Modicon
Modbus Protocol
Reference Guide

PI–MBUS–300 Rev. J

Modicon
Modbus Protocol
Reference Guide

PI–MBUS–300 Rev. J

June 1996

MODICON, Inc., Industrial Automation Systems
One High Street
North Andover, Massachusetts 01845

Preface

This guide is written for the person who will use Modicon Modbus protocols and messages for communication in Modicon programmable controller applications. It describes how messages are constructed, and how transactions take place using Modbus protocol.

This guide should be used in conjunction with Modicon user guides for the types of networks and programmable controllers present in the application. Familiarity with your network layout, and with your control application, is assumed.

The data and illustrations in this book are not binding. We reserve the right to modify our products in line with our policy of continuous product improvement. Information in this document is subject to change without notice and should not be construed as a commitment by Modicon, Inc., Industrial Automation Systems.

Modicon, Inc. assumes no responsibility for any errors that may appear in this document. If you have any suggestions for improvements, or have found any errors in this publication, please notify us.

The following are trademarks of Modicon, Inc.:

Modbus	984	P190	SM85
ModConnect	BM85	RR85	SQ85
Modcom	BP85	SA85	

DEC® is a registered trademark of Digital Equipment Corporation.
VAX™ and DECNET™ are trademarks of Digital Equipment Corporation.

IBM® is a registered trademark of International Business Machines Corporation.
IBM AT™, IBM XT™, Micro Channel™, and Personal System/2™ are trademarks of International Business Machines Corporation.

Microsoft® and MS–DOS® are registered trademarks of Microsoft Corporation.

Western Digital® is a registered trademark of Western Digital Corporation.

Ethernet™ is a trademark of Xerox Corporation.

Printed in U. S. A.

Related Publications

Refer to the following publication for details about the application of Modicon 984 Programmable Controller systems:

GM–0984–SYS 984 Programmable Controller Systems Manual. Modicon, Inc.

Refer to the following publications for details about the application and installation of the Modbus Plus network and related communications devices:

GM–MBPL–001 Modbus Plus Network Planning and Installation Guide. Modicon, Inc.

GM–BM85–001 Modbus Plus Bridge/Multiplexer User's Guide. Modicon, Inc.

Refer to the following publication for details about the Modcom III Communications Software Library for host computer applications:

GM–MC3A–001 Modcom III Communications Software Library User's Guide. Modicon, Inc.

Contents

Figures

Chapter 1
Modbus Protocol

- Introducing Modbus Protocol
- The Two Serial Transmission Modes
- Modbus Message Framing
- Error Checking Methods

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 Modicon. For original equipment manufacturers, Modicon 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.

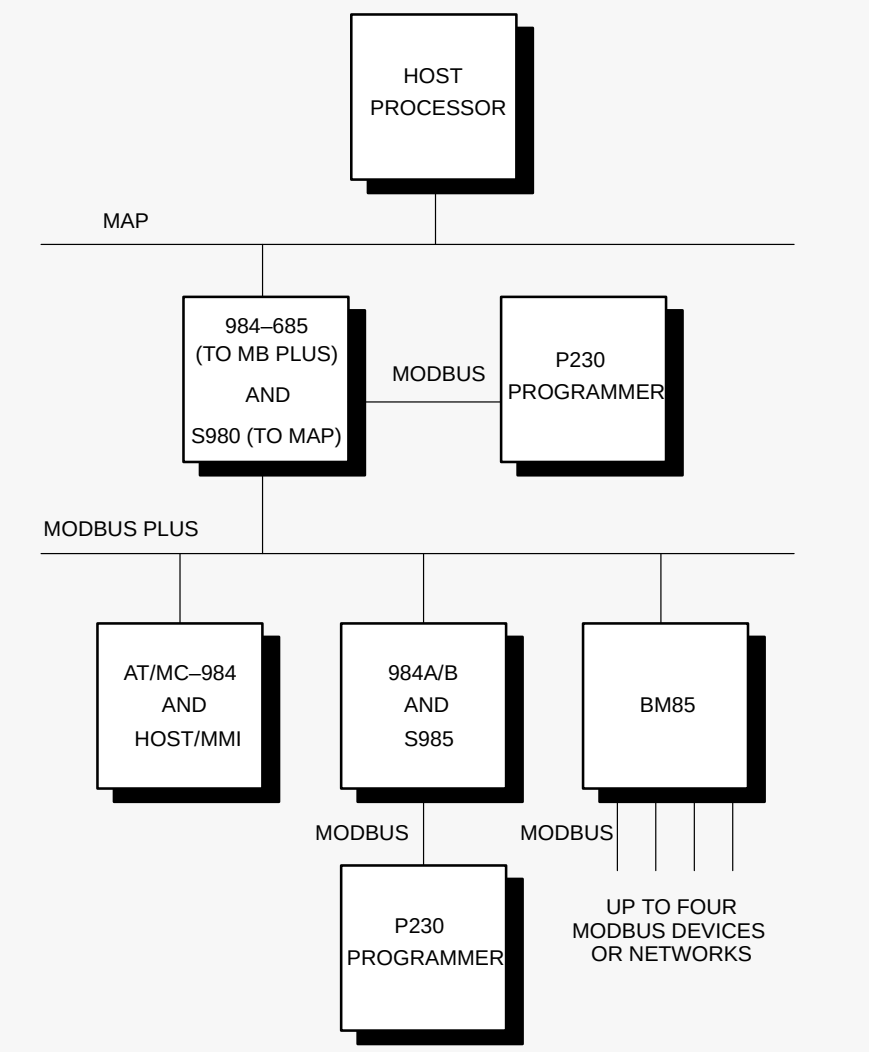

Figure 1 Overview of Modbus Protocol Application

Introducing Modbus Protocol (Continued)

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 (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.

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.

The Query–Response Cycle

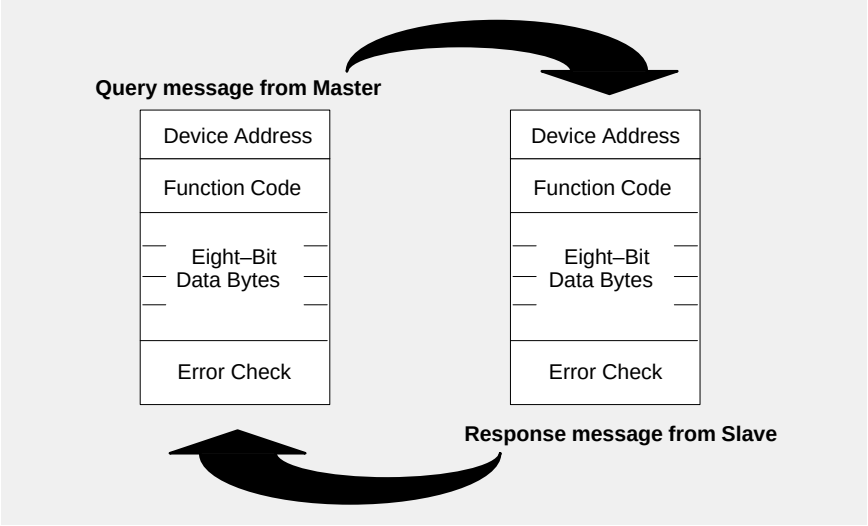

Figure 2 Master–Slave Query–Response Cycle

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.

The 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.

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 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.

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

Modbus Message Framing (Continued)

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 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.

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.

Modbus Message Framing (Continued)

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:

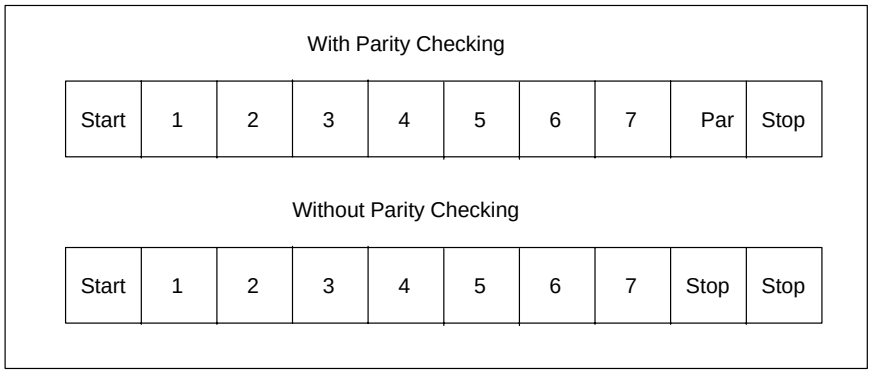

Figure 5 Bit Order (ASCII)

With RTU character framing, the bit sequence is:

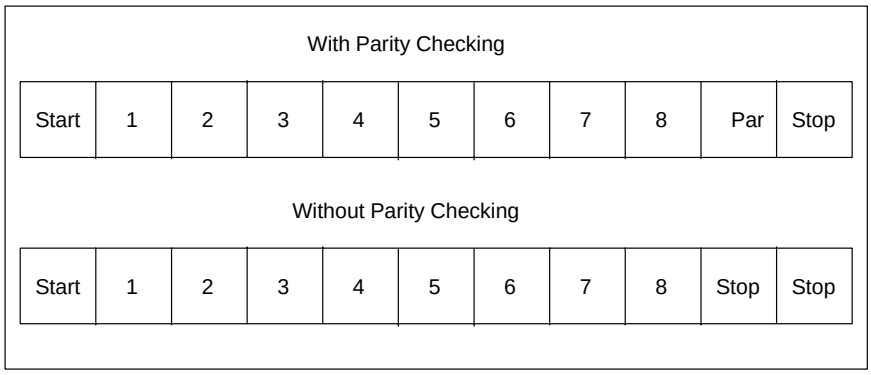

Figure 6 Bit Order (RTU)

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 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.

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.

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.

Error Checking Methods (Continued)

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.

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 contained in Appendix C.

Chapter 2
Data and Control Functions

- Modbus Function Formats
- A Summary of Function Codes
- Details of Modbus 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 7 shows an example of a Modbus query message. Figure 8 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 8 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 7 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 8 Slave Response with ASCII/RTU Framing

Modbus Function Formats (Continued)

Field Contents on Modbus Plus

Modbus messages sent on Modbus Plus networks are imbedded into the Logical Link Control (LLC) level frame. Modbus message fields consist of 8–bit bytes, similar to those used with RTU framing.

The Slave Address field is converted to a Modbus Plus routing path by the sending device. The CRC field is not sent in the Modbus message, because it would be redundant to the CRC check performed at the High–level Data Link Control (HDLC) level.

The rest of the message remains as in the standard serial format. The application software (e.g., MSTR blocks in controllers, or Modcom III in hosts) handles the framing of the message into a network packet.

Figure 9 shows how a Read Holding Registers query would be imbedded into a frame for Modbus Plus transmission.

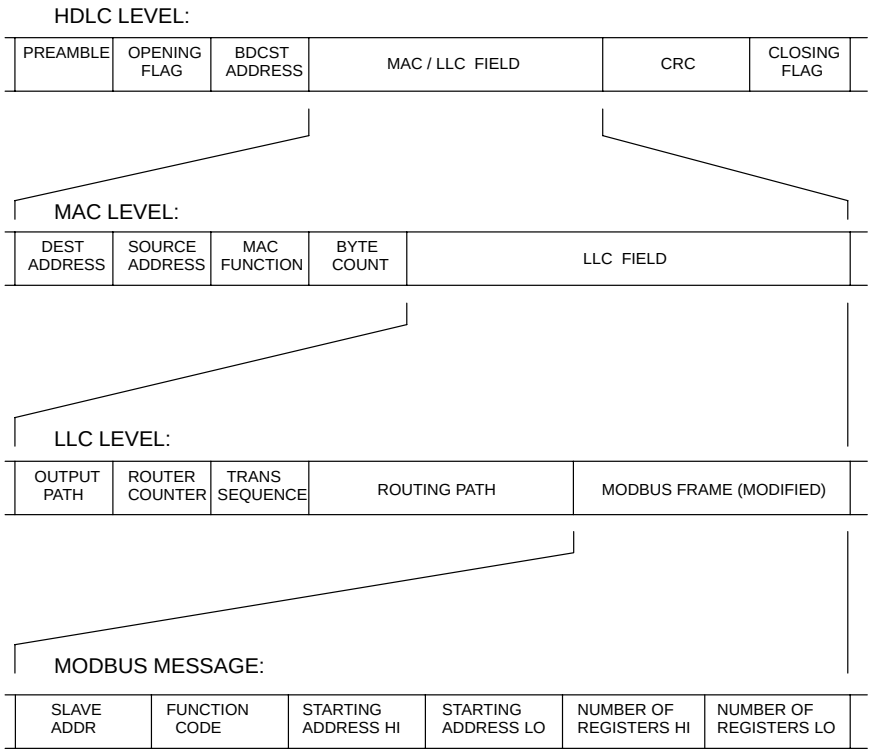

Figure 9 Field Contents on Modbus Plus

Function Codes Supported by Controllers

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

'Y' indicates that the function is supported. '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 Chapter 3)						
09	Program 484	N	Y	N	N	N	N
10	Poll 484	N	Y	N	N	N	N
11	Fetch Comm. Event Ctr.	Y	N	Y	N	N	Y
12	Fetch Comm. Event Log	Y	N	Y	N	N	Y
13	Program Controller	Y	N	Y	N	N	Y
14	Poll Controller	Y	N	Y	N	N	Y
15	Force Multiple Coils	Y	Y	Y	Y	Y	Y
16	Preset Multiple Registers	Y	Y	Y	Y	Y	Y
17	Report Slave ID	Y	Y	Y	Y	Y	Y
18	Program 884/M84	N	N	N	Y	Y	N
19	Reset Comm. Link	N	N	N	Y	Y	N
20	Read General Reference	N	N	Y	N	N	Y
21	Write General Reference	N	N	Y	N	N	Y

Code	Name	384	484	584	884	M84	984
22	Mask Write 4X Register	N	N	N	N	N	(1)
23	Read/Write 4X Registers	N	N	N	N	N	(1)
24	Read FIFO Queue	N	N	N	N	N	(1)

Notes:

(1) Function is supported in 984–785 only.

01 Read Coil Status

Description

Reads the ON/OFF status of discrete outputs (0X references, coils) in the slave. Broadcast is not supported.

Appendix B lists the maximum parameters supported by various controller models.

Query

The query message specifies the starting coil and quantity of coils to be read. Coils are addressed starting at zero: coils 1–16 are addressed as 0–15.

Here is an example of a request to read coils 20–56 from slave device 17:

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	01
Starting Address Hi	00
Starting Address Lo	13
No. of Points Hi	00
No. of Points Lo	25
Error Check (LRC or CRC)	—

Figure 10 Read Coil Status – Query

Response

The coil status in the response message is packed as one coil per bit of the data field. Status is indicated as: 1 = ON; 0 = OFF. The LSB of the first data byte contains the coil addressed in the query. The other coils follow toward the high order end of this byte, and from 'low order to high order' in subsequent bytes.

If the returned coil quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeros (toward the high order end of the byte). The Byte Count field specifies the quantity of complete bytes of data.

Here is an example of a response to the query on the opposite page:

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	01
Byte Count	05
Data (Coils 27–20)	CD
Data (Coils 35–28)	6B
Data (Coils 43–36)	B2
Data (Coils 51–44)	0E
Data (Coils 56–52)	1B
Error Check (LRC or CRC)	—

Figure 11 Read Coil Status – Response

The status of coils 27–20 is shown as the byte value CD hex, or binary 1100 1101. Coil 27 is the MSB of this byte, and coil 20 is the LSB. Left to right, the status of coils 27 through 20 is: ON–ON–OFF–OFF–ON–ON–OFF–ON.

By convention, bits within a byte are shown with the MSB to the left, and the LSB to the right. Thus the coils in the first byte are '27 through 20', from left to right. The next byte has coils '35 through 28', left to right. As the bits are transmitted serially, they flow from LSB to MSB: 20 . . . 27, 28 . . . 35, and so on.

In the last data byte, the status of coils 56–52 is shown as the byte value 1B hex, or binary 0001 1011. Coil 56 is in the fourth bit position from the left, and coil 52 is the LSB of this byte. The status of coils 56 through 52 is: ON–ON–OFF–ON–ON. Note how the three remaining bits (toward the high order end) are zero–filled.

02 Read Input Status

Description

Reads the ON/OFF status of discrete inputs (1X references) in the slave. Broadcast is not supported.

Appendix B lists the maximum parameters supported by various controller models.

Query

The query message specifies the starting input and quantity of inputs to be read. Inputs are addressed starting at zero: inputs 1–16 are addressed as 0–15.

Here is an example of a request to read inputs 10197–10218 from slave device 17:

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	02
Starting Address Hi	00
Starting Address Lo	C4
No. of Points Hi	00
No. of Points Lo	16
Error Check (LRC or CRC)	—

Figure 12 Read Input Status – Query

Response

The input status in the response message is packed as one input per bit of the data field. Status is indicated as: 1 = ON; 0 = OFF. The LSB of the first data byte contains the input addressed in the query. The other inputs follow toward the high order end of this byte, and from 'low order to high order' in subsequent bytes.

If the returned input quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeros (toward the high order end of the byte). The Byte Count field specifies the quantity of complete bytes of data.

Here is an example of a response to the query on the opposite page:

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	02
Byte Count	03
Data (Inputs 10204–10197)	AC
Data (Inputs 10212–10205)	DB
Data (Inputs 10218–10213)	35
Error Check (LRC or CRC)	—

Figure 13 Read Input Status – Response

The status of inputs 10204–10197 is shown as the byte value AC hex, or binary 1010 1100. Input 10204 is the MSB of this byte, and input 10197 is the LSB. Left to right, the status of inputs 10204 through 10197 is: ON–OFF–ON–OFF–ON–ON–OFF–OFF.

The status of inputs 10218–10213 is shown as the byte value 35 hex, or binary 0011 0101. Input 10218 is in the third bit position from the left, and input 10213 is the LSB. The status of inputs 10218 through 10213 is: ON–ON–OFF–ON–OFF–ON. Note how the two remaining bits (toward the high order end) are zero–filled.

03 Read Holding Registers

Description

Reads the binary contents of holding registers (4X references) in the slave. Broadcast is not supported.

Appendix B lists the maximum parameters supported by various controller models.

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)	—

Figure 14 Read Holding Registers – Query

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 on the opposite page:

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)	—

Figure 15 Read Holding Registers – Response

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 (3X references) in the slave. Broadcast is not supported.

Appendix B lists the maximum parameters supported by various controller models.

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)	—

Figure 16 Read Input Registers – Query

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 on the opposite page:

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)	—

Figure 17 Read Input Registers – Response

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

05 Force Single Coil

Description

Forces a single coil (0X reference) to either ON or OFF. When broadcast, the function forces the same coil reference in all attached slaves.

☞ **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.

Appendix B lists the maximum parameters supported by various controller models.

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:

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)	—

Figure 18 Force Single Coil – Query

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 on the opposite page:

RESPONSE

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)	—

Figure 19 Force Single Coil – Response

06 Preset Single Register

Description

Presets a value into a single holding register (4X reference). When broadcast, the function presets the same register reference in all attached slaves.

☞ **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.

Appendix B lists the maximum parameters supported by various controller models.

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:

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)	—

Figure 20 Preset Single Register – Query

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 on the opposite page:

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)	—

Figure 21 Preset Single Register – Response

Chapter 3
Diagnostic Subfunctions

- Modbus Function 08 – Diagnostics
- Diagnostic Subfunctions

Function 08 – Diagnostics

Description

Modbus function 08 provides a series of tests for checking the communication system between the master and slave, or for checking various internal error conditions within the slave. Broadcast is not supported.

The function uses a two–byte **subfunction code** field in the query to define the type of test to be performed. The slave echoes both the function code and subfunction code in a normal response.

Most of the diagnostic queries use a two–byte data field to send diagnostic data or control information to the slave. Some of the diagnostics cause data to be returned from the slave in the data field of a normal response.

Diagnostic Effects on the Slave

In general, issuing a diagnostic function to a slave device does not affect the running of the user program in the slave. User logic, like discretes and registers, is not accessed by the diagnostics. Certain functions can optionally reset error counters in the slave.

A slave device can, however, be forced into ‘Listen Only Mode’ in which it will monitor the messages on the communications system but not respond to them. This can affect the outcome of your application program it it depends upon any further exchange of data with the slave device. Generally, the mode is forced to remove a malfunctioning slave device from the communications system.

How This Information is Organized in Your Guide

An example diagnostics query and response are shown on the opposite page. These show the location of the function code, subfunction code, and data field within the messages.

A list of subfunction codes supported by the controllers is shown on the pages after the example response. Each subfunction code is then listed with an example of the data field contents that would apply for that diagnostic.

Query

Here is an example of a request to slave device 17 to Return Query Data. This uses a subfunction code of zero (00 00 hex in the two–byte field). The data to be returned is sent in the two–byte data field (A5 37 hex).

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	08
Subfunction Hi	00
Subfunction Lo	00
Data Hi	A5
Data Lo	37
Error Check (LRC or CRC)	—

Figure 44 Diagnostics – Query

Response

The normal response to the Return Query Data request is to loopback the same data. The function code and subfunction code are also echoed.

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	08
Subfunction Hi	00
Subfunction Lo	00
Data Hi	A5
Data Lo	37
Error Check (LRC or CRC)	—

Figure 45 Diagnostics – Response

The data fields in responses to other kinds of queries could contain error counts or other information requested by the subfunction code.

Diagnostic Codes Supported by Controllers

The listing below shows the subfunction codes supported by Modicon controllers. Codes are listed in decimal.

'Y' indicates that the subfunction is supported. 'N' indicates that it is not supported.

Code	Name	384	484	584	884	M84	984
00	Return Query Data	Y	Y	Y	Y	Y	Y
01	Restart Comm Option	Y	Y	Y	Y	Y	Y
02	Return Diagnostic Register	Y	Y	Y	Y	Y	Y
03	Change ASCII Input Delimiter	Y	Y	Y	N	N	Y
04	Force Listen Only Mode	Y	Y	Y	Y	Y	Y
05–09	Reserved						
10	Clear Ctrs and Diagnostic Reg.	Y	Y	(1)	N	N	(1)
11	Return Bus Message Count	Y	Y	Y	N	N	Y
12	Return Bus Comm. Error Count	Y	Y	Y	N	N	Y
13	Return Bus Exception Error Cnt	Y	Y	Y	N	N	Y
14	Return Slave Message Count	Y	Y	Y	N	N	N
15	Return Slave No Response Cnt	Y	Y	Y	N	N	N
16	Return Slave NAK Count	Y	Y	Y	N	N	Y
17	Return Slave Busy Count	Y	Y	Y	N	N	Y
18	Return Bus Char. Overrun Cnt	Y	Y	Y	N	N	Y
19	Return Overrun Error Count	N	N	N	Y	N	N
20	Clear Overrun Counter and Flag	N	N	N	Y	N	N
21	Get/Clear Modbus Plus Statistics	N	N	N	N	N	Y
22–up	Reserved						

Notes:

(1) Clears Counters only.

Diagnostic Subfunctions

00 Return Query Data

The data passed in the query data field is to be returned (looped back) in the response. The entire response message should be identical to the query.

Subfunction	Data Field (Query)	Data Field (Response)
00 00	Any	Echo Query Data

01 Restart Communications Option

The slave's peripheral port is to be initialized and restarted, and all of its communications event counters are to be cleared. If the port is currently in Listen Only Mode, no response is returned. This function is the only one that brings the port out of Listen Only Mode. If the port is not currently in Listen Only Mode, a normal response is returned. This occurs before the restart is executed.

When the slave receives the query, it attempts a restart and executes its power–up confidence tests. Successful completion of the tests will bring the port online.

A query data field contents of FF 00 hex causes the port's Communications Event Log to be cleared also. Contents of 00 00 leave the log as it was prior to the restart.

Subfunction	Data Field (Query)	Data Field (Response)
00 01	00 00	Echo Query Data
00 01	FF 00	Echo Query Data

08 Diagnostics (Continued)

02 Return Diagnostic Register

The contents of the slave's 16–bit diagnostic register are returned in the response.

Subfunction	Data Field (Query)	Data Field (Response)
00 02	00 00	Diagnostic Register Contents

How the Register Data is Organized

The assignment of diagnostic register bits for Modicon controllers is listed below.

In each register, bit 15 is the high–order bit. The description is TRUE when the corresponding bit is set to a logic '1'.

184/384 Diagnostic Register

Bit	Description
0	Continue on Error
1	Run Light Failed
2	T–Bus Test Failed
3	Asynchronous Bus Test Failed
4	Force Listen Only Mode
5	Not Used
6	Not Used
7	ROM Chip 0 Test Failed
8	Continuous ROM Checksum Test in Execution
9	ROM Chip 1 Test Failed
10	ROM Chip 2 Test Failed
11	ROM Chip 3 Test Failed
12	RAM Chip 5000-53FF Test Failed
13	RAM Chip 6000-67FF Test Failed, Even Addresses
14	RAM Chip 6000-67FF Test Failed, Odd Addresses
15	Timer Chip Test Failed

484 Diagnostic Register

Bit	Description
0	Continue on Error
1	CPU Test or Run Light Failed
2	Parallel Port Test Failed
3	Asynchronous Bus Test Failed
4	Timer 0 Test Failed
5	Timer 1 Test Failed
6	Timer 2 Test Failed
7	ROM Chip 0000-07FF Test Failed
8	Continuous ROM Checksum Test in Execution
9	ROM Chip 0800-0FFF Test Failed
10	ROM Chip 1000-17FF Test Failed
11	ROM Chip 1800-1FFF Test Failed
12	RAM Chip 4000-40FF Test Failed
13	RAM Chip 4100-41FF Test Failed
14	RAM Chip 4200-42FF Test Failed
15	RAM Chip 4300-43FF Test Failed

584/984 Diagnostic Register

Bit	Description
0	Illegal Configuration
1	Backup Checksum Error in High–Speed RAM
2	Logic Checksum Error
3	Invalid Node Type
4	Invalid Traffic Cop Type
5	CPU/Solve Diagnostic Failed
6	Real Time Clock Failed
7	Watchdog Timer Failed - Scan Time exceeded 250 ms.
8	No End of Logic Node detected, or quantity of end of segment words (DOIO) does not match quantity of segments configured
9	State Ram Test Failed
10	Start of Network (SON) did not begin network
11	Bad Order of Solve Table
12	Illegal Peripheral Intervention
13	Dim Awareness Flag
14	Not Used
15	Peripheral Port Stop Executed, not an error.

08 Diagnostics (Continued)

884 Diagnostic Register

Bit	Description
0	Modbus IOP Overrun Errors Flag
1	Modbus Option Overrun Errors Flag
2	Modbus IOP Failed
3	Modlbus Option Failed
4	Ourbus IOP Failed
5	Remote IO Failed
6	Main CPU Failed
7	Table RAM Checksum Failed
8	Scan Task exceeded its time limit - too much user logic
9	Not Used
10	Not Used
11	Not Used
12	Not Used
13	Not Used
14	Not Used
15	Not Used

03 Change ASCII Input Delimiter

The character ‘CHAR’ passed in the query data field becomes the end of message delimiter for future messages (replacing the default LF character). This function is useful in cases where a Line Feed is not wanted at the end of ASCII messages.

Subfunction	Data Field (Query)	Data Field (Response)
00 03	CHAR 00	Echo Query Data

04 Force Listen Only Mode

Forces the addressed slave to its Listen Only Mode for Modbus communications. This isolates it from the other devices on the network, allowing them to continue communicating without interruption from the addressed slave. No response is returned.

When the slave enters its Listen Only Mode, all active communication controls are turned off. The Ready watchdog timer is allowed to expire, locking the controls off. While in this mode, any Modbus messages addressed to the slave or broadcast are monitored, but no actions will be taken and no responses will be sent.

The only function that will be processed after the mode is entered will be the Restart Communications Option function (function code 8, subfunction 1).

Subfunction	Data Field (Query)	Data Field (Response)
00 04	00 00	No Response Returned

10 (0A Hex) Clear Counters and Diagnostic Register

For controllers other than the 584 or 984, clears all counters and the diagnostic register. For the 584 or 984, clears the counters only. Counters are also cleared upon power–up.

Subfunction	Data Field (Query)	Data Field (Response)
00 0A	00 00	Echo Query Data

08 Diagnostics (Continued)

11 (0B Hex) Return Bus Message Count

The response data field returns the quantity of messages that the slave has detected on the communications system since its last restart, clear counters operation, or power–up.

Subfunction	Data Field (Query)	Data Field (Response)
00 0B	00 00	Total Message Count

12 (0C Hex) Return Bus Communication Error Count

The response data field returns the quantity of CRC errors encountered by the slave since its last restart, clear counters operation, or power–up.

Subfunction	Data Field (Query)	Data Field (Response)
00 0C	00 00	CRC Error Count

13 (0D Hex) Return Bus Exception Error Count

The response data field returns the quantity of Modbus exception responses returned by the slave since its last restart, clear counters operation, or power–up. Exception responses are described and listed in Appendix A.

Subfunction	Data Field (Query)	Data Field (Response)
00 0D	00 00	Exception Error Count

14 (0E Hex) Return Slave Message Count

The response data field returns the quantity of messages addressed to the slave, or broadcast, that the slave has processed since its last restart, clear counters operation, or power–up.

Subfunction	Data Field (Query)	Data Field (Response)
00 0E	00 00	Slave Message Count

15 (0F Hex) Return Slave No Response Count

The response data field returns the quantity of messages addressed to the slave for which it returned no response (neither a normal response nor an exception response), since its last restart, clear counters operation, or power–up.

Subfunction	Data Field (Query)	Data Field (Response)
00 0F	00 00	Slave No Response Count

16 (10 Hex) Return Slave NAK Count

The response data field returns the quantity of messages addressed to the slave for which it returned a Negative Acknowledge (NAK) exception response, since its last restart, clear counters operation, or power–up. Exception responses are described and listed in Appendix A.

Subfunction	Data Field (Query)	Data Field (Response)
00 10	00 00	Slave NAK Count

08 Diagnostics (Continued)

17 (11 Hex) Return Slave Busy Count

The response data field returns the quantity of messages addressed to the slave for which it returned a Slave Device Busy exception response, since its last restart, clear counters operation, or power–up. Exception responses are described and listed in Appendix A.

Subfunction	Data Field (Query)	Data Field (Response)
00 11	00 00	Slave Device Busy Count

18 (12 Hex) Return Bus Character Overrun Count

The response data field returns the quantity of messages addressed to the slave that it could not handle due to a character overrun condition, since its last restart, clear counters operation, or power–up. A character overrun is caused by data characters arriving at the port faster than they can be stored, or by the loss of a character due to a hardware malfunction.

Subfunction	Data Field (Query)	Data Field (Response)
00 12	00 00	Slave Character Overrun Count

19 (13 Hex) Return IOP Overrun Count (884)

The response data field returns the quantity of messages addressed to the slave that it could not handle due to an 884 IOP overrun condition, since its last restart, clear counters operation, or power–up. An IOP overrun is caused by data characters arriving at the port faster than they can be stored, or by the loss of a character due to a hardware malfunction. This function is specific to the 884.

Subfunction	Data Field (Query)	Data Field (Response)
00 13	00 00	Slave IOP Overrun Count

20 (14 Hex) Clear Overrun Counter and Flag (884)

Clears the 884 overrun error counter and resets the error flag. The current state of the flag is found in bit 0 of the 884 diagnostic register (see subfunction 02). This function is specific to the 884.

Subfunction	Data Field (Query)	Data Field (Response)
00 14	00 00	Echo Query Data

08 Diagnostics (Continued)

21 (15 Hex) Get/Clear Modbus Plus Statistics

Returns a series of 54 16-bit words (108 bytes) in the data field of the response (this function differs from the usual two-byte length of the data field). The data contains the statistics for the Modbus Plus peer processor in the slave device.

In addition to the Function code (08) and Subfunction code (00 15 hex) in the query, a two-byte Operation field is used to specify either a 'Get Statistics' or a 'Clear Statistics' operation. The two operations are exclusive - the 'Get' operation cannot clear the statistics, and the 'Clear' operation does not return statistics prior to clearing them. Statistics are also cleared on power-up of the slave device.

The operation field immediately follows the subfunction field in the query:

— A value of 00 03 specifies the 'Get Statistics' operation.

— A value of 00 04 specifies the 'Clear Statistics' operation.

QUERY: This is the field sequence in the query:

Function	Subfunction	Operation	
08	00 15	00 03	(Get Statistics)
08	00 15	00 04	(Clear Statistics)

GET STATISTICS RESPONSE: This is the field sequence in the normal response to a Get Statistics query:

Function	Subfunction	Operation	Byte Count	Data
08	00 15	00 03	00 6C	Words 00 - 53

CLEAR STATISTICS RESPONSE: The normal response to a Clear Statistics query is an echo of the query:

Function	Subfunction	Operation
08	00 15	00 04

Modbus Plus Network Statistics

Word	Bits	Meaning
00		Node type ID:
	0	Unknown node type
	1	Programmable controller node
	2	Modbus bridge node
	3	Host computer node
	4	Bridge Plus node
	5	Peer I/O node
01	0 ... 11	Software version number in hex (to read, strip bits 12–15 from word)
	12 ... 14	Reserved
	15	Defines Word 15 error counters (see Word 15)
		Most significant bit defines use of error counters in Word 15. Least significant half of upper byte, plus lower byte, contain software version.
		Layout: | Upper Byte | Lower Byte | [ ] [——Software version in hex——] \ Most significant bit defines Word 15 error counters (see Word 15)
02		Network address for this station
03		MAC state variable:
	0	Power up state
	1	Monitor offline state
	2	Duplicate offline state
	3	Idle state
	4	Use token state
	5	Work response state
	6	Pass token state
	7	Solicit response state
	8	Check pass state
	9	Claim token state
	10	Claim response state
04		Peer status (LED code); provides status of this unit relative to the network:
	0	Monitor link operation
	32	Normal link operation
	64	Never getting token
	96	Sole station
	128	Duplicate station

Modbus Plus Network Statistics (Continued)

Word	Bits	Meaning
05		Token pass counter; increments each time this station gets the token
06		Token rotation time in ms
07	LO	Data master failed during token ownership bit map
	HI	Program master failed during token ownership bit map
08	LO	Data master token owner work bit map
	HI	Program master token owner work bit map
09	LO	Data slave token owner work bit map
	HI	Program slave token owner work bit map
10	HI	Data slave/get slave command transfer request bit map
11	LO	Program master/get master rsp transfer request bit map
	HI	Program slave/get slave command transfer request bit map
12	LO	Program master connect status bit map
	HI	Program slave automatic logout request bit map
13	LO	Pretransmit deferral error counter
	HI	Receive buffer DMA overrun error counter
14	LO	Repeated command received counter
	HI	Frame size error counter
15	If Word 1 bit 15 is *not set*, Word 15 has the following meaning:	
	LO	Receiver collision–abort error counter
	HI	Receiver alignment error counter
	If Word 1 bit 15 is *set*, Word 15 has the following meaning:	
	LO	Cable A framing error
	HI	Cable B framing error
16	LO	Receiver CRC error counter
	HI	Bad packet–length error counter
17	LO	Bad link–address error counter
	HI	Transmit buffer DMA–underrun error counter

Word	Byte	Meaning
18	LO	Bad internal packet length error counter
	HI	Bad MAC function code error counter
19	LO	Communication retry counter
	HI	Communication failed error counter
20	LO	Good receive packet success counter
	HI	No response received error counter
21	LO	Exception response received error counter
	HI	Unexpected path error counter
22	LO	Unexpected response error counter
	HI	Forgotten transaction error counter
23	LO	Active station table bit map, nodes 1 ... 8
	HI	Active station table bit map, nodes 9 ...16
24	LO	Active station table bit map, nodes 17 ... 24
	HI	Active station table bit map, nodes 25 ... 32
25	LO	Active station table bit map, nodes 33 ... 40
	HI	Active station table bit map, nodes 41 ... 48
26	LO	Active station table bit map, nodes 49 ... 56
	HI	Active station table bit map, nodes 57 ... 64
27	LO	Token station table bit map, nodes 1 ... 8
	HI	Token station table bit map, nodes 9 ... 16
28	LO	Token station table bit map, nodes 17 ... 24
	HI	Token station table bit map, nodes 25 ... 32
29	LO	Token station table bit map, nodes 33 ... 40
	HI	Token station table bit map, nodes 41 ... 48
30	LO	Token station table bit map, nodes 49 ... 56
	HI	Token station table bit map, nodes 57 ... 64
31	LO	Global data present table bit map, nodes 1 ... 8
	HI	Global data present table bit map, nodes 9 ... 16
32	LO	Global data present table bit map, nodes 17 ... 24
	HI	Global data present table bit map, nodes 25 ... 32
33	LO	Global data present table bit map, nodes 33 ... 40
	HI	Global data present table bit map, nodes 41 ... 48
34	LO	Global data present table map, nodes 49 ... 56
	HI	Global data present table bit map, nodes 57 ... 64

Modbus Plus Network Statistics (Continued)

Word	Bits	Meaning
35	LO	Receive buffer in use bit map, buffer 1–8
	HI	Receive buffer in use bit map, buffer 9 ... 16
36	LO	Receive buffer in use bit map, buffer 17 ... 24
	HI	Receive buffer in use bit map, buffer 25 ... 32
37	LO	Receive buffer in use bit map, buffer 33 ... 40
	HI	Station management command processed initiation counter
38	LO	Data master output path 1 command initiation counter
	HI	Data master output path 2 command initiation counter
39	LO	Data master output path 3 command initiation counter
	HI	Data master output path 4 command initiation counter
40	LO	Data master output path 5 command initiation counter
	HI	Data master output path 6 command initiation counter
41	LO	Data master output path 7 command initiation counter
	HI	Data master output path 8 command initiation counter
42	LO	Data slave input path 41 command processed counter
	HI	Data slave input path 42 command processed counter
43	LO	Data slave input path 43 command processed counter
	HI	Data slave input path 44 command processed counter
44	LO	Data slave input path 45 command processed counter
	HI	Data slave input path 46 command processed counter
45	LO	Data slave input path 47 command processed counter
	HI	Data slave input path 48 command processed counter
46	LO	Program master output path 81 command initiation counter
	HI	Program master output path 82 command initiation counter
47	LO	Program master output path 83 command initiation counter
	HI	Program master output path 84 command initiation counter
48	LO	Program master command initiation counter
	HI	Program master output path 86 command initiation counter
49	LO	Program master output path 87 command initiation counter
	HI	Program master output path 88 command initiation counter

Word	Bits	Meaning
50	LO	Program slave input path C1 command processed counter
	HI	Program slave input path C2 command processed counter
51	LO	Program slave input path C3 command processed counter
	HI	Program slave input path C4 command processed counter
52	LO	Program slave input path C5 command processed counter
	HI	Program slave input path C6 command processed counter
53	LO	Program slave input path C7 command processed counter
	HI	Program slave input path C8 command processed counter

Appendix A
Exception Responses

- Exception Responses
- Exception Codes

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.

Figure 46 shows 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	02
4	LRC	73

Figure 46 Master Query and Slave Exception Response

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 that only that one coil is to be read, as specified by the number of coils field (0001).

If the coil address is non–existent 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.

A listing of exception codes begins on the next page.

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 the request and is processing it, but a long duration of time will be required to do so. 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 engaged in 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.

Appendix B
Application Notes

This Appendix contains information and suggestions for using Modbus in your application.

- Maximum Query/Response Parameters for Modicon Controllers
- Estimating Serial Transaction Timing
- Application Notes for the 584 and 984A/B/X Controllers

Maximum Query/Response Parameters

The listings in this section show the maximum amount of data that each controller can request or send in a master query, or return in a slave response. All function codes and quantities are in decimal.

184/384

Function	Description	Query	Response
1	Read Coil Status	800 coils	800 coils
2	Read Input Status	800 inputs	800 inputs
3	Read Holding Registers	100 registers	100 registers
4	Read Input Registers	100 registers	100 registers
5	Force Single Coil	1 coil	1 coil
6	Preset Single Register	1 register	1 register
7	Read Exception Status	N/A	8 coils
8	Diagnostics	N/A	N/A
9	Program 484	Not supported	Not supported
10	Poll 484	Not supported	Not supported
11	Fetch Comm. Event Ctr.	N/A	N/A
12	Fetch Comm. Event Log	N/A	70 data bytes
13	Program Controller	32 data bytes	32 data bytes
14	Poll Controller	N/A	32 data bytes
15	Force Multiple Coils	800 coils	800 coils
16	Preset Multiple Registers	100 registers	100 registers
17	Report Slave ID	N/A	N/A
18	Program 884/M84	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

484

These values are for an 8K controller. See the 484 User's Guide for limits of smaller controllers.

Function	Description	Query	Response
1	Read Coil Status	512 coils	512 coils
2	Read Input Status	512 inputs	512 inputs
3	Read Holding Registers	254 registers	254 registers
4	Read Input Registers	32 registers	32 registers
5	Force Single Coil	1 coil	1 coil
6	Preset Single Register	1 register	1 register
7	Read Exception Status	N/A	8 coils
8	Diagnostics	N/A	N/A
9	Program 484	16 data bytes	16 data bytes
10	Poll 484	N/A	16 data bytes
11	Fetch Comm. Event Ctr.	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	800 coils	800 coils
16	Preset Multiple Registers	60 registers	60 registers
17	Report Slave ID	N/A	N/A
18	Program 884/M84	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

Maximum Q/R Parameters (Continued)

584

Function	Description	Query	Response
1	Read Coil Status	2000 coils	2000 coils
2	Read Input Status	2000 inputs	2000 inputs
3	Read Holding Registers	125 registers	125 registers
4	Read Input Registers	125 registers	125 registers
5	Force Single Coil	1 coil	1 coil
6	Preset Single Register	1 register	1 register
7	Read Exception Status	N/A	8 coils
8	Diagnostics	N/A	N/A
9	Program 484	Not supported	Not supported
10	Poll 484	Not supported	Not supported
11	Fetch Comm. Event Ctr.	N/A	N/A
12	Fetch Comm. Event Log	N/A	70 data bytes
13	Program Controller	33 data bytes	33 data bytes
14	Poll Controller	N/A	33 data bytes
15	Force Multiple Coils	800 coils	800 coils
16	Preset Multiple Registers	100 registers	100 registers
17	Report Slave ID	N/A	N/A
18	Program 884/M84	Not supported	Not supported
19	Reset Comm. Link	Not supported	Not supported
20	Read General Reference	(1)	(1)
21	Write General Reference	(1)	(1)

Notes:

(1) The maximum length of the entire message must not exceed 256 bytes.

884

Function	Description	Query	Response
1	Read Coil Status	2000 coils	2000 coils
2	Read Input Status	2000 inputs	2000 inputs
3	Read Holding Registers	125 registers	125 registers
4	Read Input Registers	125 registers	125 registers
5	Force Single Coil	1 coil	1 coil
6	Preset Single Register	1 register	1 register
7	Read Exception Status	N/A	8 coils
8	Diagnostics	N/A	N/A
9	Program 484	Not supported	Not supported
10	Poll 484	Not supported	Not supported
11	Fetch Comm. Event Ctr.	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	800 coils	800 coils
16	Preset Multiple Registers	100 registers	100 registers
17	Report Slave ID	N/A	N/A
18	Program 884/M84	(1)	(1)
19	Reset Comm. Link	N/A	N/A
20	Read General Reference	Not supported	Not supported
21	Write General Reference	Not supported	Not supported

Notes:

(1) The maximum length of the entire message must not exceed 256 bytes.

Maximum Q/R Parameters (Continued)

M84

Function	Description	Query	Response
1	Read Coil Status	64 coils	64 coils
2	Read Input Status	64 inputs	64 inputs
3	Read Holding Registers	32 registers	32 registers
4	Read Input Registers	4 registers	4 registers
5	Force Single Coil	1 coil	1 coil
6	Preset Single Register	1 register	1 register
7	Read Exception Status	N/A	8 coils
8	Diagnostics	N/A	N/A
9	Program 484	Not supported	Not supported
10	Poll 484	Not supported	Not supported
11	Fetch Comm. Event Ctr.	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	64 coils	64 coils
16	Preset Multiple Registers	32 registers	32 registers
17	Report Slave ID	N/A	N/A
18	Program 884/M84	(1)	(1)
19	Reset Comm. Link	N/A	N/A
20	Read General Reference	Not supported	Not supported
21	Write General Reference	Not supported	Not supported

Notes:

(1) The maximum length of the entire message must not exceed 256 bytes.

984

Function	Description	Query	Response
1	Read Coil Status	2000 coils	2000 coils
2	Read Input Status	2000 inputs	2000 inputs
3	Read Holding Registers	125 registers	125 registers
4	Read Input Registers	125 registers	125 registers
5	Force Single Coil	1 coil	1 coil
6	Preset Single Register	1 register	1 register
7	Read Exception Status	N/A	8 coils
8	Diagnostics	N/A	N/A
9	Program 484	Not supported	Not supported
10	Poll 484	Not supported	Not supported
11	Fetch Comm. Event Ctr.	N/A	N/A
12	Fetch Comm. Event Log	N/A	70 data bytes
13	Program Controller	33 data bytes	33 data bytes
14	Poll Controller	N/A	33 data bytes
15	Force Multiple Coils	800 coils	800 coils
16	Preset Multiple Registers	100 registers	100 registers
17	Report Slave ID	N/A	N/A
18	Program 884/M84	Not supported	Not supported
19	Reset Comm. Link	Not supported	Not supported
20	Read General Reference	(1)	(1)
21	Write General Reference	(1)	(1)

Notes:

(1) The maximum length of the entire message must not exceed 256 bytes.

Estimating Serial Transaction Timing

The Transaction Sequence

The following sequence of events occusr during a Modbus serial transaction. Letters in parentheses ( ) refer to the timing notes at the end of the listing.

1. The Modbus master composes the message.
2. The master device modem RTS and CTS status are checked. (A)
3. The query message is transmitted to the slave. (B)
4. The slave processes the query message. (C) (D)
5. The slave calculates an error check field. (E)
6. The slave device modem RTS and CTS status are checked. (A)
7. The response message is transmitted to the master. (B)
8. The master application acts upon the response and its data.

Timing Notes

(A) If the RTS and CTS pins are jumpered together, this time is negligible. For J478 modems, the time is about 5 ms.

(B) Use the following formula to estimate the transmission time:

$$\text{Time (ms)} = \frac{1000 \text{ X (character count) X (bits per character)}}{\text{Baud Rate}}$$

(C) The Modbus message is processed at the end of the controller scan. The worst–case delay is one scan time, which occurs if the controller has just begun a new scan. The average delay is 0.5 scan time.

The time allotted for servicing Modbus ports at the end of the controller scan (before beginning a new scan) depends upon the controller model. Timing for each model is described on the next page.

(C) Continued:

For 484 controllers the time is approximately 1.5 ms. The Modbus port is available on a contention basis with any J470/J474/J475 that is present.

For 584 and 984 controllers the time is approximately 1.5 ms for each Modbus port. The ports are serviced sequentially, starting with port 1.

For 184/384 controllers the time varies according to the amount of data being handled. It ranges from a minimum of 0.5 ms to a maximum of about 6.0 ms (for 100 registers), or 7.0 ms (for 800 coils). If a programming panel is currently being used with the controller, the Modbus port is locked out.

(D) Modbus functions 1 through 4, 15, and 16 permit the master to request more data than can be processed during the time alloted for servicing the slave's Modbus port. If the slave cannot process all of the data, it will buffer the data and process it at the end of subsequent scans.

The amount of data that can be processed during one service period at the Modbus port is as follows:

	Discretes	Registers
Micro 84	16	4
184/384	800	100
484	32	16
584	64	32
984A/B/X	64	32
984–X8X	1000	125

Note: '984–X8X' refers to 984 slot–mount models (984–385, –685, etc).

For the 884, the processing time for multiple data is as follows:

Read 768 coils: 14 scans	Force single coil: 3 scans
Read 256 inputs: 7 scans	Preset registers: 3 scans
Read 125 output registers: 5 scans	Force 768 coils: 18 scans
Read 125 input registers: 8 scans	Preset 100 registers: 10 scans

(E) LRC calculation time is less than 1 ms. CRC calculation time is about 0.3 ms for each 8 bits of data to be returned in the response.

Notes for the 584 and 984A/B/X

These application notes apply only to Modicon 584 and 984A/B/X controllers.

- **Baud Rates:** When using both Modbus ports 1 and 2, the maximum allowable *combined* baud rate is 19,200 baud.

- **Port Lockups:** When using ASCII, avoid sending ‘zero data length’ messages, or messages with no device address. For example, this is an illegal message:

  : CR LF (colon, CR, LF)

  Random port lockups can occur this kind of message is used.

- **Terminating ASCII Messages:** ASCII messages should normally terminate with a CRLF pair. With the 584 and 984A/B/X controllers, an ASCII message can terminate after the LRC field (without the CRLF characters being sent), if an interval of at least one second is allowed to occur after the LRC field. If this happens, the controller will assume that the message terminated normally.

Appendix C
LRC/CRC Generation

- LRC Generation
- 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 field's 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 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):

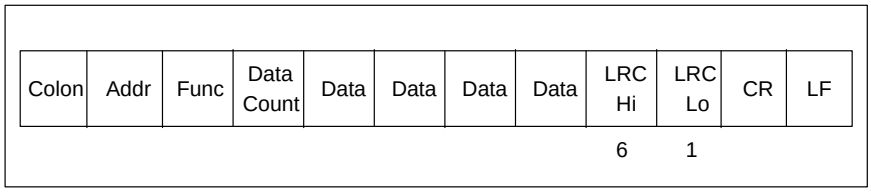

Figure 47 LRC Character Sequence

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 LRC upon       */
unsigned short usDataLen ;                  /* quantity of bytes in message        */

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

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

    return ((unsigned char)(-((char)uchLRC))) ;   /* return twos complement        */
}
```

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 contents 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):

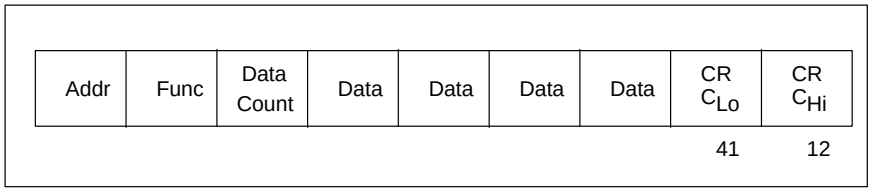

Figure 48 CRC Byte Sequence

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.

CRC Generation (Continued)

Example (Continued)

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 byte of CRC initialized        */
    unsigned char uchCRCLo = 0xFF ;         /* low byte of CRC 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
} ;
```