# MODBUS PROTOCOL

PDF format version of the MODBUS Protocol

**The original was found at:**

**http://www.http://www.modicon.com/techpubs/toc7.html**

**(In case of any discrepancies, that version should be considered accurate.)**

**Hope you find this useful!**
**Spehro Pefhany, January 2000**

Trexon INC.
3-1750 The Queensway Suite 1298 Toronto ON Canada M9C 4H5
(905) 271-4477 fax: (905) 271-9838 e-mail: info@trexon.com

# Modbus Protocol

# Chapter 1
# Modbus Protocol



**1.1** Introducing Modbus Protocol

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

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

The Modbus protocol provides the internal standard that the Modicon controllers use for parsing messages. During communications on a Modbus network, the protocol determines how each controller will know its device address, recognize a message addressed to it, determine the kind of action to be taken, and extract any data or other information contained in the message. If a reply is required, the controller will construct the reply message and send it using Modbus protocol.

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

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

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

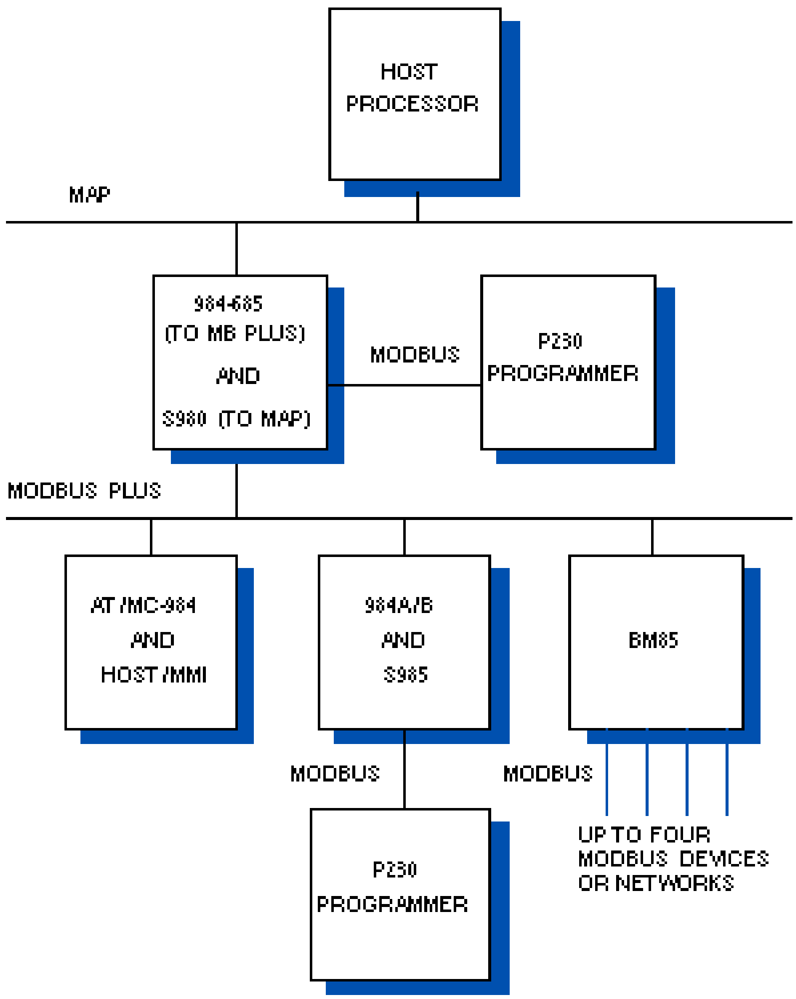



**Figure 1 Overview of Modbus Protocol Application**

**1.1.1** Transactions on Modbus Networks

Standard Modbus ports on Modicon controllers use an RS-232C compatible serial interface that defines connector pinouts, cabling, signal levels, transmission baud rates, and parity checking. Controllers can be networked directly or via modems.

Controllers communicate using a master-slave technique, in which only one device (the master) can initiate transactions (queries). The other devices (the slaves) respond by supplying the requested data to the master, or by taking the action requested in the query. Typical master devices include host processors and programming panels. Typical slaves include programmable controllers.

The master can address individual slaves, or can initiate a broadcast message to all slaves. Slaves return a message (response) to queries that are addressed to them individually. Responses are not returned to broadcast queries from the master.

The Modbus protocol establishes the format for the master's query by placing into it the device (or broadcast) address, a function code defining the requested action, any data to be sent, and an error-checking field. The slave's response message is also constructed using Modbus protocol. It contains fields confirming the action taken, any data to be returned, and an error-checking field. If an error occurred in receipt of the message, or if the slave is unable to perform the requested action, the slave will construct an error message and send it as its response.

**1.1.2** Transactions on Other Kinds of Networks

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

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

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

**1.1.3** The Query-Response Cycle

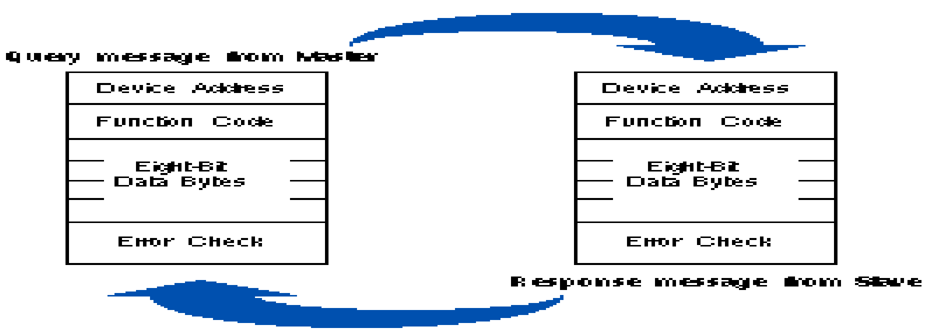



**Figure 2 Master-Slave Query-Response Cycle**

**The Query**

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

**The Response**

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

**1.2** Two Serial Transmission Modes

Controllers can be setup to communicate on standard Modbus networks using either of two transmission modes: ASCII or RTU. Users select the desired mode, along with the serial port communication parameters (baud rate, parity mode, etc), during configuration of each controller. The mode and serial parameters must be the same for all devices on a Modbus network.

The selection of ASCII or RTU mode pertains only to standard Modbus networks. It defines the bit contents of message fields transmitted serially on those networks. It determines how information will be packed into the message fields and decoded.

On other networks like MAP and Modbus Plus, Modbus messages are placed into frames that are not related to serial tranasmission. For example, a request to read holding registers can be handled between two controllers on Modbus Plus without regard to the current setup of either controller's serial Modbus port.

**1.2.1** ASCII Mode

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

**Coding System**

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

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

**Bits per Byte**

V 1 start bit

V 7 data bits, least significant bit sent first

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

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

**Error Check Field**

V Longitudinal Redundancy Check (LRC)

**1.2.2** RTU Mode

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

**Coding System**

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

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

**Bits per Byte**

V 1 start bit

V 8 data bits, least significant bit sent first

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

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

**Error Check Field**

V Cyclical Redundancy Check (CRC)

**1.3** Modbus Message Framing

In either of the two serial transmission modes (ASCII or RTU), a Modbus message is placed by the transmitting device into a frame that has a known beginning and ending point. This allows receiving devices to begin at the start of the message, read the address portion and determine which device is addressed (or all devices, if the message is broadcast), and to know when the message is completed. Partial messages can be detected and errors can be set as a result.

On networks like MAP or Modbus Plus, the network protocol handles the framing of messages with beginning and end delimiters that are specific to the network. Those protocols also handle delivery to the destination device, making the Modbus address field imbedded in the message unnecessary for the actual transmission. (The Modbus address is converted to a network node address and routing path by the originating controller or its network adapter.)

**1.3.1** ASCII Framing

In ASCII mode, messages start with a colon ( : ) character (ASCII 3A hex), and end with a carriage return-line feed (CRLF) pair (ASCII 0D and 0A hex).

The allowable characters transmitted for all other fields are hexadecimal 0 ... 9, A ... F. Networked devices monitor the network bus continuously for the colon character. Wh

# Chapter 2
# Data and Control Functions

V Read FIFO Queue

## 2.1 Modbus Function Formats

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

### 2.1.1 Data Addresses in Modbus Messages

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

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

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

V Holding register 40001 is addressed as register 0000 in the data address field of the message. The function code field already specifies a holding register operation. Therefore the 4*x* reference is implicit.

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

### 2.1.2 Field Contents in Modbus Messages

The following tables show examples of a Modbus query and normal response. Both examples show the field contents in hexadecimal, and also show how a message could be framed in ASCII or in RTU mode.

**Query**

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

**Response**

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

The master query is a Read Holding Registers request to slave device address 06. The message requests data from three holding registers, 40108 ... 40110.

**Note:** The message specifies the starting register address as 0107 (006B hex).

The slave response echoes the function code, indicating this is a normal response. The Byte Count field specifies how many eight-bit data items are being returned. It shows the count of eight-bit bytes to follow in the data, for either ASCII or RTU. With ASCII, this value is half the actual count of ASCII characters in the data. In ASCII, each four-bit hexadecimal value requires one ASCII character, therefore two ASCII characters must follow in the message to contain each eight-bit data item.

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

### How to Use the Byte Count Field

When you construct responses in buffers, use a Byte Count value that equals the count of eight-bit bytes in your message data. The value is exclusive of all other field contents, including the Byte Count field.

### 2.1.3 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 eight-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 7 shows how a Read Holding Registers query would be imbedded into a frame for Modbus Plus transmission.

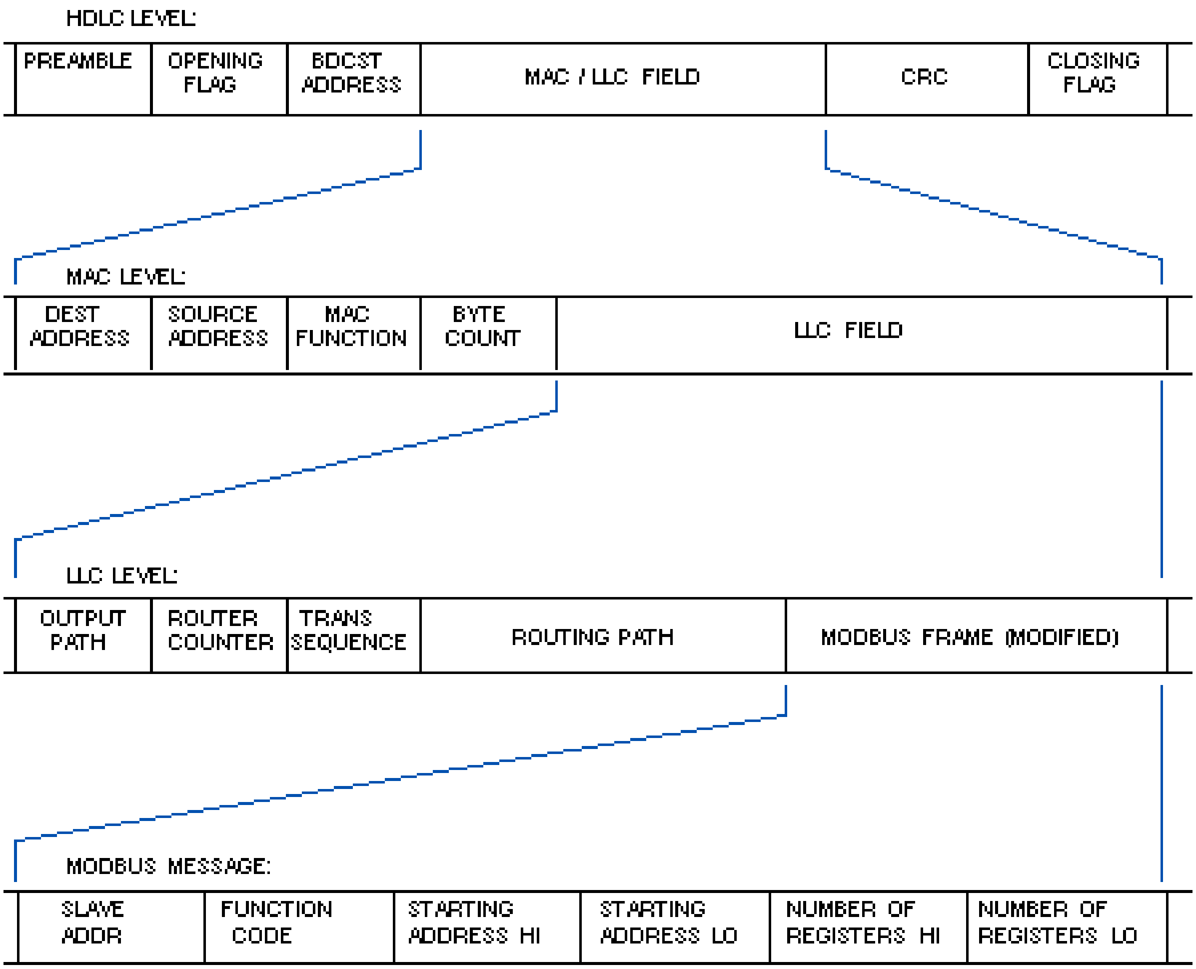



**Figure 7 Field Contents on Modbus Plus**

## 2.2 Function Codes

The listing below shows the function codes supported by Modicon controllers. Codes are listed in decimal; Y indicates that the function is supported, and N indicates that it is not supported.

| Code | Name | 384 | 484 | 584 | 884 | M84 | 984 |
|---|---|---|---|---|---|---|---|
| 01 | Read Coil Status | Y | Y | Y | Y | Y | Y |
| 02 | Read Input Status | Y | Y | Y | Y | Y | Y |
| 03 | Read Holding Registers | Y | Y | Y | Y | Y | Y |
| 04 | Read Input Registers | Y | Y | Y | Y | Y | Y |
| 05 | Force Single Coil | Y | Y | Y | Y | Y | Y |
| 06 | Preset Single Register | Y | Y | Y | Y | Y | Y |
| 07 | Read Exception Status | Y | Y | Y | Y | Y | Y |
| 08 | Diagnostics | see page NO TAG) | | | | | |
| 09 | Program 484 | N | Y | N | N | N | N |
| 10 | Poll 484 | N | Y | N | N | N | N |
| 11 | Fetch Comm Event Counter | Y | N | Y | N | N | Y |
| 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 | N | Y | Y | N |
| 21 | Write General Reference | N | N | Y | N | N | Y |
| 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) |

(1) = Function is supported in 984-785 only.

### 2.2.1 01 Read Coil Status

Reads the ON / OFF status of discrete outputs (0*x* references, coils) in the slave. Broadcast is not supported. The maximum parameters supported by various controller models are listed on page .

#### Query

The query message specifies the starting 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 query to read coils 20 ... 56 from slave device 17:

| Field Name | Example (Hex) |
|---|---|
| Slave Address | 11 |
| Function | 01 |
| Starting Address Hi | 00 |
| Starting Address Lo | 13 |
| Number of Points Hi | 00 |
| Number of Points Lo | 25 |
| Error Check (LRC or CRC) | -- |

### 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:

| 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) | -- |

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 ... 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 ... 20, from left to right. The next byte has coils 35 ... 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 ... 52 is: ON-ON-OFF-ON-ON.

**Note:** The three remaining bits (toward the high-order end) are zero-filled.

### 2.2.2 02 Read Input Status

Reads the ON / OFF status of discrete inputs (1*x* references) in the slave. Broadcast is not supported. The maximum parameters supported by various controller models are listed on page .

#### Query

The query message specifies the starting 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:

| Field Name | Example (Hex) |
|---|---|
| Slave Address | 11 |
| Function | 02 |
| Starting Address Hi | 00 |
| Starting Address Lo | C4 |
| Number of Points Hi | 00 |
| Number of Points Lo | 16 |
| Error Check (LRC or CRC) | -- |

#### 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:

| 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) | -- |

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 ... 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 ... 10213 is: ON-ON-OFF-ON-OFF-ON.

**Note:** The two remaining bits (toward the high order end) are zero-filled.

### 2.2.3 03 Read Holding Registers

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

#### Query

The query message specifies the starting register and quantity of registers to be read. Registers are addressed starting at zero- registers 1 ... 16 are addressed as 0 ... 15.

Here is an example of a request to read registers 40108 ... 40110 from slave device 17:

| Field Name | Example (Hex) |
|---|---|
| Slave Address | 11 |
| Function | 03 |
| Starting Address Hi | 00 |
| Starting Address Lo | 6B |
| Number of Points Hi | 00 |
| Number of Points Lo | 03 |
| Error Check (LRC or CRC) | -- |

#### Response

The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register, the first byte contains the high order bits and the second contains the low order bits.

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

Here is an example of a response to the query:

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

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

### 2.2.4 04 Read Input Registers

Reads the binary contents of input registers (3X references) in the slave. Broadcast is not supported. The maximum parameters supported by various controller models are listed on page .

### Query

The query message specifies the starting register and quantity of registers to be read. Registers are addressed starting at zero- registers 1 ... 16 are addressed as 0 ... 15.

Here is an example of a request to read register 30009 from slave device 17:

| Field Name | Example (Hex) |
| --- | --- |
| Slave Address | 11 |
| Function | 04 |
| Starting Address Hi | 00 |
| Starting Address Lo | 08 |
| Number of Points Hi | 00 |
| Number of Points Lo | 01 |
| Error Check (LRC or CRC) | -- |

### Response

The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register, the first byte contains the high-order bits and the second contains the low-order bits.

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:

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

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

### 2.2.5 05 Force Single Coil

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

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

### Query

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

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

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

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

### Response

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

Here is an example of a response to the query:

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

## 2.2.6 06 Preset Single Register

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

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

### Query

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

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

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

| Field Name | Example (Hex) |
|---|---|
| Slave Address | 11 |
| Function | 06 |
| Register Address Hi | 00 |
| Register Address Lo | 01 |
| Preset Data Hi | 00 |

| | |
|---|---|
| Preset Data Lo | 03 |
| Error Check (LRC or CRC) | -- |

**Response**

The normal response is an echo of the query, returned after the register contents have been preset.

Here is an example of a response to the query:

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

## 2.2.7 07 Read Exception Status

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-e.g., 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:

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

### 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:

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

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

### 2.2.8 11 (0B Hex) Fetch Comm Event Counter

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:

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

### 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:

| 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) | -- |

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.

### 2.2.9 12 (0C Hex) Fetch Comm Event Log

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:

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

### 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:

| 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) | -- |

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.

### 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.

### 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 and 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 |

### 2.2.10 15 (0F Hex) Force Multiple Coils

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

**Note:** The function will override the controller's memory protect state and 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.

### 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 and 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.

| 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) | -- |

### 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:

| 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) | -- |

## 2.2.11 16 (10 Hex) Preset Multiple Registers

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

**Note:** The function will override the controller's memory protect state. The preset 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.

### 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:

| Field Name | Example (Hex) |
|---|---|
| Slave Address | 11 |
| Function | 10 |
| Starting Address Hi | 00 |
| Starting Address Lo | 01 |
| Number of Registers Hi | 00 |
| Number of Registers Lo | 02 |
| Byte Count | 04 |
| Data Hi | 00 |
| Data Lo | 0A |
| Data Hi | 01 |
| Data Lo | 02 |
| Error Check (LRC or CRC) | -- |

### Response

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

| Field Name | Example (Hex) |
|---|---|
| Slave Address | 11 |
| Function | 10 |
| Starting Address Hi | 00 |
| Starting Address Lo | 01 |
| Number of Registers Hi | 00 |
| Number of Registers Lo | 02 |
| Error Check (LRC or CRC) | -- |

### 2.2.12 17 (11 Hex) Report Slave ID

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:

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

#### Response

The format of a normal response is shown below. The data contents are specific to each controller type.

| Field Name | Example (Hex) |
|---|---|
| Slave Address | Echo 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) | -- |

#### 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 |

**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 and 4 for further definition | | |
| 2 | RUN indicator status | | 0 =OFF |
| | | | FF =ON |
| 3 and 4 | Status word | | |
| | | Bit 0 | 0 |
| | | Bit 1 | 0 =Memory Protect OFF |
| | | | 1 =Memory Protect ON |
| | | Bits 2 and 3 | 0, 0 =184 Controller |
| | | | 1, 0 =384 Controller |
| | | Bits 4 ... 15 | Unused |

Bytes 5 ... 10, returned for a correct J347 setup and normal PIB, are:

| Byte | Content |
|---|---|
| 5 and 6 | PIB table starting address |
| 7 and 8 | Controller serial number |
| 9 and 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:

| Byte | Content |
|---|---|
| 11 and 12 | Maximum quantity of output coils |
| 13 and 14 | Output coil enable table |
| 15 and 16 | Address of input coil/run table |
| 17 and 18 | Quantity of input coils |
| 19 and 20 | Input coil enable table |
| 21 and 22 | First latch number (multiple of 16) |
| 23 and 24 | Last latch number (multiple of 16) |
| 25 and 26 | Address of input registers |
| 27 and 28 | Quantity of input registers |
| 29 and 30 | Quantity of output and holding registers |
| 31 and 32 | Address of user logic |
| 33 and 34 | Address of output coil RAM table |
| 35 and 36 | Function inhibit mask |
| 37 and 38 | Address of extended function routine |
| 39 and 40 | Address of data transfer routine |
| 41 and 42 | Address of traffic cop |
| 43 and 44 | Unused |
| 45 and 46 | Function inhibit mask |
| 47 and 48 | Address of A Mode history table |
| 49 and 50 | Request table for DX printer |
| 51 and 52 | Quantity of sequence groups |
| 53 and 54 | Address of sequence image table |
| 55 and 56 | Address of sequence RAM |
| 57 and 58 | Quantity of 50XX registers |
| 59 and 60 | Address of 50XX table |
| 61 and 62 | Address of output coil RAM image |
| 63 and 64 | Address of input RAM image |
| 65 and 66 | Delayed output start group |
| 67 and 68 | Delayed output end group |
| 69 and 70 | Watchdog line |
| 71 and 72 | RAM Address of latches |
| 73 and 74 | Quantity of delayed output groups |

**584**

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

| Byte | Contents | | |
|---|---|---|---|
| 1 | Slave ID | | 3 |
| 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 | High byte of the machine state word (configuration table word 101, 65 hex) | | |
| | | Bit 15 (MSB of byte) | 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 | 0 =Constand Sweep OFF |
| | | | 1 =Constand Sweep ON |
| | | Bit 9 | 0 =Single Sweep OFF |
| | | | 1=Single Sweep ON |
| 6 | | Bit 8 | 0 =24-bit nodes |
| | | | 1 =16-bit nodes |
| 7 | Low byte of the machine state word | | |
| | | Bit 7 | 1 =Power ON |
| | | | should never =0 (OFF) |
| | | Bit 6 | 0 =RUN indicator ON |
| | | | 1 =RUN indicator OFF |
| | | Bit 5 | 0 =Memory Protect ON |
| | | | 1 =Memory Protect OFF |
| | | Bit 4 | 0 =Battery OK |
| | | | 1 =Battery Not OK |
| 7 | | Bits 3 ... 0 | Unassigned |
| 8 | High byte of the machine stop code (configuration table word 105, 69 hex) | | |
| | | Bit 15 (MSB of byte) | 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 |

| | | |
|---|---|---|
| 9 | Low byte of the machine stop code | |
| | Bit 7 (MSB of byte) | 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 |

**984**

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

| Byte | Contents | | |
|---|---|---|---|
| 1 | Slave ID | | 9 |
| 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 | High byte of the machine state word (configuration table word 101, 65 hex) | | |
| | | Bit 15 (MSB of byte) | Unassigned |
| | | Bits 14 ... 11 | Unassigned |
| | | Bit 10 | 0 = Constand Sweep OFF |
| | | | 1 = Constand Sweep ON |
| | | Bit 9 | 0 = Single Sweep OFF |
| | | | 1 = Single Sweep ON |
| | | Bit 8 | 0 = 24-bit nodes |
| | | | 1 = 16-bit nodes |
| 7 | Low byte of the machine state word | | |
| | | Bit 7 (MSB of byte) | 1 = Power ON |
| | | | should never = 0 ( OFF) |
| | | Bit 6 | 0 = RUN indicator ON |
| | | | 1 = RUN indicator OFF |
| | | Bit 5 | 0 = Memory Protect ON |
| | | | 1 = Memory Protect OFF |
| | | Bit 4 | 0 = Battery OK |
| | | | 1 = Battery Not OK |
| | | Bits 3 ... 1 | Unassigned |
| | | Bit 0 | 0 = NO Memory downsize |

| | | | |
|---|---|---|---|
| 7 | | Bit 0 | 1 = Memory Downsize |

**Note:** 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 lo byte * 16)

State table size (16-bit words) = (Word 100 * 1024) - (Word 175 hi byte * 16)

| Byte | Contents | | |
|---|---|---|---|
| 8 | High byte of the machine stop code (configuration table word 105, 69 hex) | | |
| | | Bit 15 (MSB of byte) | 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 |
| 9 | Low byte of the machine stop code | | |
| | | Bit 7 (MSB of byte) | 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 |

### Micro 84

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

| Byte | Contents | |
|---|---|---|
| 1 | Slave ID | 0 |
| 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 |
| 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 |
| 2 | RUN indicator status | | 0 = OFF |
| | | | FF = ON |
| 3 | Current port number | | |
| 4 | Size of user logic plus state RAM in kbytes (1 word = 2 bytes) | | |
| 5 | Reserved | | |
| 6 | Hook bits | | |
| | | Bits 0 and 2 | Reserved |
| | | Bit 3 | 1 = Do not execute standard mapper |
| | | Bit 4 | 1 = Test end-of-scan hooks |
| | | Bit 5 | Reserved |
| | | Bit 6 | 1 = Do not execute standard logic solver |
| | | Bit 7 | Reserved |
| 7, 8 | Reserved | | |

### 2.2.13 20 (14 Hex) Read General Reference

Returns the contents of registers in Extended Memory file (6*x*) references. Broadcast is not supported. The function can read multiple groups of references. The groups can be separate (noncontiguous), 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 seven bytes:

V The reference type-one byte (must be specified as 6)

V The Extended Memory file number-two bytes (1 ... 10, 0001 ... 000A hex)

V The starting register address within the file-two bytes

V The quantity of registers to be read-two 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).

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

| Extended Memory 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 | User Logic | State RAM | Extended Mem Size | Last File | Last Reg (Decimal) |
|---|---|---|---|---|---|
| with AS-M785-032 Memory Cartridge | 32K | 32K | 0 | 0 | 0 |
| | 16K | 64K | 72K | 8 | 3727 |
| with AS-M785-048 Memory Cartridge | 48K | 32K | 24K | 3 | 4575 |
| | 32K | 64K | 96K | 10 | 8303 |

Examples of a query and response follow. An example of a request to read two groups of references from slave device 17 is shown. Group 1 consists of two registers from file 4, starting at register 2 (address 0001). Group 2 consists of two registers from file 3, starting at register 10 (address 0009).

| 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 Lo02 | 02 |
| Error Check (LRC or CRC) | -- |

## 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.

| Field Name | Example (Hex) |
|---|---|
| Slave Address | 11 |
| Function | 14 |
| Byte Count | 0C |
| Sub–Res 1, Byte Count | 05 |
| Sub-Req 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) | -- |

### 2.2.14 21 (15 Hex) Write General Reference

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

The function can write multiple groups of references. The groups can be separate (noncontiguous), 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 seven bytes plus the data:

V The reference type-one byte (must be specified as 6)

V The Extended Memory file number-two bytes (1 ... 10, 0001 ... 000A hex)

V The starting register address within the file-two bytes

V The quantity of registers to be written-two bytes

V The data to be written-two bytes/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).

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

| Extended Memory 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 | User Logic | State RAM | Extended Mem Size | Last File | Last Reg (Decimal) |
|---|---|---|---|---|---|
| with AS-M785-032 Memory Cartridge | 32K | 32K | 0 | 0 | 0 |
| | 16K | 64K | 72K | 8 | 3727 |
| with AS-M785-048 Memory Cartridge | 48K | 32K | 24K | 3 | 4575 |
| | 32K | 64K | 96K | 10 | 8303 |

Examples of a query and response follow. An example of a request to write one group of references into slave device 17 is shown. The group consists of three registers in file 4, starting at register 8 (address 0007).

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

### Response

The normal response is an echo of the 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-Res 1, Register Count Hi | 00 |
| Sub-Res 1, Register Count Lo | 03 |
| Sub-Req 1, Register Data Hi | 06 |
| Sub-Req 1, Register Data Lo | AF |
| Sub-Res 1, Register Data Hi | 04 |
| Sub-Res 1, Register Data Lo | BE |
| Sub-Res 1, Register Data Hi | 10 |
| Sub-Res 1, Register Data Lo | 0D |
| Error Check (LRC or CRC) | -- |

### 2.2.15 22 (16 Hex) Mask Write 4*x* Register

Modifies the contents of a specified 4*x* 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.

**Note:** This function is supported in the 984-785 controller only.

**Query**

The query specifies the 4*x* 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 And_Mask )

For example,

| | Hex | Binary |
|---|---|---|
| Current Contents | 12 | 0001 0010 |
| And_Mask | F2 | 1111 0010 |
| Or_Mask | 25 | 0010 0101 |
| And_Mask | 0D | 0000 1101 |
| Result | 17 | 0001 0111 |

**Note:** 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:** 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.

Here is an example of a Mask Write to register 5 in slave device 17, using the above mask values:

| 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) | -- |

### Response

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

| 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) | -- |

### 2.2.16 23 (17 Hex) Read / Write 4*x* Registers

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 4*x* registers, and then return the contents of another group of 4*x* registers. Broadcast is not supported.

**Note:** 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:

| 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) | -- |

## 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:

| 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) | -- |

## 2.2.17 24 (18 Hex) Read FIFO Queue

Reads the contents of a first-in first-out (FIFO) queue of 4*x* 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.

**Note:** This function is supported in the 984-785 controller only.

### Query

The query specifies the starting 4*x* 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).

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

### 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).

Here is an example of a normal response to the previous query:

| 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) | -- |

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:

V 41248 (contents 440 decimal, 01B8 hex)

V 41249 (contents 4740, 1284 hex)

V 41250 (contents 4898, 1322 hex)

# Chapter 3
# Diagnostic Subfunctions

V Get / Clear Modbus Plus Statistics

V Modbus Plus Network Statistics

## 3.1 Function 08-Diagnostics

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).

| 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) | -- |

## 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.

| 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) | -- |

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

## 3.2 Diagnostic Codes Supported by Controllers

Subfunction codes are listed in decimal; Y indicates that the subfunction is supported, and 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 Register | 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 Count | Y | Y | Y | N | N | Y |
| 14 | Return Slave Message Count | Y | Y | Y | N | N | N |
| 15 | Return Slave No Response Count | 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 Character Overrun Count | 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 | | | | | |

(1) =Clears Counters only.

### 3.2.1 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 |

### 3.2.2 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 |

### 3.2.3 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 |

### 3.2.4 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 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 |

**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 ... 15 | Not Used |

### 3.2.5 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 |

### 3.2.6 04 Force Listen Only Mode

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

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

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

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

### 3.2.7 10 (0A Hex) Clear Counters and Diagnostic Register

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

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

### 3.2.8 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 |

### 3.2.9 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 |

### 3.2.10 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. For a description of exception responses, see page .

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

### 3.2.11 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 |

### 3.2.12 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 Repsonse Count |

### 3.2.13 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. For a description of exception responses, see page .

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

### 3.2.14 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. For a description of exception responses, see page .

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

### 3.2.15 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 |

### 3.2.16 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.

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

**Note:** This function is specific to the 884.

### 3.2.17 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).

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

☞
**Note:** This function is specific to the 884.

### 3.2.18 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 cannot 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:

V -- A value of 00 03 specifies the Get Statistics operation.

V -- 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) |
| | | 00 04 (Clear Statistics) |

### Get Statistics Response

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

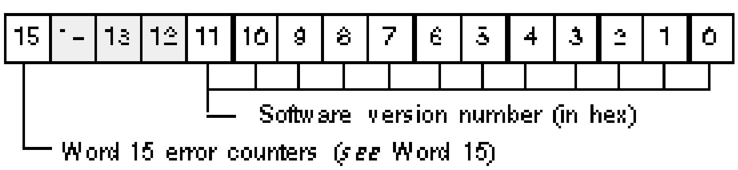
| Function | Subfunction | Operation | Byte Count | Data |
|---|---|---|---|---|
| 08 | 00 15 | 00 03 | 00 6C | Words 00 ... 53 |

## 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 |

## 3.2.19 Modbus Plus Network Statistics

| Word | | Meaning |
|---|---|---|
| 00 | **Bit** | Node type ID |
| | 0 | Unknown node type |
| | 1 | PLC node |
| | 2 | Modbus bridge node |
| | 3 | Host computer node |
| | 4 | Bridge Plus node |
| | 5 | Peer I/O node |
| 01 | **Bit** | |
| | 0 ... 11 | Software version number in hex (to read, strip bits 12–15 from word) |
| | 12 | Device supports dual cable network |
| | 13 | Device supports Peer Cop communication |
| | 14 | Device supports identity reporting |
| | 15 | Defines Word 15 error counters (see Word 15)<br>Most significant bit defines use of error counters in Word 15. Least significant half of upper byte, plus lower byte, contain software version.<br><br> |
| 02 | | Network address for this station |
| 03 | **Bit** | 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 | Monitoring link operation only—passive station |
| | 32 | Normal link operation |
| | 64 | Never getting token—sees tokens, receives none |
| | 96 | Sole station—never sees tokens |
| | 128 | Duplicate station—sees other stations with same address |
| 05 | | Token pass counter; increments each time this station gets the token |
| 06 | | Token rotation time in ms |
| 07 | **Byte** | |
| | LO | Data master failed during token ownership bit map |
| | HI | Program master failed during token ownership bit map |

**Note:** Word 07 bitmaps are used internally by the peer processor to determine which paths have already had a command sent to them during the current token ownership. This limits the number of commands per path to one during a single token ownership.

| Word | Byte | Meaning |
|---|---|---|
| 08 | LO | Data master token owner work-to-do table |
| | HI | Program master token owner work-to-do table |
| 09 | LO | Data slave token owner work-to-do table |
| | HI | Program slave token owner work-to-do table |
| 10 | LO | Data master response (now available to read) |
| | HI | Data slave command |
| 11 | LO | Program master response (now available to read) |
| | HI | Program slave command |
| 12 | LO | Program master connect status table—master paths in use |
| | HI | Program slave automatic logout request table—slaves to log out |

**Note:** Words 08 ... 12 are token owner work tables. They are bitmaps representing work that needs to be done by the node the next time it gets the token. Each byte is a bitmap corresponding to work requested of each of the eight paths of the indicated type.

| Word | Byte | Meaning |
|---|---|---|
| 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 |
| | | **Note** 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 |
| 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 (increments normally) |
| | HI | No response received error counter (increments normally 1 ... 10 times/s). Each station occasionally allows a new station to join the network, which increments this counter. If a station leaves, the remaining stations continue to increment their error counters until the station is removed from each station's map. |
| 21 | LO | Exception response received error counter—LLC layer error, illegal packet error |
| | HI | Unexpected path error counter—data packet contains illegal path field |
| 22 | LO | Unexpected response counter—packet sent to wrong destination |
| | HI | Forgotten transaction error counter—command was initiated but never completed, possibly because the response packet had the wrong path, sequence numbers, or node number. |

**Note:** Words 13 ... 22 contain pairs of 8-bit counters that pertain to certain types of error conditions as well as to successful transactions. Under normal operating conditions, the only bytes that change are word 20 LO and HI. Word 14 HI could also increment because of an MSTR or similar programming error in the application. If any other bytes increments, a possible problem exists on the network-e.g., in a single station or wiring connection.

| Word | Byte | Meaning |
|---|---|---|
| 23 | LO | Active station table bit map, nodes 8 ... 1 |
| | HI | Active station table bit map, nodes 16 ...9 |
| 24 | LO | Active station table bit map, nodes 24 ... 17 |
| | HI | Active station table bit map, nodes 32 ... 25 |
| 25 | LO | Active station table bit map, nodes 40 ... 33 |
| | HI | Active station table bit map, nodes 48 ... 41 |
| 26 | LO | Active station table bit map, nodes 56 ... 49 |
| | HI | Active station table bit map, nodes 64 ... 57 |

☞
**Note:** Words 23 ... 26 contain the active station bitmaps. An active station is any one that has sent packets of data over the network.

| Word | Byte | Meaning |
|---|---|---|
| 27 | LO | Token station table bit map, nodes 8 ... 1 |
| | HI | Token station table bit map, nodes 16 ...9 |
| 28 | LO | Token station table bit map, nodes 24 ... 17 |
| | HI | Token station table bit map, nodes 32 ... 25 |
| 29 | LO | Token station table bit map, nodes 40 ... 33 |
| | HI | Token station table bit map, nodes 48 ... 41 |
| 30 | LO | Token station table bit map, nodes 56 ... 49 |
| | HI | Token station table bit map, nodes 64 ... 57 |

☞
**Note:** Words 27 ... 30 contain the token station table bitmaps. A token station is any one that has token-passing capabilities.

| Word | Byte | Meaning |
|---|---|---|
| 31 | LO | Global data present table bit map, nodes 8 ... 1 |
| | HI | Global data present table bit map, nodes 16 ...9 |
| 32 | LO | Global data present table bit map, nodes 24 ... 17 |
| | HI | Global data present table bit map, nodes 32 ... 25 |
| 33 | LO | Global data present table bit map, nodes 40 ... 33 |
| | HI | Global data present table bit map, nodes 48 ... 41 |
| 34 | LO | Global data present table map, nodes 56 ... 49 |
| | HI | Global data present table bit map, nodes 64 ... 57 |

**Note:** Words 31 ... 34 contain the global data present table bitmaps. Each time a station passes a token, it also passes the global data, even if there are zero bytes of global data. When one station sees another pass the token with global data, it sets its bit in its table for that other station. The bit remains set until the station reads the global data from that other station, after which the bit is cleared. A second read of global data indicates that no global data is present.

**Note:** In screen 2 of the MBPSTAT program, the number of global data words present is indicated under the station number. If this field is filled with spaces, then MBPSTAT has requested the global data from a second time before the other station passed the token.

| Word | Byte | Meaning |
|---|---|---|
| 35 | LO | Receive buffer in use bit map, buffer 8 ... 1 |
| | HI | Receive buffer in use bit map, buffer 16 ... 9 |
| 36 | LO | Receive buffer in use bit map, buffer 24 ... 17 |
| | HI | Receive buffer in use bit map, buffer 32 ... 25 |
| 37 | LO | Receive buffer in use bit map, buffer 40 ... 33 |
| | HI | Station management command processed initiation counter |

**Note:** The LO bytes of words 35 ... 37 indicate the use of the internal receive buffers within the peer processor.

| Word | Byte | Meaning |
|---|---|---|
| 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 |
| 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 |

# Chapter 4
# Exception Responses



### 4.1 Exception Responses

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

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

V If the slave does not receive the query due to a communication error, no response is returned. The master program will eventually process a timeout condition for the query.

V If the slave receives the query, but detects a communication error (parity, LRC, or CRC), no response is returned. The master program will eventually process a timeout condition for the query.

V If the slave receives the query without a communication error, but cannot handle it (for example, if the request is to read a nonexistent coil or register), the slave will return an exception response informing the master of the nature of the error.

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

#### Function Code Field

In a normal response, the slave echoes the function code of the original query in the function code field of the response. All function codes have a most significant bit (MSB) of 0 (their values are all below 80 hexadecimal). In an exception response, the slave sets the MSB of the function code to 1. This makes the function code value in an exception response exactly 80 hexadecimal higher than the value would be for a normal response.

With the function code's MSB set, the master's application program can recognize the exception response and can examine the data field for the exception code.

#### Data Field

In a normal response, the slave may return data or statistics in the data field (any information that was requested in the query). In an exception response, the slave returns an exception code in the data field. This defines the slave condition that caused the exception. Here is an example of a master query and slave exception response. The field examples are shown in hexadecimal.

**Query**

| Byte | Contents | Example |
|---|---|---|
| 1 | Slave Address | 0A |
| 2 | Function | 01 |
| 3 | Starting Address Hi | 04 |
| 4 | Starting Address Lo | A1 |
| 5 | Number of Coils Hi | 00 |
| 6 | Number of Coils Lo | 01 |
| 7 | LRC | 4F |

**Exception Response**

| Byte | Contents | Example |
|---|---|---|
| 1 | Slave Address | 0A |
| 2 | Function | 81 |
| 3 | Exception Code | 02 |
| 4 | LRC | 73 |

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

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

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

### 4.2 Exception Codes

| Code | Name | Meaning |
|---|---|---|
| 01 | ILLEGAL FUNCTION | The function code received in the query is not an allowable action for the slave. If a Poll Program Complete command was issued, this code indicates that no program function preceded it. |
| 02 | ILLEGAL DATA ADDRESS | The data address received in the query is not an allowable address for the slave. |
| 03 | ILLEGAL DATA VALUE | A value contained in the query data field is not an allow-able value for the slave |
| 04 | SLAVE DEVICE FAILURE | An unrecoverable error occurred while the slave was at-tempting to perform the requested action. |
| 05 | ACKNOWLEDGE | The slave has accepted a request and is processing it, but a long duration of time is required. This response is returned to prevent a timeout error from occurring in the master. The master can next issue a Poll Program Complete message to determine if processing is completed. |
| 06 | SLAVE DEVICE BUSY | The slave is processing a long-duration program command. The master should retransmit the message later when the slave is free. |
| 07 | NEGATIVE ACKNOWLEDGE | The slave cannot perform the program function received in the query. This code is returned for an unsuccessful programming request using function code 13 or 14 decimal. The master should request diagnostic or error information from the slave. |
| 08 | MEMORY PARITY ERROR | The slave attempted to read extended memory, but detected a parity error in the memory. The master can retry the request, but service may be required on the slave device. |

# Chapter 5
# Application Notes

V Maximum Query / Response Parameters

V Estimating Serial Transaction Timing

V Application Notes for the 584 and 984A / B / X

## 5.1 Maximum Query / Response Parameters

The listings 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 |
|---|---|---|---|
| 01 | Read Coil Status | 800 coils | 800 coils |
| 02 | Read Input Status | 800 inputs | 800 inputs |
| 03 | Read Holding Registers | 100 registers | 100 registers |
| 04 | Read Input Registers | 100 registers | 100 registers |
| 05 | Force Single Coil | 1 coil | 1 coil |
| 06 | Preset Single Register | 1 register | 1 register |
| 07 | Read Exception Status | N/A | 8 coils |
| 08 | Diagnostics | N/A | N/A |
| 09 | Program 484 | Not supported | Not supported |
| 10 | Poll 484 | Not supported | Not supported |
| 11 | Fetch Comm Event Counter | 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 |
|---|---|---|---|
| 01 | Read Coil Status | 512 coils | 512 coils |
| 02 | Read Input Status | 512 inputs | 512 inputs |
| 03 | Read Holding Registers | 254 registers | 254 registers |
| 04 | Read Input Registers | 32 registers | 32 registers |
| 05 | Force Single Coil | 1 coil | 1 coil |
| 06 | Preset Single Register | 1 register | 1 register |
| 07 | Read Exception Status | N/A | 8 coils |
| 08 | Diagnostics | N/A | N/A |
| 09 | Program 484 | 16 data bytes | 16 data bytes |
| 10 | Poll 484 | N/A | 16 data bytes |
| 11 | Fetch Comm Event Counter | 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 |

**584**

| Function | Description | Query | Response |
|---|---|---|---|
| 01 | Read Coil Status | 2000 coils | 2000 coils |
| 02 | Read Input Status | 2000 inputs | 2000 inputs |
| 03 | Read Holding Registers | 125 registers | 125 registers |
| 04 | Read Input Registers | 125 registers | 125 registers |
| 05 | Force Single Coil | 1 coil | 1 coil |
| 06 | Preset Single Register | 1 register | 1 register |
| 07 | Read Exception Status | N/A | 8 coils |
| 08 | Diagnostics | N/A | N/A |
| 09 | Program 484 | Not supported | Not supported |
| 10 | Poll 484 | Not supported | Not supported |
| 11 | Fetch Comm Event Counter | 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) |

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

## 884

| Function | Description | Query | Response |
|---|---|---|---|
| 01 | Read Coil Status | 2000 coils | 2000 coils |
| 02 | Read Input Status | 2000 inputs | 2000 inputs |
| 03 | Read Holding Registers | 125 registers | 125 registers |
| 04 | Read Input Registers | 125 registers | 125 registers |
| 05 | Force Single Coil | 1 coil | 1 coil |
| 06 | Preset Single Register | 1 register | 1 register |
| 07 | Read Exception Status | N/A | 8 coils |
| 08 | Diagnostics | N/A | N/A |
| 09 | Program 484 | Not supported | Not supported |
| 10 | Poll 484 | Not supported | Not supported |
| 11 | Fetch Comm Event Counter | 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 |

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

## M84

| Function | Description | Query | Response |
|---|---|---|---|
| 01 | Read Coil Status | 64 coils | 64 coils |
| 02 | Read Input Status | 64 inputs | 64 inputs |
| 03 | Read Holding Registers | 32 registers | registers |
| 04 | Read Input Registers | 4 registers | 4 registers |
| 05 | Force Single Coil | 1 coil | 1 coil |
| 06 | Preset Single Register | 1 register | 1 register |
| 07 | Read Exception Status | N/A | 8 coils |
| 08 | Diagnostics | N/A | N/A |
| 09 | Program 484 | Not supported | Not supported |
| 10 | Poll 484 | Not supported | Not supported |
| 11 | Fetch Comm Event Counter | 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 |

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

## 984

| Function | Description | Query | Response |
|---|---|---|---|
| 01 | Read Coil Status | 2000 coils | 2000 coils |
| 02 | Read Input Status | 2000 inputs | 2000 inputs |
| 03 | Read Holding Registers | 125 registers | 125 registers |
| 04 | Read Input Registers | 125 registers | 125 registers |
| 05 | Force Single Coil | 1 coil | 1 coil |
| 06 | Preset Single Register | 1 register | 1 register |
| 07 | Read Exception Status | N/A | 8 coils |
| 08 | Diagnostics | N/A | N/A |
| 09 | Program 484 | Not supported | Not supported |
| 10 | Poll 484 | Not supported | Not supported |
| 11 | Fetch Comm Event Counter | Not supported | Not supported |

| | | | |
|---|---|---|---|
| 12 | Fetch Comm Event Log | Not supported | 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) |

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

### 5.2 Estimating Serial Transaction Timing

The following sequence of events occurs 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:

Time (ms) = 1000 * (character count) * (bits/character)
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 half the 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.

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 eight bits of data to be returned in the response.

## 5.3 Notes for the 584 and 984A / B / X

### 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 1 s is allowed to occur after the LRC field. If this happens, the controller will assume that the message has terminated normally.

# Chapter 6
# LRC / CRC Generation

V LRC Generation

V CRC Generation

**6.1 LRC Generation**

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

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

**Generating an LRC**

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

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

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

**Placing the LRC into the Message**

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

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

**Figure 8 LRC Character Sequence**

**Example**

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

```
unsigned char *auchMsg ;        A pointer to the message buffer
con-
                                                                    taining
binary data to be used for
                                                                     generating
the LRC

unsigned short usDataLen ;      The quantity of bytes in the
                                                                    message
buffer.
```

The function returns the LRC as a type unsigned char.

**LRC Generation Function**

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

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

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

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

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

**6.2 CRC Generation**

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

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

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

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

**Generating a CRC**

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

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

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

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

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

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

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

**Step 7** When the CRC is placed into the message, its upper and lower bytes must be swapped as described below.

**Placing the CRC into the Message**

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

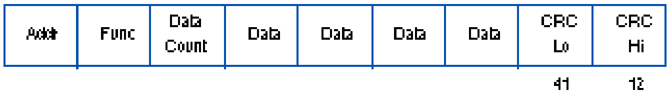



**Figure 9 CRC Byte Sequence**

**Example**

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

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

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

The function takes two arguments:

```
unsigned char *puchMsg ;        A pointer to the message buffer
                                                                        containing
binary data to be used
                                                                        for
generating the CRC

unsigned short usDataLen ;      The quantity of bytes in the
                                                                        message
buffer.
```

The function returns the CRC as a type unsigned short.

**CRC Generation Function**

```
unsigned short CRC16(puchMsg, usDataLen)

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

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

/* table
  */

        while (usDataLen--)             /* pass through message buffer
  */
                {
                uIndex = uchCRCHi ^ *puchMsgg++ ;       /* calculate the CRC
 */
```

```
                uchCRCHi = uchCRCLo ^ auchCRCHi[uIndex} ;
                uchCRCLo = auchCRCLo[uIndex] ;
                }

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

**High Order Byte Table**

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

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

**Low Order Byte Table**

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

static char auchCRCLo[] = {
0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06,
0x07, 0xC7, 0x05, 0xC5, 0xC4, 0x04, 0xCC, 0x0C, 0x0D, 0xCD,
0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09,
0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A,
0x1E, 0xDE, 0xDF, 0x1F, 0xDD, 0x1D, 0x1C, 0xDC, 0x14, 0xD4,
0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13, 0xD3,
0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3,
0xF2, 0x32, 0x36, 0xF6, 0xF7, 0x37, 0xF5, 0x35, 0x34, 0xF4,
0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A,
0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29,
0xEB, 0x2B, 0x2A, 0xEA, 0xEE, 0x2E, 0x2F, 0xEF, 0x2D, 0xED,
0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26,
0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60,
```

```
0x61, 0xA1, 0x63, 0xA3, 0xA2, 0x62, 0x66, 0xA6, 0xA7, 0x67,
0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC, 0xAD, 0x6D, 0xAF, 0x6F,
0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68,
0x78, 0xB8, 0xB9, 0x79, 0xBB, 0x7B, 0x7A, 0xBA, 0xBE, 0x7E,
0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C, 0xB4, 0x74, 0x75, 0xB5,
0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71,
0x70, 0xB0, 0x50, 0x90, 0x91, 0x51, 0x93, 0x53, 0x52, 0x92,
0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54, 0x9C, 0x5C,
0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B,
0x99, 0x59, 0x58, 0x98, 0x88, 0x48, 0x49, 0x89, 0x4B, 0x8B,
0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C, 0x8C,
0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42,
0x43, 0x83, 0x41, 0x81, 0x80, 0x40
} ;
```