0xB0, 0x50, 0x90, 0x91, 0x51, 0x93, 0x53, 0x52, 0x92,
0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54, 0x9C, 0x5C,
0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B,
0x99, 0x59, 0x58, 0x98, 0x88, 0x48, 0x49, 0x89, 0x4B, 0x8B,
0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C, 0x8C,
0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42,
0x43, 0x83, 0x41, 0x81, 0x80, 0x40
} ;
```