36

Modbus

Mário de Sousa
University of Porto

Paulo Portugal
University of Porto

36.1 Introduction

The Modbus protocol was created in 1979 by Modicon* as a means of sharing data between their PLCs (programmable logic controllers). Modicon took the approach of openly publishing its specification and allowing its use by anyone without asking for royalties. These factors, along with the simplicity of the protocol itself, allowed it to become the first widely accepted *de facto* standard for industrial communication.

Although initially a proprietary protocol controlled only by Modicon, it is since 2004 controlled by a community of users and suppliers of automation equipment, known as Modbus-IDA. This nonprofit organization oversees the evolution of the protocol and seeks to drive its adoption by continuing to openly distribute the protocol specifications [1–4] and providing an infrastructure for device compatibility certification.

Currently, Modbus is used in equipment ranging from the smallest network-enabled sensors and actuators to complex automation controllers and SCADA (supervisory control and data acquisition) systems. Additionally, many implementations are freely available in source code form in a great variety of programming languages. Readers interested in obtaining the Modbus standards or one of the many available implementations of the protocol are suggested to access the Modbus-IDA Web site at http://www.mobus.org.

36.2 Modbus Interaction and Data Models

The Modbus communication protocol provides a means whereby one device may read and write data to memory areas located on another remote device. The typical example of a remote device is an RTU (remote terminal unit) providing a physical interface (inputs and outputs) to an industrial process, with

* At the time, Modicon was a PLC Manufacturer. It is now a brand of Schneider Electric.

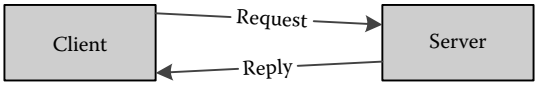

FIGURE 36.1 Client–server interaction model.

little to no processing capability, whereas the device taking the initiative to read and write data is usually a PLC. These two devices interact over the Modbus protocol using the client–server interaction model (Figure 36.1). The client (PLC) takes the initiative of asking the server (RTU) to write and read data that is present in the RTU. The server merely responds to these requests, never taking the initiative to start a data exchange.

A client may make requests to several distinct Modbus servers, usually one at a time. Likewise, the server may respond to requests coming from several clients. Whether or not the server is capable of processing simultaneous requests coming from several clients depends on the server, although typically these are processed one after the other.

The organization of the memory areas on the server to which the client may write to or read from is defined by the Modbus protocol itself. There are four distinct memory areas (Table 36.1); two of them are organized as 16 bit registers, whereas the other two are composed of arrays of single bits. Likewise, two of the memory areas provide read and write access permissions, while the other two may only be read from.

Typically, an RTU acting as a Modbus server will allow its physical digital inputs to be read through the "discrete inputs" memory area, the digital outputs to be read and written to through the "coils" memory area, the status of any analog inputs will be read using the "input registers" memory area, and the values of any analog outputs will be read and changed through the "holding registers" memory area. Nevertheless, it is completely up to the server manufacturer to decide the semantics attributed to the data located in these memory areas. In other words, the Modbus protocol does not define what will happen on the server when a specific memory address is written with some determined value. How the memory areas are organized, and where the data are actually stored, will depend on the type of device acting as a Modbus server.

An example would be a variable speed drive that controls the rotation speed of an electrical motor. When acting as a Modbus server, it may map the holding registers memory area to the area where the drive stores its configuration parameters (maximum and minimum motor voltage, motor speed, etc.), and the input registers memory area to the speed drive internal status registers (current speed, current output voltage, etc.).

Another example would be a graphical HMI (human–machine interface) terminal, which graphically displays a representation of the current state of an industrial process and permits an operator to modify operating parameters using specific buttons and dials. The HMI terminal, acting as a Modbus client, obtains the data values corresponding to the industrial process' state from the controlling PLC, which acts as the Modbus server. This PLC might use the holding register memory area to store the current state of a continuous variable of the process under control (e.g., volume of liquid in a tank), and the coil's memory area to store the flags indicating which on-screen buttons the operator is currently pressing.

TABLE 36.1 Memory Areas of the Modbus Data Model

Memory Area Name	Address—Range	Register Size (Bits)	Access Permissions
Input registers	1–65,536	16	Read
Holding registers	1–65,536	16	Read/write
Discrete inputs	1–65,536	1	Read
Coils	1–65,536	1	Read/write

The devices acting as Modbus servers may even decide to map several Modbus memory areas to the same internal physical memory. In these cases, the Modbus memory areas may overlap, and writing to a holding register, for example, may also change the state of the corresponding input register residing on the same physical memory.

Note, however, that although the Modbus Data Model specifies that elements in each memory area are addressed using values ranging from 1 to 65,536, unfortunately these address values are in actual fact encoded on the frames sent over the "wire" between the client and server as ranging from 0 to 65,535. This usually leads to many errors of "off by one," as the manuals of some device manufacturers specify how the Modbus memory areas are mapped onto to the device's internal architecture assuming the 1–65,536 addressing range, whereas other device manufacturers assume the 0–65,535 addressing range. When reading a manual one can never be sure which addressing range is being used, unless this is explicitly stated, which is rare. However, if the manual makes any reference to an address value of "0," then usually one may safely assume that the 0–65,535 addressing range is being considered.

36.3 Modbus Protocol Architecture

The Modbus protocol is organized as a two-layer protocol (Figure 36.2).

The upper layer, called the "Modbus application layer," defines the functions or services that a Modbus client may request of a Modbus server, and how these requests are encoded onto a message or APDU (application layer protocol data unit). It also defines how the servers have to reply to each function, and the actions they should take on behalf of the client. This layer is further explained in Section 36.4.

The lower layer defines how the upper layer APDUs are encapsulated and encoded onto frames to be sent over the wire by the underlying physical layer, and how the server devices are addressed. There are three distinct versions of this lower layer. The ASCII (American Standard Code for Information Interchange) version encodes the upper layer APDU as ASCII characters, whereas the RTU and TCP versions use direct byte representation. The RTU and ASCII versions are expected to be sent over EIA/TIA-232 or EIA/TIA-485 (commonly known as RS232 and RS485). The TCP version, as the name implies, is sent over TCP connections established over the IP protocol. Although it is common that the IP protocol frames are then sent over an Ethernet LAN (local area network), there is no reason that they may not use any other underlying network, including the global Internet.

As the RTU and ASCII versions are very similar, they are both described in the same Section 36.5. Section 36.6 will explain the TCP version.

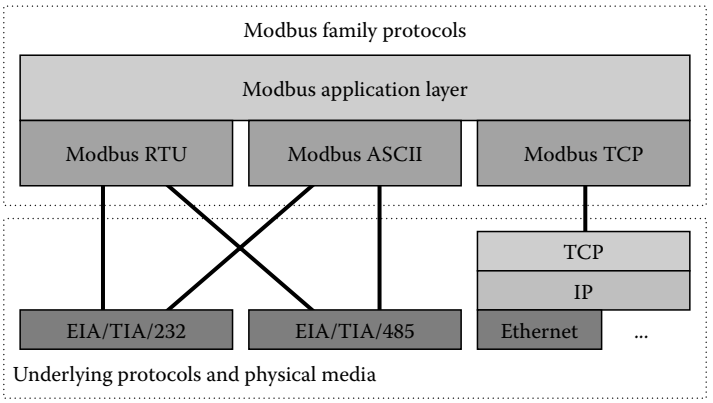

FIGURE 36.2 Organization of Modbus protocols.

36.4 Modbus Application Layer

The Modbus protocol defines three distinct APDUs (Figure 36.3) used in the Modbus application layer. All three APDUs start with a single-byte value indicating the Modbus function being requested or to which a reply is being made. Following the "function" byte value come all data and parameters of that specific function. Data and parameters have a variable number of bytes, depending on the function in question, and the number of memory registers that are being accessed.

All APDUs are, however, limited in size to a maximum of 253 bytes, due to limitations imposed by the underlying EIA/TIA-485 layer. The Modbus protocol also specifies that all 16 bit addresses and data items are encoded using big-endian representation. This means that when a numerical quantity larger than a single byte is transmitted, the most significant byte is sent first. Nevertheless, some device manufacturers allow the user to specify whether the device should use big or little-endian encoding.

The request APDU is sent by the client to the server. Upon successful completion of the desired function, the server replies with a response APDU. If the server encounters an error, the server notifies the client with an exception response APDU.

36.4.1 Data Access Functions

The Modbus protocol defines a large list of functions. The most often used functions are those associated with accessing the memory areas (Table 36.2).

All functions listed in Table 36.2, with the exception of functions 0x14, 0x15, 0x16, and 0x18, simply request that some data be read or written to a specific memory area. Function codes 0x05 and 0x06 are used to write to a single element (coil or holding register, respectively). The remaining functions allow the client to read from or write to multiple contiguous elements of the same memory area.

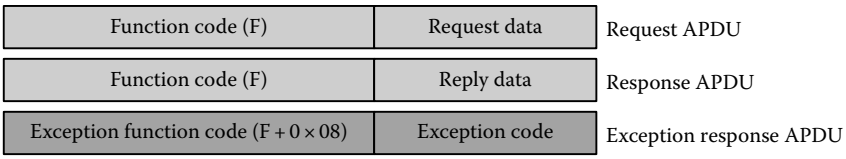

FIGURE 36.3 General format of APDU frames.

TABLE 36.2 Functions Used for Data Access

Memory Area	Function Name	Function Code (Hex)	Addressable Elements	Possible Response Error Codes
Discrete Inputs	Read discrete inputs	0x02	1–2000	01, 02, 03, 04
Coils	Read coils	0x01	1–2000	01, 02, 03, 04
Coils	Write single coil	0x05	1	01, 02, 03, 04
Coils	Write multiple coils	0x0F	1–1976	01, 02, 03, 04
Input registers	Read input registers	0x04	1–125	01, 02, 03, 04
Holding registers	Read holding registers	0x03	1–125	01, 02, 03, 04
Holding registers	Write single register	0x06	1	01, 02, 03, 04
Holding registers	Write multiple registers	0x10	1–123	01, 02, 03, 04
Holding registers	Read/write multiple registers	0x17	1–121 (write) 1–125 (read)	01, 02, 03, 04
Holding registers	Mask write register	0x16	1	01, 02, 03, 04
Holding registers	Read FIFO queue	0x18	1–32	01, 02, 03, 04
Files	Read file record	0x14		01, 02, 03, 04, 08
Files	Write file record	0x15		01, 02, 03, 04, 08

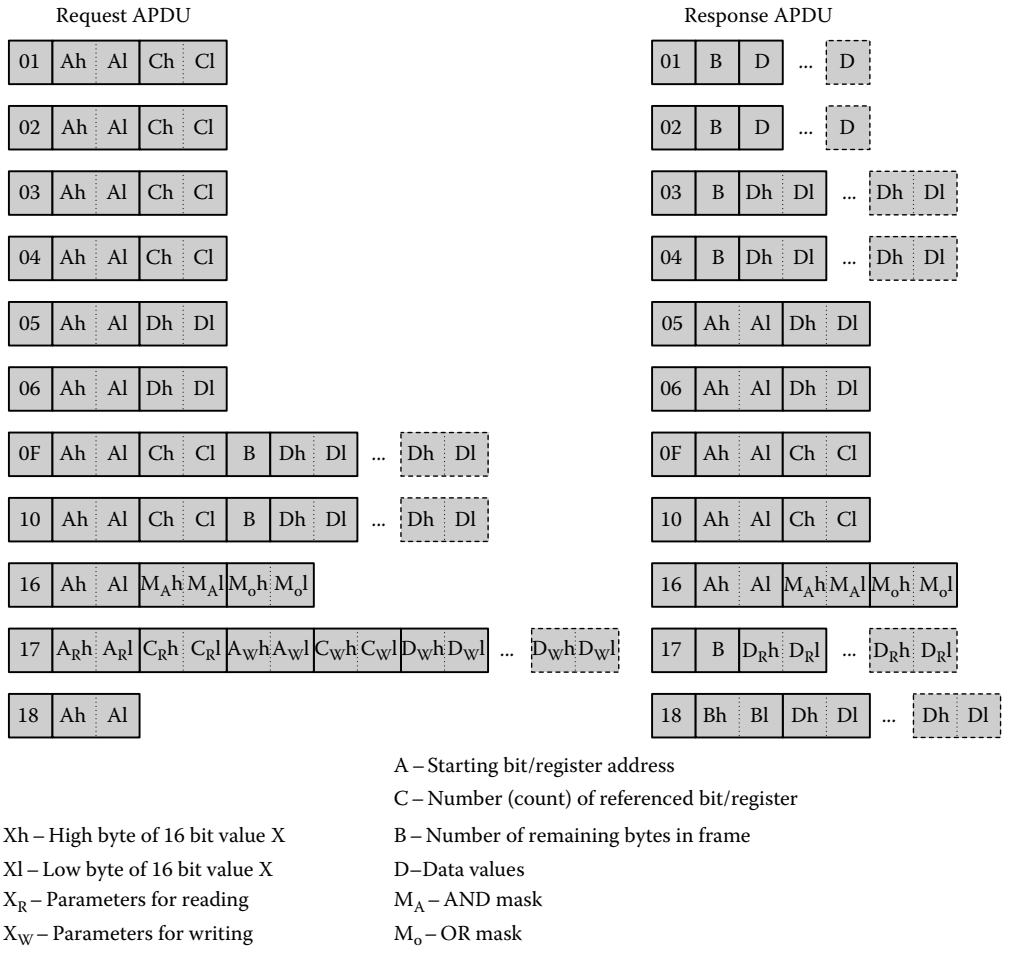

FIGURE 36.4 Format of data access functions APDUs.

Note that the maximum number of addressable elements is limited by the maximum size of the APDU. For example, for function code 0x0F, the request APDU starts off with 6 bytes containing the function code and respective parameters (see Figure 36.4), leaving a maximum of 247 (equal to 253 – 6) available bytes in which to send the data. The data, consisting of 1 bit elements, are packed 8 per byte, resulting in a maximum of 1976 (247 × 8) addressable elements.

Function 0x16 (Mask Write Register) changes the value stored in a single holding register. However, unlike the other functions, the new value is not obtained directly from the data sent in the APDU, but rather is obtained by applying a logical operation using the current value of the holding register with two masks, an AND mask and an OR mask, sent with the APDU.

New_value = (Current_value AND And_mask) OR (Or_mask AND (NOT And_mask))

Using this function the client can switch off certain bits and set other bits of the holding register, while simultaneously leaving the remaining bits unchanged, in a single atomic operation.

Function 0x18 (Read FIFO Queue) allows the client to request reading a FIFO queue from the server. The FIFO queue must be stored within the holding registers memory area in the server and consists of a first register containing the number of elements in the queue (queue count register), followed by the values of each element in the queue in the following registers (queue data registers). The client merely

indicates in the request APDU the address of the queue count register, and the server will reply with the value contained in this queue count register, followed by the value of each queue data register, up to a maximum of 31 data registers.

For each function, the format of the request APDU sent by the client, and the respective response APDU with which the server replies, may be found in Figure 36.4. Note that in the functions accessing bit-sized memory areas (i.e., functions 0x01, 0x02, 0x05, 0x0F), the bits are sent packed 8 per byte. For example, a function reading or writing 20 bits will result in the state of the first 16 bits being packed into 2 bytes, and the remaining 4 bits sent on the least significant bits of the third data byte.

The Modbus protocol includes two additional functions for data access. These functions (0x14 for reading and 0x15 for writing) are referred to in the base Modbus specification documents as "Read File Record" and "Write File Record," but in the specification of the Modbus/TCP version are called "Read General Reference" and "Write General Reference." A file contains 10,000 records (addressed from 0 to 9,999), each record being a 16 bit element. Each file is referenced by its number, ranging from 0 to 65,535. Note that the memory referenced by these files is independent from the memory areas defined in Table 36.1. For details regarding these functions, which are seldomly implemented by Modbus servers, interested readers are encouraged to consult the freely available Modbus specifications [1–4].

36.4.2 Diagnostic Functions

The second large group of Modbus functions allows the client to obtain diagnostic information from the server (Table 36.3). These functions are only available on serial line devices, and many commercially available Modbus servers do not implement some or all of these functions.

With function 0x07 (Read Exception Status) the client may read 8 bits of device-specific exception status information. Function 0x08 (Diagnostic) is used to obtain diagnostic information from the server, or to have the server execute some auto-diagnostic routines that should not affect the normal operation of the server. The function is followed by a subfunction code indicating which diagnostic information is being requested (bus message count, communication error count, etc.), or which diagnostic routine should be executed (restart communication, force listen only mode, etc.).

With function 0x0B (Get Communication Event Counter) the client can obtain a status word as well as an event count of the server's communication event counter. This event counter is incremented once for each successful message completion. Function 0x0C (Get Communication Event Log) returns the same data as function 0x0B, plus the message count (number of messages processed since last restart) and a field of 64 event bytes. These event bytes contain the status result of the last 64 messages that were either sent or received by the server.

With function 0x11 (Report Slave ID), a client may obtain the run status of the server device (run/stop), a device-specific identification byte, and some additional device-specific 249 bytes of data.

The Modbus specification includes one additional function, 0x2B, with two subfunctions. One subfunction (0x0E) allows the client to obtain device identification (*Mandatory*: vendor name, product code, major and minor revision; *Optional*: vendor URL, product name, model name, user application name) all in ASCII string format. The second subfunction may be used to tunnel other communication protocols inside a Modbus connection.

TABLE 36.3 Modbus Function Codes for Diagnostic Purposes

Function Name	Function Code (Hex)	Possible Response Error Codes
Read exception status	0x07	01, 04
Diagnostic	0x08	01, 03, 04
Get communication event counter	0x0B	01, 04
Get communication event log	0x0C	01, 04
Report slave ID	0x11	01, 04

For more details regarding these functions, as well as their corresponding frame formats, the interested reader is once again encouraged to consult the Modbus specification [1].

36.4.3 Device Classes

Modbus server and client devices are classified into classes [3], depending on which of the above functions they implement.

Class 0 includes the minimum useful set of functions:

- Read holding registers (0x03)
- Write multiple registers (0x10)

Class 1 defines a more complete set of the most commonly used functions:

- Read coils (0x01)
- Read discrete inputs (0x02)
- Read input registers (0x04)
- Write single coil (0x05)
- Write single register (0x06)
- Read exception status (0x07)

Class 2 devices support all data transfer functions:

- Write multiple coils (0x0F)
- Read file record (0x14)
- Write file record (0x15)
- Mask write register (0x16)
- Read/write multiple registers (0x17)
- Read FIFO queue (0x18)

36.4.4 Error Handling

Error checking starts as soon as the server receives a request APDU. At this time it will start off by verifying the validity of the function code, address values, and data values (in this order) and, upon the first error encountered, will reply with an exception response APDU with error codes 1, 2, or 3, respectively (Table 36.4). After passing these initial tests, the actions corresponding to the function indicated in the request APDU are executed. If an error occurs during this processing, additional error codes may be generated and sent to the client using the exception response APDUs.

TABLE 36.4 Modbus Exception Codes

Code (Hex)	Name	Comments
0x01	Illegal function	
0x02	Illegal data address	
0x03	Illegal data value	
0x04	Slave device failure	
0x05	Acknowledge	Not commonly used. Indicates that the slave will reply later to the request.
0x06	Slave device busy	
0x08	Memory parity error	Used only in function codes 0x14 and 0x15.
0x0A	Gateway path unavailable	Only used by gateways.
0x0B	Gateway target device failed to respond	Only used by gateways.

36.5 Modbus Serial

A Modbus serial network comprises multiple devices connected to a physical medium (e.g., a serial bus). Due to the shared nature of the physical medium, a mechanism is required to regulate medium access. For this reason, Modbus serial follows the master–slave interaction model (Figure 36.5), with the Modbus client becoming the master, and the Modbus servers taking the role of slaves. The master is responsible for initiating the communication by sending requests to the slaves, one request at a time. These, in turn, reply to the master with the requested data. The request/reply exchange can be performed in one of two ways:

- *Unicast mode*: The master sends a request to a specific slave. The slave processes this request and replies to the master.
- *Broadcast mode*: The master sends a request to all slaves. The slaves process this request, but do not reply to the master.

The master only starts a request/reply exchange once the previous exchange has finished.

Each Modbus serial network may only have one master. The number of slaves is limited to 247. A device can be both master and slave, but not at the same time. Therefore, a device can change its role whenever appropriate. In practice, roles are usually static since the Modbus protocol itself does not specify how the changing of roles may be managed.

36.5.1 Frames

The request/reply exchange is performed using request and reply APDUs, which are transmitted within frames that always have the same structure (Figure 36.6):

- *Address*: This field contains the slave address. In a request frame, it identifies the destination device, while in a reply frame it identifies the sender. Each slave has a unique address in the range 1–247. Address 0 is used (by the master) for broadcasting messages. The range 248–255 is reserved.
- *Modbus APDU*: This field is the application layer PDU (Section 36.4).
- *CRC or LRC*: This field is used for error detection purposes. The content depends on the transmission mode (RTU or ASCII) being used. For details, see Section 36.5.4.

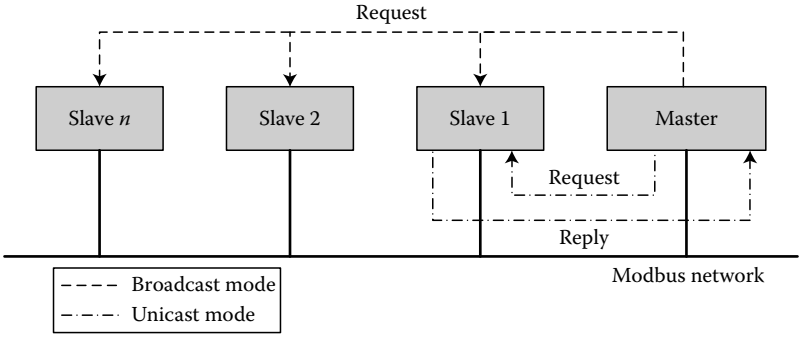

FIGURE 36.5 Request/reply exchange.

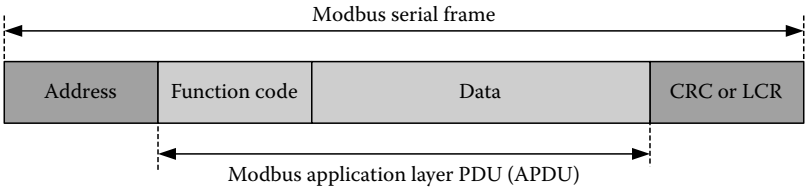

FIGURE 36.6 Modbus serial frame.

In Modbus serial, frames exchanged between master and slave devices can be transmitted in one of two modes: RTU or ASCII. Each mode defines different ways to code the frame contents (i.e., bytes) for transmission. What most distinguishes both modes are the smaller frames produced by the RTU mode. Consequently, for the same baud rate, the RTU mode achieves a higher throughput. There are, however, circumstances that may lead one to choose the ASCII mode. These will be discussed below.

All devices must implement the RTU mode, with the ASCII mode being optional. A device may support both modes, but never simultaneously. Within a network all devices must use the same transmission mode; the communication cannot occur otherwise since both modes are incompatible.

36.5.2 RTU Mode

The RTU mode uses an asynchronous approach for data transmission. Each byte, within a frame, is transmitted using an 11 bit character (Figure 36.7):

- 1 start bit (ST), used for the initial synchronization
- 8 bits, the data coded in binary with the least significant bit sent first
- 1 parity bit (PT), used for error detection
- 1 stop bit (SP), to ensure a minimum idle time between consecutive character transmissions

All devices must support the even parity checking method, but the user may choose any other method (even, odd, or no parity) on devices that support them. However, all devices within the same network must all be configured to use the same method. If the no parity method is used, then the number of stop bits must be 2. This guarantees that 1 byte will always be coded with 11 bits.

The absence of frame delimiters (header or trailer) may lead to synchronization errors during frame reception, i.e., the receiver may have difficulty in identifying the beginning or the end of each frame. To overcome this, it becomes necessary that the time interval between consecutive characters (within the same frame) does not exceed a certain value; otherwise the next character could be easily interpreted by the receiver as the beginning of a new frame. In the same manner, if consecutive frames are separated by a very small time interval, they could be interpreted as a single frame. In both cases, received data will be misinterpreted which will lead to communication errors. To avoid this issue, the transmission of a frame must meet the following timing requirements (Figure 36.8):

- Each frame must be transmitted, one character after the other with a minimum idle interval between them of 1.5 character times.* If this interval is exceeded, the receiver considers that the frame is incomplete and should discard it.
- Consecutive frames must be separated by an idle interval of at least 3.5 character times.

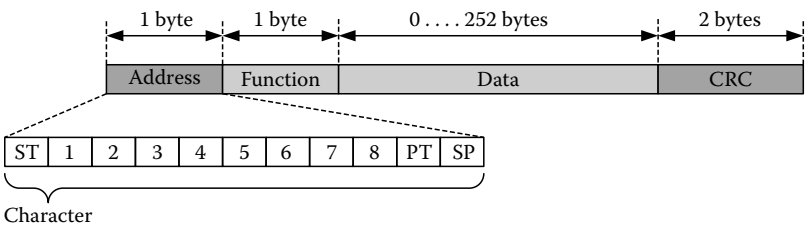

FIGURE 36.7 RTU frame transmission.

* Time necessary for the transmission of 1.5 × 11 bits.

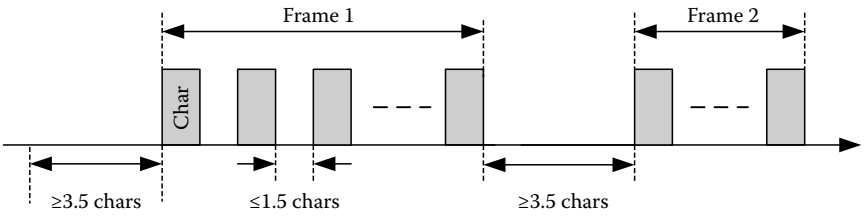

FIGURE 36.8 RTU timing requirements.

From this description, it is easy to understand that some care must be taken when implementing an RTU driver, particularly with regard to the accuracy of the timers used. Although efficient, the RTU mode requires a tight control of timing. For more details on implementation aspects, readers are suggested to consult [4].

36.5.3 ASCII Mode

The ASCII mode also employs an asynchronous transmission. In this case, however, each byte is transmitted as two ASCII characters (Figure 36.9).

The coding process is as follows: each 8 bit byte is divided into two nibbles of 4 bits each. One nibble contains the 4 most significant bits in the byte, and the other the 4 least significant bits. Each nibble is then coded using its hexadecimal representation on the ASCII alphabet. For example, a byte with value 0xA1 (or 161 in decimal) is transmitted as the characters "A" followed by "1." The character "A" is sent as the byte with value 0x41 which is the ASCII representation of the "A" character. Likewise, the character "1" is transmitted as its ASCII value of 0x31.

Each ASCII character is transmitted using 10 bits as follows:

- 1 start bit (ST)
- 7 data bits,* the ASCII character ("0"-"9," "A"-"F") with the least significant bit sent first
- 1 Parity bit (PT)
- 1 Stop bit (SP)

The requirements for parity checking are the same as those of the RTU mode.

Contrary to the RTU mode, a frame transmitted in the ASCII mode has a header and a trailer. The header identifies the beginning of a frame and is represented by the ":" ASCII character (0x3A). The trailer identifies the end of the frame, and comprises two ASCII characters: CR (Carriage Return—0x0D) and LF (Line Feed—0x0A), transmitted in this order.

The use of headers and trailers helps the reception process, since the receiver has only to wait for the ":" character to detect the beginning of a frame. Conversely, the end of a frame is detected when the

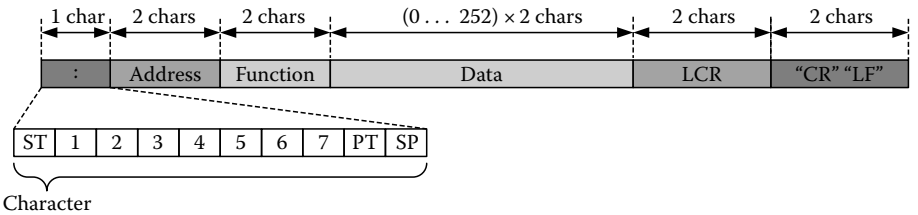

FIGURE 36.9 ASCII frame transmission.

* Each ASCII character is coded using 7 bits.

sequence "CR" "LF" occurs. Since this mode does not specify a maximum time interval between consecutive characters, this implies that this interval could theoretically assume any value. However, this could lead to a situation in which the receiver would block, e.g., when the sender crashes during a frame transmission, whereupon the receiver would wait indefinitely for the remaining characters. In order to prevent these situations, the receiver should implement a timeout mechanism. When a timeout occurs, the received data are discarded and the receiver waits for the next frame. In [4], it is suggested to use 1 s for the timeout value. However, it is possible to use other values depending on the network topology. Further details can be found in [4].

From the discussion, it is easy to understand that the ASCII mode is simpler to implement and does not pose any particular timing requirements. However, the price to pay is the doubling of the frame size when compared to the RTU mode.

36.5.4 Error Detection

Error detection mechanisms are located at two levels, namely, the character level and frame level.

At the character level, a parity checking method is used. The sender computes the parity of each character using the parity method chosen (even, odd, or no parity) and sets the parity bit accordingly. The receiver, using the same method of the sender, computes the expected parity bit, and compares it with the received parity bit. If they are different, an error has occurred and the entire frame is discarded. It is easy to see that the error detection capability of this method is quite limited. For example, when two bits within a character are flipped (due to a transmission error), the error will not be detected.

Error detection at the frame level uses a CRC (cyclic redundancy check) in the RTU mode or LRC (longitudinal redundancy check) in the ASCII mode. Although these methods are conceptually different, the methodology for their use is the same. The sender computes the CRC or LRC value using a predefined algorithm which is applied to the frame contents: Address + APDU fields. This value is then appended to the frame. The receiver computes the CRC or the LRC using the same algorithm, and compares the computed results with the received ones. If they are different, it concludes that the frame has errors and should therefore be discarded. The CRC value is computed through a polynomial division of the frame using $X^{16} + X^{15} + X^2 + 1$ as the generator polynomial. It produces a 16 bit value, which is transmitted with the lower byte first. The LRC value is computed using a modulo-2 sum (XOR-sum) of all bytes of the frame and produces an 8 bit value. A comparison between both methods shows that the CRC has a higher error detection capability. The CRC and LRC values are transmitted using the same methodology used for the remaining characters. Details about CRC or LRC computation can be found in [4].

36.5.5 Physical Layer

Modbus serial supports two physical layers: RS485 [5] and RS232 [6]. The RS485 consists of a multi-drop network able to support multiple devices (typically 32 devices per network segment). Several segments can be connected by using repeaters, thus increasing the maximum number of devices in the network. It supports 1 master (at a time) and up to 247 slaves, and it can be used for larger distances (up to 1000 m) with high baud rates. The RS232 is a point-to-point solution with just 1 master and 1 slave. It is used for short distances (<20 m) with low to medium baud rates.

All devices must support an RS485 interface, while the RS232 is optional. Both transmission modes (RTU and ASCII) can be used on either interface type. Moreover, all devices within a network must have the same configuration, comprising the following parameters: transmission mode (RTU or ASCII), parity type (even, odd, or no parity), and baud rate (bits/s). All devices must support 9.6 and 19.2 kbps rates, with the latter being the default value. Further details about the characteristics of each interface and their application to Modbus networks can be found in [4].

36.6 Modbus TCP

A Modbus TCP network consists of multiple devices connected through a TCP/IP network, interacting following the client–server model. A client sends a request to a server, which in turn responds to the client with the requested data. This transaction (request/response exchange) is performed by sending Modbus TCP frames through a TCP connection previously established between the client and the server. Connection establishment and management are handled by the TCP/IP protocol and occur independently of the Modbus protocol. A Modbus server listens on port 502 for requests from clients that wish to establish a new connection with the server. This port is presently reserved (and registered) for Modbus applications [2].

Some Modbus server devices may support multiple connections from distinct clients simultaneously. Similarly, clients may establish multiple connections to distinct servers. Additionally, the Modbus TCP protocol allows a client to send multiple Modbus requests to the same server over the same connection, even before the arrival of the replies to previous requests. A device may be both a client and server at the same time.

36.6.1 Frames

Modbus TCP frames (request and response) consist of a MBAP header (Modbus Application Protocol header) plus the application layer APDU (Figure 36.10).

The MBAP header comprises several fields:

- *Transaction identifier*: Since a client can issue several concurrent transactions over the same TCP connection, the responses are not guaranteed to arrive in the same order that the requests were sent. It is therefore necessary to have an identifier for each transaction in order to match the request and response frames. The client initializes this field when it performs a request. The server echoes this value in the response frame.
- *Protocol identifier*: This is used to identify the protocol; it currently always has the value 0.
- *Length*: This field indicates the size (in bytes) of the unit identifier field plus the APDU. Data transfer in a TCP connection is performed as a stream of bytes, which could lead to a situation where several frames are waiting to be read in the reception buffer. To identify frame boundaries in these situations, the frame length must be known.
- *Unit identifier*: This field is used to identify the destination device (a slave). It is used mainly by gateways between Modbus/TCP and Modbus serial networks, where the gateway, upon receiving a Modbus/TCP frame, needs to know the identification of the slave on the Modbus serial network that should receive that frame. Modbus/TCP devices that are not gateways usually ignore this field.

Unlike Modbus serial, Modbus TCP frames do not have an error detection field. This was considered unnecessary as the TCP/IP stack already includes several error detection mechanisms. Further details about the implementation of Modbus TCP can be found in [2].

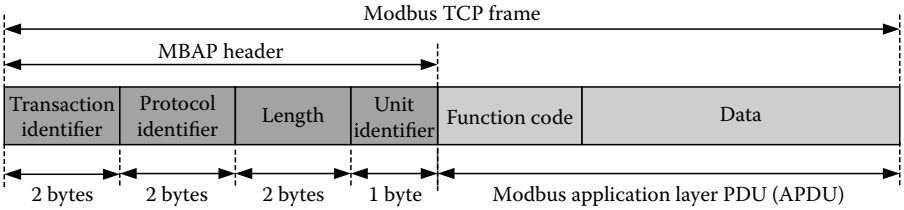

FIGURE 36.10 Modbus TCP frame.

36.7 Example

A typical example of a small-scale project using a Modbus network is the automation of a rope-making machine. This machine consists of two physically separated parts: a braiding component that winds the twines into a rope and a traction component that pulls out the built rope at the correct speed. Each component is driven by an electrical motor connected to a variable speed drive. The braiding component additionally contains several digital and analog sensors to detect possible faults (e.g., twine breaking) and digital and analog actuators (e.g., status signaling lights). The automation logic is handled by a single small PLC, and an HMI graphical terminal is used to interact with the operator (Figure 36.11). A SCADA graphical interface on the plant manager's computer also connects to the PLC over the Modbus TCP network in order to obtain manufacturing data (e.g., up-time, meters of rope produced).

In this project, the PLC communicates with the two variable speed drives over a Modbus serial network, with the PLC acting as the master and the drives acting as slaves. The digital and analog I/Os are handled by an RTU, which, along with the graphical terminal and the SCADA, connects to the PLC over a Modbus TCP network (TCP/IP over Ethernet). The PLC acts as a client to the Modbus server in the RTU and as a server to the Modbus clients in the graphical terminal and SCADA. Note that on the Modbus TCP network: (1) the PLC acts as both a client and a slave simultaneously, (2) three clients coexist on the same network, and (3) the server on the PLC is accessed simultaneously by two clients (the graphical terminal and the SCADA).

In order to show in more detail how the data are actually accessed remotely over the Modbus protocol the connection between the PLC and the RTU will be explained further. A typical example of a Modbus slave device is the RTU of the STB series of products by Schneider Electric. These RTUs are modular, consisting of a network adapter, followed by one or more physical interface modules. The following example (Figure 36.12) considers an RTU with a Modbus/TCP network interface (NIP 2212), followed by the following modules: power supply (PDT 3100), four digital inputs (DDI 3420), four digital outputs (DDO 3410), two analog inputs (AVI 1270), and two analog outputs (AVO 1250).

Although the Modbus protocol defines bit-sized memory areas, this particular RTU does not make use of them; in fact, it only uses the holding registers memory area. It maps the four digital inputs onto the four least significant bits of one 16 bit holding register, the four digital outputs onto the four least significant bits of another register, and each of the analog inputs/outputs onto their own register. Additionally, it uses some registers to store status information.

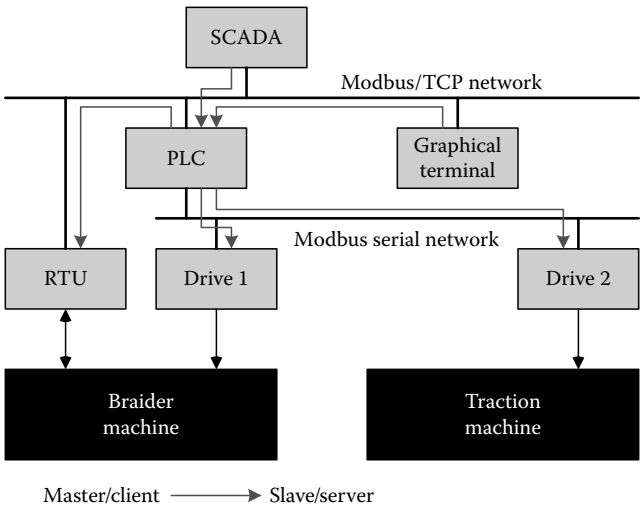

FIGURE 36.11 Example machine and automation equipment.

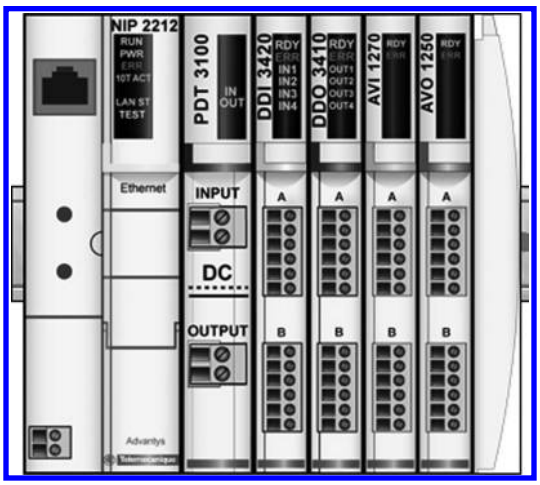

FIGURE 36.12 Physical aspect of RTU with configuration as described in the example.

Figure 36.13 shows a table with the exact mapping of I/Os and status data to the holding register memory area, for an RTU configured as in Figure 36.11. All addresses in this table start with the digit 4, which is the method used by this RTU's supplier to indicate that these refer to registers in the holding register memory area. The remaining digits represent the real register address.

The holding registers used are grouped into two contiguous areas. One area, starting at address 0001, will be used as an output memory area, i.e., the Modbus client may write new data to these registers, which will then be used by the RTU to drive the physical outputs. The data stored in the second contiguous area, starting at address 5392, will be updated by the RTU in such a way as to reflect the actual values in the physical inputs and outputs. Although the Modbus protocol allows a Modbus client to write new data values to

I/O data values

Node Number	Module Name	Input Address	Input Value	Format	Output Address	Output Value	Format
1	STB DDI 3420	45392	0	dec			dec
		45393	0	dec			dec
2	STB DDO 3410	45394	0	dec	40001	0	dec
		45395	0	dec			dec
3	STB AVI 1270	45396	3248	dec			dec
		45397	0	dec			dec
		45398	3264	dec			dec
		45399	0	dec			dec
4	STB AVO 1250	45400	48	dec	40002	0	dec
		45401	48	dec	40003	0	dec
				dec			dec
				dec			dec
				dec			dec
				dec			dec
				dec			dec

FIGURE 36.13 Mapping of I/Os onto Modbus holding registers for the example RTU.

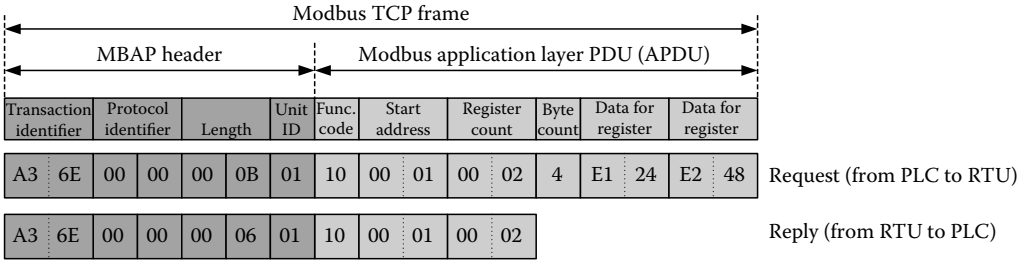

FIGURE 36.14 Example Modbus transaction between the PLC and the RTU.

holding registers, this particular RTU will ignore any attempt to write to registers in this second contiguous area. Registers in this second contiguous area should therefore be considered as being "read only."

As may be seen from this table, the four digital inputs of the first DDI 3420 module are mapped onto the holding register at address 5392. The least significant 4 bits of the following register at address 5393 will contain status information (short-circuit, power failure) related to these same inputs.

A Modbus master changes the outputs on the next module, the DDO 3410 with four digital outputs, by writing the desired state to the holding register at address 0001. This DDO 3410 module uses another two registers in the input area: the first (at address 5394) echoes the value stored in the register 0001, while the following (at address 5395) stores the actual status of the physical outputs.

The AVI 1270 module uses the registers 5396 and 5398 to store the input voltage of each analog input, while the other two registers, at addresses 5397 and 5399, store status information (over and under-voltage errors and warnings). The AVO 1250 uses the registers at addresses 0002 and 0003 to receive the voltage values that should be applied to the analog outputs, and the registers at 5400 and 5401 store status information (over and under-voltage errors).

An example of a possible exchange of Modbus message frames between the PLC and this particular RTU (functioning as Modbus server) is described in Figure 36.14. In this exchange, the PLC uses the Modbus function "Write Multiple Registers" to change the analog output values stored in registers 0002 and 0003 to the values 0xE124 and 0xE248, respectively. Note that the starting address of 0002 is sent on the wire as 0001 since this particular RTU follows the Modbus specification to the letter, and all documentation assumes that the memory areas start at address 1, which is then sent on the wire as 0.

Acronyms

APDU	Application layer protocol data unit
ASCII	American Standard Code for Information Interchange
CRC	Cyclic redundancy check
EIA	Electronics Industries Association
FIFO	First in first out
HMI	Human–machine interface
IP	Internet protocol
LAN	Local area network
LRC	Longitudinal redundancy check
MBAP	Modbus application protocol
PLC	Programmable logic controller
RTU	Remote terminal unit
SCADA	Supervisory control and data acquisition
TCP	Transmission control protocol
TIA	Telecommunications Industry Association

References

1. Modbus Application Protocol Specification, v1.1b, Modbus-IDA, North Grafton, MA, December 28, 2006.
2. Modbus Messaging on TCP/IP Implementation Guide v1.0b, Modbus-IDA, North Grafton, MA, October 24, 2006.
3. Open Modbus/TCP Specification, Andy Swales, Schneider Electric SA, France, March 29, 1999.
4. MODBUS over Serial Line Specification and Implementation Guide V1.02, Modbus-IDA, North Grafton, MA, 2006.
5. Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems, ANSI/TIA/EIA-485-A-1998.
6. Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange, ANSI/TIA/EIA-232-F-1997.