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.

No part of this document may be reproduced in any form or by any means, electronic or mechanical, without the express written permission of Modicon, Inc., Industrial Automation Systems. All rights reserved.

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.

Copyright © 1996, Modicon, Inc.
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

Chapter 1 Modbus Protocol 1

Introducing Modbus Protocol 2
Transactions on Modbus Networks 4
Transactions on Other Kinds of Networks 4
The Query–Response Cycle 5
The Two Serial Transmission Modes 6
ASCII Mode 6
RTU Mode 7
Modbus Message Framing 8
ASCII Framing 8
RTU Framing 9
How the Address Field is Handled 10
How the Function Field is Handled 10
Contents of the Data Field 11
Contents of the Error Checking Field 12
How Characters are Transmitted Serially 13
Error Checking Methods 14
Parity Checking 14
LRC Checking 15
CRC Checking 16

Chapter 2 Data and Control Functions 17

Modbus Function Formats 18
How Numerical Values are Expressed 18
Data Addresses in Modbus Messages 18
Field Contents in Modbus Messages 18
Field Contents on Modbus Plus 20
Function Codes Supported by Controllers 22
01 Read Coil Status 24
02 Read Input Status 26
03 Read Holding Registers 28
. . . . . .

Chapter 3 Diagnostic Subfunctions 73

Function 08 – Diagnostics 74
Diagnostic Codes Supported by Controllers 76
Diagnostic Subfunctions 77
00 Return Query Data 77
01 Restart Communications Option 77
02 Return Diagnostic Register 78
03 Change ASCII Input Delimiter 81
04 Force Listen Only Mode 81
10 (0A Hex) Clear Counters and Diagnostic Register 81
11 (0B Hex) Return Bus Message Count 82
12 (0C Hex) Return Bus Communication Error Count 82
13 (0D Hex) Return Bus Exception Error Count 82
14 (0E Hex) Return Slave Message Count 83
15 (0F Hex) Return Slave No Response Count 83
16 (10 Hex) Return Slave NAK Count 83
17 (11 Hex) Return Slave Busy Count 84
18 (12 Hex) Return Bus Character Overrun Count 84
19 (13 Hex) Return IOP Overrun Count (884) 84
20 (14 Hex) Clear Overrun Counter and Flag (884) 85
21 (15 Hex) Get/Clear Modbus Plus Statistics 86
Modbus Plus Network Statistics 87

Appendix A Exception Responses 93

Exception Responses 94
Exception Codes 96

Appendix B Application Notes 99

Maximum Query/Response Parameters 100
Estimating Serial Transaction Timing 106
Notes for the 584 and 984A/B/X 108

Appendix C LRC/CRC Generation 109

LRC Generation 110
CRC Generation 112

Figures

Figure 1 Overview of Modbus Protocol Application 3
Figure 2 Master–Slave Query–Response Cycle 5
Figure 3 ASCII Message Frame 8
Figure 4 RTU Message Frame 9
Figure 5 Bit Order (ASCII) 13
Figure 6 Bit Order (RTU) 13
Figure 7 Master Query with ASCII/RTU Framing 19
Figure 8 Slave Response with ASCII/RTU Framing 19
Figure 9 Field Contents on Modbus Plus 21

Figure 10 Read Coil Status – Query 24
Figure 11 Read Coil Status – Response 25

Figure 12 Read Input Status – Query 26
Figure 13 Read Input Status – Response 27

Figure 14 Read Holding Registers – Query 28
Figure 15 Read Holding Registers – Response 29

Figure 16 Read Input Registers – Query 30
Figure 17 Read Input Registers – Response 31

Figure 18 Force Single Coil – Query 32
Figure 19 Force Single Coil – Response 33

Figure 20 Preset Single Register – Query 34
Figure 21 Preset Single Register – Response 35

Figure 22 Read Exception Status – Query 36
Figure 23 Read Exception Status – Response 37

Figure 24 Fetch Communications Event Counter – Query 38
Figure 25 Fetch Communications Event Counter – Response 39

Figure 26 Fetch Communications Event Log – Query 40
Figure 27 Fetch Communications Event Log – Response 41

Figure 28 Force Multiple Coils – Query 45
Figure 29 Force Multiple Coils – Response 45

Figure 30 Preset Multiple Registers – Query 46
Figure 31 Preset Multiple Registers – Response 47

Figure 32 Report Slave ID – Query 48
Figure 33 Report Slave ID – Response 49

Figure 34 Read General Reference – Query 60
Figure 35 Read General Reference – Response 61

Figure 36 Write General Reference – Query 64
Figure 37 Write General Reference – Response 65

Figure 38 Mask Write 4X Register – Query 67
Figure 39 Mask Write 4X Register – Response 67

Figure 40 Read/Write 4X Registers – Query 68
Figure 41 Read/Write 4X Registers – Response 69

Figure 42 Read FIFO Queue – Query 70
Figure 43 Read FIFO Queue – Response 71

Figure 44 Diagnostics – Query 75
Figure 45 Diagnostics – Response 75

Figure 46 Master Query and Slave Exception Response 95

Figure 47 LRC Character Sequence 110
Figure 48 CRC Byte Sequence 113

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 formtroae6,

The c TwÏÈ(propriTbus protocol.)TjÏÈcol.vid

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.

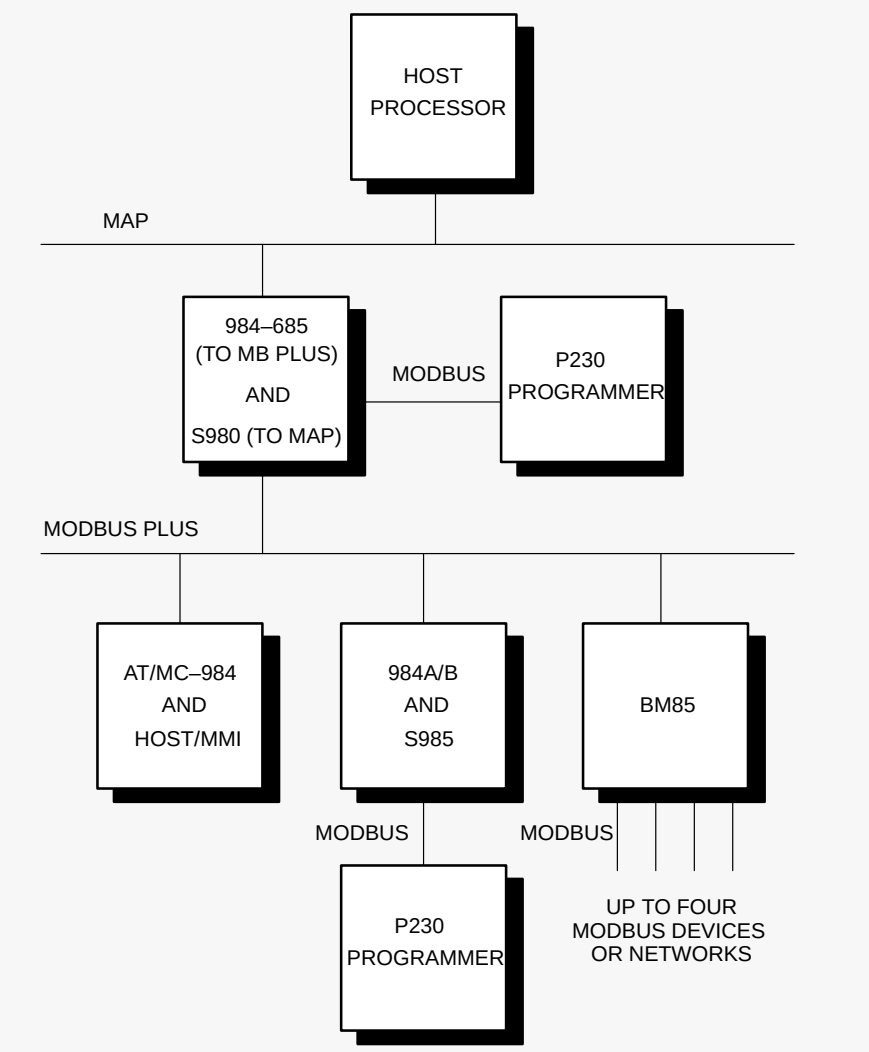


Figure 1 Overview of Modbus Protocol Application

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

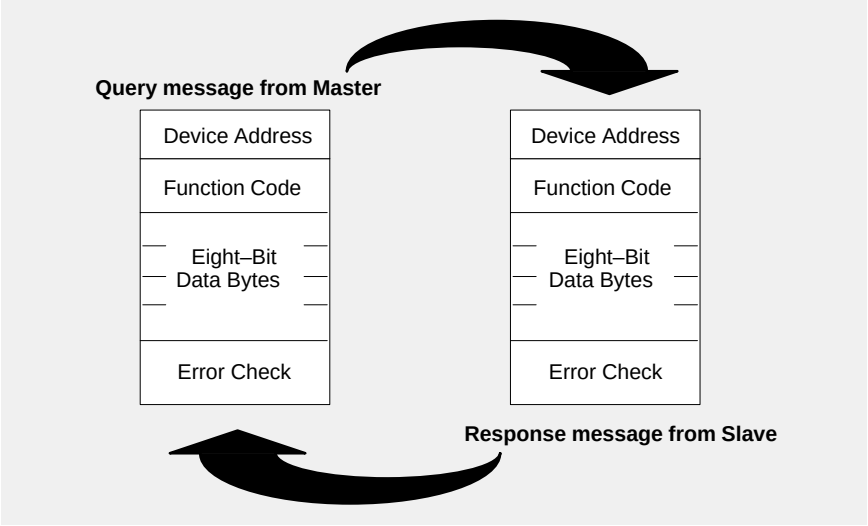


Figure 2 Master–Slave Query–Response Cycle

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:

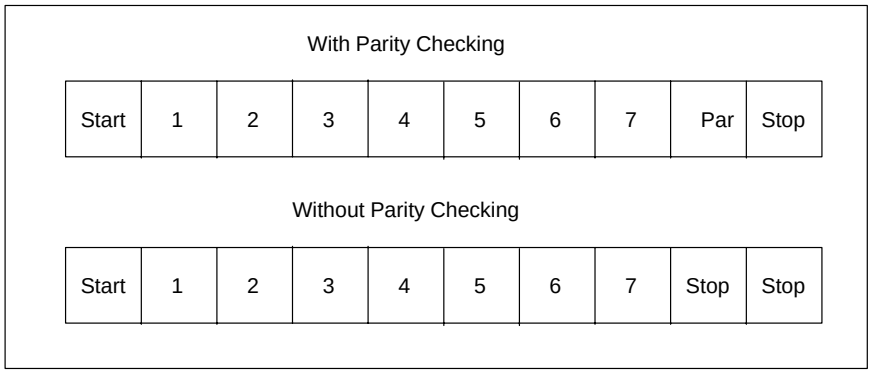


Figure 5 Bit Order (ASCII)

With RTU character framing, the bit sequence is:

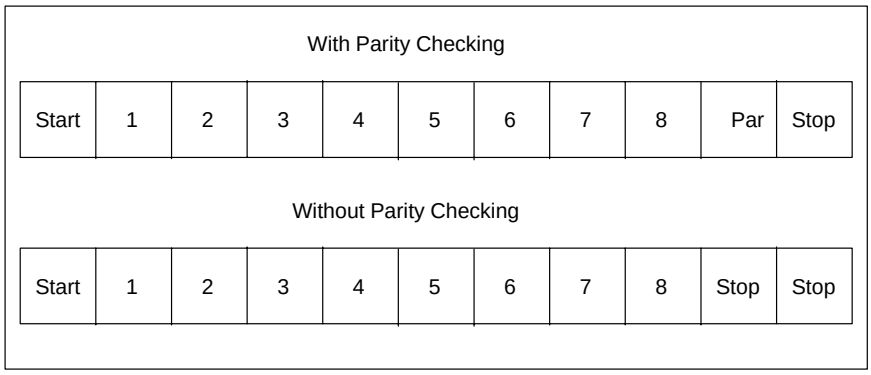


Figure 6 Bit Order (RTU)

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.

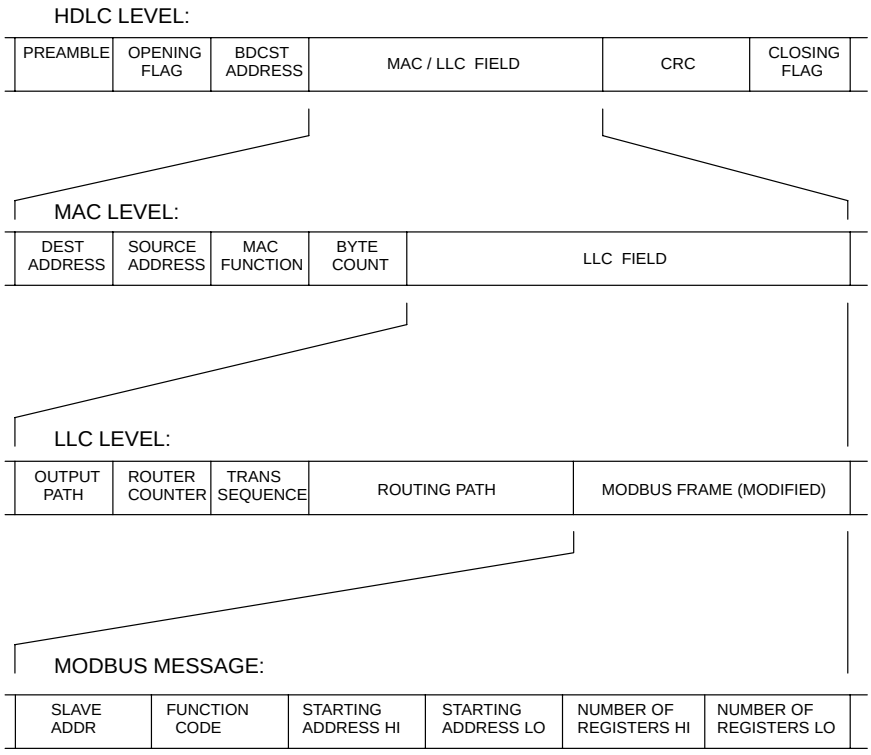


Figure 9 Field Contents on Modbus Plus

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

07 Read Exception Status

Description

Reads the contents of eight Exception Status coils within the slave controller. Certain coils have predefined assignments in the various controllers. Other coils can be programmed by the user to hold information about the contoller's status, for example, 'machine ON/OFF', 'heads retracted', 'safeties satisfied', 'error conditions exist', or other user–defined flags. Broadcast is not supported.

The function provides a simple method for accessing this information, because the Exception Coil references are known (no coil reference is needed in the function). The predefined Exception Coil assignments are:

Controller Model	Coil	Assignment
M84, 184/384, 584, 984	1 – 8	User defined
484	257	Battery Status
	258 – 264	User defined
884	761	Battery Status
	762	Memory Protect Status
	763	RIO Health Status
	764–768	User defined

Query

Here is an example of a request to read the exception status in slave device 17:

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	07
Error Check (LRC or CRC)	—

Figure 22 Read Exception Status – Query

Response

The normal response contains the status of the eight Exception Status coils. The coils are packed into one data byte, with one bit per coil. The status of the lowest coil reference is contained in the least significant bit of the byte.

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

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	07
Coil Data	6D
Error Check (LRC or CRC)	—

Figure 23 Read Exception Status – Response

In this example, the coil data is 6D hex (0110 1101 binary). Left to right, the coils are: OFF–ON–ON–OFF–ON–ON–OFF–ON. The status is shown from the highest to the lowest addressed coil.

If the controller is a 984, these bits are the status of coils 8 through 1.

If the controller is a 484, these bits are the status of coils 264 through 257. In this example, coil 257 is ON, indicating that the controller's batteries are OK.

11 (0B Hex) Fetch Comm Event Counter

Description

Returns a status word and an event count from the slave's communications event counter. By fetching the current count before and after a series of messages, a master can determine whether the messages were handled normally by the slave. Broadcast is not supported.

The controller's event counter is incremented once for each successful message completion. It is not incremented for exception responses, poll commands, or fetch event counter commands.

The event counter can be reset by means of the Diagnostics function (code 08), with a subfunction of Restart Communications Option (code 00 01) or Clear Counters and Diagnostic Register (code 00 0A).

Query

Here is an example of a request to fetch the communications event counter in slave device 17:

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	0B
Error Check (LRC or CRC)	—

Figure 24 Fetch Communications Event Counter – Query

Response

The normal response contains a two–byte status word, and a two–byte event count. The status word will be all ones (FF FF hex) if a previously–issued program command is still being processed by the slave (a busy condition exists). Otherwise, the status word will be all zeros.

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

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	0B
Status HI	FF
Status Lo	FF
Event Count Hi	01
Event Count Lo	08
Error Check (LRC or CRC)	—

Figure 25 Fetch Communications Event Counter – Response

In this example, the status word is FF FF hex, indicating that a program function is still in progress in the slave. The event count shows that 264 (01 08 hex) events have been counted by the controller.

12 (0C Hex) Fetch Comm Event Log

Description

Returns a status word, event count, message count, and a field of event bytes from the slave. Broadcast is not supported.

The status word and event count are identical to that returned by the Fetch Communications Event Counter function (11, 0B hex).

The message counter contains the quantity of messages processed by the slave since its last restart, clear counters operation, or power–up. This count is identical to that returned by the Diagnostic function (code 08), subfunction Return Bus Message Count (code 11, 0B hex).

The event bytes field contains 0-64 bytes, with each byte corresponding to the status of one Modbus send or receive operation for the slave. The events are entered by the slave into the field in chronological order. Byte 0 is the most recent event. Each new byte flushes the oldest byte from the field.

Query

Here is an example of a request to fetch the communications event log in slave device 17:

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	0C
Error Check (LRC or CRC)	—

Figure 26 Fetch Communications Event Log – Query

Response

The normal response contains a two–byte status word field, a two–byte event count field, a two–byte message count field, and a field containing 0-64 bytes of events. A byte count field defines the total length of the data in these four fields.

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

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	0C
Byte Count	08
Status HI	00
Status Lo	00
Event Count Hi	01
Event Count Lo	08
Message Count Hi	01
Message Count Lo	21
Event 0	20
Event 1	00
Error Check (LRC or CRC)	—

Figure 27 Fetch Communications Event Log – Response

In this example, the status word is 00 00 hex, indicating that the slave is not processing a program function. The event count shows that 264 (01 08 hex) events have been counted by the slave. The message count shows that 289 (01 21 hex) messages have been processed.

The most recent communications event is shown in the Event 0 byte. Its contents (20 hex) show that the slave has most recently entered the Listen Only Mode.

The previous event is shown in the Event 1 byte. Its contents (00 hex) show that the slave received a Communications Restart.

The layout of the response's event bytes is described on the next page.

12 (0C Hex) Fetch Comm Event Log (Continued)

What the Event Bytes Contain

An event byte returned by the Fetch Communications Event Log function can be any one of four types. The type is defined by bit 7 (the high–order bit) in each byte. It may be further defined by bit 6. This is explained below.

Slave Modbus Receive Event

This type of event byte is stored by the slave when a query message is received. It is stored before the slave processes the message. This event is defined by bit 7 set to a logic ‘1’. The other bits will be set to a logic ‘1’ if the corresponding condition is TRUE. The bit layout is:

Bit	Contents
0	Not Used
1	Communications Error
2	Not Used
3	Not Used
4	Character Overrun
5	Currently in Listen Only Mode
6	Broadcast Received
7	1

Slave Modbus Send Event

This type of event byte is stored by the slave when it finishes processing a query message. It is stored if the slave returned a normal or exception response, or no response. This event is defined by bit 7 set to a logic ‘0’, with bit 6 set to a ‘1’. The other bits will be set to a logic ‘1’ if the corresponding condition is TRUE. The bit layout is:

Bit	Contents
0	Read Exception Sent (Exception Codes 1-3)
1	Slave Abort Exception Sent (Exception Code 4)
2	Slave Busy Exception Sent (Exception Codes 5-6)
3	Slave Program NAK Exception Sent (Exception Code 7)
4	Write Timeout Error Occurred
5	Currently in Listen Only Mode

6	1
7	0

Slave Entered Listen Only Mode

This type of event byte is stored by the slave when it enters the Listen Only Mode. The event is defined by a contents of 04 hex. The bit layout is:

Bit	Contents
0	0
1	0
2	1
3	0
4	0
5	0
6	0
7	0

Slave Initiated Communication Restart

This type of event byte is stored by the slave when its communications port. is restarted. The slave can be restarted by the Diagnostics function (code 08), with subfunction Restart Communications Option (code 00 01).

That function also places the slave into a ‘Continue on Error’ or ‘Stop on Error’ mode. If the slave is placed into ‘Continue on Error’ mode, the event byte is added to the existing event log. If the slave is placed into ‘Stop on Error’ mode, the byte is added to the log and the rest of the log is cleared to zeros.

The event is defined by a contents of zero. The bit layout is:

Bit	Contents
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0

15 (0F Hex) Force Multiple Coils

Description

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

☞ **Note** The function will override the controller's memory protect state and a coil's disable state. The forced state will remain valid until the controller's logic next solves each coil. Coils will remain forced if they are 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 references to be forced. Coils are addressed starting at zero: coil 1 is addressed as 0.

The reguested ON/OFF states are specified by contents of the query data field. A logical '1' in a bit position of the field requests the corresponding coil to be ON. A logical '0' requests it to be OFF.

The following page shows an example of a request to force a series of ten coils starting at coil 20 (addressed as 19, or 13 hex) in slave device 17.

The query data contents are two bytes: CD 01 hex (1100 1101 0000 0001 binary). The binary bits correspond to the coils in the following way:

Bit:	1	1	0	0	1	1	0	1		0	0	0	0	0	0	0	1
Coil:	27	26	25	24	23	22	21	20		–	–	–	–	–	–	29	28

The first byte transmitted (CD hex) addresses coils 27-20, with the least significant bit addressing the lowest coil (20) in this set.

The next byte transmitted (01 hex) addresses coils 29-28, with the least significant bit addressing the lowest coil (28) in this set. Unused bits in the last data byte should be zero–filled.

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	0F
Coil Address Hi	00
Coil Address Lo	13
Quantity of Coils Hi	00
Quantity of Coils Lo	0A
Byte Count	02
Force Data Hi (Coils 27-20)	CD
Force Data Lo (Coils 29-28)	01
Error Check (LRC or CRC)	—

Figure 28 Force Multiple Coils – Query

Response

The normal response returns the slave address, function code, starting address, and quantity of coils forced.

Here is an example of a response to the query shown above.

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	0F
Coil Address Hi	00
Coil Address Lo	13
Quantity of Coils Hi	00
Quantity of Coils Lo	0A
Error Check (LRC or CRC)	—

Figure 29 Force Multiple Coils – Response

16 (10 Hex) Preset Multiple Registers

Description

Presets values into a sequence of holding registers (4X references). When broadcast, the function presets the same register references in all attached slaves.

☞ **Note** The function will override the controller's memory protect state. The preset values will remain valid in the registers until the controller's logic next solves the register contents. The register values will remain if they are 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 references to be preset. Registers are addressed starting at zero: register 1 is addressed as 0.

The requested preset values are specified in the query data field. 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. Data is packed as two bytes per register.

Here is an example of a request to preset two registers starting at 40002 to 00 0A and 01 02 hex, in slave device 17:

QUERY

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

Figure 30 Preset Multiple Registers – Query

Response

The normal response returns the slave address, function code, starting address, and quantity of registers preset.

Here is an example of a response to the query shown above.

RESPONSE

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

Figure 31 Preset Multiple Registers – Response

17 (11 Hex) Report Slave ID

Description

Returns a description of the type of controller present at the slave address, the current status of the slave Run indicator, and other information specific to the slave device. Broadcast is not supported.

Query

Here is an example of a request to report the ID and status of slave device 17:

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	11
Error Check (LRC or CRC)	—

Figure 32 Report Slave ID – Query

Response

The format of a normal response is shown below. The data contents are specific to each type of controller. They are listed on the following pages.

RESPONSE

Field Name	Contents
Slave Address	Echo of Slave Address
Function	11
Byte Count	Device Specific
Slave ID	Device Specific
Run Indicator Status	00 = OFF, FF = ON
Additional Data	Device Specific
. . .	
Error Check (LRC or CRC)	—

Figure 33 Report Slave ID – Response

A Summary of Slave IDs

These are the Slave ID codes returned by Modicon controllers in the first byte of the data field:

Slave ID	Controller
0	Micro 84
1	484
2	184/384
3	584
8	884
9	984

17 (11 Hex) Report Slave ID (Continued)

184/384

The 184 or 384 controller returns a byte count of either 4 or 74 (4A hexadecimal). If the controller's J347 Modbus Slave Interface is setup properly, and its internal PIB table is normal, the byte count will be 74. Otherwise the byte count will be 4.

The four bytes that are always returned are:

Byte	Contents
1	Slave ID (2 for 184/384). See bytes 3, 4 for further definition.
2	RUN indicator status (0 = OFF, FF = ON)
3, 4	Status word: Bit 0 = 0 Bit 1 = Memory Protect status (0 = OFF, 1 = ON) Bit 2, 3 = Controller type: Bit 2 = 0 and Bit 3 = 0 indicates 184 Bit 2 = 1 and Bit 3 = 0 indicates 384 Bits 4 - 15 = Unused

The additonal 70 bytes returned for a correct J347 setup and normal PIB are:

Byte	Contents
5, 6	PIB table starting address
7, 8	Controller serial number
9, 10	Executive ID
	Bytes 11 - 74 contain the PIB table. This data is valid only if the controller is running (as shown in Byte 2). The table is as follows:
11, 12	Maximum quantity of output coils
13, 14	Output coil enable table
15, 16	Address of input coil/run table
17, 18	Quantity of input coils
19, 20	Input coil enable table
21, 22	First latch number (must be multiple of 16)
23, 24	Last latch number (must be multiple of 16)

25, 26	Address of input registers
27, 28	Quantity of input registers
29, 30	Quantity of output and holding registers
31, 32	Address of user logic
33, 34	Address of output coil RAM table
35, 36	Function inhibit mask
37, 38	Address of extended function routine
39, 40	Address of data transfer routine
41, 42	Address of traffic cop
43, 44	Unused
45, 46	Function inhibit mask
47, 48	Address of ‘A’ Mode history table
49, 50	Request table for DX printer
51, 52	Quantity of sequence groups
53, 54	Address of sequence image table
55, 56	Address of sequence RAM
57, 58	Quantity of 50XX registers
59, 60	Address of 50XX table
61, 62	Address of output coil RAM image
63, 64	Address of input RAM image
65, 66	Delayed output start group
67, 68	Delayed output end group
69, 70	Watchdog line
71, 72	RAM Address of latches
73, 74	Quantity of delayed output groups

17 (11 Hex) Report Slave ID (Continued)

584

The 584 controller returns a byte count of 9, as follows:

Byte	Contents
1	Slave ID (3 for 584)
2	RUN indicator status (0 = OFF, FF = ON)
3	Quantity of 4K sections of page 0 memory
4	Quantity of 1K sections of state RAM
5	Quantity of segments of user logic
6, 7	Machine state word (configuration table word 101, 65 hex). The word is organized as follows: *Byte 6:* Bit 15 (MSB of byte 6) = Port 1 setup Bit 14 = Port 2 setup Bit 13 = Port 1 address set Bit 12 = Port 2 address set Bit 11 = Unassigned Bit 10 = Constant Sweep status (0 = Constand Sweep OFF, 1 = ON) Bit 9 = Single Sweep status (0 = Single Sweep OFF, 1 = ON) Bit 8 = 16/24-bit nodes (0 = 24-bit nodes, 1 = 16-bit nodes) *Byte 7:* Bit 7 (MSB of byte 7) = Power ON (1 = ON, should never = 'OFF') Bit 6 = RUN indicator status (0 = ON, 1 = OFF) Bit 5 = Memory Protect status (0 = ON, 1 = OFF) Bit 4 = Battery OK (0 = OK, 1 = Not OK) Bits 3 - 0 = Unassigned

8, 9 Machine stop code (configuration table word 105, 69 hex). The word is organized as follows:

Byte 8:
Bit 15 (MSB of byte 8) = Peripheral port stop (controlled stop)
Bit 14 = Unassigned
Bit 13 = Dim awareness
Bit 12 = Illegal peripheral intervention
Bit 11 = Multirate solve table invalid
Bit 10 = Start of Node did not start segment
Bit 9 = State RAM test failed
Bit 8 = No End of Logic detected, or bad quantity of segments

Byte 9:
Bit 7 (MSB of byte 9) = Watchdog timer expired
Bit 6 = Real time clock error
Bit 5 = CPU diagnostic failed
Bit 4 = Invalid traffic cop type
Bit 3 = Invalid node type
Bit 2 = Logic checksum error
Bit 1 = Backup checksum error
Bit 0 = Illegal configuration

17 (11 Hex) Report Slave ID (Continued)

984

The 984 controller returns a byte count of 9, as follows:

Byte	Contents
1	Slave ID (9 for 984)
2	RUN indicator status (0 = OFF, FF = ON)
3	Quantity of 4K sections of page 0 memory
4	Quantity of 1K sections of state RAM
5	Quantity of segments of user logic
6, 7	Machine state word (configuration table word 101, 65 hex). The word is organized as follows: *Byte 6:* Bit 15 (MSB of byte 6) = Unassigned Bits 14 - 11 = Unassigned Bit 10 = Constant Sweep status (0 = Constand Sweep OFF, 1 = ON) Bit 9 = Single Sweep status (0 = Single Sweep OFF, 1 = ON) Bit 8 = 16/24-bit nodes (0 = 24-bit nodes, 1 = 16-bit nodes) *Byte 7:* Bit 7 (MSB of byte 7) = Power ON (1 = ON, should never = 'OFF') Bit 6 = RUN indicator status (0 = ON, 1 = OFF) Bit 5 = Memory Protect status (0 = ON, 1 = OFF) Bit 4 = Battery OK (0 = OK, 1 = Not OK) Bits 3 - 1 = Unassigned Bit 0 = Memory downsize flag (0 = NO, 1 = Downsize

Memory Downsize: Bit 0 of the Machine State word defines the use of the memory downsize values in words 99, 100, and 175 (63, 64, and AF hexadecimal) of the configuration table. If bit 0 = logic 1, downsizing is calculated as follows:

Page 0 size (16-bit words) = (Word 99 * 4096) – (Word 175 low byte * 16)

State table size (16 bit words) = (Word 100 * 1024) – (Word 175 high byte * 16)

8, 9 Machine stop code (configuration table word 105, 69 hex). The word is organized as follows:

Byte 8:
Bit 15 (MSB of byte 8) = Peripheral port stop (controlled stop)
Bit 14 (984A, B, X) = Extended memory parity failure
Bit 14 (Other 984) = Bad IO traffic cop
Bit 13 = Dim awareness
Bit 12 = Illegal peripheral intervention
Bit 11 = Bad segment scheduler table
Bit 10 = Start of Node did not start segment
Bit 9 = State RAM test failed
Bit 8 = No End of Logic detected, or bad quantity of segments

Byte 9:
Bit 7 (MSB of byte 9) = Watchdog timer expired
Bit 6 = Real time clock error
Bit 5 (984A, B, X) = CPU diagnostic failed
Bit 5 (Other 984) = Bad coil used table
Bit 4 = S908 remote IO head failure
Bit 3 = Invalid node type
Bit 2 = Logic checksum error
Bit 1 = Coil disabled while in RUN mode
Bit 0 = Illegal configuration

17 (11 Hex) Report Slave ID (Continued)

Micro 84

The Micro 84 controller returns a byte count of 8, as follows:

Byte	Contents
1	Slave ID (0 for Micro 84)
2	RUN indicator status (0 = OFF, FF = ON)
3	Current port number
4	Memory size (1 = 1K, 2 = 2K)
5	Unused (all zeros)

484

The 484 controller returns a byte count of 5, as follows:

Byte	Contents
1	Slave ID (1 for 484)
2	RUN indicator status (0 = OFF, FF = ON)
3	System state
4	First configuration byte
5	Second configuration byte

884

The 884 controller returns a byte count of 8, as follows:

Byte	Contents
1	Slave ID (8 for 884)
2	RUN indicator status (0 = OFF, FF = ON)
3	Current port number
4	Size of user logic plus state RAM, in kilobytes (1 word = 2 bytes)
5	Reserved
6	Hook bits: Bits 0 - 2 = Reserved Bit 3 = Mapper bypass: 1 = Do not execute standard mapper Bit 4 = End of Scan tests: 1 = Test end of scan hooks Bit 5 = Reserved Bit 6 = Logic Solver bypass: 1 = Do not execute standard logic solver Bit 7 = Reserved
7, 8	Reserved

20 (14Hex) Read General Reference

Description

Returns the contents of registers in Extended Memory file (6XXXXX) references. Broadcast is not supported.

The function can read multiple groups of references. The groups can be separate (non–contiguous), but the references within each group must be sequential.

Query

The query contains the standard Modbus slave address, function code, byte count, and error check fields. The rest of the query specifies the group or groups of references to be read. Each group is defined in a separate ‘sub-request’ field which contains 7 bytes:

- The reference type: 1 byte (must be specified as 6)
- The Extended Memory file number: 2 bytes (1 to 10, hex 0001 to 000A)
- The starting register address within the file: 2 bytes
- The quantity of registers to be read: 2 bytes.

The quantity of registers to be read, combined with all other fields in the expected response, must not exceed the allowable length of Modbus messages: 256 bytes.

The available quantity of Extended Memory files depends upon the installed size of Extended Memory in the slave controller. Each file except the last one contains 10,000 registers, addressed as 0000-270F hexadecimal (0000-9999 decimal).



Note The addressing of Extended Register (6XXXXX) references differs from that of Holding Register (4XXXX) references.

The lowest Extended Register is addressed as register ‘zero’ (600000).

The lowest Holding Register is addressed as register ‘one’ (40001).

For controllers other than the 984–785 with Extended Registers, the last (highest) register in the last file is:

Ext Mem Size	Last File	Last Register (Decimal)
16K	2	6383
32K	4	2767
64K	7	5535
96K	10	8303

For the 984–785 with Extended Registers, the last (highest) register in the last file is shown in the two tables below.

984–785 with AS–M785–032 Memory Cartridge:

User Logic	State RAM	Ext Mem Size	Last File	Last Register (Decimal)
32K	32K	0	0	0
16K	64K	72K	8	3727

984–785 with AS–M785–048 Memory Cartridge:

User Logic	State RAM	Ext Mem Size	Last File	Last Register (Decimal)
48K	32K	24K	3	4575
32K	64K	96K	10	8303

Examples of a query and response are provided starting on the next page.

20 (14 Hex) Read General Reference (Continued)

An example of a request to read two groups of references from slave device 17 is shown below.

Group 1 consists of two registers from file 4, starting at register 1 (address 0001).
Group 2 consists of two registers from file 3, starting at register 9 (address 0009).

QUERY Field Name	Example (Hex)
Slave Address	11
Function	14
Byte Count	0E
Sub–Req 1, Reference Type	06
Sub–Req 1, File Number Hi	00
Sub–Req 1, File Number Lo	04
Sub–Req 1, Starting Addr Hi	00
Sub–Req 1, Starting Addr Lo	01
Sub–Req 1, Register Count Hi	00
Sub–Req 1, Register Count Lo	02
Sub–Req 2, Reference Type	06
Sub–Req 2, File Number Hi	00
Sub–Req 2, File Number Lo	03
Sub–Req 2, Starting Addr Hi	00
Sub–Req 2, Starting Addr Lo	09
Sub–Req 2, Register Count Hi	00
Sub–Req 2, Register Count Lo	02
Error Check (LRC or CRC)	—

Figure 34 Read General Reference – Query

Response

The normal response is a series of ‘sub-responses’, one for each ‘sub-request’. The byte count field is the total combined count of bytes in all ‘sub-responses’. In addition, each ‘sub-response’ contains a field that shows its own byte count.

RESPONSE Field Name	Example (Hex)
Slave Address	11
Function	14
Byte Count	0C
Sub–Res 1, Byte Count	05
Sub–Res 1, Reference Type	06
Sub–Res 1, Register Data Hi	0D
Sub–Res 1, Register Data Lo	FE
Sub–Res 1, Register Data Hi	00
Sub–Res 1, Register Data Lo	20
Sub–Res 2, Byte Count	05
Sub–Res 2, Reference Type	06
Sub–Res 2, Register Data Hi	33
Sub–Res 2, Register Data Lo	CD
Sub–Res 2, Register Data Hi	00
Sub–Res 2, Register Data Lo	40
Error Check (LRC or CRC)	—

Figure 35 Read General Reference – Response

21 (15Hex) Write General Reference

Description

Writes the contents of registers in Extended Memory file (6XXXXX) references. Broadcast is not supported.

The function can write multiple groups of references. The groups can be separate (non–contiguous), but the references within each group must be sequential.

Query

The query contains the standard Modbus slave address, function code, byte count, and error check fields. The rest of the query specifies the group or groups of references to be written, and the data to be written into them. Each group is defined in a separate ‘sub-request’ field which contains 7 bytes plus the data:

- The reference type: 1 byte (must be specified as 6)
- The Extended Memory file number: 2 bytes (1 to 10, hex 0001 to 000A)
- The starting register address within the file: 2 bytes
- The quantity of registers to be written: 2 bytes
- The data to be written: 2 bytes per register.

The quantity of registers to be written, combined with all other fields in the query, must not exceed the allowable length of Modbus messages: 256 bytes.

The available quantity of Extended Memory files depends upon the installed size of Extended Memory in the slave controller. Each file except the last one contains 10,000 registers, addressed as 0000-270F hexadecimal (0000-9999 decimal).



Note The addressing of Extended Register (6XXXXX) references differs from that of Holding Register (4XXXX) references.

The lowest Extended Register is addressed as register ‘zero’ (600000).

The lowest Holding Register is addressed as register ‘one’ (40001).

For controllers other than the 984–785 with Extended Registers, the last (highest) register in the last file is:

Ext Mem Size	Last File	Last Register (Decimal)
16K	2	6383
32K	4	2767
64K	7	5535
96K	10	8303

For the 984–785 with Extended Registers, the last (highest) register in the last file is shown in the two tables below.

984–785 with AS–M785–032 Memory Cartridge:

User Logic	State RAM	Ext Mem Size	Last File	Last Register (Decimal)
32K	32K	0	0	0
16K	64K	72K	8	3727

984–785 with AS–M785–048 Memory Cartridge:

User Logic	State RAM	Ext Mem Size	Last File	Last Register (Decimal)
48K	32K	24K	3	4575
32K	64K	96K	10	8303

Examples of a query and response are provided starting on the next page.

21 (15 Hex) Write General Reference (Continued)

An example of a request to write one group of references into slave device 17 is shown below.

The group consists of three registers in file 4, starting at register 7 (address 0007).

QUERY Field Name	Example (Hex)
Slave Address	11
Function	15
Byte Count	0D
Sub–Req 1, Reference Type	06
Sub–Req 1, File Number Hi	00
Sub–Req 1, File Number Lo	04
Sub–Req 1, Starting Addr Hi	00
Sub–Req 1, Starting Addr Lo	07
Sub–Req 1, Register Count Hi	00
Sub–Req 1, Register Count Lo	03
Sub–Req 1, Register Data Hi	06
Sub–Req 1, Register Data Lo	AF
Sub–Req 1, Register Data Hi	04
Sub–Req 1, Register Data Lo	BE
Sub–Req 1, Register Data Hi	10
Sub–Req 1, Register Data Lo	0D
Error Check (LRC or CRC)	—

Figure 36 Write General Reference – Query

Response

The normal response is an echo of the query.

RESPONSE Field Name	Example (Hex)
Slave Address	11
Function	15
Byte Count	0D
Sub–Req 1, Reference Type	06
Sub–Req 1, File Number Hi	00
Sub–Req 1, File Number Lo	04
Sub–Req 1, Starting Addr Hi	00
Sub–Req 1, Starting Addr Lo	07
Sub–Req 1, Register Count Hi	00
Sub–Req 1, Register Count Lo	03
Sub–Req 1, Register Data Hi	06
Sub–Req 1, Register Data Lo	AF
Sub–Req 1, Register Data Hi	04
Sub–Req 1, Register Data Lo	BE
Sub–Req 1, Register Data Hi	10
Sub–Req 1, Register Data Lo	0D
Error Check (LRC or CRC)	—

Figure 37 Write General Reference – Response

22 (16Hex) Mask Write 4X Register

Description

Modifies the contents of a specified 4XXXX register using a combination of an AND mask, an OR mask, and the register's current contents. The function can be used to set or clear individual bits in the register. Broadcast is not supported.

This function is supported in the 984–785 controller only.

Query

The query specifies the 4XXXX reference to be written, the data to be used as the AND mask, and the data to be used as the OR mask.

The function's algorithm is:

Result = (Current Contents AND And_Mask) OR (Or_Mask AND $\overline{\text{And_Mask}}$)

For example:		**Hex**	**Binary**
Current Contents	=	12	0001 0010
And_Mask	=	F2	1111 0010
Or_Mask	=	25	0010 0101
$\overline{\text{And_Mask}}$	=	0D	0000 1101
Result	=	17	0001 0111

Note that if the Or_Mask value is zero, the result is simply the logical ANDing of the current contents and And_Mask. If the And_Mask value is zero, the result is equal to the Or_Mask value.

Note that the contents of the register can be read with the Read Holding Registers function (function code 03). They could, however, be changed subsequently as the controller scans its user logic program.

An example of a Mask Write to register 5 in slave device 17, using the above mask values, is shown on the next page.

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	16
Reference Address Hi	00
Reference Address Lo	04
And_Mask Hi	00
And_Mask Lo	F2
Or_Mask Hi	00
Or–Mask Lo	25
Error Check (LRC or CRC)	—

Figure 38 Mask Write 4X Register – Query

Response

The normal response is an echo of the query. The response is returned after the register has been written.

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	16
Reference Address Hi	00
Reference Address Lo	04
And_Mask Hi	00
And_Mask Lo	F2
Or_Mask Hi	00
Or–Mask Lo	25
Error Check (LRC or CRC)	—

Figure 39 Mask Write 4X Register – Response

23 (17Hex) Read/Write 4X Registers

Description

Performs a combination of one read and one write operation in a single Modbus transaction. The function can write new contents to a group of 4XXXX registers, and then return the contents of another group of 4XXXX registers. Broadcast is not supported. This function is supported in the 984–785 controller only.

Query

The query specifies the starting address and quantity of registers of the group to be read. It also specifies the starting address, quantity of registers, and data for the group to be written. The byte count field specifies the quantity of bytes to follow in the write data field.

Here is an example of a query to read six registers starting at register 5, and to write three registers starting at register 16, in slave device 17:

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	17
Read Reference Address Hi	00
Read Reference Address Lo	04
Quantity to Read Hi	00
Quantity to Read Lo	06
Write Reference Address Hi	00
Write Reference Address Lo	0F
Quantity to Write Hi	00
Quantity to Write Lo	03
Byte Count	06
Write Data 1 Hi	00
Write Data 1 Lo	FF
Write Data 2 Hi	00
Write Data 2 Lo	FF
Write Data 3 Hi	00
Write Data 3 Lo	FF
Error Check (LRC or CRC)	—

Figure 40 Read/Write 4X Registers – Query

Response

The normal response contains the data from the group of registers that were read. The byte count field specifies the quantity of bytes to follow in the read data field.

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

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	17
Byte Count	0C
Read Data 1 Hi	00
Read Data 1 Lo	FE
Read Data 2 Hi	0A
Read Data 2 Lo	CD
Read Data 3 Hi	00
Read Data 3 Lo	01
Read Data 4 Hi	00
Read Data 4 Lo	03
Read Data 5 Hi	00
Read Data 5 Lo	0D
Read Data 6 Hi	00
Read Data 6 Lo	FF
Error Check (LRC or CRC)	—

Figure 41 Read/Write 4X Registers – Response

24 (18Hex) Read FIFO Queue

Description

Reads the contents of a First–In–First–Out (FIFO) queue of 4XXXX registers. The function returns a count of the registers in the queue, followed by the queued data. Up to 32 registers can be read: the count, plus up to 31 queued data registers. The queue count register is returned first, followed by the queued data registers.

The function reads the queue contents, but does not clear them. Broadcast is not supported.

This function is supported in the 984–785 controller only.

Query

The query specifies the starting 4XXXX reference to be read from the FIFO queue. This is the address of the pointer register used with the controller's FIN and FOUT function blocks. It contains the count of registers currently contained in the queue. The FIFO data registers follow this address sequentially.

An example of a Read FIFO Queue query to slave device 17 is shown below. The query is to read the queue starting at the pointer register 41247 (04DE hex).

QUERY

Field Name	Example (Hex)
Slave Address	11
Function	18
FIFO Pointer Address Hi	04
FIFO Pointer Address Lo	DE
Error Check (LRC or CRC)	—

Figure 42 Read FIFO Queue – Query

Response

In a normal response, the byte count shows the quantity of bytes to follow, including the queue count bytes and data register bytes (but not including the error check field).

The queue count is the quantity of data registers in the queue (not including the count register).

If the queue count exceeds 31, an exception response is returned with an error code of 03 (Illegal Data Value).

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

RESPONSE

Field Name	Example (Hex)
Slave Address	11
Function	18
Byte Count Hi	00
Byte Count Lo	08
FIFO Count Hi	00
FIFO Count Lo	03
FIFO Data Reg 1 Hi	01
FIFO Data Reg 1 Lo	B8
FIFO Data Reg 2 Hi	12
FIFO Data Reg 2 Lo	84
FIFO Data Reg 3 Hi	13
FIFO Data Reg 3 Lo	22
Error Check (LRC or CRC)	—

Figure 43 Read FIFO Queue – Response

In this example, the FIFO pointer register (41247 in the query) is returned with a queue count of 3. The three data registers follow the queue count. These are: 41248 (contents 440 decimal -- 01B8 hex); 41249 (contents 4740 -- 1284 hex); and 41250 (contents 4898 -- 1322 hex).

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 fÏÈ[e slave oadcast

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:

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

(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):

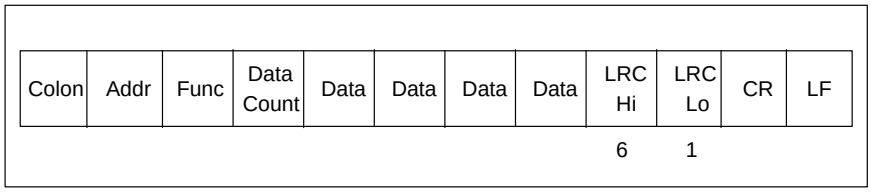


Figure 47 LRC Character Sequence

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):

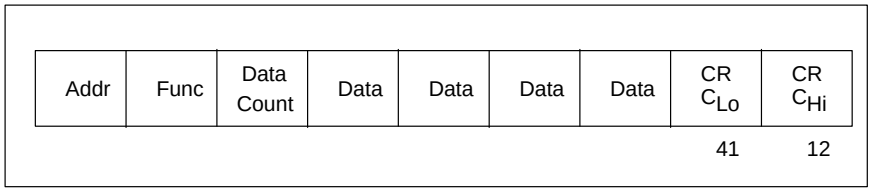


Figure 48 CRC Byte Sequence

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
} ;
```