Modbus

Modbus is a serial communications protocol originally published by Modicon (now Schneider Electric) in 1979 for use with its programmable logic controllers (PLCs). Modbus has become a *de facto* standard communication protocol and is now a commonly available means of connecting industrial electronic devices.[1] Modbus is popular in industrial environments because it is openly published and royalty-free. It was developed for industrial applications, is relatively easy to deploy and maintain compared to other standards, and places few restrictions other than the size on the format of the data to be transmitted. The Modbus uses the RS485 as its physical layer. It is possible to use the DC-BUS as power line communication physical layer to save wires.

Modbus enables communication among many devices connected to the same network, for example, a system that measures temperature and humidity and communicates the results to a computer. Modbus is often used to connect a supervisory computer with a remote terminal unit (RTU) in supervisory control and data acquisition (SCADA) systems. Many of the data types are named from industry usage of Ladder logic and its use in driving relays: a single-bit physical output is called a *coil*, and a single-bit physical input is called a *discrete input* or a *contact*.

The development and update of Modbus protocols have been managed by the Modbus Organization[2] since April 2004, when Schneider Electric transferred rights to that organization.[3] The Modbus Organization is an association of users and suppliers of Modbus-compliant devices that advocates for the continued use of the technology.[4]

Contents

Limitations

- Since Modbus was designed in the late 1970s to communicate to programmable logic controllers, the number of data types is limited to those understood by PLCs at the time. Large binary objects are not supported.
- No standard way exists for a node to find the description of a data object, for example, to determine whether a register value represents a temperature between 30 and 175 degrees.
- Since Modbus is a master/slave protocol, there is no way for a field device to "report an exception" (except over Ethernet TCP/IP, called open-mbus) – the master node must routinely poll each field device and look for changes in the data. This consumes bandwidth and network time in applications where bandwidth may be expensive, such as over a low-bit-rate radio link.
- Modbus is restricted to addressing 254 devices on one data link, which limits the number of field devices that may be connected to a master station (once again, Ethernet TCP/IP is an exception).
- Modbus transmissions must be contiguous, which limits the types of remote communications devices to those that can buffer data to avoid gaps in the transmission.
- Modbus protocol itself provides no security against unauthorized commands or interception of data.[5]

Modbus object types

The following is a table of object types provided by a Modbus slave device to a Modbus master device:

Object type	Access	Size	Address Space
Coil	Read-write	1 bit	00001 - 09999
Discrete input	Read-only	1 bit	10001 - 19999
Input register	Read-only	16 bits	30001 - 39999

Holding register Read-write 16 bits 40001 - 49999

Protocol versions

Versions of the Modbus protocol exist for serial port and for Ethernet and other protocols that support the Internet protocol suite. There are many variants of Modbus protocols:

- *Modbus RTU* — This is used in serial communication and makes use of a compact, binary representation of the data for protocol communication. The RTU format follows the commands/data with a cyclic redundancy check checksum as an error check mechanism to ensure the reliability of data. Modbus RTU is the most common implementation available for Modbus. A Modbus RTU message must be transmitted continuously without inter-character hesitations. Modbus messages are framed (separated) by idle (silent) periods.
- *Modbus ASCII* — This is used in serial communication and makes use of ASCII characters for protocol communication. The ASCII format uses a longitudinal redundancy check checksum. Modbus ASCII messages are framed by leading colon (":") and trailing newline (CR/LF).
- *Modbus TCP/IP or Modbus TCP* — This is a Modbus variant used for communications over TCP/IP networks, connecting over port 502.[6] It does not require a checksum calculation, as lower layers already provide checksum protection.
- *Modbus over TCP/IP or Modbus over TCP or Modbus RTU/IP* — This is a Modbus variant that differs from Modbus TCP in that a checksum is included in the payload as with Modbus RTU.
- *Modbus over UDP* — Some have experimented with using Modbus over UDP on IP networks, which removes the overheads required for TCP.[7]
- *Modbus Plus (Modbus+, MB+ or MBP)* — Modbus Plus is proprietary to Schneider Electric and unlike the other variants, it supports peer-to-peer communications between multiple masters.[8] It requires a dedicated co-processor to handle fast HDLC-like token rotation. It uses twisted pair at 1 Mbit/s and includes transformer isolation at each node, which makes it transition/edge-triggered instead of voltage/level-triggered. Special hardware is required to connect Modbus Plus to a computer, typically a card made for the ISA, PCI or PCMCIA bus.
- *Pemex Modbus* — This is an extension of standard Modbus with support for historical and flow data. It was designed for the Pemex oil and gas company for use in process control and never gained widespread adoption.
- *Enron Modbus* — This is another extension of standard Modbus developed by Enron Corporation with support for 32-bit integer and floating-point variables and historical and flow data. Data types are mapped using standard addresses.[9] The historical data serves to meet an American Petroleum Institute (API) industry standard for how data should be stored.[*citation needed*]

Data model and function calls are identical for the first 4 variants of protocols; only the encapsulation is different. However the variants are not interoperable, nor are the frame formats.

Communications and devices

Each device communicating (transferring data) on a Modbus is given a unique address.

On Modbus RTU, Mobus ASCII and Modbus Plus which are all Rs-485 single cable multi-drop networks, only the node assigned as the Master may initiate a command. All other devices are slaves and respond to requests and commands.

For the protocols using Ethernet such as Modbus TCP, any device can send out a Modbus command thus all can act as a Masters, although normally, only one device acts as a Master.

There are many modems and gateways that support Modbus, as it is a very simple and often copied protocol. Some of them were specifically designed for this protocol. Different implementations use wireline, wireless communication, such as in the ISM band, and even Short Message Service (SMS) or General Packet Radio Service (GPRS). One of the more common designs of wireless networks makes use of mesh networking. Typical problems that designers have to overcome include high latency and timing issues.

Commands

Modbus commands can instruct a Modbus Device to:

- change the value in one of its registers, that is written to Coil and Holding registers.
- read an I/O port: Read data from a Discrete and Coil ports,
- command the device to send back one or more values contained in its Coil and Holding registers.

A Modbus command contains the Modbus address of the device it is intended for (1 to 247). Only the addressed device will respond and act on the command, even though other devices might receive it (an exception is specific broadcastable commands sent to node 0, which are acted on but not acknowledged).

All Modbus commands contain checksum information to allow the recipient to detect transmission errors.

Frame formats

A Modbus "frame" consists of an Application Data Unit (ADU)Cite error: There are `<ref>` tags on this page without content in them (see the help page). , which encapsulates a Protocol Data Unit (PDU):[10]

- ADU = Address + PDU + Error check,
- PDU = Function code + Data.

The byte order for values in Modbus data frames is most significant byte of a multi-byte value is sent before the others. All Modbus variants use one of the following frame formats.[1]

Modbus RTU frame format (primarily used on asynchronous serial data lines like RS-485/EIA-485)

Name	Length (bits)	Function
Start	28	At least 3½ character times of silence (mark condition)
Address	8	Station address
Function	8	Indicates the function code; e.g., read coils/holding registers
Data	$n \times 8$	Data + length will be filled depending on the message type
CRC	16	Cyclic redundancy check
End	28	At least 3½ character times of silence between frames

Note about the CRC:

- Polynomial: $x^{16} + x^{15} + x^{2} + 1$ (CRC-16-ANSI also known as CRC-16-IBM, normal hexadecimal algebraic polynomial being `8005` and reversed `A001`).
- Initial value: 65,535.
- Example of frame in hexadecimal: `01 04 02 FF FF B8 80` (CRC-16-ANSI calculation from `01` to `FF` gives `80B8`, which is transmitted **least** significant byte **first**).

Modbus ASCII frame format (primarily used on 7- or 8-bit asynchronous serial lines)

Name	Length (bytes)	Function
Start	1	Starts with colon `:` (ASCII hex value is `3A`)
Address	2	Station address
Function	2	Indicates the function codes like read coils / inputs
Data	$n \times 2$	Data + length will be filled depending on the message type
LRC	2	Checksum (Longitudinal redundancy check)
End	2	Carriage return – line feed (CR/LF) pair (ASCII values of `0D`, `0A`)

Address, function, data, and LRC are all capital hexadecimal readable pairs of characters representing 8-bit values (0–255). For example, 122 (7 × 16 + 10) will be represented as `7A`.

LRC is calculated as the sum of 8-bit values, negated (two's complement) and encoded as an 8-bit value. Example: if address, function, and data encode as 247, 3, 19, 137, 0, and 10, their sum is 416. Two's complement (−416) trimmed to 8 bits is 96 (e.g. 256 × 2 − 416), which will be represented as `60` in hexadecimal. Hence the following frame: `:F7031389000A60<CR><LF>`.

Modbus TCP frame format (primarily used on Ethernet networks)

Name	Length (bytes)	Function
Transaction identifier	2	For synchronization between messages of server and client
Protocol identifier	2	0 for Modbus/TCP
Length field	2	Number of remaining bytes in this frame
Unit identifier	1	Slave address (255 if not used)
Function code	1	Function codes as in other variants
Data bytes	*n*	Data as response or commands

Unit identifier is used with Modbus/TCP devices that are composites of several Modbus devices, e.g. on Modbus/TCP to Modbus RTU gateways. In such case, the unit identifier tells the Slave Address of the device behind the gateway. Natively Modbus/TCP-capable devices usually ignore the Unit Identifier.

Available function/command codes

The various reading, writing and other operations are categorized as follows.[11] The most primitive reads and writes are shown in bold. A number of sources use alternative terminology, for example *Force Single Coil* where the standard uses *Write Single Coil*.[12]
Prominent entities within a Modbus slave are:

- Coils: readable and writable, 1 bit (off/on)
- Discrete Inputs: readable, 1 bit (off/on)
- Input Registers: readable, 16 bits (0 to 65,535), essentially measurements and statuses
- Holding Registers: readable and writable, 16 bits (0 to 65,535), essentially configuration values

Modbus function codes

Function type			**Function name**	**Function code**	**Comment**
Data Access	Bit access	Physical Discrete Inputs	**Read Discrete Inputs**	2	
		Internal Bits or Physical Coils	**Read Coils**	1	
			Write Single Coil	5	
			Write Multiple Coils	15	
	16-bit access	Physical Input Registers	**Read Input Registers**	4	
		Internal Registers or Physical Output Registers	**Read Multiple Holding Registers**	3	
			Write Single Holding Register	6	
			Write Multiple Holding Registers	16	
			Read/Write Multiple Registers	23	
			Mask Write Register	22	
			Read FIFO Queue	24	
	File Record Access		Read File Record	20	
			Write File Record	21	
Diagnostics			Read Exception Status	7	serial only
			Diagnostic	8	serial only
			Get Com Event Counter	11	serial only
			Get Com Event Log	12	serial only
			Report Slave ID	17	serial only
			Read Device Identification	43	
Other			Encapsulated Interface Transport	43	

Format of data of requests and responses for main function codes

Requests and responses follow frame formats described above. This section gives details of data formats of most used function codes.

Function code 1 (read coils) and function code 2 (read discrete inputs)

Request:

- Address of first coil/discrete input to read (16-bit)
- Number of coils/discrete inputs to read (16-bit)

Normal response:

- Number of bytes of coil/discrete input values to follow (8-bit)
- Coil/discrete input values (8 coils/discrete inputs per byte)

Value of each coil/discrete input is binary (0 for off, 1 for on). First requested coil/discrete input is stored as least significant bit of first byte in reply.
If number of coils/discrete inputs is not a multiple of 8, most significant bit(s) of last byte will be stuffed with zeros.
For example, if eleven coils are requested, two bytes of values are needed. Suppose states of those successive coils are *on, off, on, off, off, on, on, on, off, on, on*, then the response will be `02 E5 06` in hexadecimal.

Because the byte count returned in the reply message is only 8 bits wide and the protocol overhead is 5 bytes, a maximum of 2008 (251 x 8) discrete inputs or coils can be read at once.

Function code 5 (force/write single coil)

Request:

- Address of coil (16-bit)
- Value to force/write: 0 for off and 65,280 (FF00 in hexadecimal) for on

Normal response: same as request.

Function code 15 (force/write multiple coils)

Request:

- Address of first coil to force/write (16-bit)
- Number of coils to force/write (16-bit)
- Number of bytes of coil values to follow (8-bit)
- Coil values (8 coil values per byte)

Value of each coil is binary (0 for off, 1 for on). First requested coil is stored as least significant bit of first byte in request.
If number of coils is not a multiple of 8, most significant bit(s) of last byte should be stuffed with zeros. See example for function codes 1 and 2.

Normal response:

- Address of first coil (16-bit)
- number of coils (16-bit)

Function code 4 (read input registers) and function code 3 (read holding registers)

Request:

- Address of first register to read (16-bit)
- Number of registers to read (16-bit)

Normal response:

- Number of bytes of register values to follow (8-bit)
- Register values (16 bits per register)

Because the number of bytes for register values is 8-bit wide and maximum modbus message size is 256 bytes, only 125 registers for Modbus RTU and 123 registers for Modbus TCP can be read at once.[13]

Function code 6 (preset/write single holding register)

Request:

- Address of holding register to preset/write (16-bit)
- New value of the holding register (16-bit)

Normal response: same as request.

Function code 16 (preset/write multiple holding registers)

Request:

- Address of first holding register to preset/write (16-bit)
- Number of holding registers to preset/write (16-bit)
- Number of bytes of register values to follow (8-bit)
- New values of holding registers (16 bits per register)

Because register values are 2-bytes wide and only 127 bytes worth of values can be sent, only 63 holding registers can be preset/written at once.

Normal response:

- Address of first preset/written holding register (16-bit)
- Number of preset/written holding registers (16-bit)

Exception responses

For a normal response, slave repeats the function code. Should a slave want to report an error, it will reply with the requested function code plus 128 (hex `0x80`) (3 becomes 131 = hex `0x83`), and will only include one byte of data, known as the *exception code*.

Main Modbus exception codes

Code	Text	Details
1	Illegal Function	Function code received in the query is not recognized or allowed by slave
2	Illegal Data Address	Data address of some or all the required entities are not allowed or do not exist in slave
3	Illegal Data Value	Value is not accepted by slave
4	Slave Device Failure	Unrecoverable error occurred while slave was attempting to perform requested action
5	Acknowledge	Slave has accepted 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. Master can next issue a *Poll Program Complete* message to determine whether processing is completed
6	Slave Device Busy	Slave is engaged in processing a long-duration command. Master should retry later
7	Negative Acknowledge	Slave cannot perform the programming functions. Master should request diagnostic or error information from slave
8	Memory Parity Error	Slave detected a parity error in memory. Master can retry the request, but service may be required on the slave device
10	Gateway Path Unavailable	Specialized for Modbus gateways. Indicates a misconfigured gateway
11	Gateway Target Device Failed to Respond	Specialized for Modbus gateways. Sent when slave fails to respond

Coil, discrete input, input register, holding register numbers and addresses

Some conventions govern how access to Modbus entities (coils, discrete inputs, input registers, holding registers) are referenced.

It is important to make a distinction between entity *number* and entity *address*:

- Entity *numbers* combine entity type and entity location within their description table.
- Entity *address* is the starting address, a 16-bit value in the data part of the Modbus frame. As such its range goes from 0 to 65,535

In the traditional standard, *numbers* for those entities start with a digit, followed by a number of 4 digits in the range 1–9,999:

- coils *numbers* start with **0** and span from **0**0001 to **0**9999,
- discrete input *numbers* start with **1** and span from **1**0001 to **1**9999,
- input register *numbers* start with **3** and span from **3**0001 to **3**9999,
- holding register *numbers* start with **4** and span from **4**0001 to **4**9999.

This translates into *addresses* between 0 and 9,998 in data frames. For example, in order to read holding registers starting at *number* 40001, corresponding *address* in the data frame will be 0 with a function code of 3 (as seen above). For holding registers starting at *number* 40100, *address* will be 99. Etc.

This limits the number of *addresses* to 9,999 for each entity. A *de facto* referencing extends this to the maximum of 65,536.[14] It simply consists of adding one digit to the previous list:

- coil *numbers* span from **0**00001 to **0**65536,
- discrete input *numbers* span from **1**00001 to **1**65536,
- input register *numbers* span from **3**00001 to **3**65536,
- holding register *numbers* span from **4**00001 to **4**65536.

When using the extended referencing, all *number* references must have exactly 6 digits. This avoids confusion between coils and other entities. For example, to know the difference between holding register #40001 and coil #40001, if coil #40001 is the target, it must appear as #040001.

JBUS mapping

Another *de facto* protocol closely related to Modbus appeared after it, and was defined by PLC brand April Automates, the result of a collaborative effort between French companies Renault Automation and Merlin Gerin et Cie in 1985: **JBUS**. Differences between Modbus and JBUS at that time (number of entities, slave stations) are now irrelevant as this protocol almost disappeared with the April PLC series which AEG Schneider Automation bought in 1994 and then made obsolete. However the name JBUS has survived to some extent.

JBUS supports function codes 1, 2, 3, 4, 5, 6, 15, and 16 and thus all the entities described above. However numbering is different with JBUS:

- Number and address coincide: entity #*x* has address *x* in the data frame.
- Consequently, entity number does not include the entity type. For example, holding register #40010 in Modbus will be holding register #9, located at address 9 in JBUS.
- Number 0 (and thus address 0) is not supported. Slave should not implement any real data at this number and address and it can return a null value or throw an error when requested.

Implementations

Almost all implementations have variations from the official standard. Different varieties might not communicate correctly between equipment of different suppliers. Some of the most common variations are:

- Data types
    - IEEE floating-point number
    - 32-bit integer
    - 8-bit data
    - Mixed data types
    - Bit fields in integers
    - Multipliers to change data to/from integer. 10, 100, 1000, 256 ...
- Protocol extensions
    - 16-bit slave addresses
    - 32-bit data size (1 address = 32 bits of data returned)
    - Word-swapped data

Trade group

Modbus Organization, Inc. is a trade association for the promotion and development of Modbus protocol.[2]

Modbus Plus

Despite the name, Modbus Plus[15] is not a variant of Modbus. It is a different protocol, involving token passing.

It is a proprietary specification of Schneider Electric, though it is unpublished rather than patented. It is normally implemented using a custom chipset available only to partners of Schneider.

See also

- CAN Bus