MODBUS

THE MANUAL

September 2018

About the author

Author Rob Hulsebos has been involved with industrial networks since their beginning in 1993. He studied Computer Science with a specialization in data communication. Working as a software-engineer for a PLC vendor, he made implementations for Profibus, AS-Interface, Bitbus, Ethernet, TCP/IP, various proprietary protocols, and several Modbus versions.

As of 1998 he has been active as a teacher for Mikrocentrum (Eindhoven, The Netherlands), providing courses on the basics of industrial networks, Profibus and industrial Ethernet, for more than 3500 students. He also publishes about ongoing developments in industrial networking for the Dutch trade press, and has written several books about this subject.

In 2010, Rob found the missing link during the reverse engineering of the Stuxnet virus, after which the operation of the virus could be explained. Since then, Rob is active in the field of industrial cybersecurity, where his experience in software-development and the implementations of protocol stacks helps to improve industrial products.

But as of 2018 Rob is still using Modbus! In recent years he made two now implementations of Modbus for an industrial controls vendor. Customers encountering Modbus for the first time still face the same issues as twenty years ago: lack of documentation. For unknown reasons no-one ever wrote a book about Modbus, despite its popularity. This book is the first attempt to fill that void, where all the theory of Modbus and Rob's personal experience are put together. Undoubtedly there are many improvements possible to this edition; the author welcomes your comments and ideas about improvements!

Contact the author at:

Het Kempke 8, 5672 PL Nuenen, The Netherlands

-or-

r dot hulsebos at onsnet dot nu

CHAPTER 1. PAST AND FUTURE

1. The history

The original development of Modbus was done by Modicon in 1979 for use with its own PLC's. Later Modicon was acquired by AEG which itself was acquired by Schneider, which is still considered the 'owner' of Modbus, even though the intellectual rights were transferred to the Modbus Organization (www.modbus.org). Around 2000, with the start of industrial Ethernet, Schneider launched Modbus/TCP which became very popular due to the absence of serious competitors, which were in development until ca. 2005. Even then, protocols like Ethernet/IP and ProfiNet still had to develop their market share, so Modbus/TCP was the most popular industrial Ethernet protocol until ca. 2012. In 2018 it was still the #4 of all industrial Ethernet protocols.

Around 2002, Schneider found Modbus/TCP to be technically obsolete. It decided to develop a modern industrial Ethernet protocol in cooperation with the German company Jetter. Later they christened the new protocol "IDA" and formed a trade association also called IDA, attracting other vendors (such as Phoenix Contact), with the idea of creating an "open" protocol specification, to compete with ProfiNet. A first version of the IDA-protocol was released in 2002. When the IDA association members could find no common ground on how to develop IDA further, both Schneider and Phoenix left IDA. This left both companies without any modern industrial Ethernet technology. Phoenix subsequently joined ProfiNet (Siemens), and Schneider joined Ethernet/IP (Rockwell, Allen/Bradley). It was then decided to integrate Modbus/TCP into the Ethernet/IP specification, allowing it to run concurrently with Ethernet/IP.

IDA continued for a few years, but showed no results. It merged with the Modbus User's Organization, to become the Modbus/IDA group. After a few years, the IDA developments were silently stopped and files and documents removed from the website.

On the 12th of April 2002 Schneider transferred its "right, title and interest' in the protocol copyright on Modbus to the Modbus/IDA group, a non-profit organization formed in 2002 to advance the use of Modbus in the world.

Despite the advances in other industrial Ethernet protocols, Modbus/TCP stood its ground. Due to Schneider moving on to Ethernet/IP, it was expected that Modbus/TCP would not be developed any further. But due to weaknesses in the protocol, Modbus/TCP devices became the #1 target for hackers searching the Internet for industrial equipment to hack. Many studies were made about a "Secure Modbus", but nothing much happened. Then in 2015 Schneider released the M580 controller with a version of Modbus called "Secure Modbus/TCP". It uses the "IPSec" (IP Secure) protocol, giving authentication of devices and detection of rogue network messages, without sacrificing speed. In August 2018, there was the surprising announcement of the release of the specification for "Modbus/TCP Security" [MBUSSECURE].

2. Where do we find Modbus?

Modbus is found in lots of equipment, ranging from very simple embedded devices to PLC's and (industrial) controllers and SCADA products. Due to the simplicity of Modbus and the low costs for electronic parts it is a good fit for embedded devices. In many SCADA systems the Modbus implementation is often available for free, making it the first choice for smaller industrial systems. Traditionally this is almost always the Modbus/ASCII or /RTU version, as it can be run over simple serial ports. For embedded devices Modbus/TCP is also more and more seen, because Ethernet-implementations have become very cheap.

Modbus is also available in the (industrial Ethernet protocol) called Ethernet/IP, as Schneider has chosen this protocol as its standard "Industrial Ethernet" protocol. As a consequence, to help existing customers to migrate from Modbus to Ethernet/IP, the Ethernet/IP system was extended to also officially support Modbus (also eased by Schneider becoming a member in the board of directors of the Ethernet/IP user group).

Modbus/TCP is also often found in devices supporting TCP/IP. When TCP/IP is available, an implementation of Modbus/TCP can be easily added. Since many industrial Ethernet protocols (i.e. ProfiNet) support TCP/IP for all non-real-time tasks, one can use Modbus/TCP next to the real-time protocol.

Finally, we see Modbus/TCP appear in cybersecurity devices called "intrusion detection systems", "firewalls" or "secure router". This is because many Modbus devices are not properly protected against incorrect network messages, messages from other devices, messages with unusual commands, etc. The protection devices scan all Modbus traffic, and reject all Modbus messages not fitting certain criteria. It is not that other industrial protocols are invulnerable, only that Modbus/TCP is very popular and it easy to implement.

3. The specification

The Modbus User's Group (www.modbus.org) is the entry point for all specifications about Modbus. On this website, the specification documents are freely available. The current (2018) version number is 1.1b3, which was last changed in 2012. In comparison with the previous version, not much was changed, except now it has become a "client/server" protocol instead of a "master/slave" protocol. Technically this does not make any difference, and the Modbus protocol itself also hasn't changed. Furthermore the specification was written to be more conformant to "standardese" jargon and abbreviations, which makes it more difficult to read.

The reason for changing the terminology is unknown; it may have to do with historical reasons, or with marketing reasons, as client/server sounds more sophisticated. Anyway, the change will bring confusion[1] to a market which for more than 30 years has learned to live with the master/slave terminology.

[1] *Personally, I find the effort could have better been spent in changing the ancient Modicon PLC terminology, like "coils". Who knows what a coil does nowadays in an industrial controller?*

Before the Modbus User's Group existed, Schneider was the owner of the specification. The original specification document is called "PI_MBUS-300", and can still be found on internet in many places. For reference purposes, it is also available from www.modbus.org. Although outdated, the document is surprisingly easy to read, so may be a good starting point for learning about the Modbus/ASCII and RTU protocols.

After the introduction of Modbus/TCP, the Modbus User's Group has rewritten the specification, so the description of the function codes is now separate from the physical layer specification. Also, the specification has been updated with more examples and flowcharts, to remove ambiguity on how to process incoming network messages.

Be aware that many Modbus implementations do not follow the specifications to the letter, especially the error handling.

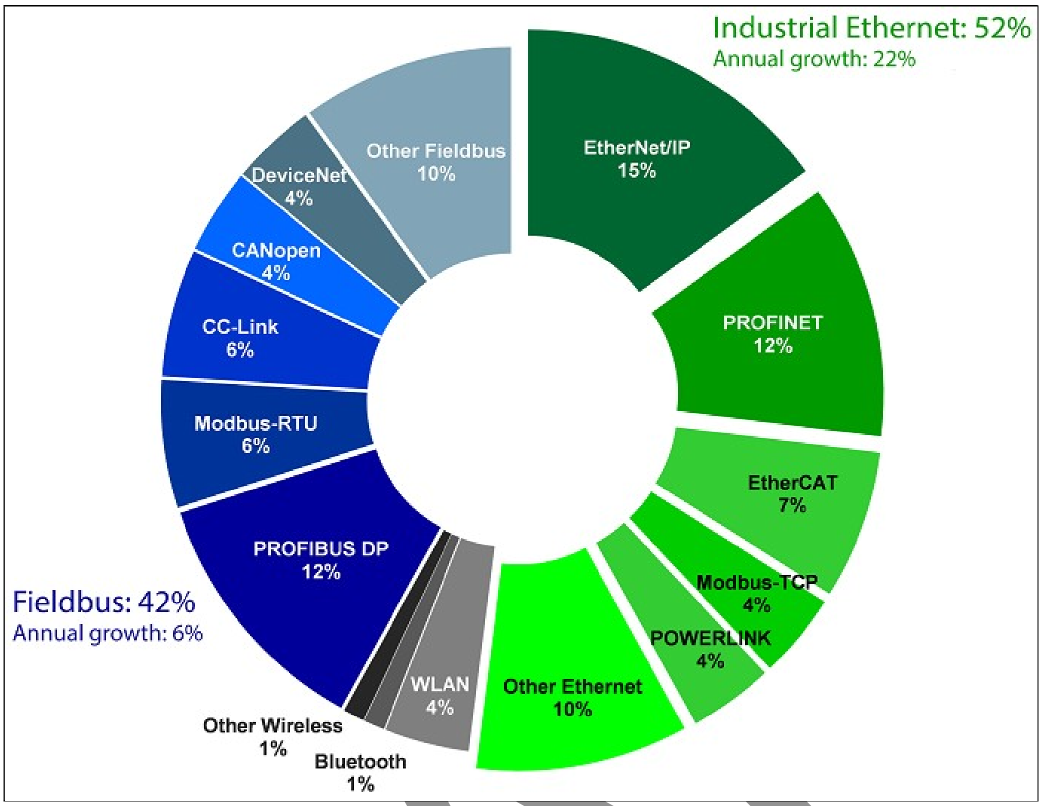
4. The market position

The industrial network market consists of several hundreds of protocols, many small contenders, and a few with large market shares. We see this in every application area: machinery, discrete automation, process installations, building automation, automotive, embedded systems, etc. Historically, Modbus has a PLC-background, so it is not a surprise that it is still used in that market. But Modbus is also strong in process automation applications and in building automation. Surprisingly it is also popular in small embedded systems, due to its simplicity and low cost.

Around the year 2000, the migration to Ethernet-based industrial network protocols started, creating "industrial Ethernet". Modbus/TCP was on the market very early, and this helped to boost its market share. Implementations for Modbus/TCP were up-and-running, while many other protocols existed only on paper. For many years, Modbus/TCP was the most-popular industrial Ethernet protocol.

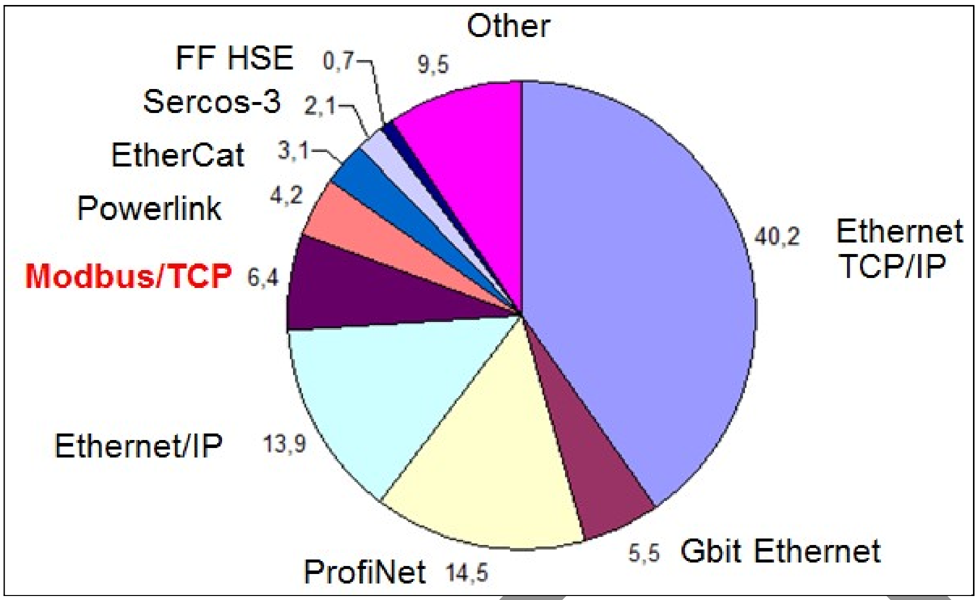
This changed slowly after 2005, but today (2016) Modbus is still the #3 most popular industrial Ethernet protocol, after ProfiNet and Ethernet/IP. It is difficult to find good market statistics, as this depends on whom is asked (vendors, customers), in which continent (America's, Europe or Asia), and which market is researched. Also, many questionnaires fail to recognize the distinction between (for example) Modbus, Modbus/TCP, TCP/IP and Ethernet: if you are a Modbus/TCP user on Ethernet, which technology should you tick in a questionnaire? And what if you use Modbus/TCP in Ethernet/IP?

The Swedish company HMS publishes a yearly (since 2015) overview of the market shares of industrial network protocols, based on sales of their own products. In 2015 the serial Modbus was the 2nd most popular fieldbus protocol (7% market share) and Modbus/TCP the 4th Ethernet protocol (4%). In 2017, the market shares hadn't changed much (see figure).



The products of HMS do not cover the complete industrial network market, i.e. no protocols listed typically used in process automation or building automation. Mainly, protocols that are used in machinery automation and embedded systems are listed. These are application areas where Modbus is not very popular; nevertheless Modbus/TCP is still listed as the #4 protocol of industrial Ethernet, and #2 of the first generation industrial networks. Probably the figures would change considerably if protocols from other parts of industry would be counted in.

Market researcher IMS published a study about the industrial Ethernet market shares in 2011 and 2015. Below are their projections for 2015, which have not changed much from the figures for 2011. Note that these diagrams show that Modbus/TCP's market-share is in decline against the other industrial Ethernet protocols *used for high-speed control*, which Modbus/TCP is not meant for.



5. Where can I use Modbus?

Modbus is a very general protocol, allowing for use in many sorts of application. I have seen it in machinery automation, process control systems, building automation, test systems, ships, press brakes, safety controllers, etc. After all, "a bit is a bit" and the network protocol doesn't much care what the bit is used for.

There is one exception to this: Modbus cannot (may not!) be used for safety applications, for example a PLC reading/setting safety I/O signals, such as for emergency stops and light curtains. Why not? The protocol is not "safe" enough; it cannot guarantee that the data sent and received is of the quality necessitated by the legal requirements for safety systems. It also cannot guarantee that the safety system's components will respond in time.

Now, Modbus is not unique – it turns out that most modern industrial network protocols cannot be used in safety applications (for the same reasons). In order to allow this, an additional layer of software[2] is needed to add the extra functionality. These have been written for various industrial network protocols, giving us ProfiSafe (=on top of Profibus/DP), CIP Safety (on top of CIP), Ethercat Safety, and many more. Unfortunately there is *no* Modbus/Safety!

6. The future of Modbus

Even though Modbus is approaching its 40th birthday, it is still "alive and kicking". Despite its simplicity (as compared to modern industrial network protocols), or perhaps because of its

[2] *This publication is not the right place to describe what this extra layer must do, but a book has been written about safety networks: Reinert, "Sichere Bussysteme für die Automation" Hüthig Verlag 2001, ISBN 3-7785-2797-5 (yes, it is in German).*

simplicity, it is still very popular. It is still being implemented in new devices, due to its low cost requirements for electronics and software.

Despite this popularity, the Modbus User's Group is not active in designing new extensions to Modbus. Modbus/TCP was the last major innovation (and a very successful one at that!).

Surprisingly, Schneider Electronics launched the "M580 EPAC" (Ethernet Programmable Automation Controller) with Modbus/TCP allowing the use of the "IPSec" (IP Secure). IPSec adds a level of safety to a network: confidentiality (by encrypting data), integrity (no modification to data), authentication (knowing that you are communicating with the right party), and anonymity (not knowing with whom you are communicating). The M580 does not support all of IPSec; the data part of a TCP/IP network message is not encrypted. TCP/IP's administrative fields in a network message are encrypted. This still prevents modification of network messages, and the insertion of false messages in a TCP/IP data stream. So a hacker can still listen in on the data being transmitted, but he cannot influence the communication in Modbus/TCP. The advantage of not encrypting all data is that the speed of Modbus/TCP is not decreased.

According to a publication of security-company DigitalBond it became known that Schneider was working on an implementation of "Secure Modbus/TCP" (or Modbus/TLS). A presentation on Youtube (https://www.youtube.com/watch?v=kgvFWYv7Wwk) given during the S4x17 security conference (January 2017) gives more details about ongoing developments. Finally, in August 2018 the specification became publicly available, see [MBUSSECURE].

7. Usage in industrial Ethernet protocols

Modbus/TCP can also be used within ProfiNet and Beckhoff's Ethercat and any other industrial Ethernet protocol with support for TCP/IP. All these protocols have a method of guaranteeing their real-time behaviour, and they allow non-real time traffic using TCP/IP if spare bandwidth is available.

8. Literature

[MBUS300]	PI_MBUS_300.PDF June 1996	Version J of the "Modicon Modbus Protocol Reference Guide" was *the* Modbus specification for several decades. It is still being referenced by many vendors. The document is no longer formally valid; it has been split up in two separate documents (see below), describing the physical layer and application layer.
[MBUSSERIAL]	Modbus_over_serial_line_V1.02.pdf December 2006	A formalized description of the serial line interface, valid for Modbus/ASCII and Modbus/RTU.
[MBUSAPPL]	Modbus_Application_Protocol_V1_1b3.pdf	The description of all the commands

		(function codes), valid for Modbus/ASCII, Modbus/RTU and Modbus/TCP.
[MBUSSECURE]	Modbus/TCP Security MB-TCP-Security-v21_2018-07-24	The specification of the Modbus/TCP Security protocol.

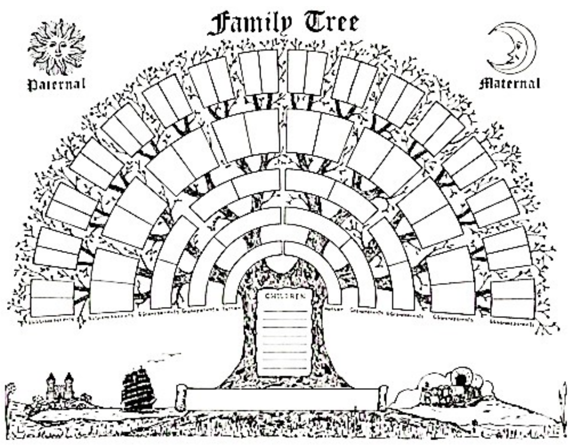
All documents can be retrieved directly from the website www.modbus.org. Older versions can be found everywhere on internet.

CHAPTER 2. VERSIONS

1. The family

Modbus is not a single protocol, but a whole family, developed during three decades. Some of them are still used, others have already disappeared. Here's an overview (not chronological):

Modbus/ASCII	The first version, where the messages are sent as readable text (hence the "ASCII"). Occasionally it is still used.
Modbus/RTU	The successor of Modbus/ASCII, about twice as fast due to half the overhead. This is still a very popular version for use on RS232 or RS485 networks.
Modbus/TCP	The "industrial Ethernet" version of Modbus/RTU, running on top of TCP and therefor usable on almost anything that has a TCP/IP interface. For several years Modbus/TCP was the most popular industrial Ethernet protocol, and is today (2016) still #3!
Modbus/UDP	A variant of Modbus/TCP, using UDP (User Datagram Protocol) instead of TCP. This is much faster than TCP, allows for sending broadcast messages, but is less reliable than TCP. Some vendors claim to support Modbus/UDP, but there is no official specification for this protocol, so it is very likely that devices from different vendors cannot operate with each other as they will have completely different implementations.
Modbus/SFB	(Sequential Frequency Band) is a very special version of Modbus, as it is an implementation of Modbus on top of another protocol: Intel's "Bitbus". Twenty years ago this was a very fast technology (375 Kbit/s). SFB is hardly seen in practice, and Intel has stopped with supporting Bitbus in its processors.
Modbus/1	An alternative name for Modbus/ASCII or /RTU. It came into being when Modbus/2 was developed, and the need arose to distinguish the 'older' versions. But the name never caught on.
Modbus/2 (also called Modbus-II)	Originally thought as the successor of Modbus/1, this version was hardly ever used due to its difficult cabling methodology (RG6 coaxial cable). It has now disappeared from the market.



Modbus/+	The “ModbusPlus” (sometimes called: MBP or MB+) protocol uses a token-passing protocol on an RS485 wired network running at 1 Mbit/s. Because its specification was not put in the public domain, it was mainly used in Schneider systems.
Modbus/SL	The name used for serial Modbus capabilities inside Ethernet/IP.
Modbus/TCP Security	The name of the secure version Modbus/TCP, released in 2018.
Enron Modbus	A variant of Modbus/RTU developed by Enron for use in the process industry, supporting 32-bit integer numbers and floating-point numbers (see http://www.simplymodbus.ca/enron_history.htm for more details).
JBus	A French version of Modbus/RTU, for 99% identical to it but with a few small changes, originally popular with Telemecanique/April PLC’s, but today hardly seen anymore.
Secure Modbus/TCP	A version of Modbus/TCP using the “IPSec” protocol, released by Schneider in 2015 in its own M580 PLC’s.
UMAS	“Unified Messaging Application Services” is a proprietary protocol[3] of Schneider which uses Modbus/TCP as transport. It uses a special function code (90) to transport UMAS-protocol messages in the data part of a Modbus message. UMAS allows access to tags with a name (for example, “ActualPressure”) instead of a physical address (i.e., 00020). By using Modbus this way, modern Schneider PAC controllers can use UMAS, while devices without UMAS-support can still use the traditional Modbus function codes. Since UMAS is a protocol completely different from Modbus, we will not discuss it in this publication.

In the remainder of this publication, we will mainly focus on Modbus/RTU. Where appropriate, the differences with Modbus/ASCII will be described. Modbus/TCP is itself based on Modbus/RTU, and where appropriate we will describe the details.

[3] *The specification of UMAS is not publicly available, but on www.lirasenlared.xyz/2017/08/the-unity-umas-protocol-part-i.html much information can be found.*

2. Differences between the members of the family

Although all are members of the same family, this does not mean that they are “on speaking terms” with each other. This is usually due to different ways of cabling, network speeds, protocol implementations, etc. It is best to assume that there is **zero** compatibility.

Luckily, in practice one doesn’t encounter all the members of the Modbus family, probably only Modbus/ASCII, Modbus/RTU and Modbus/TCP. These operate more or less identically, so protocol converters can be bought.

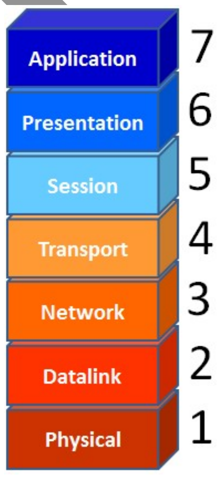
Enron Modbus and JBus are both similar to Modbus/RTU, you should be able to connect this to a Modbus/RTU network with little difficulty.

CHAPTER 3. THE OSI-MODEL AND MODBUS

Understanding Modbus is not difficult, since the protocol specification is quite small (some 50 pages), compare this to the documents for Profibus, which needs around 600 pages, or Ethernet (2000+ pages). Nevertheless, it is still a lot of information. The well-known "OSI 7-layer model" is the world's leading reference architecture on networks, and understanding the OSI-model helps in understanding how network protocols work.

1. Why seven layers?

A network protocol is sometimes very complex, taking several hundreds to thousands of pages to describe. Without any logical structure there is no way to properly design a protocol, or to implement it. It is also difficult to compare network protocols.



So all possible network functionality is divided into seven layers. Why seven and no six, or eight? After a lot of discussions the compromise was: 7. Some 20 years later, it appeared that 8 would have been better, sometimes layer 2 is split in two. But the OSI-model was not changed.

The names of the seven layers are standardised, see the picture on the right. With the easy-to-remember sentence "All People Seem To Need Data Processing" you have the 6 starting letters of the top 6 layers (A, P, S, T, N, P) helping to remember the full names.

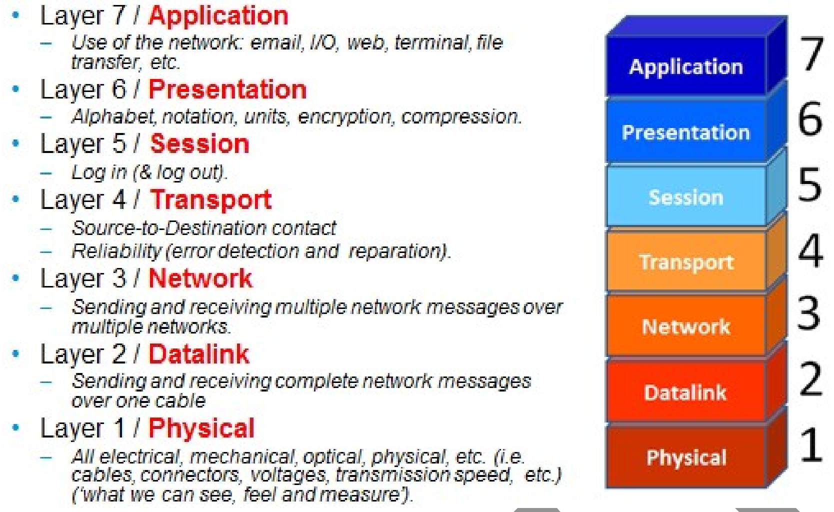
Which functionality is in which layer? OSI makes a difference between layers 1..4 and 5..7. The **first** group is meant for transmission of network messages from A to B. The **second** group is meant for the application: what do I do with a network, how do I use it?

Function of each layer

What does each layer do? That depends on what needs to be done by the protocol.

Starting at the bottom, the **physical** layer, here we see everything that we can see, feel and measure: electronics, cables, shielding, connectors, electrical voltages, etc. These allow us to transmit 0 and 1 bits from here to there.

One layer higher, the **datalink** layer, those separate 0 and 1 bits are combined to 'network messages'. These bits belong together, and are delivered as a group on a receiving device. To which device the message is delivered is determined by a 'network address', which must be unique on the network. Measures are also taken to detect damaged bits (i.e. sent as 0 but received as 1, or vice-versa). If so, the datalink layer at the receiving device ignores the whole message.



- Layer 7 / **Application**
  - *Use of the network: email, I/O, web, terminal, file transfer, etc.*
- Layer 6 / **Presentation**
  - *Alphabet, notation, units, encryption, compression.*
- Layer 5 / **Session**
  - *Log in (& log out).*
- Layer 4 / **Transport**
  - *Source-to-Destination contact*
  - *Reliability (error detection and reparation).*
- Layer 3 / **Network**
  - *Sending and receiving multiple network messages over multiple networks.*
- Layer 2 / **Datalink**
  - *Sending and receiving complete network messages over one cable*
- Layer 1 / **Physical**
  - *All electrical, mechanical, optical, physical, etc. (i.e. cables, connectors, voltages, transmission speed, etc.) ('what we can see, feel and measure').*

At the **network** layer, we see functionality to transmit network messages over multiple networks in a row (A, B, C, ...), step by step, until it arrives at the final destination. If the amount of data is too large to be sent in one network message, the data is split up over multiple smaller network messages ("fragments"). The network layer checks that all fragments arrive at the final destination. If some fragment is missing, the network layer at the receiving device ignores all other fragments.

The **transport** layer handles the communication between the originator of a message and its destination. Usually they are on the same network, but in case they are not, it is the network layer (see above) that handles this (but the transport layer is unaware of this).

The transport layer can also detect errors, and repair them. The physical / datalink / network layers can also detect errors, but usually they do **not** repair[4] them. This is now the task of the **transport** layer: it detects all sorts of errors, and repairs them (within certain limits of course). If the repair action(s) fail, the transport layer given an error, so it is known what happened. So the transport error adds reliability to a network. Layers 5, 6 and 7 do not have to worry about reliability anymore; if the transport layer says it is bad, it is bad! Nobody can repair errors better than the transport layer can (of course, if the transport layer protocol is made to detect & repair errors; there are also transport layer protocols that don't do this, for example because it is not needed or because errors are detected & repaired in the application software).

The **session** layer's task is to establish, check and stop communication sessions between two devices, authentication (who are you?) and authorization (checking whether *you* are allowed to do certain actions), etc. The session may layer may also handle the merging of several data streams, i.e. audio and video of a live stream, which must be perfectly synchronous.

The **presentation** layer takes care of how data is transferred: which alphabet do we use (ASCII, Unicode, or something else?).How are floating point numbers transmitted? Or strings? Which mathematical units do we use (kilometres or miles?). Do we send data not-encoded or encoded, and

[4] As usual, there are exceptions to this: the so-called "Forward Error Correction" as in Bluetooth, because it is more efficient in wireless environments with their high bit error rate. FEC corrects errors on the receiver of a network message, instead of detecting errors and subsequently asking for a retransmission ("Backward Error Correction" - BEC), which takes more time (but is more efficient in wired networks with a low bit error rate).

if so, according to which algorithm? Do we compress data or not, and if so, according to which algorithm?

Finally, at the **application** layer we see functionality to be used by us: web browsing, email sending and receiving, file transfer, remote login, gaming, streaming audio, video, network printing, file sharing, skype, social media, etc. With Modbus, we see here the functionality that we can use: read, write, and control a PLC, or a remote I/O module, a drive, a terminal, etc.

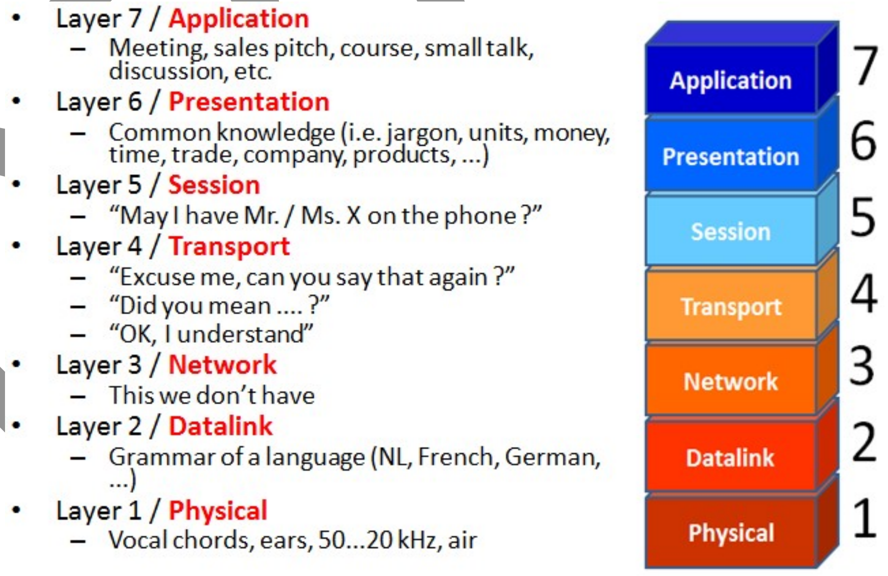
> Note that the application layer is **not** the application (a mistake made very often); it provides services to the application software. For example, a web browser uses the application layer protocol (such as HTTP) to get webpages from a remote server.

Taken together, the physical layer and the application layer are the two are the most visible parts of a network to any user. Every network needs wires (or wireless) and offers functionality to its user.

2. The human OSI model

The OSI-model is often mentioned in presentations about network. But it is seldom explained in a proper way, leaving the listeners in a state of confusion: why is this relevant? I therefor often use an analogy.

Surprising as it may seem, the OSI model is also applicable to communication between humans. We employ exactly the same ways of communication as a network protocol does. Communication between humans is also prone to errors, and we have developed ways to handle this. The "software" for this is "downloaded" when we are young (toddlers), or we learn our social behaviour.



- Layer 7 / **Application**
  - Meeting, sales pitch, course, small talk, discussion, etc.
- Layer 6 / **Presentation**
  - Common knowledge (i.e. jargon, units, money, time, trade, company, products, ...)
- Layer 5 / **Session**
  - "May I have Mr. / Ms. X on the phone ?"
- Layer 4 / **Transport**
  - "Excuse me, can you say that again ?"
  - "Did you mean .... ?"
  - "OK, I understand"
- Layer 3 / **Network**
  - This we don't have
- Layer 2 / **Datalink**
  - Grammar of a language (NL, French, German, ...)
- Layer 1 / **Physical**
  - Vocal chords, ears, 50...20 kHz, air

Starting at the bottom, our human physical layer is mostly the air, over which we talk and listen, using frequency modulation between some 20 Hz and 20 kHz and with amplitude modulation (speak soft / shout), via our vocal chords and ears. But this is not the only physical layer we have; deaf people can transmit sign language with hand movements and 'read' this visually, or blind people can feel Braille script.

The next higher layer is the grammar of the language we use, which is independent of the physical layer: Dutch, French, English, German… Humans do not have a network layer; we cannot talk from A to B with many intermediate humans in between. Messages get garbled and we have no social system in place that we can use to 'route' how a message must be sent onwards to its destination.

The human transport layer exists, and is used to detect errors in communication, although not always flawless due to the ambiguities in the grammar of our languages. But we can ask a person to repeat something that we didn't understand correctly, we can acknowledge messages, we can ask questions back, etc.

Our session layer consists of the social conventions on how to start a conversation, i.e. by "Excuse please, may I …", or with colleagues or family members by calling out their name first, attracting their attention. Once the session is 'opened', the real conversation can begin. When ready, we finish by "Thank you", "Goodbye!", "Be seeing you", etc. signalling the session is finished. When using the phone, there are similar conventions.

The human presentation layer handles conventions like: which currency are we using (Euro's, pounds, dollars), how are distances measured (kilometres, miles), which alphabet is used (Latin, Chinese, Russian, etc.), how are fractional numbers written (2.5 or 2,5), how is time handled (AM:PM or 24-hours system), etc. In daily life this is almost invisible, as we just use these conventions without thinking about it. But when one goes on a business trip abroad it becomes immediately that there must be agreement between speakers ("Is this price in € or $?")

Finally, the human application layer describes how people interact: are we having a business meeting, a holiday party, are we listening to a concert, talking to our children or to our parents, are you being pulled over by a police officer, teaching in front of a class, commanding soldiers, etc. In each of these examples we interact differently.

3. Identical layers

Whenever two parties are communicating to each other, be it humans or electronic devices, there is one very important rule:

Both must have the same implementations for ***all*** seven OSI-layers.

Let's take two humans as example. Our usual way of communicating (talking/listening) fails if one person is deaf. Another common physical layer must now be found. So write everything on paper, which the other person can read. Works fine, unless the person is blind. We could then use Braille script. We can continue to use the same language (= the higher OSI-layers can still be the same!)

Unless, of course, one person speaks/writes English, and the other Dutch. Despite having a common physical layer, there is now a mismatch at the datalink and communication is impossible – we hear/read the other's messages, but cannot understand them.

The solution could be for one person to use the other language (= download a 2nd protocol layer in your brain), or to use an interpreter. In both cases there is a translation between languages, and communication will work.

When using electronic devices, we see exactly the same. A USB-device cannot communicate with an Ethernet device; the physical layer is different (connector, wire, speed) as is the datalink layer. You need a converter. Modbus devices also cannot communicate with (say) Profibus devices, as the layer 7 protocol is different. Even when both use RS485 as physical layer, there is not the same implementation in all 7 layers, so it still is no going to work.

*I use this example because I see this going wrong in practice very often. A user has two devices with an RS485 interface, and therefore assumes that communication must be possible. But this **also** depends on the implementation of the 6 other protocol layers.*

4. Examples of OSI-layers

When we look at the protocols used in OSI layer 7, we see a lot of familiar abbreviations. Not surprisingly, because we often work directly with applications using these protocols. Below we give a few examples, and the tasks done by each protocol:

Sending / receiving email: SMTP, POP3

Contact mail server, get incoming mail, show mail headers, send mail, erase, handle address book, set spam filter, ...

File transfer: FTP, NFS, Samba, CIFS,

List files, copy file, delete file, rename file, get file, put file, create directory, delete directory, disconnect disk, ...

Remote login: Telnet, Rlogin,

Login, logout, move / click mouse, send command, get response, ...

Webbrowsing: HTTP, HTTPS

Contact webserver, get page, get picture, use secure transfer, ...

Remote I/O: Modbus, Profibus/DP, AS-Interface, CAN, ...

Read inputs, set outputs, get diagnostics, write parameter / setpoint, give command, ...

A device may support multiple application protocols in parallel. A good example of this is your PC / mobile phone / tablet. It allows web browsing, email handling, video / audio streams, etc. All these are handled concurrently by the operating system on the device. Very simple devices probably only support one application protocol – for example, Modbus!

Examples of OSI-layer 6

Regarding data presentation, one of the oldest is “ASN.1” (Abstract Syntax Notation 1), but this is seldom used in industrial network protocols. Profibus/FMS used the “FER” (Fieldbus Encoding Rules), a simpler implementation of ASN.1, but this also has not caught on.

In practice many protocols simply follow hard-coded presentation rules, such as the “Big Endian” or “Little Endian” format for multi-byte datatypes (in effect saying: “User, you handle it!”). Floats and doubles are encoded as specified in IEEE-754. Strings have many different formats: ASCII, Unicode. File formats are plenty: ZIP, TIFF, JPG, MP3…

Modbus devices often follow the “Big Endian” format, but not always! We will discuss how Modbus represents its data in chapter 7.

Examples of OSI-layer 5

Layer 5 protocols are relatively unknown, but there are several. To name a few: ASP (AppleTalk Session Protocol), PPTP (Point-to-Point Tunnelling Protocol), NFS (Network File System), etc.

Examples of OSI-layer 4

There are two very well-known transport layer protocols: TCP and UDP (both from the TCP/IP family of protocols). They have completely different designs, which can be summarized as follows: everything that TCP does, UDP doesn’t. TCP is ultra-reliable, So UDP is much faster, but less reliable. Appendix C will explain this in more detail.

Examples of OSI-layer 3

There is one very widely used network protocol: IP. IP stands for “Internet Protocol”, and is (as the name says) the basis of the world-wide web.

In the past, when (telephone) modems were used to communicate on internet, it turned out that IP had too much overhead. Subsequently, a more efficient version of IP called “SLIP” (Serial Line IP) was developed which better fitted the bandwidth-constrained modems. Nowadays this is not an issue anymore, and SLIP is not used anymore.

Examples of OSI-layer 2

Each network protocol has a layer 2, so we can name any protocol here: Modbus, Profibus, Ethernet, USB, …

Examples of OSI-layer 1

The physical part of a network is what every user sees. Some well-known standards:

RS232, the “serial” interface, also called “COM port" on PC's. Each pin in the connector has a dedicated function. Voltages uses are in the range -15V (for a '1') to +15V (for a '0'). One data bit can be sent at a time, hence the name 'serial'. Speeds are in the range 110 bit/s up to 115 Kbit/s. Two devices can be connected.

Centronics, the "parallel" interface on PC's used to connect printers in the past. Eight data bits can be sent at a time, hence the name "parallel". The speed depends on the electrical handshake between the two devices.

USB, the "Universal Serial Bus" to connect small equipment. It can power devices via 2 wires, and communicate over 2 other wires over a maximum distance of 5 meter. Speeds are in the range from 1,2 Mbit/s to 9,6 Gbit/s.

Ethernet, the high-speed network link to connect PC's to the LAN.

WiFi, the wireless version of Ethernet. Network messages are sent radiographically on the 2,4 GHz band, at a speed between 11 Mbit/s and several Gbit/s (for the modern versions). Distances that can be covered can be multiple kilometres, depending on the quality of the antennas (and having no external interference, or disturbances).

5. The OSI-model in relation to Modbus

Having explained the OSI-model's seven layers, it will be a surprise to learn that Modbus does **not** specify them all. For example, when we take Modbus/ASCII and Modbus/RTU:

a) No standard physical layer. Of course every network needs a physical layer, so every Modbus vendor must choose one. In practice, most vendors use RS232 or RS485, but even then sometimes two devices cannot work together for differences in their physical layer implementations.

b) No network layer. It is **not** possible to connect two (or more) separate Modbus networks and make one larger network out of this. If this is needed, it must be handled by the application software itself.
c) No transport layer. As a consequence, error detection and error repair automatically become the responsibility of the application software.
d) No session layer. It is not necessary to “log in” on a Modbus device – if two (or more) devices are on the same physical network, they implicitly trust each other and can communicate with each other.
e) No presentation layer. This becomes the responsibility of the application software.

And in Modbus/TCP:

a) No physical layer. In practice this will usually be the Ethernet-standard UTP (Unshielded Twisted Pair) version running at 10, 100 or 1000 Mbit/s. But wireless Ethernet is also possible, nowadays according to 802.11n, but the older (slower) version 802.11b or 802.11b are also possible. Also possible here are ADSL (telephone line) or GSM (mobile phone).
b) No datalink layer. It is not necessary for Modbus/TCP to specify this, as this is completely ‘hidden’ by the IP protocol. If two devices both have TCP/IP, in principle they can communicate with each other, as long as there is a communications path. Usually we see the Ethernet datalink protocol used, but it can also be the datalinks of ADSL, WiFi, GSM, etc.
c) No session layer (see above).
d) No presentation layer (see above).

The differences between Modbus/ASCII and RTU, and Modbus/TCP are mainly caused by the use of TCP/IP. When using this protocol suite, one becomes independent of the underlying datalink layer and physical layer, giving enormous flexibility. TCP/IP itself adds the network layer and the transport layer, allowing communication over LAN’s and over internet, in a very reliable way.

CHAPTER 4. THE PHYSICAL LAYER

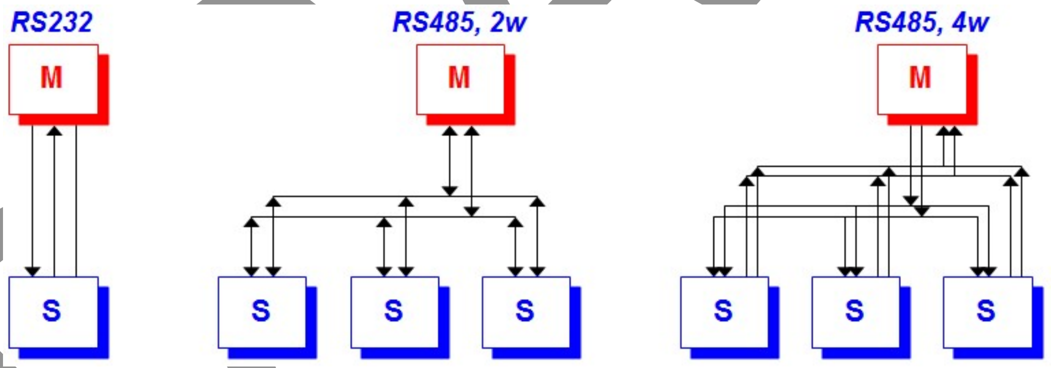
1. Missing physical layer

Modbus/ASCII and Modbus/RTU

The serial Modbus versions do **not** specify which physical layer must be used for the network. This is strange, since every network (Modbus and all others alike) require a physical layer: a copper wire, coax cable, fibre-optic cable, infrared, twisted-pair, wireless radio, etc. In order for two devices two communicate, both **must** have the **same** physical layer implementation. Because Modbus does not specify one, every vendor must choose something. It is thus very well possible that two Modbus devices from different vendors cannot communicate, due to physical layer differences.

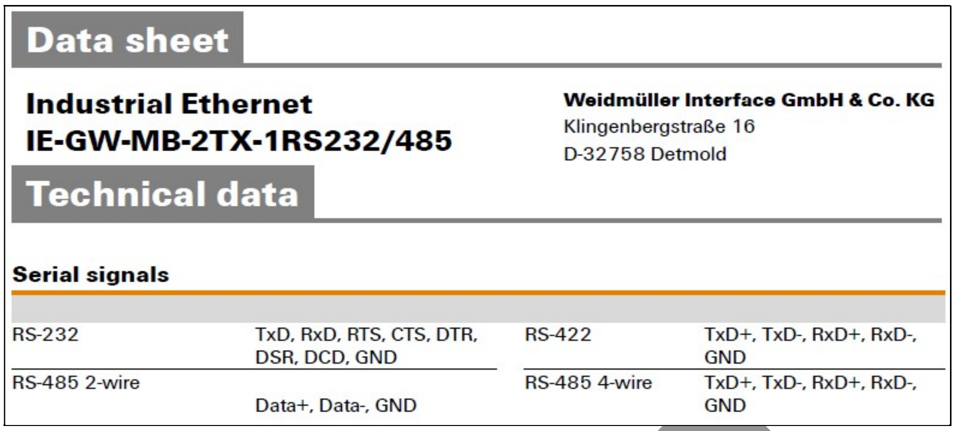
This sounds worse than we see in practice, because a device with a strange physical layer implementation cannot communicate with anyone else in the world, which does not help to improve the sales figures for such a device. Therefore, most Modbus devices use one of the following physical layers:

- RS232. The simplest version employs 3 wires (two for data, + GND).
- RS422/RS485, 2-wire or 4-wire. Sometimes an additional GND is necessary, so one can end up with 3 or 5 wires instead.



The device marked "M" is the 'master' of the network; the devices marked "S" are the slaves. These roles are explained in more detail in the next chapter.

Which physical layer a device implements is normally found in the documentation. An example:



Data sheet

Industrial Ethernet
IE-GW-MB-2TX-1RS232/485

Weidmüller Interface GmbH & Co. KG
Klingenbergstraße 16
D-32758 Detmold

Technical data

Serial signals

RS-232	TxD, RxD, RTS, CTS, DTR, DSR, DCD, GND	RS-422	TxD+, TxD-, RxD+, RxD-, GND
RS-485 2-wire	Data+, Data-, GND	RS-485 4-wire	TxD+, TxD-, RxD+, RxD-, GND

Modbus/TCP

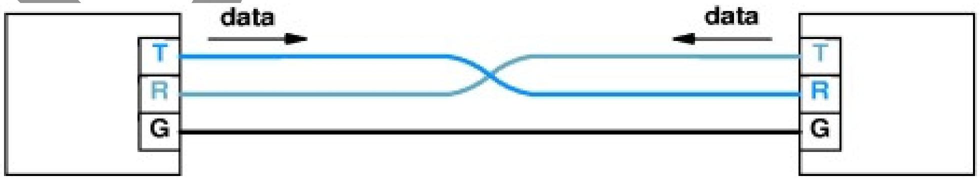
Modbus/TCP also doesn't specify a physical layer. It doesn't have to; it runs on **any** physical layer that is capable of handling TCP, in practice this is usually Ethernet, but it can also be WiFi (wireless Ethernet), ADSL (telephone line), fibre-optic cable, coax cable, microwave, satellite, GSM, the internet, etc. This gives enormous flexibility in using Modbus/TCP!

Theoretically, you could allow communication between two Modbus/TCP devices via internet, one at home and the other at the south pole, and this would take the same effort as connecting two Modbus/TCP devices in a factory (except for getting the device installed on the south pole).

2. RS232

RS232 (Recommended Standard 232) is one of the oldest physical layers still in use in industrial automation. The first version was developed in the 1960's for the purpose of connecting terminals to modems. In the 1980's it became popular as the "COM" port on a PC, for connecting equipment such as mice, modems, printers etc. As of this century the popularity of RS232 declined rapidly, due to the rise of USB which allowed for more versatility and higher-speeds. But for industrial applications and embedded equipment RS232 is still an often-used interface, due to its low cost and simplicity.

The simplest RS232 link consists of only three wires. The "transmit" output of the left device is connected to the "receive" input of the right device, and vice-versa. The third wire is the "ground", the electrical reference of 0V.



The maximum distance that can be achieved with RS232 is often specified as only 15 meters. It is a little-known fact that this distance depends on the electrical characteristics of the cable; with so-called "low impedance" cable distances of 50 meters and more can be achieved.

RS232 is a low-speed link: allowable bitrates start at 50 bit/s, maximum bitrate is 115200 bit/s, but many devices support only 19200 or 38400 or 57600. In order for two devices to communicate, they must have a common bitrate.

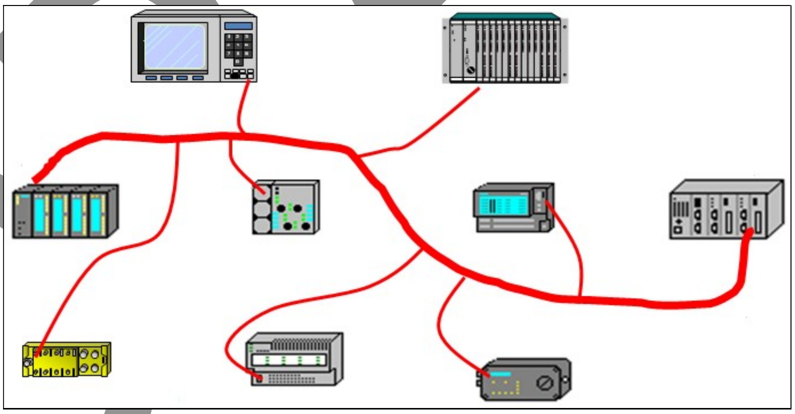
More information
Appendix 13 gives more detailed information about RS232.

3. RS422 and RS485

RS485 is a very commonly used physical layer for industrial networks. It is not only used in Modbus, but also in popular protocols like Profibus. In contrast to RS232, RS485 is much better suited for industrial applications, as it allows for much longer networks (up to 1200 meter), more devices (usually maximum 32, sometimes even 64 or 128) and higher speeds (up to 10 Mbit/s).

RS422 is the little brother of RS485, more limited in its capabilities (max 10 devices) and not often seen in combination with Modbus. However, many vendors describe that their equipment uses RS422 due to a common misunderstanding: a 2-wire network is RS485; a 4-wire network is RS422. This is not true; both RS422 and RS485 can have 2-wire and 4-wire variants.

An RS485 network consists of a main network cable called “trunk”, on which devices can be attached via “stub” lines. All devices are electrically connected to each other, which make that there are rules to be followed in order to guarantee the correct electrical functioning of the network: trunk length, stub lengths, number of devices, speed, etc. Failure to comply with these rules will cause communication problems between some devices, or a network that doesn’t function at all.

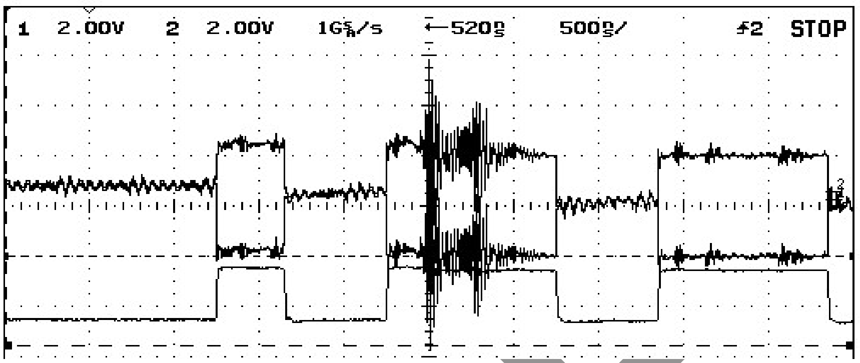


RS485 exists in two versions, one using two signal wires, and another using four signal wires. Nowadays, mostly the two-wire version is used, but sometimes one encounters the four-wire version.

Connecting two-wire RS485 devices to four-wire RS485 devices may not always work, due to the four-wire devices hearing their own transmissions back. Ask the vendor of a four-wire device to find out whether it may be connected on a 2-wire network.

Filtering out noise

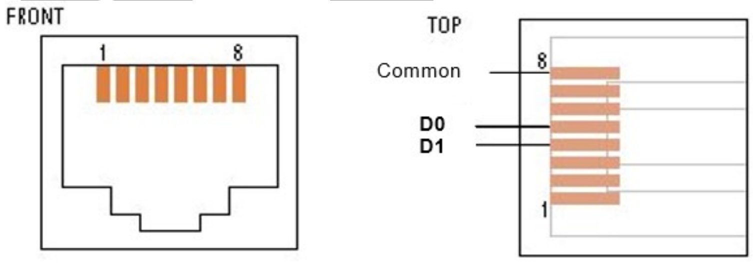
A good reason for using RS485 in industrial application is that it very resilient to electrical disturbances caused by EMC, due to the “balanced transmission” technique. All data is sent over two wires, and the voltage difference between the two wires determines whether a ‘0’ or a ‘1’ is sent / received. Electric disturbances influence the voltages on both wires in an equal amount, so the voltage difference remains the same, in effect cancelling out the noise. This behaviour makes RS485 especially capable



An example of the voltages on the two network wires (middle), and the voltage difference (below). Large amount of noise have no effect on the voltage difference!

The connector

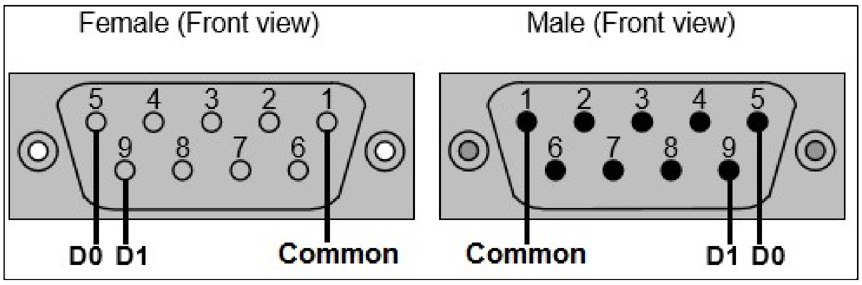
In [MBUS300] no specification was given about the connector to be used. In the later [MBUSSERIAL], both the well-known RJ45 (used in Ethernet), and the 9-pin sub-D connector are described.



Signals on the RJ45 connector for the two-wire RS485 version.

The RJ45 has become popular in the last decade for use with non-Ethernet networks, due its small size, low cost, and cheap cabling (thanks to Ethernet). On a device, always the female connector is found. Unfortunately, RJ45 is often confused as being the Ethernet-interface of a device, which doesn’t have to be so. More confusing is that for Modbus/TCP devices it *is* the Ethernet interface.

The 9-pin sub-D connector can be found in either its male or female version.



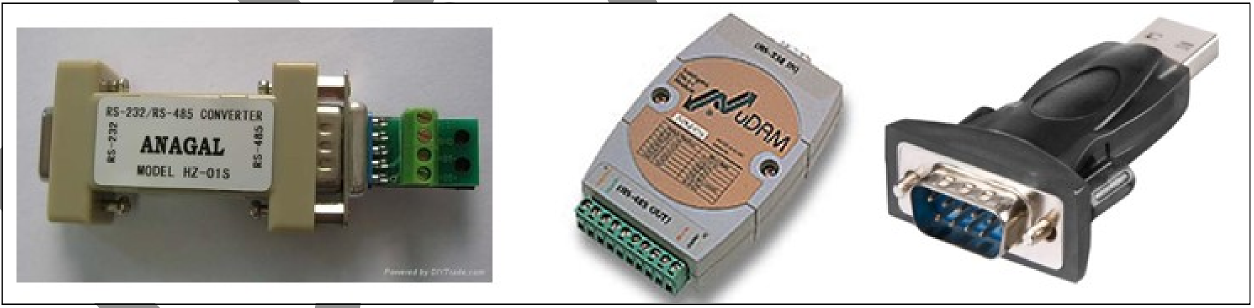
Note that many vendors have decided to use other connectors than the RJ45 or 9-pin sub-D. And even when one of these connectors is found on a device, it may still be that the signals are on different pins. This makes wiring large Modbus networks always a time-consuming task.

More information
Appendix 14 gives more detailed information about RS485.

4. Conversion

Sometimes, a device has a RS232 interface, but it must connect to a device using RS485 (of vice-versa). This is not directly possible, due to the electrical differences. A converter is necessary, which are readily available on the market.

Also increasingly common is the need to connect a PC *without* RS232 port to a Modbus device using RS232 or RS485. This is easy to do, as there are many USB-to-RS232 or USB-to-RS485 converters on the market.



Converters without power supply
Many converters do not require an external power-supply, as they can get power from USB (5V / 0.5A) or from any unused RS232 modem control output signals. This is very useful especially as no additional power supply is needed (and associated 110/230V outlet), but it also has its drawbacks, which are often not well described.

So take care when using RS232/RS485 converters:

- Due to the limited amount of power, it is **not** possible to have a 32-device network *and* running at very high speeds *and* over long distances. Usually you can have only ***one*** of these three.

- On laptops, RS232 signals are often at -9V or +9V, instead of the maximum -15V / + 15V. This further limits the capabilities on RS485.
- The smallest converters have no space for a galvanic isolation between the RS232 and RS485 electronics. This means that when there is a voltage spike on the RS485, the RS232 electronics may get damaged, as could be the motherboard (usually a PC or PLC).
- The modem control signals cannot be used.

In any permanent system, the author recommends the use of externally powered, galvanically isolated converters.

Fibre optic converters

When there is a need to cover longer distances than RS485 allows, the use of fibre-optic converters is a good solution. Usually they are used in pairs, so all data sent to the first converter comes out unmodified at the second converter. This is done completely transparently, allowing use with any network protocol, including Modbus/ASCII or /RTU.

CHAPTER 5. DATALINK LAYER

The Modbus datalink layer specifies the (binary) format of the network messages, the speed with which messages are transmitted, the way devices operate on the cable and how errors are detected (but not repaired).

1. Masters and slaves

On any bus-wired network (like RS485), there must always be a mechanism to prevent multiple devices transmitting at the same time. Modbus implements the so-called “master/slave” algorithm to prevent simultaneous transmissions from multiple devices. It works as follows:

- On the network, exactly **one** device is designated to execute the “master” role. Usually this is a PLC, or a PC, or an intelligent device.
- All other devices on the network execute the “slave” role. Every slave needs a (unique) network address, set by the user.

The cooperation between the master and the slaves is as follows:

- The master transmits a network message, meant for exactly one slave. This is indicated in the network message by the “slave address” field (see below).
- All slaves receive this network message.
- Every slave compares the “slave address” field in the network message to its own slave address (as configured earlier).
- If there is **no** match, a slave ignores the network message completely.
- If there ***is*** a match, the slave must execute the command in the network message.

- In the meantime, the master is waiting for the answer (it may not send any other network message until the answer is received).
- When the slave is ready, it sends an answer back to the master.
- The master processes the answer, and can then start again.

By this way of working, there can always be only **one** active transmitting device on the network; the master and the slaves alternate.

> *Why is this important? Remember that on the RS485 Modbus versions all devices share the same network cable. Having multiple devices transmit at the same time would mean that they disturb each other's traffic. On RS232 this cannot occur, due to the separate RX / TX lines, but one cannot have more than one device anyway. With Modbus/TCP it should theoretically be possible to have multiple devices active at the same time… if only the software would support it!*

This very simple way of working has some serious consequences:

- The master plays an important role on a Modbus network. If the master fails, the whole network stops.
- The master **must** wait for an answer of a slave. If a slave is slow in answering, the master is also slowed down. If a slave never answers, the master cannot continue. As this is disastrous for the remainder of the network, this error is usually detected and handled at higher levels, so after a while the master can continue controlling the remaining slaves.
- As slaves may never send messages autonomously, slaves cannot send alarms (or other important messages) when the need arises.
- As slaves may only send answers to the master, slaves **cannot** communicate with each other. Every slave communicates with the master, but with nobody else.

It is an error to have a network with more than one master. As a second master operates unware of the presence of the first master, it may sends its own messages simultaneous with those of the first master. The net result is that both messages are disturbed. One *may* be lucky: when a message of the second master is sent in the idle time of the network, but then at a certain moment a slave is going to send a message, and this may also interfere with another transmission. Apparently random errors, delays and retransmissions are the result.

Client/server

Probably due to the political connotation of the terms master and slave, the Modbus User Group has changed the specification. The terminology used today is that Modbus is a "client/server" network. Luckily, the way of working of Modbus is not changed at all!

2. Network addresses

As on any network, devices must have a unique 'network address', sometimes also called 'slave address' or 'unit identifier'. This is needed because every device hears *all* transmissions of all other devices; how can a slave know which message is destined for him? Therefore a network address must be set on a device. Modbus *only* requires network addresses for slaves; a master doesn't need one.

The network addresses must be configured on all slaves before they are switched on. How this is to be set, may be determined by the vendor of the device(s). Note that each slave address may be used only once on any network; it is a gross error to have two (or more) devices with the same address!

Network addresses in Modbus are 8 bits in size, so can have the value 0..255. However, the Modbus specification mentions:

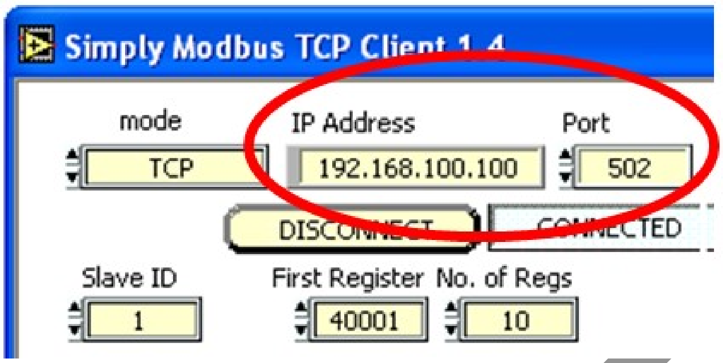
- Address "0" is meant for commands sent as 'broadcast' (see below).
- Addresses 248..255 are reserved for "future extensions". But these extensions have not been written for more than 30 years. Therefore, some vendors allow usage of these network addresses.

A "broadcast" is a special feature of Modbus. When a master sends a network message with value 0 where normally the slave address is used, the message gets a special meaning: it is meant for **all** slaves on the network.

After having received a broadcast, a slave may **not** send a reply message back, not even error messages. This is done to prevent multiple slaves sending these replies simultaneously, causing the messages to be lost. A consequence of this is that the master does not know whether all slaves have received the broadcast, and/or whether they could execute it without any problem. Also, commands that by their very nature exist to return data to the master, such as "Read", can of course not be used with broadcasts.

The Modbus specification [MBUSSERIAL] specifically mentions that all slave devices ***must*** *support broadcasts. The older specification [MBUS300] does not require this. It is therefore very common to find slave devices that do* ***not*** *support broadcast. Even worse: it is the experience of the author that most devices do not support it!*

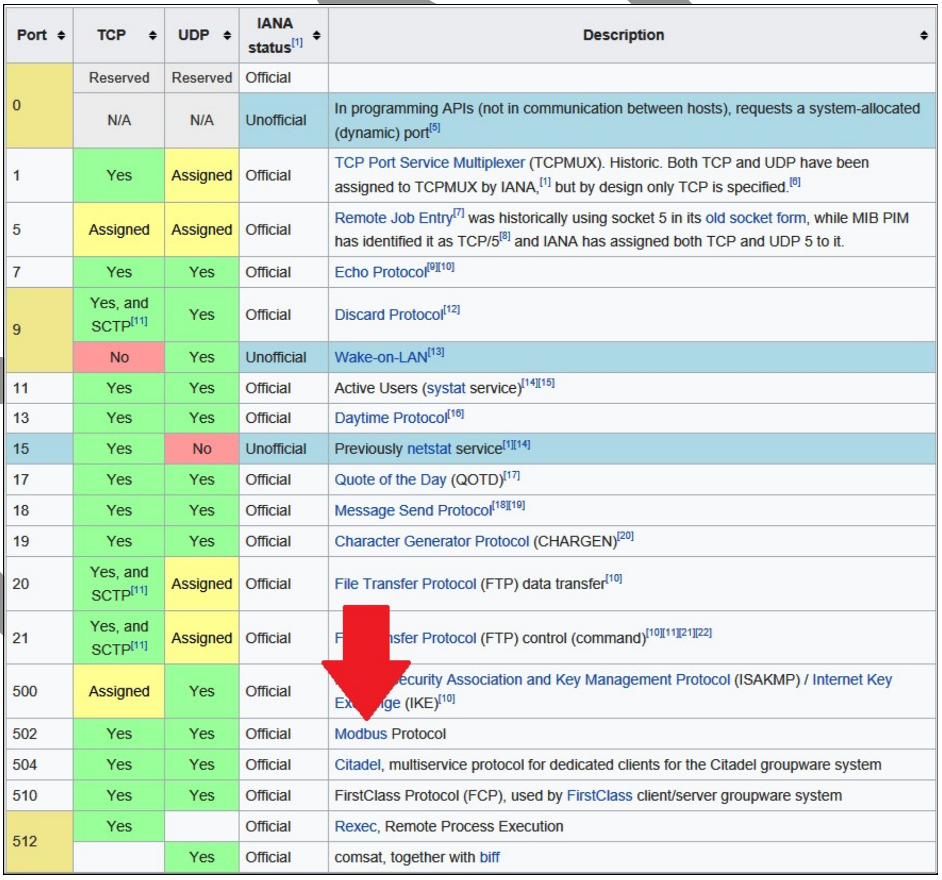
Modbus/TCP differences
When using Modbus/TCP, each device (also the master!) must have an "IP address". It is 32 bit in size, and written down in "dotted" notation, for example: 172.16.5.35. These addresses are normally assigned by the network administrator, either a "static" (fixed) setting known at power-up, or a "dynamic" setting where the IP-address is assigned on request by a DHCP server available somewhere on the network.


Simply Modbus TCP Client 1.4

mode: TCP
IP Address: 192.168.100.100
Port: 502
DISCONNECT
CONNECTED
Slave ID: 1
First Register: 40001
No. of Regs: 10

Modbus/TCP does not have a ‘broadcast’ address, because TCP does not support broadcasts, but only point-to-point communication.

Additionally to the IP-address, any Modbus/TCP server must have a “port”. This is a feature of TCP/IP allowing concurrent usage of multiple network protocols on the same device. The port (a number in the range 1..65535) indicates which protocol is used, and TCP/IP uses this information to send incoming messages on to the right protocol handler. One such well-known port is 80, used for webservers. Modbus/TCP uses port 502. The number 502 has been officially assigned by the Internet authority (on request of Schneider).



Port	TCP	UDP	IANA status[1]	Description
0	Reserved	Reserved	Official	
	N/A	N/A	Unofficial	In programming APIs (not in communication between hosts), requests a system-allocated (dynamic) port[5]
1	Yes	Assigned	Official	TCP Port Service Multiplexer (TCPMUX). Historic. Both TCP and UDP have been assigned to TCPMUX by IANA,[1] but by design only TCP is specified.[6]
5	Assigned	Assigned	Official	Remote Job Entry[7] was historically using socket 5 in its old socket form, while MIB PIM has identified it as TCP/5[8] and IANA has assigned both TCP and UDP 5 to it.
7	Yes	Yes	Official	Echo Protocol[9][10]
9	Yes, and SCTP[11]	Yes	Official	Discard Protocol[12]
	No	Yes	Unofficial	Wake-on-LAN[13]
11	Yes	Yes	Official	Active Users (systat service)[14][15]
13	Yes	Yes	Official	Daytime Protocol[16]
15	Yes	No	Unofficial	Previously netstat service[1][14]
17	Yes	Yes	Official	Quote of the Day (QOTD)[17]
18	Yes	Yes	Official	Message Send Protocol[18][19]
19	Yes	Yes	Official	Character Generator Protocol (CHARGEN)[20]
20	Yes, and SCTP[11]	Assigned	Official	File Transfer Protocol (FTP) data transfer[10]
21	Yes, and SCTP[11]	Assigned	Official	F[illegible]sfer Protocol (FTP) control (command)[10][11][21][22]
500	Assigned	Yes	Official	[illegible]ecurity Association and Key Management Protocol (ISAKMP) / Internet Key Ex[illegible]ge (IKE)[10]
502	Yes	Yes	Official	Modbus Protocol
504	Yes	Yes	Official	Citadel, multiservice protocol for dedicated clients for the Citadel groupware system
510	Yes	Yes	Official	FirstClass Protocol (FCP), used by FirstClass client/server groupware system
512	Yes		Official	Rexec, Remote Process Execution
		Yes	Official	comsat, together with biff

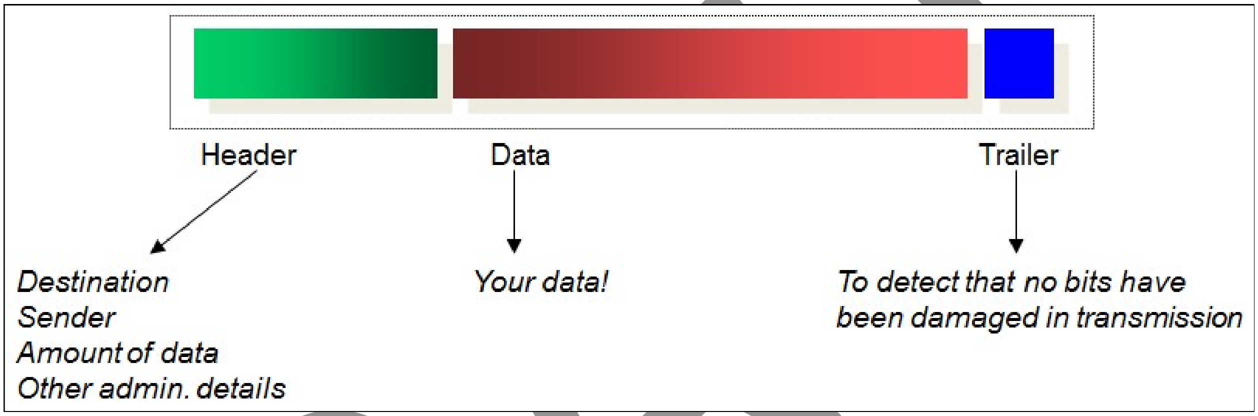
Some vendors allow different ports (i.e. 503 or higher) but in most cases their usage is completely unnecessary. The port is chosen on each server, and may be set to the same value on all devices on a network.

For outgoing Modbus/TCP traffic, a port on the master is also needed. Normally this is automatically assigned by the TCP/IP protocol stack; it does not have to be configured by the user.

3. Message formats

Modbus has a message format not unlike all other datalink protocols. A network message consists of three parts:

- a "header" with administrative information, followed by …
- the "data" section in which the user-provided data is kept, and …
- at the end a short "trailer" (also called: "footer") which is used to detect any damaged bits while the message was in transmission.

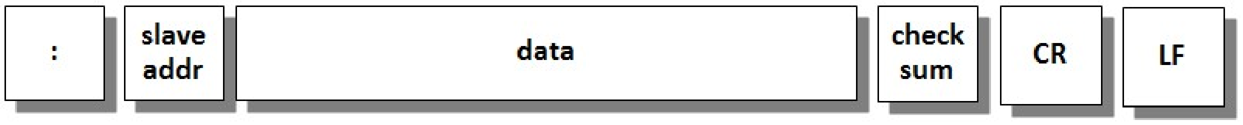


Modbus has three different message formats, one for Modbus/ASCII, one for Modbus/RTU, and one for Modbus/TCP. The data part is depending on which function code is used (see chapter 6).

Modbus/ASCII

The Modbus/ASCII format is as follows:

- Each message starts with the colon character ':';
- Next comes the network address of the slave for which the message is intended, or the network address of the slave which sends a message;
- Next comes the data part (of a variable length);
- Next comes the "checksum" (explained below);
- And the message ends with the characters CR and LF (Carriage Return, Line Feed[5]).

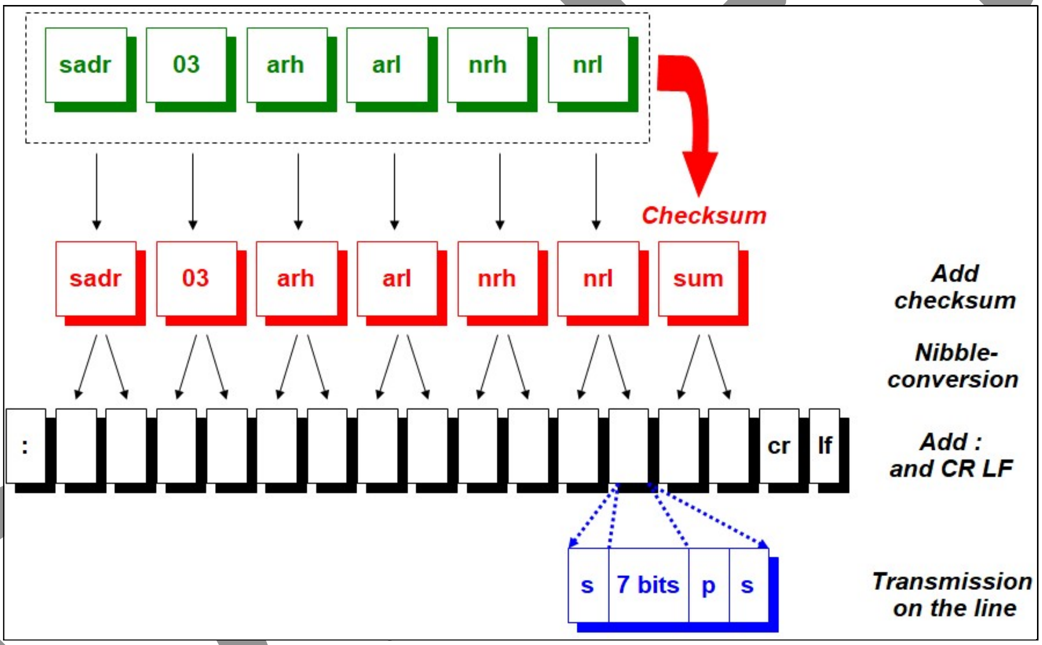


[5] *Theoretically, Modbus/ASCII allows the LF character to be set to a character more to your liking with a special command. The author has never seen this used, however.*

Except for the starting colon, the carriage return and the line feed, Modbus/ASCII has a unique method of transmitting the characters (bytes) in a network message, unlike any other network the author has ever seen. It goes as follows:

- Each byte (8 bits) is split in two “nibbles” of 4 bits each.
- Each nibble, with a binary value in the range 0000…1111 (= decimal 0 to 15) is seen as a hexadecimal value coded with the hexadecimal digit: 0123456789ABCDEF.
- Each hexadecimal digit is converted to the corresponding ASCII-code for that character. The ASCII table uses 7 bits characters. So, bits 0000 are translated to digit 0 which has ASCII value (hex)30. Bits 0001, digit 1, have ASCII value (hex) 41. Bits 1010, digit A, becomes ASCII value (hex)41. Etc.

So, any (8-bit) byte is converted into two 7-bit ASCII characters. These are then transmitted according the serial format, i.e. with a start bit, (optional) parity bit, and a stop bit. The following picture shows the format of a message for function 3 with its two parameters “ar” (address of first register) and “nr” (number of registers to read):



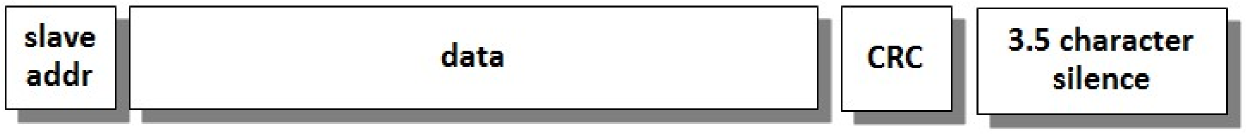
On the receiving devices, the reverse translation takes place: the ASCII-characters are converted back to their 4 bit nibble, and two successive nibbles are stored in a byte. All bytes following the colon are assembled, and when the CR LF is received the message is complete and can be sent up (in the protocol stack) for further processing.

Modbus/RTU

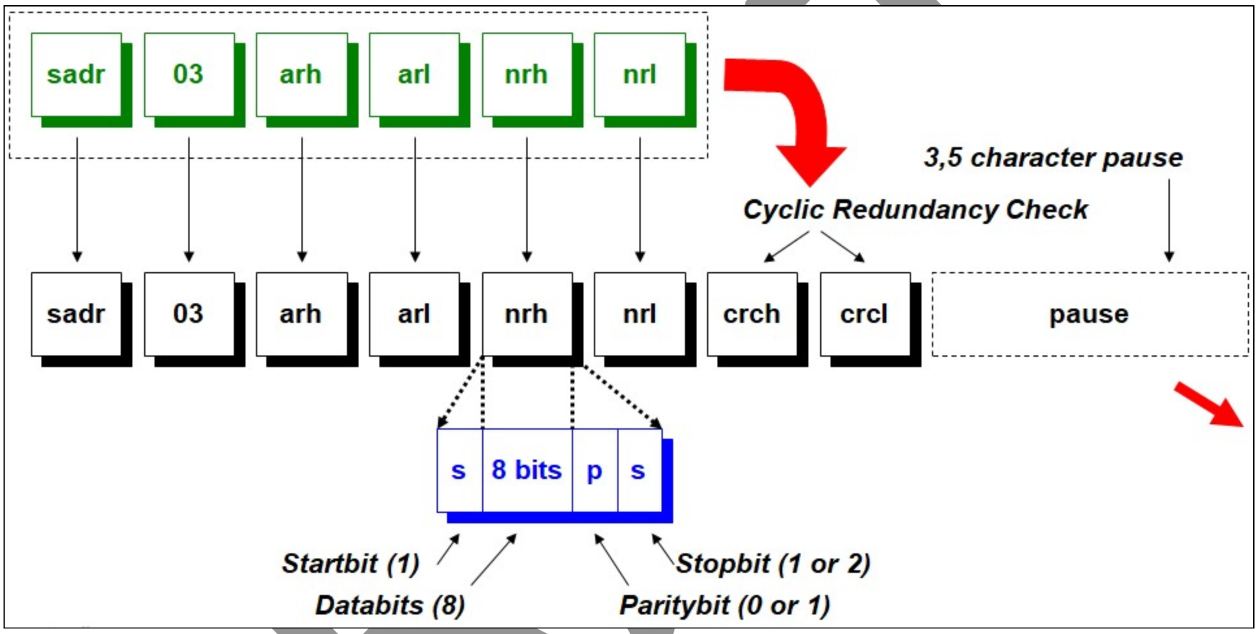
The Modbus/RTU format is slightly different:

- There is no dedicated starting character; the message just starts with the slave address (same function as above);
- Next comes the data part (of a variable length);
- Next comes the “CRC” (Cyclic Redundancy Check, explained below);

- And the message ends when no more data comes for at least the duration of (the transmission time of) 3.5 characters.

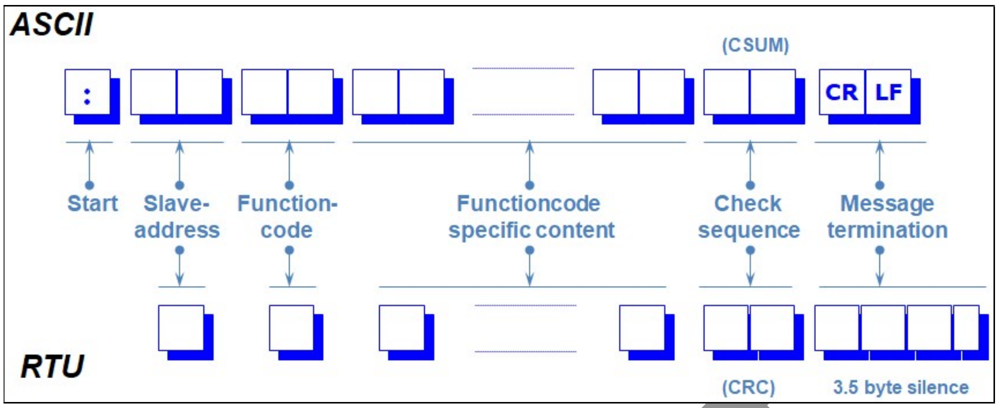


When a message is in transmission, there may not be more than 3.5 character time silence between any two characters in the message. If this occurs, the receiving devices will interpret this as the end of the message, and start processing all data received so far. Because the message is not complete, it will be ignored.



Difference in overhead

The difference in overhead between Modbus/ASCII and Modbus/RTU is substantial. From the original 8 bits in a byte in memory, to 20 bits on the network for Modbus/ASCII. As we will see below, Modbus/RTU also has some overhead, but only 3 bits per byte (11 bits in total). So roughly Modbus/ASCII takes about two times (20/11) more network bandwidth than Modbus/RTU does, which makes Modbus/ASCII considerably slower in use (at the same bitrate). This also depends on the (software) processing speeds of all devices on the network, so in real life the difference may be smaller than the factor 2.



For example, suppose that the master wants to send a message with function code 3 to slave 6 to read 3 registers starting at 107 (hex 6B). The raw data of the message are the bytes "06 03 00 6B 00 03". On a Modbus/ASCII network, this would result in the transmission of 17 bytes:

```
    3A 30 36 30 33 30 30 36 42 30 30 30 33 CSUM1 CSUM2 0D 0A
```

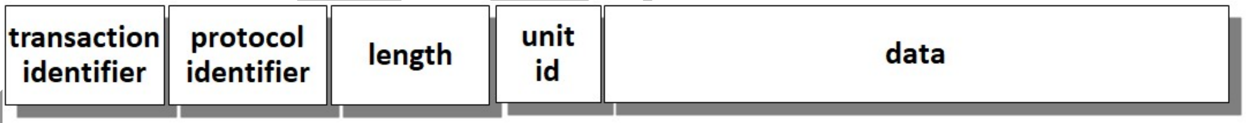
On Modbus/RTU, only 11.5 bytes are needed:

```
    06 03 00 6B 00 03 CRC1 CRC2 <silence>
```

Note that the information contents of the message are equal, only the transmission format differs.

Modbus/TCP

The Modbus/TCP format resembles that of Modbus/RTU; it adds an own header, but does not have a CRC and silence period. A CRC is not necessary, because checks on data corruption are done by the TCP protocol: any received but corrupted data is either repaired by TCP, **or** an error is given. This means that the receiver of Modbus/TCP messages never has to worry about handling of corrupted data.



The fields have the following meaning:

<u>**Transaction ID**</u>

The "transaction identifier" is a 16-bit number that is given a value on the Modbus/TCP client, and must be echoed back by the server in the response message. What the value of this id is, differs per vendor. Some implementations always use the same value, others use an increasing number (5, 6, 7, ...) and there are also implementations that use it for own purposes.

The author prefers the system where the master increases the transaction id for each new command, as it makes it very easy to follow the traffic between a client and a server with a network analyser tool like Wireshark. It also allows for easy detection of any missing *messages or duplicate messages (usually caused by faulty server protocol stack implementations).*

Protocol ID

The "protocol identifier" is a field that for the last two decades has never been used in Modbus/TCP. It is a 16-bit number, and must have the value 0. Probably this field was invented by Schneider to allow for new versions of Modbus/TCP.

Number of bytes that follow

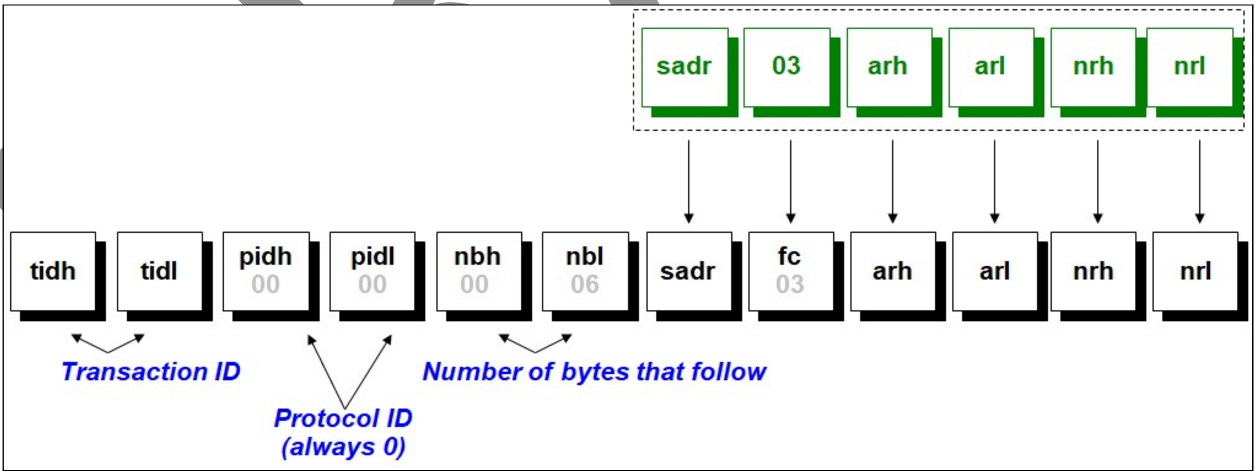
This is a 16-bit field indicating the amount of data that follows. This is necessary in TCP[6] because otherwise the receiver doesn't know how much data is coming.

Unit ID

The "unit id" is an 8-bit field that is often unused. It corresponds to the "slave address" field in Modbus/ASCII and Modbus/RTU network messages. Because of the use of TCP/IP it is not necessary to select devices this way, as the TCP/IP network address is used instead. So in most cases this field is given the value 0.

In Modbus/TCP-to-Modbus/RTU converters, the unit id can be used to provide the slave address to which the message must be passed on. All Modbus/RTU slaves are accessed via the converter, which has only one IP-address. By using the unit id, the converter knows to which slave the message is to be sent (Modbus/RTU devices have no IP-address).

The following picture shows (using the earlier used function 3 example) how a Modbus/TCP message is sent:



Modbus/TCP has no need for a checksum or CRC, as a much better protection against corrupted data is provided by Ethernet (if used), and TCP/IP itself. It is also not needed to delimit any message

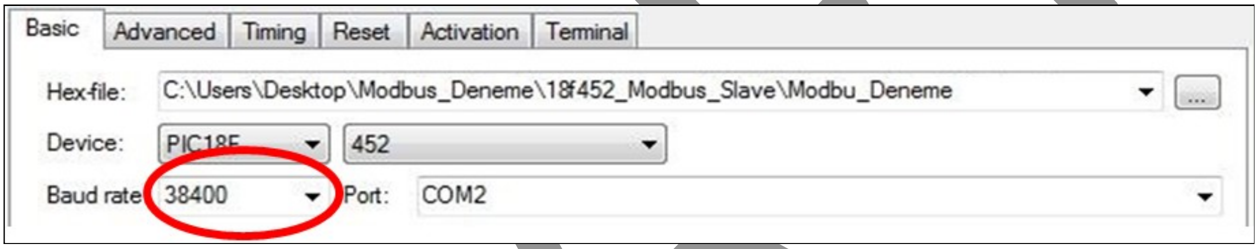
[6] *Many other network protocols using TCP do this too. Only when all messages have the same lengths such a field isn't necessary, but in Modbus messages have no predictable length.*

with special start- or stop-characters or periods of silence, as the Modbus/TCP implementation always 'knows' how much data it can expect.

Same protocol!
It is important to realize that all devices on the network must talk the same protocol, since Modbus/ASCII devices cannot communicate with Modbus/RTU devices (and vice-versa) and with Modbus/TCP devices (and vice versa). Also Modbus/RTU devices cannot communicate with Modbus/TCP devices (and vice-versa), even though the message formats are very similar; a converter is needed[7].

3. Bitrate

When using RS232 or RS485, **all** devices on the same cable must be set to the same bitrate. Which bitrates are supported may differ per device; consult the vendor's documentation for more information. If there is no bitrate common to all devices, a solution could be to have two (or more) separate networks.



Hex-file: C:\Users\Desktop\Modbus_Deneme\18f452_Modbus_Slave\Modbu_Deneme
Device: PIC18F 452
Baud rate 38400 Port: COM2

Furthermore, all devices must also have the same number of data bits (7 or 8), the same parity bit settings (even/odd/none), and the same number of stopbits (1 or 2).

When using Modbus/TCP, bitrate plays no role as this is handled by the underlying physical layer. When this is Ethernet, devices with different bitrates can still communicate with each other as modern Ethernet switches convert bitrates when necessary (between 10, 100, and 1000 Mbit/s). When using wireless Ethernet (WiFi), the access points automatically set the most optimal bitrate depending on the distances between two devices and the wireless signal quality.

4. Serial transmission format

Modbus/ASCII and Modbus/RTU follow the serial transmission format as described in appendix A. In Modbus/TCP, on Ethernet, a byte is just transmitted as: a byte.

5. Parity bit and stop bit(s)

When using RS232 or RS485, all devices on the network must have the **same** setting for the parity bit: none, odd, or even. It is recommended to always use a parity bit, as this increases the reliability of the network due to a better capability to detect damaged bits (at the expense of taking up 10%

[7] *The converter is needed anyway, because of the different wiring.*

more bandwidth). There is no difference in error detection quality between the odd or even parity methods. Most networks use even parity.

Related to the parity setting is the number of stop bits:

- When no parity bit is used, 2 stop bits must be used.
- When using the odd or even parity method, 1 stop bit must be used.

The usage of two stopbits is uncommon in modern serial communications software, and some vendors might not support it. Check in advance whether the equipment/software that you intend to buy supports this

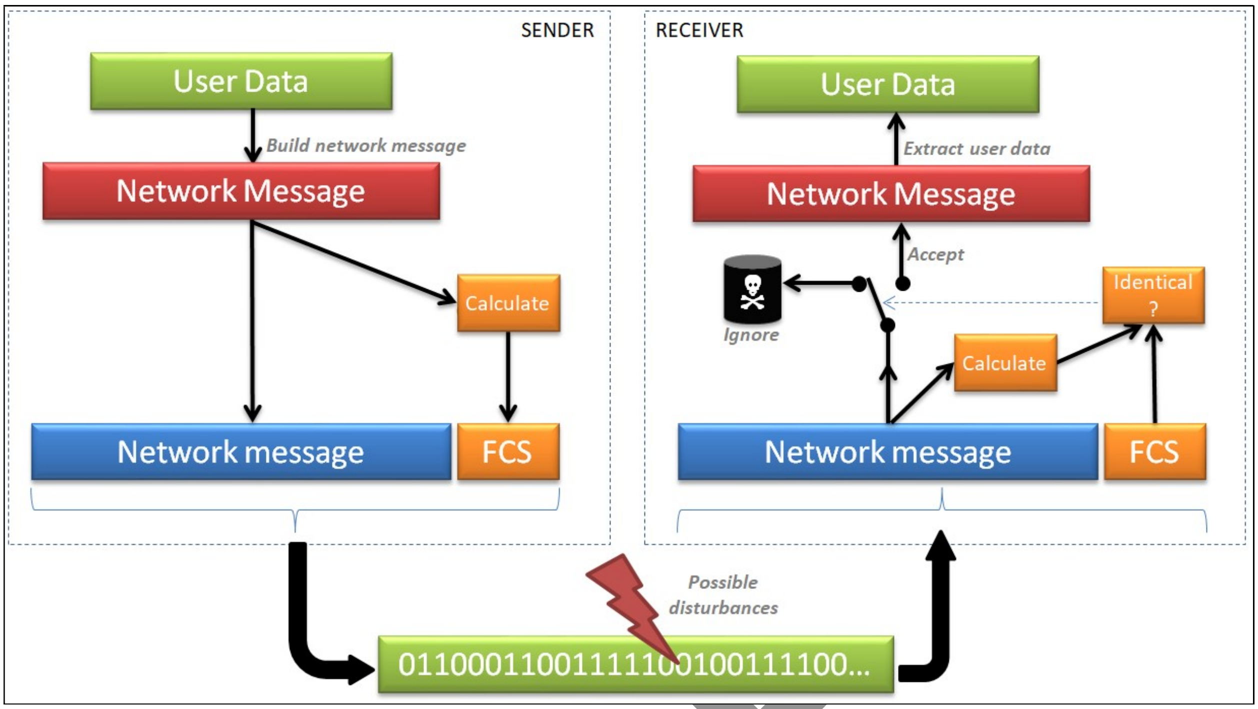
Many vendors also allow the use of ***one*** *stop bit (instead of two) when using no parity. It is recommended to configure the network this way if possible, as it saves one completely unnecessary bit per character (10% less overhead!).*

Modbus/TCP
When using Modbus/TCP, no parity bit is used, this is never used in TCP and also not on Ethernet. TCP has a more advanced "checksum" algorithm (see below).

6. Checksum
A "checksum" is a more advanced method of detection of corrupted bits in a network than parity, because the latter operates on only a single byte, while the checksum takes *all* bytes in a message into account. This means that the complete loss of a single byte, or the insertion of a byte, can also be detected. Additionally, a checksum is able to catch[8] errors that parity cannot: multiple damaged bits in a byte.

> *In the Modbus specification the checksum is called "LRC" (Longitudinal Redundancy Check), to distinguish it from the "Transverse Redundancy Check" (TRC) which is normally called "parity". In this publication, we will use "checksum" and "parity".*

[8] *There is one error a checksum cannot detect: the removal or insertion of bytes in a message with the value zero. This is simply because the value of a checksum doesn't change when (not) adding zero. For this reason, network messages protected by checksums must have an additional measure to detect this, for example using only fixed length messages, or via a "length" field indicating how much data is present in a network message.*



As the name says, it is a “sum”. The values of all bytes in the message are added together. This result (the sum) is then sent too. The receiver of the message does the same addition; if there is no damaged data then the received checksum will have the same value as the recalculated checksum. If the recalculated checksum has a different value, then the message will be rejected.

Example

As an example, suppose that a Modbus message has the following contents:

0A	01	04	A1	00	01

Simply add these 6 bytes (in hexadecimal), the sum is B1. If the value is greater than (hex) 100, subtract (hex) 100, repeatedly if needed. Modbus then specifies to calculate the 2’s complement value (meaning: subtract the value from (hex) 100), giving 4F, and this is then the checksum transmitted:

0A	01	04	A1	00	01	4F

The receiver shall also execute the checksum calculation, and if it gives the same result the message is undamaged.[9]

Error detection quality

An 8-bit checksum can detect the following errors:

- 100% of all network messages with one corrupted bit

[9] *For Modbus implementers: a trick is possible: the receiver can check the correctness of the message by adding the values of all bytes, including the checksum byte. If the result of the addition (modulo 256) is 0, then the message is undamaged.*

- 100% of all network messages with 2, 3, 4, 5, 6, 7 or 8 consecutive corrupted bits (a so-called “burst error”)
- 99,6% (= 255/256) of all network messages with 9 or more consecutive corrupted bits

Modbus/ASCII uses the checksum algorithm because it is easy to calculate. At a time when CPU’s were still quite slow, this was an important advantage over the computationally intensive CRC algorithm of Modbus/RTU (see below). However, the checksum algorithm has it weaknesses, and with modern CPU’s the CRC algorithm processing power is not a problem anymore.

Undetected corrupted data

Note that the checksum algorithm does not detect **all** possible corrupted data, so *sometimes* a corrupted bit will pass the checks anyway and end up in the application, which might then take wrong decisions. Of course the question is then: how likely is this to occur? Once per day would be unacceptable, once per million years would not be a problem. This is called the “residual error rate”.

The standard IEC-60870-5-1 gives some data about this subject, although the algorithm used there is the one as used in (serial) Profibus. But since Profibus also works with parity bits and a checksum, and the use of these doesn’t differ much from the usage in Modbus, the data from the IEC-60870-5-1 is probably a good indication on how Modbus behaves.

The IEC-60870 shifts the problem back to the user. A checksum (or any error detection algorithm) is not a solution for badly wired networks, devices with non-conformant electronics, or networks which have too many electronic disturbance sources near them (such as: frequency converters, electric motors, soft starters, welding equipment, etc.). If this causes too much corrupted bits, the likelihood that a checksum will **not** detect a corrupted bit increases. Two examples (from the IEC-60870-5-1):

- Suppose that 1 bit per 100000 is disturbed. The likelihood that one such disturbed bit is **not** detected is: $5{\cdot}10^{-17}$. Multiply this by the bitrate of the network, say 115200 bit/s, and then the likelihood that a bit is disturbed in a second is $5{,}76{\cdot}10^{-12}$, or: once per 5505 years.. One undetected error per five thousand years is probably a very acceptable figure.

- Now suppose that 1 bit per 10000 is disturbed. The likelihood that this is not detected is: 10^{-9}, which translated to (@ 115200 bits/s) to once per 200 days. This is an unacceptable figure.

The calculation shows that a small deterioration of network quality can have a big effect on application reliability. Conversely, a network with fewer disturbances will function even better. The example disturbance probability of 1 bit per 100000 is already good; when there are even less disturbances the residual error rate decreases very quickly: 10 times less errors is a 10 times less undetected errors. For example, a network with 1 disturbed bit per 10 million will have a residual error rate of once per 550500 years[10].

[10] *I often got complaints from users who could not explain strange errors in their data. This is then often attributed to “corrupted data” on the network (“It’s not my fault”). Normally I don’t believe such explanations, given the data above. After investigation, it is usually a programming error.*

Modbus/TCP

Modbus/TCP does not use a checksum, as TCP does this already: it uses a 32-bit checksum internally. When corrupted data is detected, TCP will execute repair actions automatically; this is not even visible for the application layer.

7. Cyclic Redundancy Check (CRC)

The Cyclic Redundancy Check (CRC) is much better in the detection of transmission errors than the checksum algorithm. Most modern industrial networks therefor use a CRC.

The value of the CRC is calculated on the message data in the Modbus message, just as the checksum is calculated. It is then appended to the network message. The receiver of the message does the same CRC calculation, and compares the result with the received CRC. If there are no transmission errors, then the received CRC will have the same value as the recalculated CRC. If the recalculated CRC has a different value, then the message will be rejected.

It could also happen that a transmission error occurs in the CRC value sent in the network message. What happens next at the receiver of this message[11]?

The mathematics behind a CRC calculation is too complex to explain here in detail, we will use an analogy[12] instead. Basically what is done is to make one large binary number of all the data in the network message, for example when there are 20 bytes of data then this is a 20*8=160 bit binary number. Next we do a longhand division: our 160 bits divided by a 16-bit number which has been specially selected (the so-called "syndrome"). When the division is finished, there will be a remainder, and this is our CRC value which is appended to the network message.

The receiver of the network message will do the same longhand division. When there is a transmission error in one of our 160 bits, then there will be a different remainder.

A CRC calculation is very CPU-intensive; a badly written implementation will slow down the system. The Modbus specification gives an example of an implementation in the C programming language, which is much more efficient than longhand division algorithms.

Modbus/RTU uses a 16-bit CRC algorithm, mathematically described as $x^{16}+x^{15}+x^{2}+1$ or as "polynomial 8005", and colloquially known as "CRC-16", "CRC-16-IBM" or "CRC-16-ANSI". There are many more CRC-algorithms working on 16 bits data, but Modbus only uses the one according to the above mathematical description.

[11] *The answer is: there is no match between the calculated CRC and the received CRC, so the user data is considered to be damaged, while in fact it isn't. But there is no way to know this! (unfortunately).*

[12] *Explained in more detail on https://en.wikipedia.org/wiki/Cyclic_redundancy_check.*

I could not find data about the error detection qualities of the Modbus CRC-16 algorithm, as comparison therefore some data about the CCITT-16 algorithm, which can detect:

- Any arbitrary combination of an odd number of corrupted bits;
- Any arbitrary combination of two corrupted bits;
- At least 99,8% of any arbitrary combination of 4 or 6 corrupted bits;
- At least 99,99% of any arbitrary combination of 8 or 10 corrupted bits;
- At least 99,999% of any arbitrary combination of 12, 14 or 16 corrupted bits;
- All consecutive 16 corrupted bits;
- 99,9985% of more than 16 consecutive corrupted bits.

This shows us that *sometimes* a damaged bit passes all security checks, and will end up in the application software. As already discussed earlier (checksum quality) the likelihood that a damaged bits slips by is very, very small, with CRC much smaller than with a checksum.

Modbus/TCP

When using Modbus/TCP with Ethernet as physical layer, network messages are also protected by Ethernet's 32-bit CRC (on top of the TCP checksum). This comes in place of the parity bit and the Modbus/RTU CRC. This is why the Modbus/TCP messages do not contain a CRC like in Modbus/RTU; it would not add any extra error detection quality, and would only increase the processing load on the CPU due to the double work.

8. Error detection

The detection of errors in messages is done differently in Modbus/ASCII and /RTU, and Modbus/TCP.

Modbus/ASCII and / RTU detect errors at three levels:

- Per individual character in a message (via parity, see above).
- The amount of idle time between two successive characters in a message
- The "checksum" or the "CRC" in each message.

Per character, the following checks are performed:

- Each character must start with a start-bit of value '0'.
- The number of data bits that follow (7 or 8) must be correct.
- The received value for the parity bit must have the same value as the locally recalculated value of the parity, based on the data bits' values.
- The stop bit(s) must have a bit value of '1'.

Normally these checks are performed by the serial I/O controller chip, a "UART" (Universal Asynchronous Receiver Transmitter). If a problem is detected, a "framing error" or a "parity error" is given. The consequence is that the data is discarded, a character is missing from the network message, and this in turn will cause the entire message to be discarded.

At the 2nd level, the characters in a message must follow each other back-to-back. After the stop bit of the first character, ideally immediately the start bit of the next character follows. But especially on older (slower) CPU's and UART's this is impossible to achieve. Modbus allows for a short pause between two subsequent characters, but not more than 1.5 character transmission time.

At the 3rd level, the checksum (or CRC) is calculated on the received data. The result of this calculation must match the checksum (or CRC) as received in the message. If they do **not** match, there is an error somewhere (Modbus doesn't know where), so the whole message is discarded. You also do not get an error message reported back. Perhaps the device has a diagnostic counter that increments, so you can see later how much of this type of error has occurred. Modbus has a special "function code 8" that can be used[13] to read out diagnostics counters.

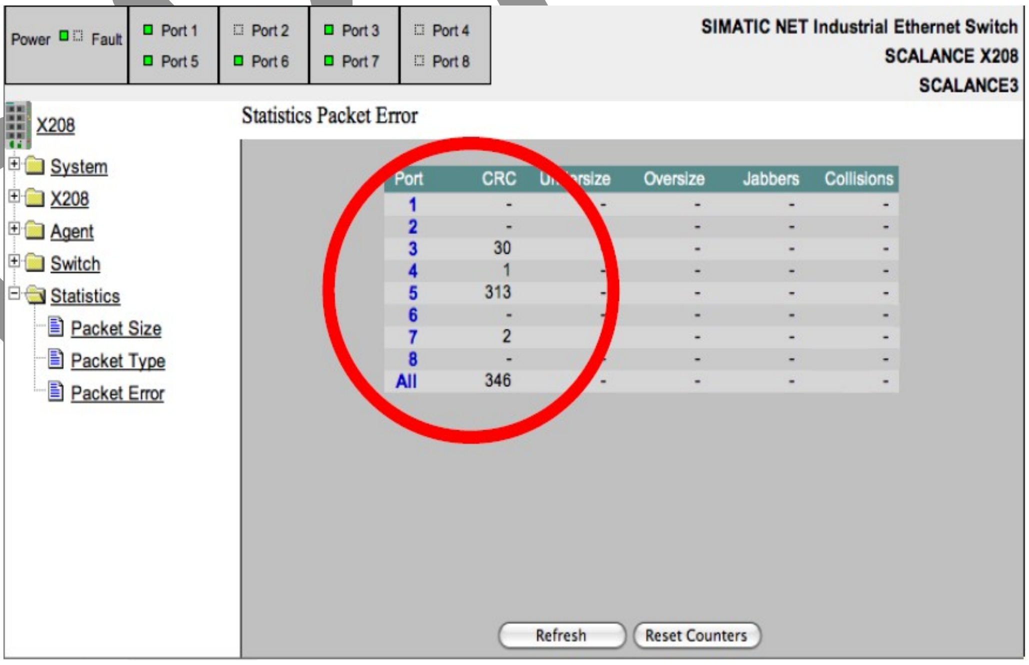
If the message passes all these checks, it is passed on to the application layer. This layer will execute its own checks on the contents of the data in the message. Any error found here is not caused by problems on the network, but usually by programming errors in the application software.

Modbus/TCP

In Modbus/TCP, errors are detected and handled differently.

If Ethernet is used at the lowest level, damaged bits in network messages are detected by Ethernet's usage of a CRC (its own). When the message is damaged, this is detected by the receiving device's Ethernet controller, and not further processed. This can also occur in any intermediate switch (or router) between the sending and receiving device.

When managed switches are used, a diagnostics counter will be incremented; its value (and many other diagnostics) can be read out via the SNMP protocol (Simple Network Management Protocol).



SIMATIC NET Industrial Ethernet Switch
SCALANCE X208
SCALANCE3

Statistics Packet Error

Port	CRC	Undersize	Oversize	Jabbers	Collisions
1	-	-	-	-	-
2	-	[illegible]	-	-	-
3	30	[illegible]	-	-	-
4	1	-	-	-	-
5	313	-	-	-	-
6	-	-	-	-	-
7	2	[illegible]	-	-	-
8	-	-	-	-	-
All	346	-	-	-	-

[13] *Unfortunately it is often not implemented.*

A managed switch with an embedded webserver can show statistics counters on the number of errors, such as CRC errors, per port. Here port 5 would be a prime candidate for further investigation.

If the message arrives undamaged at the final destination, it is passed on to TCP/IP for further processing. Both IP and TCP have their own error-detection algorithms. Errors at this level are of another category, for example:

- Messages received in a different order than transmitted (this will be corrected);
- Messages completely missing (see below);
- Duplicated messages (these are filtered out);

Whenever a message is discarded by Ethernet, TCP will detect this, and ask the original sending device to retransmit the message. This is completely handled autonomously by TCP; the application software on the original sending device will be unaware that this has happened.

TCP will attempt to repair any error it sees, and may do so for several minutes before giving up and reporting an error. When TCP does so, be assured that it has done the utmost to repair any problem, and little else can be done to improve on this.

This behaviour of TCP/IP may sometimes cause large delays in message processing, causing the application software to stall [14] *for several minutes too (if this is not properly programmed).*

As TCP is so good in repairing network problems, it makes no sense for the application software to try to improve on this. Nevertheless, many Modbus/TCP implementations are actually rewritten Modbus/RTU implementations, and you still get retries.

If the message passes all these checks, it is passed on to the application layer, and handled the same as in Modbus/ASCII and /RTU (see above).

[14] *I have seen machines stall for 12 minutes while the protocol stack was busy repairing unrepairable communication errors before finally reporting a failure. This was on a WindowsXP machine using DCOM, which seemed to double any TCP timeout "just in case". After having experienced this a few times, the operators just power-cycled the machine, this made the machine working again in 3 minutes.*

CHAPTER 6. APPLICATION LAYER

Before we can discuss the application layer functionality of the Modbus protocol, it is necessary to understand how a Modicon PLC works. Even though most Modbus users never physically encounter a real Modicon (or its successor: Schneider) PLC, the Modbus protocol still reflects how such a PLC is interfaced. The original Modbus specifications [MBUS300] assumed Modicon knowledge to be present in the reader, in [MBUSAPPL] more is explained about the Modicon PLC memory model.

1. Modicon PLC functioning and interface

A PLC (Programmable Logic Controller) is a generic electronics device used in industrial automation, which can be programmed to handle a specific task automatically and repeatedly. It has 'sensors' with which it can know the state of the outside world, and 'actuators' to influence it.

Sensors can be of a 'digital' type, or an 'analogue' type. Digital sensors can only have two states: on/off, high/low, open/close, slow/fast, stop/move, 0/1, yes/no, left/right, etc. Analogue sensors usually measure a physical value, i.e. a temperature, voltage, pressure, intensity, flow, RPM, etc. of which there can be an infinite number of values.

Actuators also exist in 'digital' and 'analogue' forms. Digital actuators have two states, and are used to control equipment: a motor turning or stopping; opening or closing a door or valve, switching a device on or off, etc. An analogue actuator can be used on a device to set a certain RPM, pressure on a pump, voltage on a circuit, programming a heater or AC, etc.

A Schneider PLC, with from left to right: a power-supply, the PLC itself, I/O modules and network interfaces. Source: Schneider Electric.

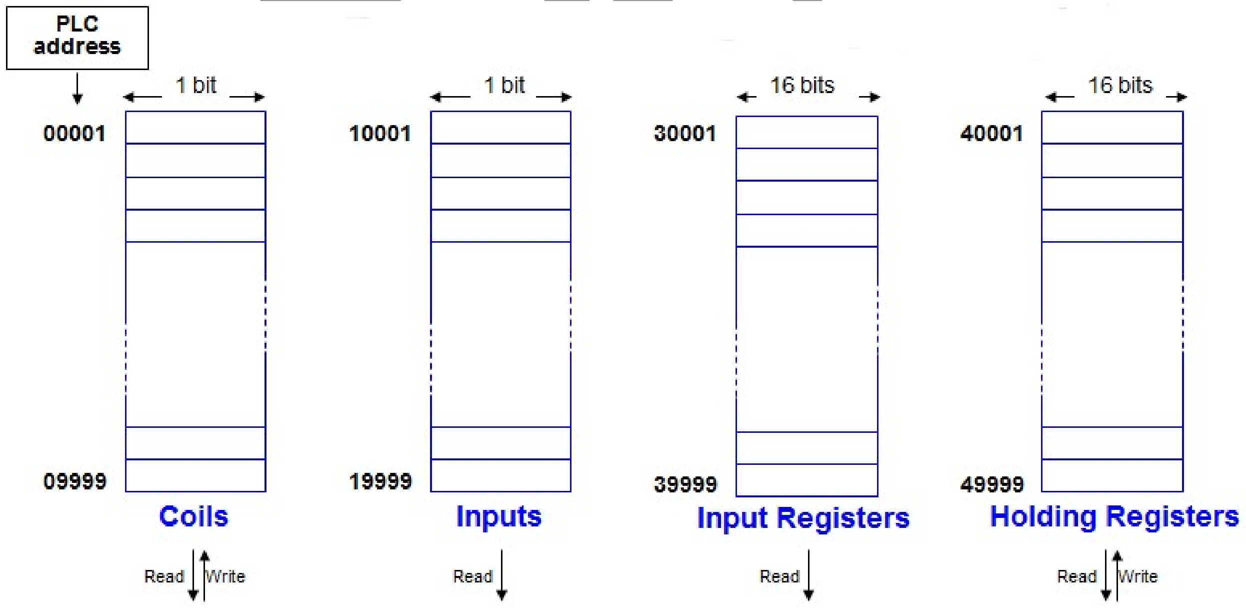
A PLC contains the program that reads in the current state of all its sensors, decides what state the system is in and what step should be taken next, and uses this knowledge to program the actuators. In effect, the PLC is controlling a machine or production line. PLC's are programmed for their task by a software-engineer on a desktop-PC or laptop. Via a network connection the program is then loaded into the PLC, after which it can function autonomously.

The processing capacity of a PLC is limited, and for larger machines one PLC cannot control the whole machine. Multiple PLC's are then used, each one controlling a part of the system. But, since they are controlling the same machine, they must coordinate their efforts. An "industrial network" is then used to allow the PLC's to communicate with each other. Modbus is a well-known example of such a network (there are a few hundred others).

Industrial networks (and Modbus) are also used to connect PLC's to so-called "visualisation systems" or "human machine interfaces". Often these are PC's with graphics programming software which shows the operators the state of the machine, what products it is making, whether the machine is up to speed, where something goes wrong, etc. The visualisation system is able to draw these diagrams based on information extracted from the PLC's.

A Modicon PLC stores the current status of all its digital inputs and outputs, and its analogue inputs and outputs, in its internal memory. Additionally, a PLC has scratchpad memory that can be used for storage of intermediate results. And finally the PLC has memory in which its program is stored.

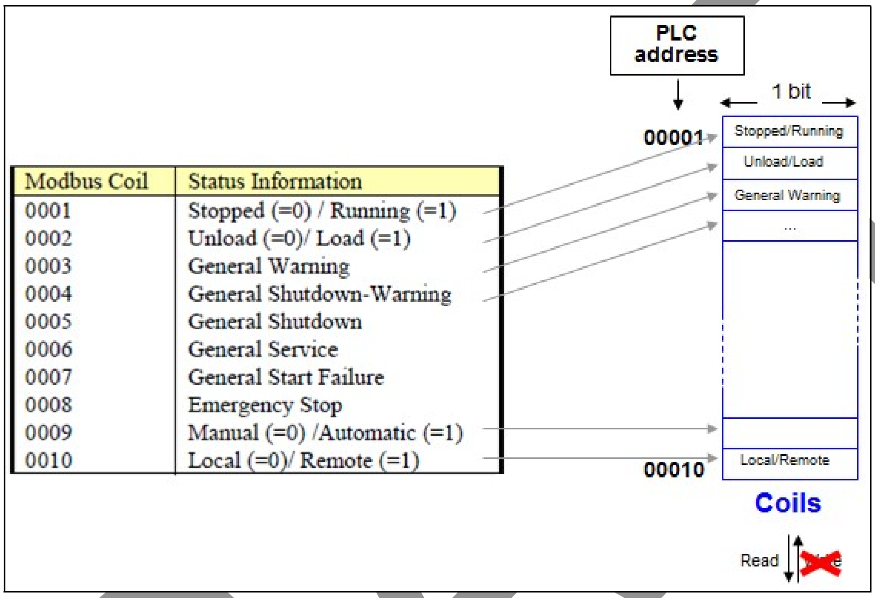
A Modicon PLC has four memory areas: "coils" (= digital outputs), "inputs" (digital), "input registers" (analogue) and "holding registers" (analogue outputs and scratchpad memory). Coils, inputs and registers are numbered starting from 1, up to 9999 in older devices, or up to 65535 in modern devices. Vendors often implement devices which have less than this number, i.e. when the device does not have so many coils, inputs or registers.



The Modicon memory model is old-fashioned. Modern PLC's operate according to the IEC-61131 standard, which has storage for inputs (of any type) called %I, outputs (of any type) called %Q, and scratchpad memory called %M. When such a PLC has a Modbus interface, the IEC-61131 style memory is mapped onto the Modicon PLC memory areas according to a vendor-specific way; consult the vendor's documentation for this.

What happened to memory area 20000 – 29999? This memory area was used by Modicon PLC's with so-called "sequencers". When later in Modicon 484 PLC's this functionality was obsolete, this memory area was not re-used for other purposes for compatibility reasons.

Example:



Modbus Coil	Status Information
0001	Stopped (=0) / Running (=1)
0002	Unload (=0)/ Load (=1)
0003	General Warning
0004	General Shutdown-Warning
0005	General Shutdown
0006	General Service
0007	General Start Failure
0008	Emergency Stop
0009	Manual (=0) /Automatic (=1)
0010	Local (=0)/ Remote (=1)

An example of vendor documentation describing the functionality of the first 10 coils. This vendor does not support the function code to set coils.

Input registers and holding registers are 16 bit in size. At the time the Modbus specification was written, this was large enough to hold number in the range 0..65535 or -32768..32767. Nowadays, 32-bit numbers and even 64-bit numbers are customary. This sometimes gives problems; see chapter 6 for more information about this.

*Some master devices **disallow** access to bits or registers with addresses > 9999. This is very old-fashioned behaviour, and anyway such checks should be implemented on the slave devices, not on the master device.*

2. Bit / register addressing

Modbus employs a confusing addressing scheme for all bits and registers. As mentioned above, all bits and registers are numbered from 1..9999. To this an offset is added which specifies the memory area:

Coils	Offset 0
Inputs	Offset 10000 (decimal)

Input Registers	Offset 30000
Holding Registers	Offset 40000
Extended Memory	Offset 60000

For example, holding register 56 may also be called 40056, and 34567 is the same as input register 4567. Conversely, if documentation mentions an address 45678, you know automatically that it is holding register 5678.

As we will see later, in the network messages themselves the offset is **never** used – it is the combination of function code and offset that determines the real bit / register address. So you see values like 56 and 4567. Additionally, Modbus always subtracts 1 from these values, and this result is stored in a network message. For example, register 34567 is coded in the network message as 4567.

Which addresses must be used in the application software depends on how the vendor wants this. Some want you to use the offsets; if you want to read holding register 56 you must program 40056, and the software subtracts the 40000 for you. Other vendors allow you to use address 56, but then you must also instruct the software what you mean: a coil, an input register, an output register or a holding register?

A problem arises when using addresses >= 10000: does value 40050 correspond to input register 10050 (30000 + 10050), or holding register 50 (40000 + 50)? To prevent this overflow problem, "**extended addressing**" is used with the following offsets:

Coils	Offset 0
Inputs	Offset 100000 (decimal)
Input Registers	Offset 300000
Holding Registers	Offset 400000
Extended Memory	Offset 600000

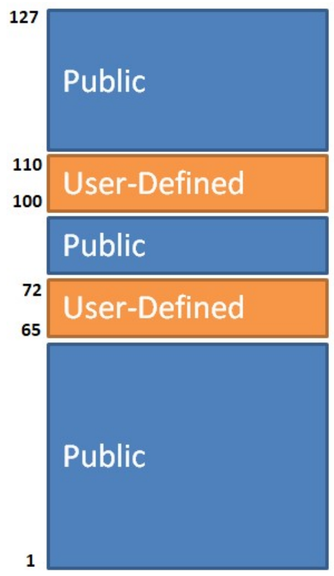
For example, holding register 567 may also be called 400567, and 300987 is the same as input register 987 (and 30987). As described above, these offsets are **never** used in network messages.

Whether application software wants to use offset 10000/30000/40000 or 100000/300000/400000, depends on the vendor.

In documentation, some vendors use the offsets, but others do not. The same holds for software packages: sometimes you must program without offsets, and sometimes they must be added (after which the software package subtracts the offset immediately before sending it out in a network message).

3. Function codes

Commands that can be sent over a Modbus network are not called like that; instead they are called "function code" (fc). This is real Modbus jargon, used often in documentation, in software etc., so we will use that terminology too.



Which function codes exist is described in the Modbus specification (explained in more detail below). These are called "public" function code, for example those in range 1..64, 73..99 or 111..127. There are also a few 'user-defined' function codes, which can be used for very user-specific functionality. The reason for allocating a few ranges of function code is that users know there are available, and that Modbus won't use them in the future.

Having 'user defined' function code means in practice that both the master/client and slaves/servers must come from the same supplier. It doesn't make sense to sell a slave with support for a certain function code, but is controlled by a master who does not have that function code.

Apart from the memory access function codes, there are also function codes for local loopback and diagnostics, and for Modicon PLC programming and control[15].

The maximum amount of application data that can be read or written in a single network message is: 250 bytes (125 registers, 2000 bits). Sometimes a lower limit of 240 bytes (120 registers, 1920 bits) is seen, as this was mentioned in the original Modbus specification [MBUS300] and many vendors copied that.

4. Function code numbering

All function codes have a number in the range 1..127. These numbers are fairly well-known within the Modbus community, and after a while one knows them by heart. Customers will **always** need to know which function codes are implemented in equipment.

The table below gives the (decimal) numbers + descriptions of the known function codes. Note the Modicon PLC specific jargon: “coil”, “input register”, “holding register” and “output register”. Note that some function codes have multiple names, for example one from [MBUS300] and one from [MBUSAPPL].

FC	**Description (and alternative names)**
01	Read Coil Status / Read Coils / Read Output Status
02	Read Input Status / Read Discrete Inputs
03	Read Holding Registers
04	Read Input Registers
05	Force Single Coil / Write Single Coil
06	Preset Single Register / Write Single Register
07	Read Exception Status
08	Loopback Test / Diagnostic
09	Program 484 (Modicon 484 only)
10	Poll 484 / Poll Program Complete (Modicon 484 only)
11	Fetch / Get Communications Event Counter
12	Fetch / Get Communications Event Log
13	Program (Modicon PLC only)
14	Program Controller / Poll Program Complete (Modicon PLC only)
15	Force Multiple Coils / Write Multiple Coils
16	Preset Multiple Registers / Write Multiple Registers
17	Report Slave ID / Report Server ID
18	Program 884/M84 (Modicon PLC only)
19	Reset Communications Link
20	Read General Reference (Modicon PLC only) / Read File Record
21	Write General Reference (Modicon PLC only) / Write File Record
22..64	**Reserved for future extensions**
22	Mask Write 4X Register / Mask Write Register
23	Read-Write 4X Registers / Read-Write Multiple Registers

[15] *The author has never seen these function codes used, as they are too specific for that brand of equipment.*

24	Read FIFO Queue (Modicon PLC only)
43	Read Device Identification / Encapsulated Interface Transport / CANOpen General Reference
65..72	**Free for users**
73..99	**Reserved for future extensions**
90	Tunneling of the UMAS protocol
91	Used by the SEMI extensions
100..110	**Free for users**
111..124	**Reserved for future extensions**
125..127	Reserved

The Modbus User Group has listed which function codes are to be used for what purposes. The latest version of the Modbus specification [MBUSAPPL] is clearer about this than earlier versions of the specification [MBUS300], which mainly reserved all function codes for Modicon.

In practice this didn't make much sense, since there is no "Modbus Police" which checks Modbus implementations worldwide for violations of the specification, so vendors did what they wanted anyway. With the current specification, the Modbus User Group specifically allows for user-defined function codes in two clearly defined regions, which decreases the likelihood that a future Modbus extension clashes with user-defined function codes.

Function codes 65..72 and 100..110 are reserved for user-defined function codes, for example when a certain type of application want to do something very special that is impossible to do with the standard function codes. Some vendors make good use of this, such as Schneider with function code 90 for their UMAS protocol. Note that for these to work, both the master AND the slave(s) must support those function codes, in practice this means that both will come from the same vendor.

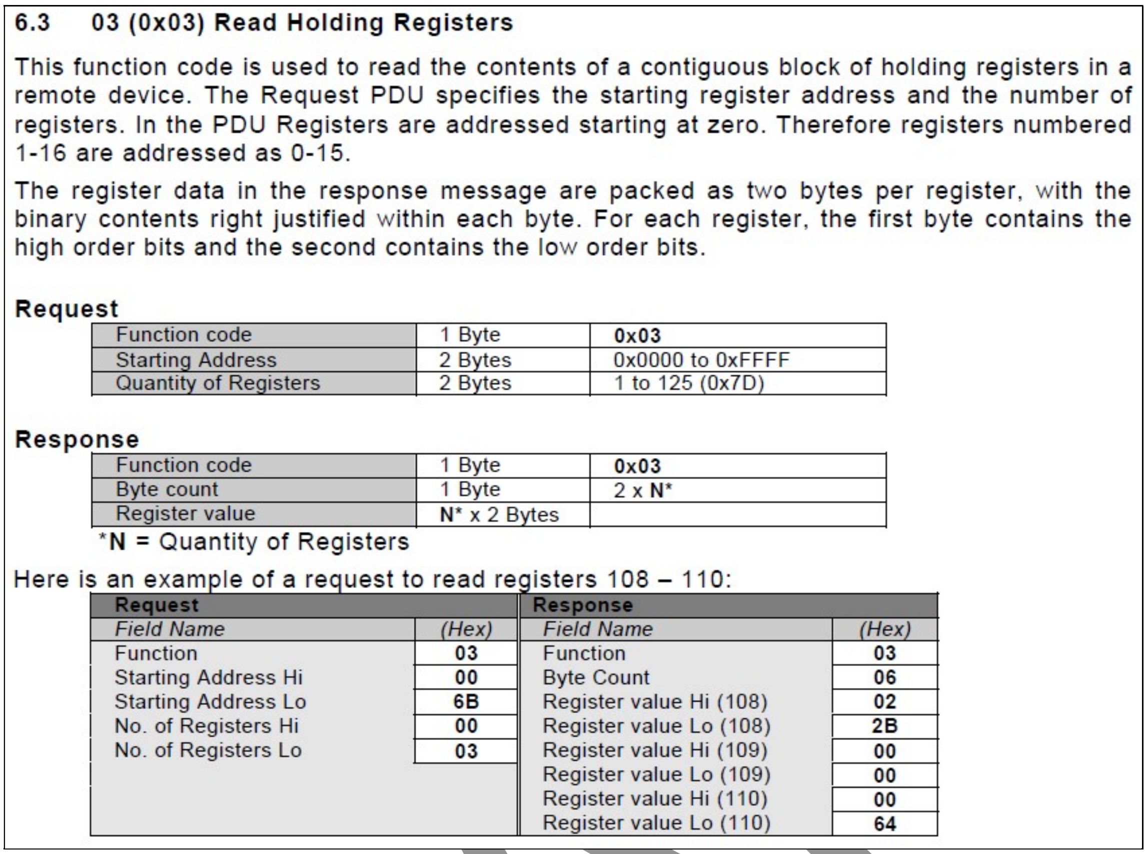
There is no central registration of user-defined function codes, so it could happen that two vendors both use the same function code (say, 73) but their implementation may be completely different. So the advice is that, whenever a user-defined function code is to be used in a project, careful analysis is done of the implementations.

Function codes 125..127 are, according to the Modbus specification reserved for future extensions, but in appendix A in the specification it is written that they are 'specifically' reserved, but it does not mention for what purpose (or for whom)[16].

What does a function code do?

Per function code the specification describes what it does, how the request network message is built, how the response message is built, which errors can be given, and an example. This makes the Modbus specification very easy to read and understand. An example:

[16] *In older literature it appears that these function codes were once used for reprogramming activities of Modicon PLC firmware.*



6.3 03 (0x03) Read Holding Registers

This function code is used to read the contents of a contiguous block of holding registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore registers numbered 1-16 are addressed as 0-15.

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.

Request

Function code	1 Byte	**0x03**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 125 (0x7D)

Response

Function code	1 Byte	**0x03**
Byte count	1 Byte	2 x **N***
Register value	**N*** x 2 Bytes	

*__N__ = Quantity of Registers

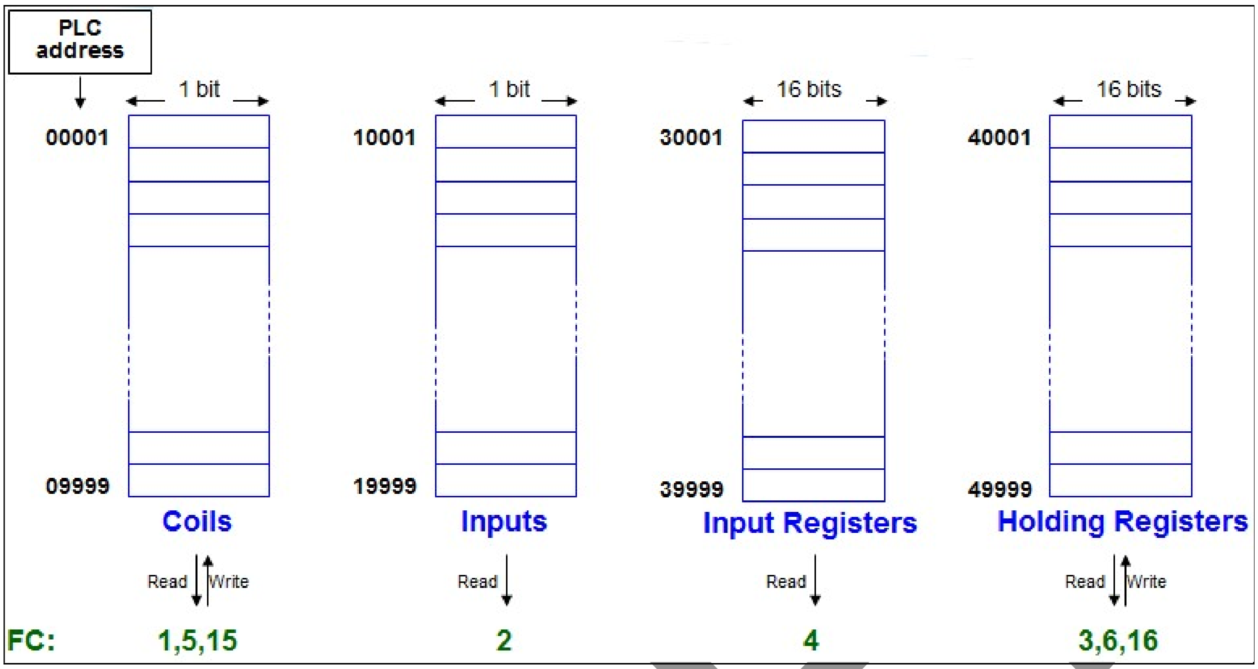
Here is an example of a request to read registers 108 – 110:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**03**	Function	**03**
Starting Address Hi	**00**	Byte Count	**06**
Starting Address Lo	**6B**	Register value Hi (108)	**02**
No. of Registers Hi	**00**	Register value Lo (108)	**2B**
No. of Registers Lo	**03**	Register value Hi (109)	**00**
		Register value Lo (109)	**00**
		Register value Hi (110)	**00**
		Register value Lo (110)	**64**

An example taken from the Modbus specification about function code 03 (Read Holding Registers). It shows the description, the format of the "request" being 5 bytes in size, and the "response" format being 2 bytes plus a variable part depending on the amount of registers needed.

5. Relation to the Modicon PLC model

For the most commonly used function codes, their relation to the Modicon PLC model is as follows. Each memory area has its own specific function codes. Of course, input memory areas can only be read (with function codes 2 and 4).

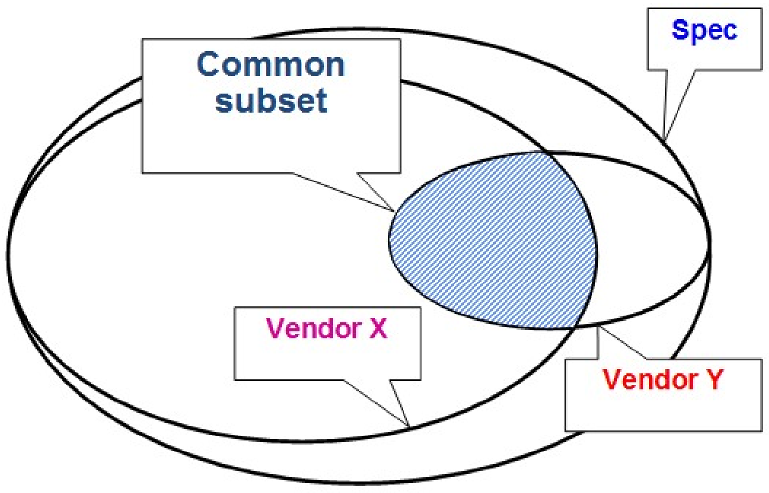


Digital outputs (coils and holding registers) can be set, but their current status can *also* be read back (what did I set that output to again?). PLC programmers like this feature, so their programs do not have to store the state of each coil in scratchpad memory for later use.

The ability to set coils via Modbus is potentially very dangerous, as the PLC-setting of a coil is then modified without the PLC application's knowledge. Many vendors therefore do ***not*** *support function codes 05 (Force Single Coil) or 15 (Write Multiple Coils).*

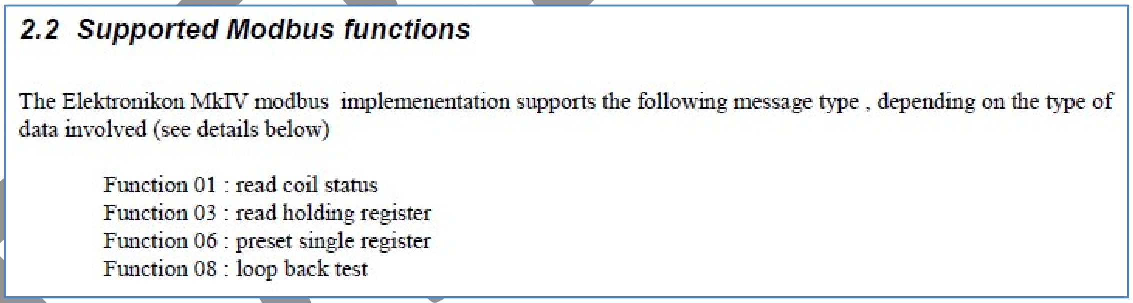
6. Vendor choice

The list of all possible function code in Modbus is: just paper. It is a specification of what *can* be used in a Modbus device, not what *must* be implemented. Any vendor may decide for himself which function codes are useful for the equipment he makes and sells; when there is no need for having certain function codes they can be left out. So the simplest Modbus device has to support one function code only. This freedom holds both for master devices and slave devices.



Because a Modbus network can use a function code only when both the master is able to send it **and** the addressed slave is able to execute it, any Modbus user should always check in advance whether there is a match in supported function codes. If there is no match, the function code cannot be used in an application. Only the common (overlapping) subset can be used.

The place to check this is always the documentation (manual) of a device, which usually lists which function codes are implemented. Usually they are given by their number, sometimes a name, but since the name is not standardized it is best to always use the function code numbers. In practice we see that function codes 03 and 16[17] are almost always implemented, since these two function codes correspond more-or-less to the 'read' and 'write' functions that almost every device needs.

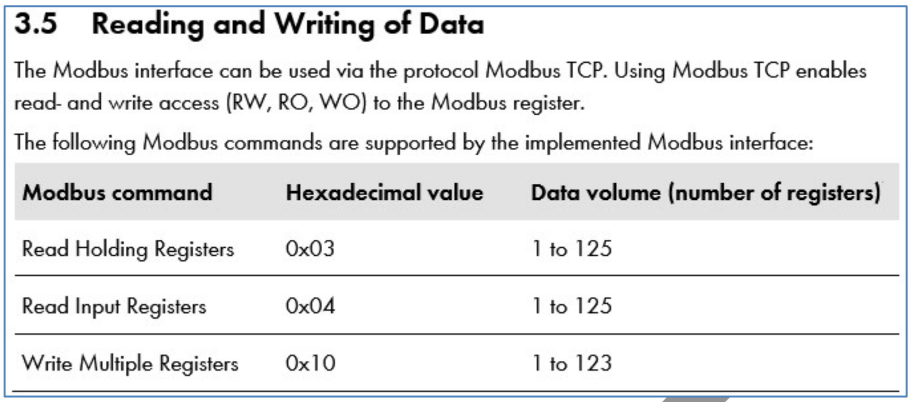

2.2 Supported Modbus functions

The Elektronikon MkIV modbus implemenentation supports the following message type , depending on the type of data involved (see details below)

Function 01 : read coil status
Function 03 : read holding register
Function 06 : preset single register
Function 08 : loop back test

An example taken from a vendor's documentation listing which function codes are supported. Source: Atlas-Copco.

[17] *Some vendors mention the supported function codes with their hexadecimal value, so this would be 10. There is no reason to get confused, since function code 10 (decimal) is only available on Modicon PLC's (which are no longer sold).*

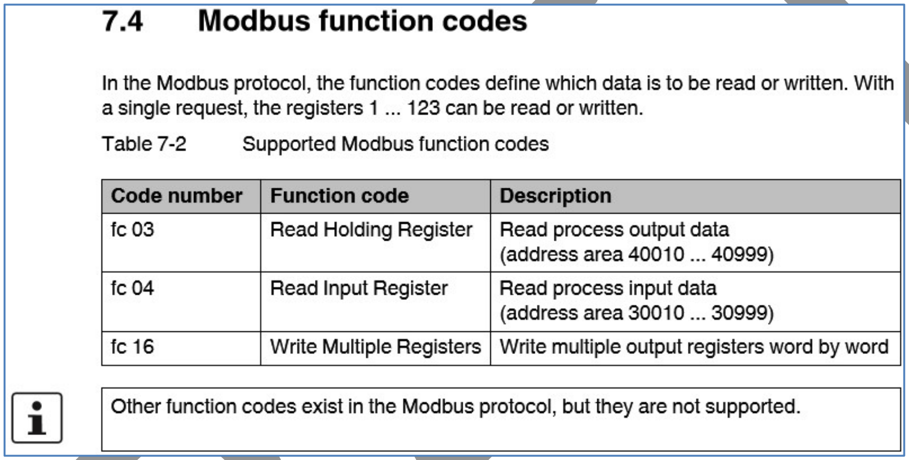

3.5 Reading and Writing of Data

The Modbus interface can be used via the protocol Modbus TCP. Using Modbus TCP enables read- and write access (RW, RO, WO) to the Modbus register.

The following Modbus commands are supported by the implemented Modbus interface:

Modbus command	Hexadecimal value	Data volume (number of registers)
Read Holding Registers	0x03	1 to 125
Read Input Registers	0x04	1 to 125
Write Multiple Registers	0x10	1 to 123

Another example taken from a vendor's manual listing which function codes are supported. Note the hexadecimal notation for function codes 3, 4 and 16. Source: SMA


7.4 Modbus function codes

In the Modbus protocol, the function codes define which data is to be read or written. With a single request, the registers 1 ... 123 can be read or written.

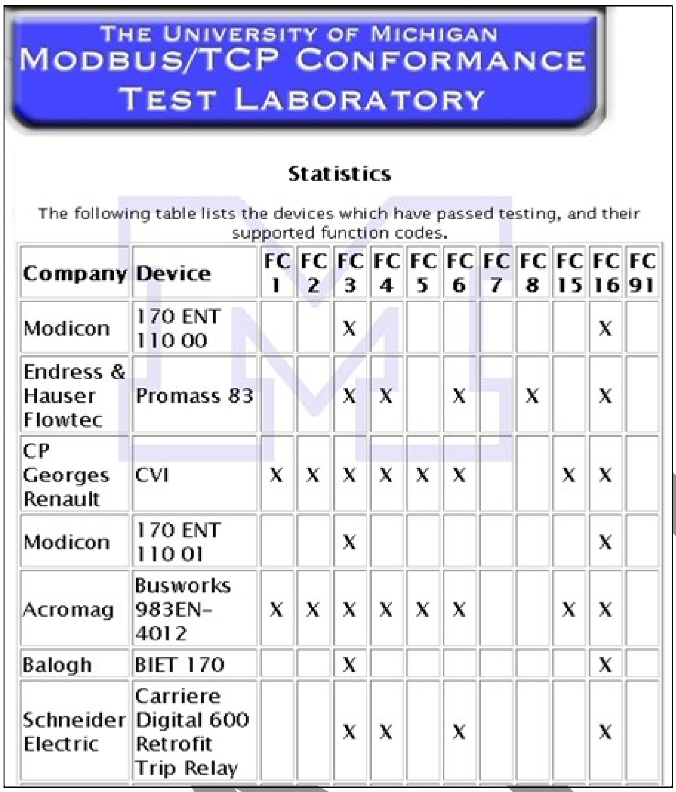
Table 7-2 Supported Modbus function codes

Code number	Function code	Description
fc 03	Read Holding Register	Read process output data (address area 40010 ... 40999)
fc 04	Read Input Register	Read process input data (address area 30010 ... 30999)
fc 16	Write Multiple Registers	Write multiple output registers word by word

Other function codes exist in the Modbus protocol, but they are not supported.

Yet another example from a vendor's manual listing which function codes are supported, and clearly warning the user that other function codes are NOT supported. Source: Phoenix Contact.

Differences between vendors can be substantial. The following table lists the devices from various vendors and the function codes each device supports.


THE UNIVERSITY OF MICHIGAN
MODBUS/TCP CONFORMANCE
TEST LABORATORY

Statistics

The following table lists the devices which have passed testing, and their supported function codes.

Company	Device	FC 1	FC 2	FC 3	FC 4	FC 5	FC 6	FC 7	FC 8	FC 15	FC 16	FC 91
Modicon	170 ENT 110 00			X							X	
Endress & Hauser Flowtec	Promass 83			X	X		X		X		X	
CP Georges Renault	CVI	X	X	X	X	X	X			X	X	
Modicon	170 ENT 110 01			X							X	
Acromag	Busworks 983EN-4012	X	X	X	X	X	X			X	X	
Balogh	BIET 170			X							X	
Schneider Electric	Carriere Digital 600 Retrofit Trip Relay			X	X		X				X	

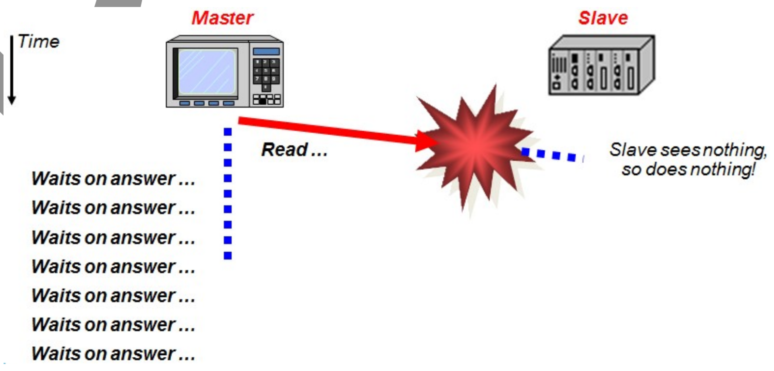
Example from the (no longer operational) Modbus conformance test lab at the University of Michigan, showing the differences in supported function codes in various devices.

7. Error handling on the master

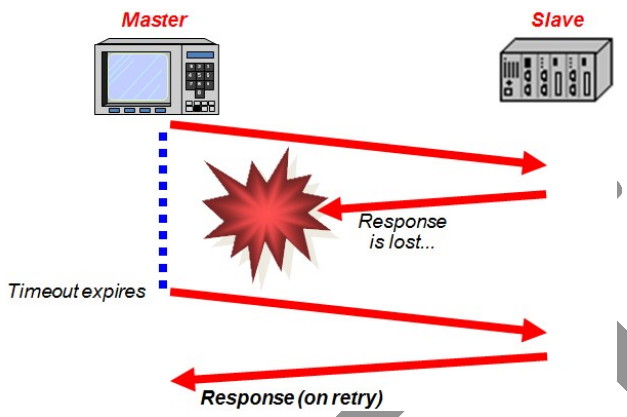
In the previous chapter, we've discussed the parity bit, the checksum and the CRC as three methods to detect damaged bits in a network message. Such network messages are summarily discarded by every receiver, as we only know that *something* is wrong, but we do not know *where*.

After this error-detection follows **error-reparation**, this is the task always done by the master. There are three scenarios:

1) The master's message to a slave is damaged, the slave ignores it. This means that the slave doesn't execute the function code, and will send nothing back to the master.



2) The master's message to a slave is received without errors. This means that the slave will execute the function code, and send an answer message back to the master, but then this message is damaged. The master will ignore it.



3) The slave to whom a message is destined is not operational / not started / has crashed, not connected to the network, or not properly configured. Nobody will execute the function code, and nothing will be sent back to the master.

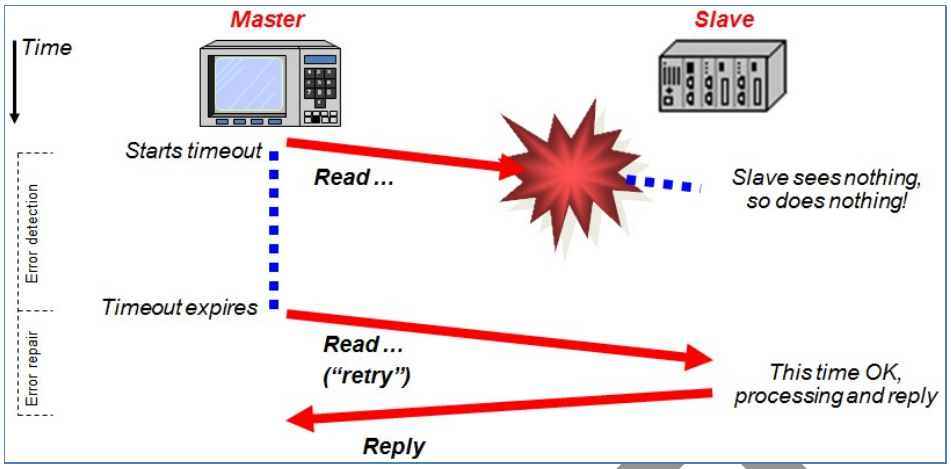
In all these three scenarios', the result is that the master does not receive a good answer message back. Now the master/slave protocol requires that the master *must* wait for an answer. It will be clear that this is not a clever way of working, since this will cause the master to wait forever. The network then stops completely[18].

Stupid as this may sound, there are masters that work this way!

A more clever way for the master is to wait for a short while ("perhaps the reply will come if I'm patient"), give an error to the application software, and then continue. In most implementations the amount of time to wait is called the "timeout period", which can be configured / set / programmed in the master implementation.

An even more clever way to handle errors on the master is not directly give an error, but re-send the original message, in the hope that this time it will not be corrupted. When the slave now receives the message without corrupted data, it can be processed on the slave, after which the response is sent back. With this so-called "retry" the master has automatically detected **and** repaired the error, without the application having to be involved.

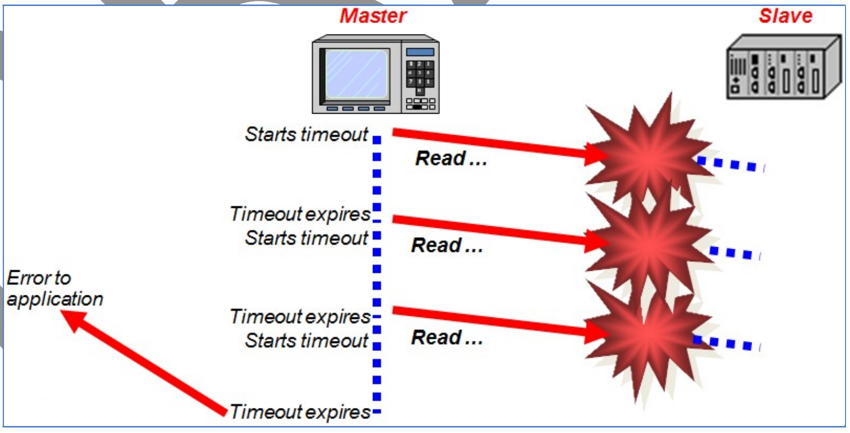
[18] *As errors always occur on any network (sooner or later), this is ticking time bomb. Reboot everything!*



Retries are specified to be done in every network protocol, except… Modbus! In the Modbus specification there is not a single word about this. But because retries are so useful, vendors add this capability to their master implementation. But as there is no standard, each vendor does it in his own way; master implementations from different vendors may vary widely. For example:

- Is only one retry done, or multiple (and how many?)
- Can different slaves have different settings for the number of retries?
- What is the timeout setting: the same for every slave, or a unique setting per slave?
- Which error is given to the application software when the retries didn't help?
- Which diagnostics is available?

There should always be a maximum number of retries that the master executes in a row for a certain slave. If this is not done, it could happen that the master continues retrying for ever, in case a slave is switched off, disconnected, or otherwise unable to respond. What the maximum number of retries can be, may differ per vendor, but is usually somewhere in the range 1..5[19].



When the maximum number of retries is reached, this is an indication of serious problems:

- Either the slave is completely non-functional, or
- There are lots of electrical problems on the network.

[19] Such small numbers of retries are also seen in other industrial network protocols, like AS-Interface, who uses 2 retries, and Profibus/DP which by default has 1 retry (but can be configured to have any in the range 1..8).

It is often easy to distinguish between these two cases: when only one slave is non-functional, there will be retries for this slave only, but not for the other slaves. When there are electrical network problems, most likely **all** slaves will have retries.

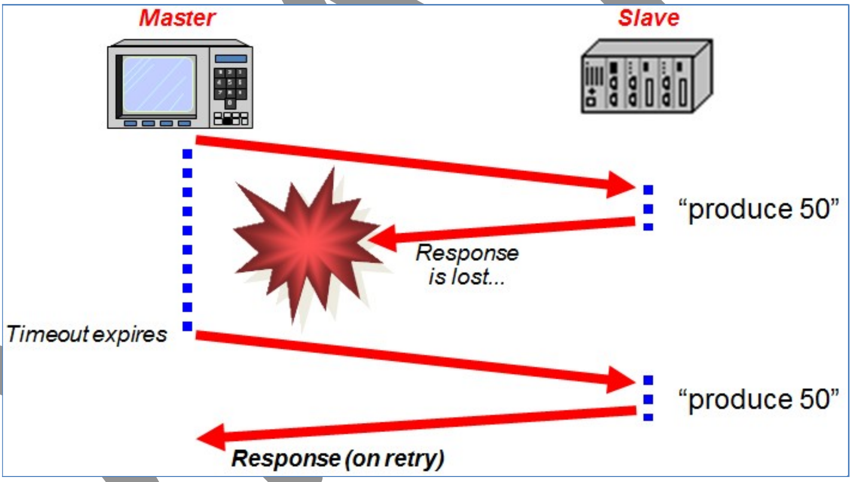
It is therefore useful to have a master-implementation which keeps track of the number of retries per slave, instead of the just the total number of retries for the whole network.

Some retries are evil!

Good as retries may seem, there is one aspect that should be taken care of: how does the slave detect that it receives a retry? It cannot distinguish a retried message, as it is bit-for-bit identical to the original message. As far as the slave is concerned, it is a new message, and it **must** be executed.

Processing retries is not always OK. This is especially so when a slave changes it state, starts certain actions, modifies its environment, etc. This is because of the nature of retries: the master sends the **same** command again.

But suppose that we are in a situation where the original command was received *and* processed, and a response was sent back, but the response is lost. The master then sends a retry, but for the slave this is a new command. It does not know what the master is doing (a retry), because it does not know that the 1st response was lost.



In the example above, the slave is instructed to produce 50 units (of some product on a production line). When finished, it duly reports its result to the master, which then gets lost. The master then sends the retry, and the slave produces *another* 50 units. The response is then sent back to the master, who now thinks his production order of 50 is OK. But actually 100 units are produced.

A similar problem may occur when the slave modifies its environment, i.e. "turn left 30 degrees" or "increase temperature by 5 degrees". Every state-changing command is dangerous if received and processed multiple times.

In all modern industrial network protocols, there is a mechanism to detect retries messages, which are then filtered out. A simple way to do this is to use a sequence number, incrementing by 1 for each new message (56, 57, 58, 59…). When a retry is done, the slave may see the same sequence number more than once (57, 57, 58, 58, 59….), and it can now properly handle this.

Unfortunately, Modbus has **no** such mechanism! This means that every state-changing command on a slave must be implemented in such a way that multiple reception of the same command is no problem. For example, by using an toggle bit (0/1/0/1…), a strobe bit (0 →1), etc. This also needs support on the master side, usually the application program.

Whenever possible, avoid the use of relative data (relative to a previous command or actual state of a slave); only use absolute values, i.e. “turn left to course 85 degrees” or “increase temperature to 35 degrees”.

8. Error handling on the slaves

Whenever a slave receives an undamaged message with a request from the master, it will first check the message for any possible errors. This are not transmission errors, but logical errors made on the master, for example a programming error.

If such an error is detected, processing of the request is stopped, and an error message is sent back to the master. Such errors are called “exceptions” in Modbus terminology. They are usually caused by programming errors, and as such occur often in new systems; but when all programming errors are solved, they should no longer occur.

The exceptions are numbered from 1 to 11, and have the following meaning:

1 Illegal Function	The slave/server does not support the function code that the master/client sent. Usually this is caused by a programming error (by the programmer not reading the documentation about supported function code). Once this is solved, this exception should never occur anymore.
2 Illegal Data Address	The coil / input / output / holding register does not exist. Usually this is caused by a programming error on the master. Example: a vendor may specify that its device has 200 holding registers (1..200). If an attempt is made to read / write holding register 300, exception 2 is given.
3 Illegal Data Value	A field in the Modbus message has a value which is invalid. Example: the maximum amount of registers that can be accessed with some commands is 120, but the field in the message can contain a much higher value. On any value > 120 this exception 3 is given. In the Modbus specification [MBUSAPPL] it is specifically mentioned that this exception code must **not** be used on register contents outside a valid range for the application software. Exception 4 must be used for this.

4 Slave Device Failure	An unrecoverable error occurred in the slave while it was handling the request. Example: a vendor may specify that a holding register may be set to values in the range 1..5. If an attempt is made to set this holding register to the value 6, exception 4 is given. Note that if a function code is used to write multiple registers, it is unclear at which coil or holding register the error was detected.
5 Acknowledge	The slave is processing a request, but as this will take a very long time, it sends this exception back to inform the master, who can then do something else instead of waiting. So actually this isn't an erroneous situation; this exception is just sent to complete the master/slave transaction cycle so that the master can continue with something else on the network (and come back later to check whether the slave is now ready or still busy). This exception code can only be given with function code 10 "Poll Program Complete", which is specific for a Modicon PLC so it is unlikely that you will ever get this exception code back.
6 Slave Device Busy	The slave/server is temporarily unable to handle new request, as it is doing something else. The master/client is advised to retransmit the command at a later time, hopefully the slave/server is then free.
7 Negative Acknowledge	The slave cannot perform the programming request sent per function code 13 or 14. This is a specific error related to Modicon PLC's. Note that this exception code is no longer listed in the current Modbus specification [MBUSAPPL].
8 Memory Parity Error	There is a problem with the extended memory (offset 6XXXX) in the slave, which appears to hold corrupted information. This a specific error related to function codes 20 and 21 for Modicon PLC's, not seen on modern Modbus devices which have no memory with parity.
9	Not used.
10 Gateway Path Unavailable	When using gateways (typically to convert from Modbus/TCP to Modbus/RTU), this exception is given when the gateway does not know how to reach the slave. This typically occurs when the gateway is not properly configured.

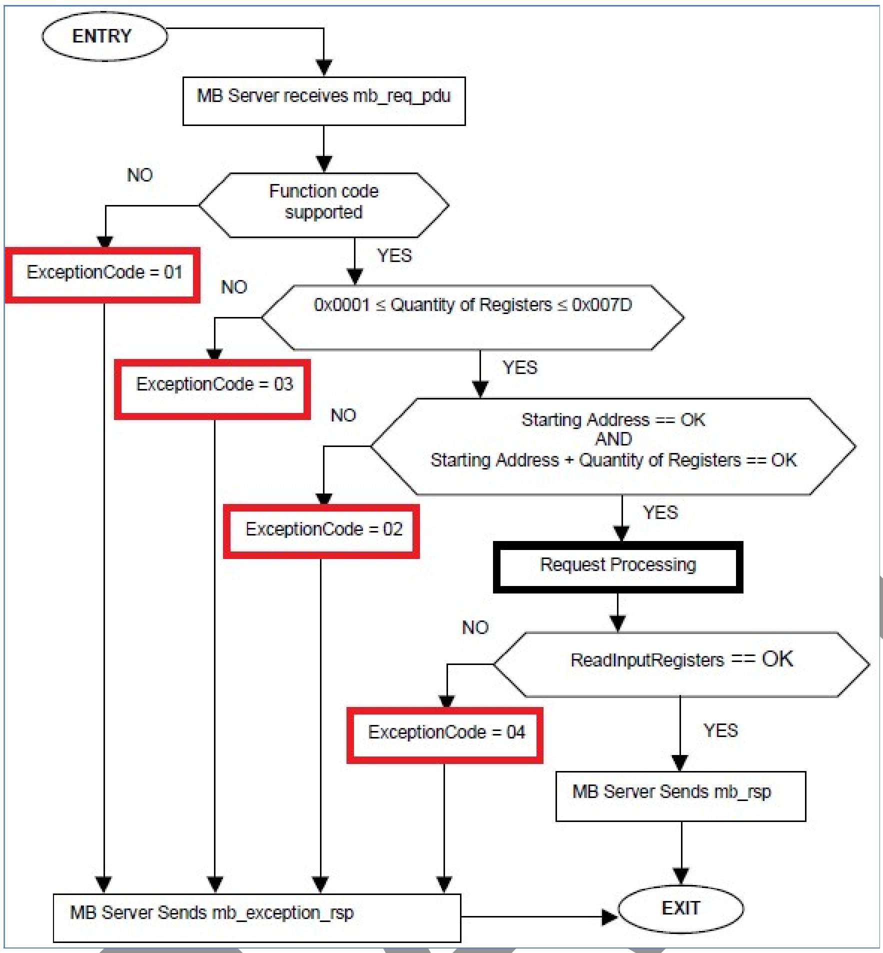
	Note that when using Modbus/TCP, the field “unit identifier” in the network message is often enough to indicate to the gateway which slave is intended, when the unit identifier is equal to the slave address. But this does not have to be the case when the gateway has multiple serial networks connected.
11 Gateway Target Device Failed To Respond	When using gateways (typically to convert from Modbus/TCP to Modbus/RTU), this exception is given when the gateway has sent a request message to a slave, but this slave did not respond within the configured time. This typically occurs when the slave is not connected, not powered, or its software is not running. The fault could also be at the master (client), when using a wrong slave address. Note that there are also Modbus/TCP gateways on the market that do not use this exception, so the Modbus/TCP master (client) must still implement its own timeout handling.

*No exception code is sent back when the message was sent as “broadcast”. This is because a slave **never** sends a reply back on broadcasts, even when the message contains an error. So very carefully check application software on a slave (or observing its behaviour, i.e. change in outputs or LEDs or a display) to see whether the broadcast was actually processed.*

In the original Modbus specification [MBUS300], the exceptions were just mentioned, but not much information was given on when each exception was to be used. For example, some vendors just reported back exception 4 “Slave Device Failure” on every error detected. Others used exceptions 1, 2 and 3 instead[20]. This made that it was difficult to generically handle exceptions on a Modbus master; they could have different meanings depending on the slave.

After the specification [MBUSAPPL] was updated for Modbus/TCP, a detailed flowchart on how each function code is to be processed in a slave was provided, giving more clarity. Hopefully, this will make the run-time behaviour of Modbus devices more similar, although it is not guaranteed that vendors will follow the flowchart. The example below shows the flowchart for the processing of function code 04 “Read Input Registers”. Such flowcharts are now available for every function code described in [MBUSAPPL].

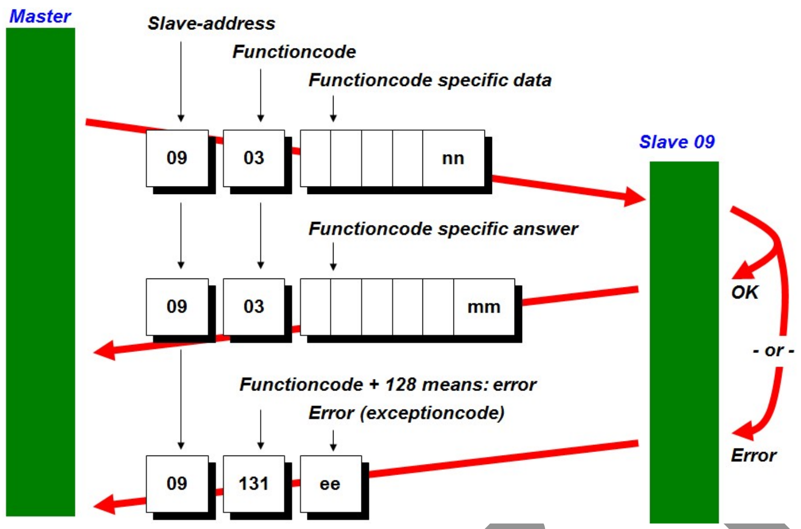
[20] *Personally, a long time ago I made software for a device which reported exception code 3 upon illegal values for a register, whereas today I know I would have to use exception code 4.*



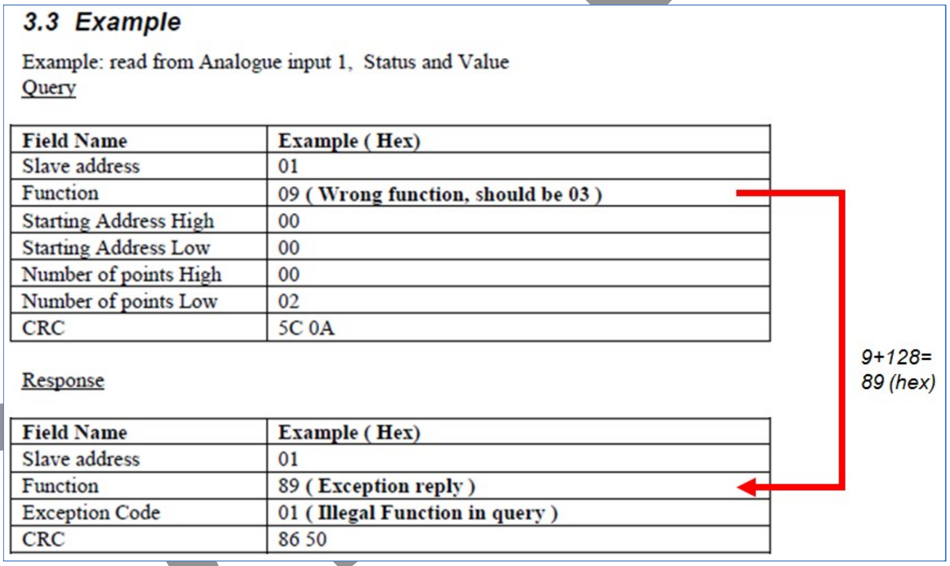
Example of a flowchart for the processing of incoming function code 04 (Read Input Registers), which clearly shows which exception must when be given.

9. Master / slave cooperation

As discussed earlier, the master **always** starts a message exchange. It sends a request to one specific slave, and the slave processes the function code contained in that message. When all is OK, a response message is sent back to the master. When an error is detected, an exception message is sent back (see the flowchart above). For example, when using function code 3:



An exception message is very short. It contains the slave's address, followed by the function code ***plus*** 128 (hex 0x80), followed by the exception code itself (see the previous section). What the master does with the exception just being reported depends on the vendor's implementation.



3.3 Example

Example: read from Analogue input 1, Status and Value

Query

Field Name	Example (Hex)
Slave address	01
Function	09 (**Wrong function, should be 03**)
Starting Address High	00
Starting Address Low	00
Number of points High	00
Number of points Low	02
CRC	5C 0A

Response

Field Name	Example (Hex)
Slave address	01
Function	89 (**Exception reply**)
Exception Code	01 (**Illegal Function in query**)
CRC	86 50

Example of a vendor's documentation showing the handling of an unsupported function code on this device, which triggers an exception code 01.

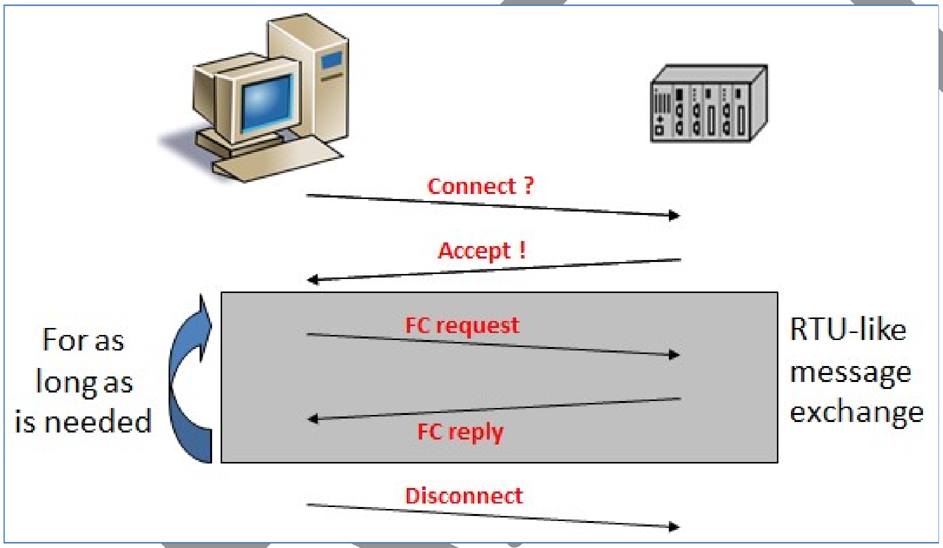
10. Connection setup

A major difference between Modbus/TCP, and Modbus/ASCII and RTU, is that the first is "connection oriented", and the second/third are "connectionless". The connection oriented way of working is enforced by TCP.

Connection oriented communication is identical to the way humans use the telephone. You set up the connection by dialling a number, and waiting for the other party to respond. If so, the connection is established and the conversation can start. At some moment the conversation is finished, and the connection is stopped. If the other party doesn't pick up the phone, you know something's amiss and it makes no sense to talk – the connection is not established. Probably you'll try later.

TCP works the same: before two devices can communicate with each other, they must also set up the connection. When accepted by the other party, communication can start (for an infinite amount of time). If no more communication is needed, the communication can be stopped (disconnect).

The advantage of connection oriented communication is that one gets a warning if no further communication is possible. The telephone system will give a signal to both parties that you've been disconnected. TCP will give an error to both devices that they've been disconnected. It continuously checks the connection status; it does this automatically without any involvement from the client or the server.



Exactly how a connection is to be set-up in Modbus/TCP is **not** specified by Modbus, and as such it depends on the vendor. Some vendors require that the application program must issue a special command to the Modbus/TCP software in order to start it setting up the connection. Others may choose to start automatically after power-on, or after the application software issues the first function code.

There is **no** function code defined to set up the connection.

CHAPTER 7. DATA PRESENTATION

Modbus has only two datatypes: “bits” (0/1), and 16-bit “registers”. Bits can be read and written, one by one, or in larger groups. Registers can also be read and written, one by one, or in larger groups. Different function codes are available for this (see chapter 6).

Simple as it may seem, the way groups of bits are to be handled in Modbus also has a relation to the CPU (processor) the software is running on. This is also true for registers. Sometimes conversions are necessary. These conversions must often be programmed as part of the application software on the master device.

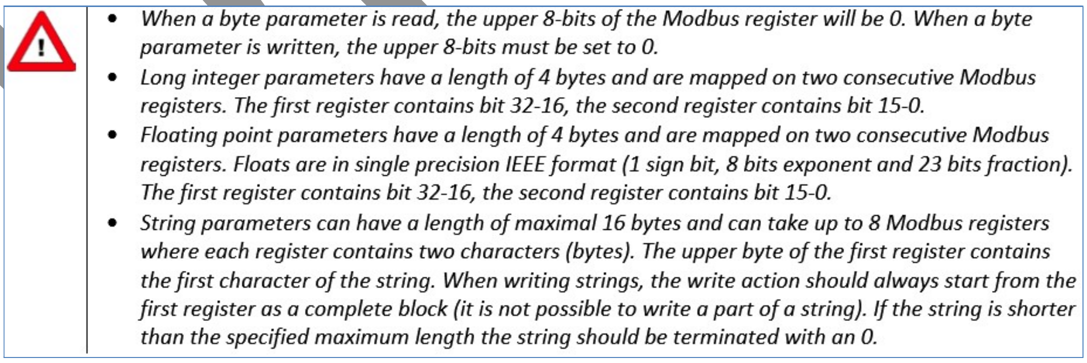
Bits and registers (and groups of them) are no longer the only datatypes used in modern industrial systems. What can one encounter in practice:

- Registers of 32 or even 64 bits, in signed or unsigned” format,
- Floating point numbers of 16, 32, 64, 80 or 128 bits,
- Character strings, in ASCII or Unicode.

Less common are:

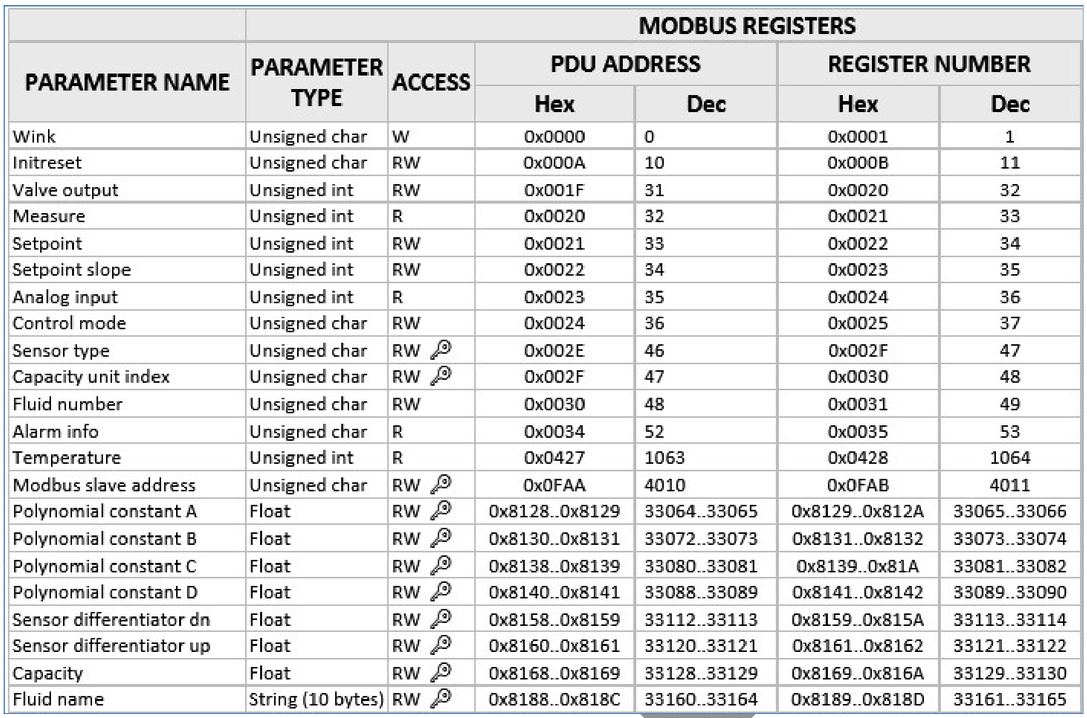
- Fixed point numbers
- Binary-coded decimal
- Characters
- Records.

Modbus does not natively know these datatypes, which makes that different vendors devise different ways of implementing these datatypes. This often comes as a surprise during commissioning!



- *When a byte parameter is read, the upper 8-bits of the Modbus register will be 0. When a byte parameter is written, the upper 8-bits must be set to 0.*
- *Long integer parameters have a length of 4 bytes and are mapped on two consecutive Modbus registers. The first register contains bit 32-16, the second register contains bit 15-0.*
- *Floating point parameters have a length of 4 bytes and are mapped on two consecutive Modbus registers. Floats are in single precision IEEE format (1 sign bit, 8 bits exponent and 23 bits fraction). The first register contains bit 32-16, the second register contains bit 15-0.*
- *String parameters can have a length of maximal 16 bytes and can take up to 8 Modbus registers where each register contains two characters (bytes). The upper byte of the first register contains the first character of the string. When writing strings, the write action should always start from the first register as a complete block (it is not possible to write a part of a string). If the string is shorter than the specified maximum length the string should be terminated with an 0.*

Excerpt from a vendor’s manual where <u>their</u> way of handling different datatypes is described.

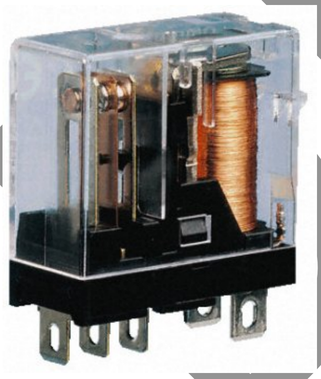


PARAMETER NAME	PARAMETER TYPE	ACCESS	MODBUS REGISTERS			
			PDU ADDRESS		REGISTER NUMBER	
			Hex	Dec	Hex	Dec
Wink	Unsigned char	W	0x0000	0	0x0001	1
Initreset	Unsigned char	RW	0x000A	10	0x000B	11
Valve output	Unsigned int	RW	0x001F	31	0x0020	32
Measure	Unsigned int	R	0x0020	32	0x0021	33
Setpoint	Unsigned int	RW	0x0021	33	0x0022	34
Setpoint slope	Unsigned int	RW	0x0022	34	0x0023	35
Analog input	Unsigned int	R	0x0023	35	0x0024	36
Control mode	Unsigned char	RW	0x0024	36	0x0025	37
Sensor type	Unsigned char	RW 🔑	0x002E	46	0x002F	47
Capacity unit index	Unsigned char	RW 🔑	0x002F	47	0x0030	48
Fluid number	Unsigned char	RW	0x0030	48	0x0031	49
Alarm info	Unsigned char	R	0x0034	52	0x0035	53
Temperature	Unsigned int	R	0x0427	1063	0x0428	1064
Modbus slave address	Unsigned char	RW 🔑	0x0FAA	4010	0x0FAB	4011
Polynomial constant A	Float	RW 🔑	0x8128..0x8129	33064..33065	0x8129..0x812A	33065..33066
Polynomial constant B	Float	RW 🔑	0x8130..0x8131	33072..33073	0x8131..0x8132	33073..33074
Polynomial constant C	Float	RW 🔑	0x8138..0x8139	33080..33081	0x8139..0x81A	33081..33082
Polynomial constant D	Float	RW 🔑	0x8140..0x8141	33088..33089	0x8141..0x8142	33089..33090
Sensor differentiator dn	Float	RW 🔑	0x8158..0x8159	33112..33113	0x8159..0x815A	33113..33114
Sensor differentiator up	Float	RW 🔑	0x8160..0x8161	33120..33121	0x8161..0x8162	33121..33122
Capacity	Float	RW 🔑	0x8168..0x8169	33128..33129	0x8169..0x816A	33129..33130
Fluid name	String (10 bytes)	RW 🔑	0x8188..0x818C	33160..33164	0x8189..0x818D	33161..33165

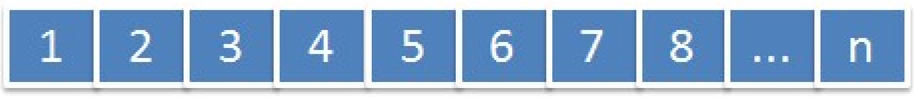
Excerpt from a vendor's manual describing the various datatypes (second column) used for various parameters.

1. Bits

Two types of bits are known, the "coils" and the "inputs". They correspond to a PLC's digital outputs and inputs, respectively. The name "coils" refers to the usage of relays, before the solid-state electronic was capable of handling the voltages and currents.



Bits are accessed with function codes 1 and 5. A slave can have up to 65535 bits in its memory. They are numbered sequentially from 1 to 65535; bit number 0 does not exist (according to the Modbus specification). The function of each bit is determined by the device's supplier and is normally documented in the manual.

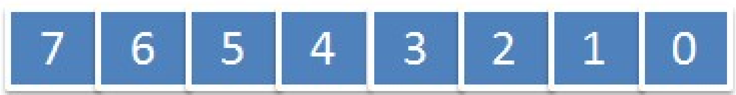


Writing a new value for a single coil will pose no problem for any application (using function code 05 – Force Single Coil). More difficult are function codes 01 (Read Coil Status) and 02 (Read Input Status), both of which allow for a group of bits to be read.

Simply as it seems, in practice the numbering of bits is not always intuitive. This is caused by different numbering systems for groups of bits in a byte, which is different from the Modbus convention shown in the picture above. And for larger groups of bits, i.e.in groups of 16 or more, there are two different numbering conventions, as used on (Intel) PC's and on PLC's according to the programming standard IEC-61131.

What usually happens is that the programmer finds the communication to be functioning perfectly, but that wrong data is read or written. Usually this is blamed on the network protocol implementation, but in fact the wrong bit locations are used.

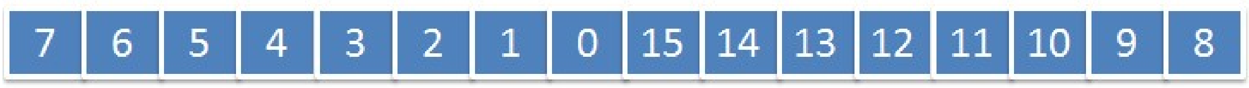
For example, suppose that you are using a PLC with 8 digital outputs (coils), numbered 0..7, and you want to read the value of coil 1. That would be the "leftmost" coil according to the Modbus numbering system. But on most modern CPUs', the following convention is used to number the 8 bits in a byte:



It now depends on the implementation software on the slave / server which bit you *really* get. Perhaps the programmer thought that "bit 1 is bit 1", so you get the value of the bit at the second-right position. Or, the programmer also could have thought, "bit 1 is the leftmost", so then you get the value of output 7".

16-bit systems

When using 16 bits or more together in a "word" (what Modbus calls: "register") there are three different number systems in existence. This is Intel's way of numbering bits in 16 bits:



This is Motorola's:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

And this in use on a PLC according to IEC-61131:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Again it depends on the vendor what you have to send on Modbus to get the value of a certain bit. Suppose you want the left-most bit, do you have to request bit 7, 15 or 0? Carefully document the relation between the device coil / input numbering you are using and the application software numbering, otherwise you will not understand it any more in a month (nor will any successors).

Accessing bit 0 is sometimes impossible, because Modbus starts counting at 1. The workaround is simply to not use bit 0 in a PLC program.

2. Integers / words

A 16-bit register is the standard Modbus datatype for holding registers, analogue inputs and analogue outputs. With 16 bits, a value in the following ranges can be represented[21]:

-	0..65536	the so-called "unsigned" values.
-	-32768..+32767	the so-called "signed" values.

When a register is signed, its value is binary coded according to the "2's-complement" standard[22] which is used by all modern CPU's for the last 3 decades. The value of bit 15 (the leftmost bit) determines:

- When it is a zero, the value is a positive number.
- When it is a one, the value is a negative number.

An example of the possible values of a byte (only a few of the 65536 shown):

Binary Value	Decimal Value	Binary Value	Decimal Value
00000000.00000000	0	**11111111.11111111**	-1
00000000.00000001	1	**11111111.11111110**	-2
00000000.00000010	2	**11111111.11111101**	-3
00000000.00000011	3	**11111111.11111100**	-4
00000000.00000100	4	**11111111.11111011**	-5

[21] *It is not possible to determine whether a group of 16 bits is signed or unsigned just by looking at the bits; you always need the vendor's documentation for this.*

[22] *There also exists a "1's complement" standard, allowing for values to be represented in the range -32767..+32767, with support for a -0 and a +0, but I have never seen this used.*

00000000.00000101	5	**11111111.11111010**	-6
00000000.00000110	6	**11111111.11111001**	-7
00000000.00000111	7	**11111111.11111000**	-8
00000001.00000000	256	**11111111.00000000**	-257
01111111.11111111	32767	**10000000.00000000**	-32768

The byte with (leftmost) bits 15..8 is called the “Most Significant Byte” (MSB) or (alternatively) the “High” or “H” byte. The other byte, containing (rightmost) bits 7..0 is called the “Least Significant Byte”(LSB), the “Low” or “L” byte.

How the H and L bytes are stored in memory depends on the CPU of that system. Some store the H / L bytes in memory exactly in this sequence: H first, then L. This is called the “Big Endian” format, formerly used on Motorola CPU’s (hence it why it also called the “Motorola” format).

Others store the H / L bytes in the sequence: L first, then H. This is called the “Little Endian” format, which is popular because Intel CPU’s use it. There is no particular advantage of using one above the other, it is just: there are two possible choices, one group of vendors chose this sequence, other vendors chose the other sequence.

You have to know how the CPU works this in order to calculate the value of 16-bit data. For example, suppose that the memory contains a byte with value 0 and next a byte with value 1. On a CPU using the “Big Endian” format, these two bytes would have the value:

V = 0 * 256 + 1 = 1

On a CPU using the “Little Endian” format:

V = 0 + 1 * 256 = 256

Another example: suppose two successive bytes have the values 10 and 20.

The CPU with “Big Endian” format has:	V= 10 * 256 + 20 = 2580
The CPU with “Little Endian” format has:	V = 10 + 256 * 20 = 5130

On **any** network protocol (not just Modbus!) the distinction between Big Endian / Little Endian format is relevant. Bytes in a message are sent and received sequentially, so for a 16 bit field there are two bytes, and which one is the H byte and which one is the L byte? The receiving CPU has to know this, in order to store the bytes in the right sequence in its memory (swapping them), in order to calculate V in the right way.

Modbus uses “Big Endian”

Modbus specifies the “Big Endian” format as mandatory for all 16-bit fields in the function code headers. It is then logical to assume that this is also the format to be used for application data. In the current Modbus specification this is indeed written down as such [MBUSAPPL, section 4.2].

But in the earlier Modbus specification [MBUS300] this was not specified as clearly, except mentioning it in the examples that the "Hi" byte must always be transferred first[23]. But some vendors thought 'if it is not specified that it can legally be done differently', and as a result they transmit the 16-bit fields in the function code headers as "Big Endian", but user data (registers) in the "Little Endian" format.

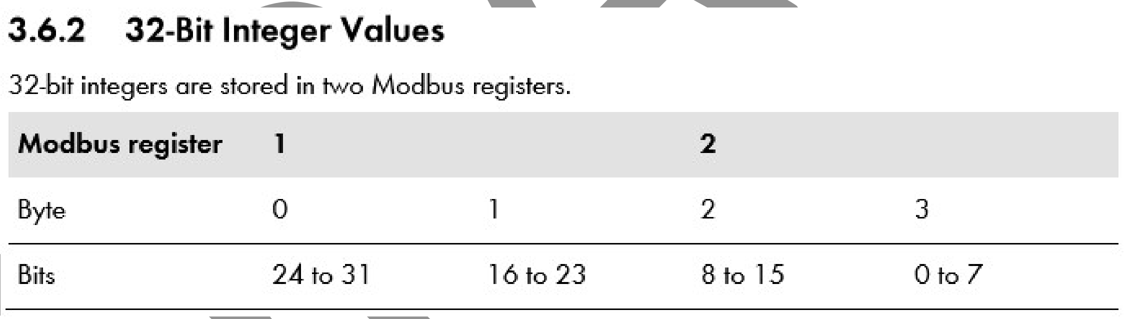
This is a *big* mistake; its will put a huge conversion effort on the network master (for Modbus/ASCII and /RTU); I haven't seen it on Modbus/TCP (yet) perhaps those implementers read the [MBUSAPPL] specification.

3. Long integers / double words

When 16 bits are not enough to store data in, the next step would be to use 32 or 64 bits. In software-engineering terms, we call this a "double word" (DWORD) or "long word" (32 bits), or a "long long word" or a "quad word" (64 bits). Note that these terms are not standardized!

The bits 32..16 are called the "Most Significant Word" (MSW), and the bits 15..0 the "Least Significant Word" (LSW). In Modbus you need 2 registers to store the MSW and LSW; usually 2 consecutive registers are chosen.

The following example is taken from a vendor's documentation, showing how 32 bits are stored in two (consecutive) registers 1 and 2, with register 1 containing the MSW and register 2 the LSW.



3.6.2 32-Bit Integer Values

32-bit integers are stored in two Modbus registers.

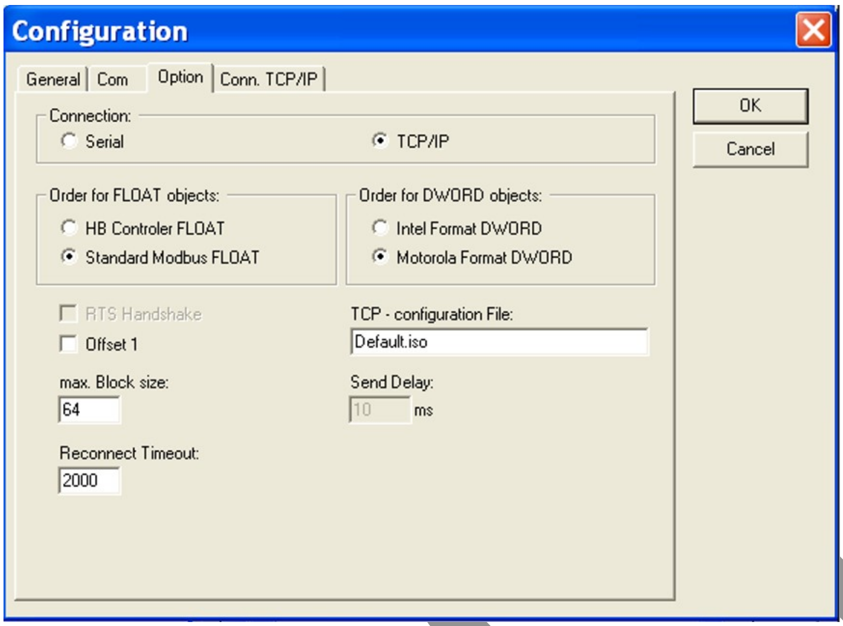
Modbus register	1		2	
Byte	0	1	2	3
Bits	24 to 31	16 to 23	8 to 15	0 to 7

Source: SMA

Big Endian / Little Endian?

In the Modbus network messages, usually the MSW is transferred first, followed by the LSW. But vendors may do this differently, just as with the MSB and LSB of a register. Consult the vendor documentation for this; some vendors allow you to configure the behaviour of their device.

[23] *For me this is clear enough: Big-Endian. And so many others read it so as well.*



Configuration

General | Com | Option | Conn. TCP/IP

Connection: Serial [✓ TCP/IP]

Order for FLOAT objects: HB Controler FLOAT [✓ Standard Modbus FLOAT]

Order for DWORD objects: Intel Format DWORD [✓ Motorola Format DWORD]

RTS Handshake

Offset 1

max. Block size: 64

Reconnect Timeout: 2000

TCP - configuration File: Default.iso

Send Delay: 10 ms

OK

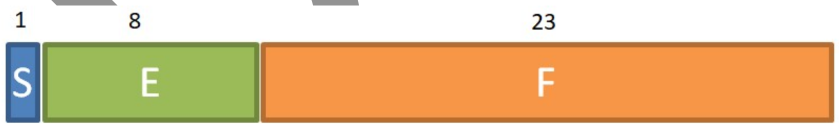
Cancel

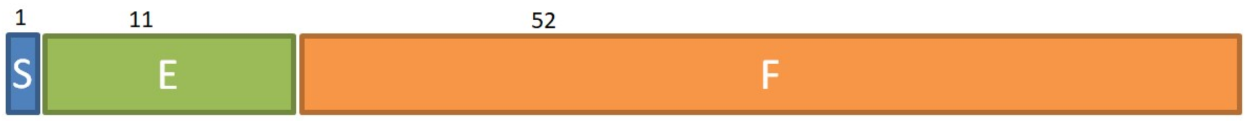
Some vendors take advantage of the freedom that the transmission format is not specified, and they use the “Little Endian” / Intel standard: transmit the LSW first, followed by the MSW. This is advantageous when both the master and the slave run on an Intel CPU, because the registers R1 and R2 do not have to be swapped twice, giving a little performance improvement[24].

4. Floating point

At the time Modbus was invented, using floating-point numbers in calculations was very “expensive”: it required costly special coprocessors or (very slow!) software packages. It was not standardized; which bit pattern corresponded to a certain value differed per vendor. This made exchange of floating-point data over a network with equipment from different vendors difficult.

This all changed in the 90’s of the previous century with the IEEE-754 standard, which is now adopted by all processor vendors. Unfortunately Modbus does not specify how floating-point values must be stored in registers.





[24] *This performance improvement is so small that it is completely dwarfed by the effort needed to explain customers what to do, and the effort spent by users trying to find out why their network seems to come up with apparently very strange data.*

The IEEE-754 "double" format for single-precision (top) and double precision (bottom), consisting of 1 "sign" bit, an 8 or 11 bit exponent and a 23 or 52-bit fractional part (sometimes also called "mantissa").

IEEE-754 specifies varies floating-point formats: 32 bit size, 64 bit size, 80 bit size, 128 bit size. Until recently, the 32 bit format was the most common, until the rise of 64 bit processors in PC's. For industrial systems the 32 bit "single precision" format is still widely used, giving numbers with 6 (decimal) digits of precision with a maximum range of 10^{38}. The 64-bit "double precision" format has 15 (decimal) digits of precision, with a maximum range of 10^{308}.

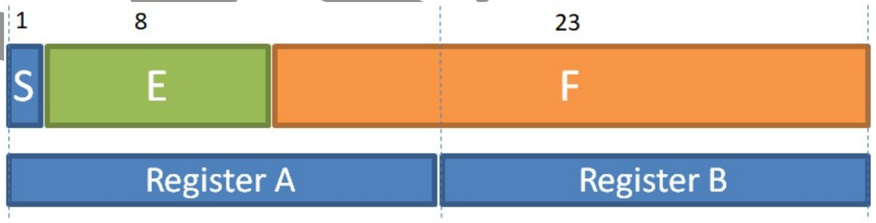
This publication is not the right place to explain the inner workings of IEEE-754, which may easily fill dozens of pages. Troves of information can be found via internet.

Transferring 32 or 64 bits floats can simply be done by taking 2 or 4 consecutive Modbus registers, which can then be read/written with the standard function codes for accessing multiple registers.

The IEEE-754 has functionality for representation of values like -0, +infinity, -infinity, "NotANumber" bit patterns, and "denormalized number" bit patterns. Not all implementations of IEEE-754 support these. The application software must be prepared to handle these; the fact that you don't support them doesn't mean that the other side won't send them!

Big Endian / Little Endian?

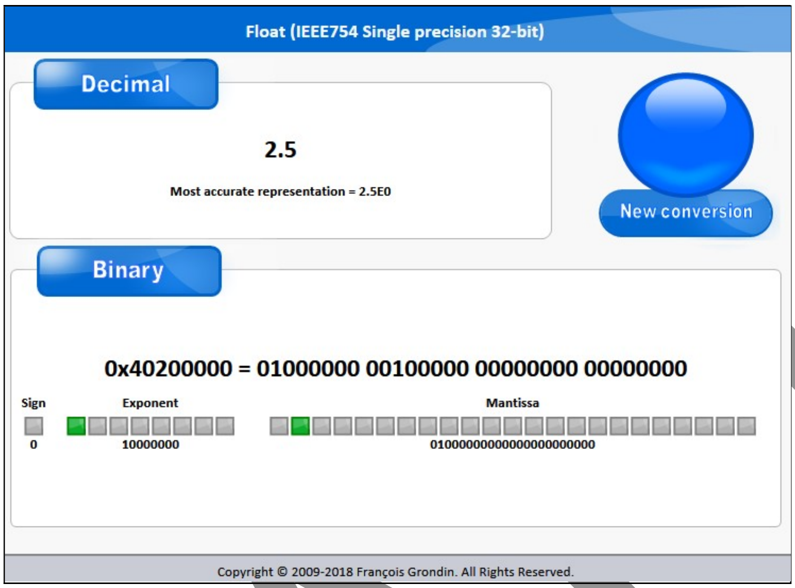
This doesn't relate to floating point values, however there are still 4 bytes which need to be transmitted in two registers (usually consecutive):



Which register contains the sign and exponent and some of the fraction bits, and which register only contains fraction bits, depends on the vendor. Some vendors transmit register A first, and other vendors register B. Within a register, you might also see the bytes swapped. So there are multiple possible ways to transmit a 32-bit floating point value:

B1	B2	B3	B4
B2	B1	B4	B3
B3	B4	B1	B2
B4	B3	B2	B1

If the vendor documentation is not clear enough about how a floating point value is transmitted, it is not easy to find the transmission format because the 32 bits do not easily convert to a human-understandable format. Although it is not very difficult to convert a bit pattern into the associated floating point value, several tools are online to assist you. To name one: www.binaryconvert.com.



Float (IEEE754 Single precision 32-bit)

Decimal

2.5

Most accurate representation = 2.5E0

Binary

0x40200000 = 01000000 00100000 00000000 00000000

Sign	Exponent	Mantissa
0	10000000	01000000000000000000000

When using 64 bit numbers, the same issues arise, but worse because there are 8 bytes involved. Fortunately 64 bit numbers are seldom seen.

5. Fixed point

Sometimes the usage of IEEE-754 is too complex, as calculations with such numbers require a lot of software, which is slow, or it requires a floating-point coprocessor (which can be expensive for smaller embedded systems). Fixed point is also often used on older equipment when floating point coprocessors were still prohibitively expensive.

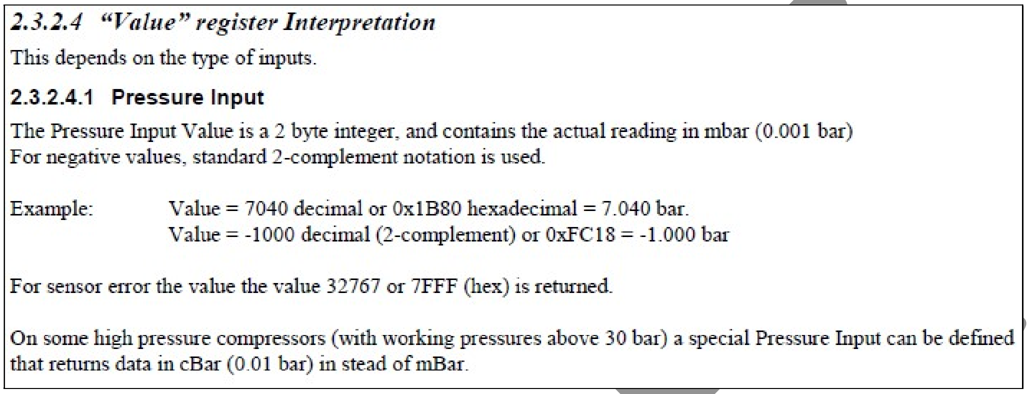
The solution is to use “fixed point numbers”. It is simply the regular integer binary representation, in which we **mentally** place a comma, on both the master and a slave.

As an example: a 16-bit number, which normally has the range 00000..65535, can be thought to have a comma after the first four digits. The maximum range than decreases from 65535 to 6553,5 (or -3276,8 .. 3276,7); but now there is one digit available for fractional results.

> For example, suppose a slave has a measured value = 123,4 that must be sent to the master. Multiply it by 10 to get an integer value: 1234, and store this in a Modbus register. Master reads this from the slave as: 1234. The application program on the master side ‘knows’ this is a fixed point number, so divides the data by 10 to get the actual value again: 123,4.

When writing data, the same algorithm is used. Suppose that the master wants to write a value 567,8 to a slave. Multiply it by 10 and send 5678 to the slave, who will then divide it by 10 again to get the real value 567,8.

It is also possible to have **two** fractional digits in a register (giving a factor 100). This limits the usable range to 0..655,35 (or to -327,68 .. 327,67). If 16 bits is too restrictive, of course it is possible to have fixed point notation with a 32-bit value stored in two (consecutive) registers.



2.3.2.4 "Value" register Interpretation

This depends on the type of inputs.

2.3.2.4.1 Pressure Input

The Pressure Input Value is a 2 byte integer, and contains the actual reading in mbar (0.001 bar)
For negative values, standard 2-complement notation is used.

Example:	Value = 7040 decimal or 0x1B80 hexadecimal = 7.040 bar.
	Value = -1000 decimal (2-complement) or 0xFC18 = -1.000 bar

For sensor error the value the value 32767 or 7FFF (hex) is returned.

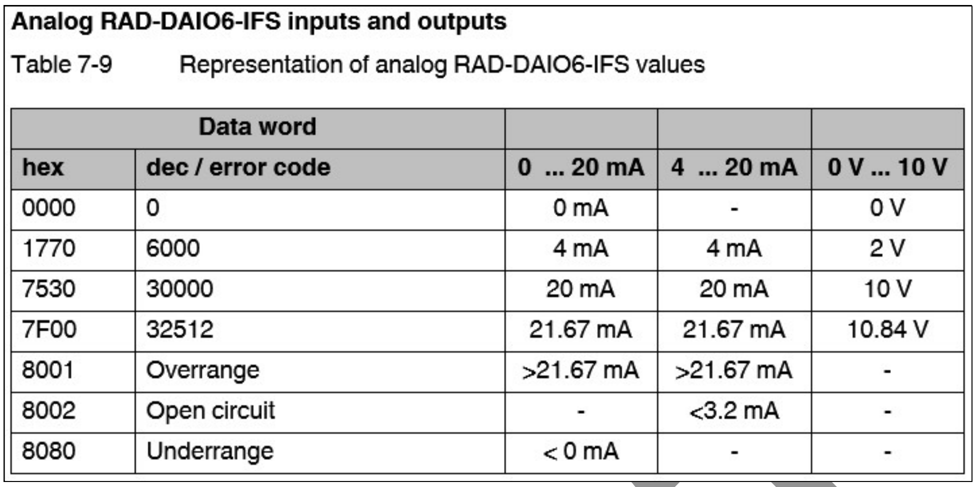
On some high pressure compressors (with working pressures above 30 bar) a special Pressure Input can be defined that returns data in cBar (0.01 bar) in stead of mBar.

You are not limited to multiplications by 10 or 100, even a factor 2 could be used – this would give you values like 3, 3½ , 4, 4½, 5, etc. As long as the master and the slave(s) agree, this is OK, entirely up to you to decide.

Note that the application software running on the master could be faced with different conversion algorithms simultaneously in use, if different slaves use different fixed point formats.

6. Scaling

This is a combination of fixed point notation with a certain offset. One sees this often with analogue signals such as 4..20 mA inputs / outputs. The example below shows such a module, which represents a value of 1 mA with the integer value 1500, and uses values > 8000 (hexadecimal) to indicate an erroneous situation. There is no standard for this; every vendor can have its own implementation.



Analog RAD-DAIO6-IFS inputs and outputs

Table 7-9 Representation of analog RAD-DAIO6-IFS values

Data word				
hex	**dec / error code**	**0 ... 20 mA**	**4 ... 20 mA**	**0 V ... 10 V**
0000	0	0 mA	-	0 V
1770	6000	4 mA	4 mA	2 V
7530	30000	20 mA	20 mA	10 V
7F00	32512	21.67 mA	21.67 mA	10.84 V
8001	Overrange	>21.67 mA	>21.67 mA	-
8002	Open circuit	-	<3.2 mA	-
8080	Underrange	< 0 mA	-	-

Excerpt from a vendor's manual, explaining which register value corresponds to a physical output setting (current or voltage) on an analogue input or output. Source: Phoenix Contact.

Not shown in the above example are analogue inputs which have a range -10V...+10V. It can be found that with one vendor the decimal values to represent these voltages are coded as follows:

-10V	hex 0000, decimal 0
0V	hex 8000, unsigned decimal 32768 / signed decimal -32768
+10V	hex FFFF, unsigned decimal 65535 / signed decimal -1

This has the strange representation that negative voltages are represented with a positive decimal value and positive voltages with a negative decimal value. Other vendors may (more logical) code the analogue values as follows:

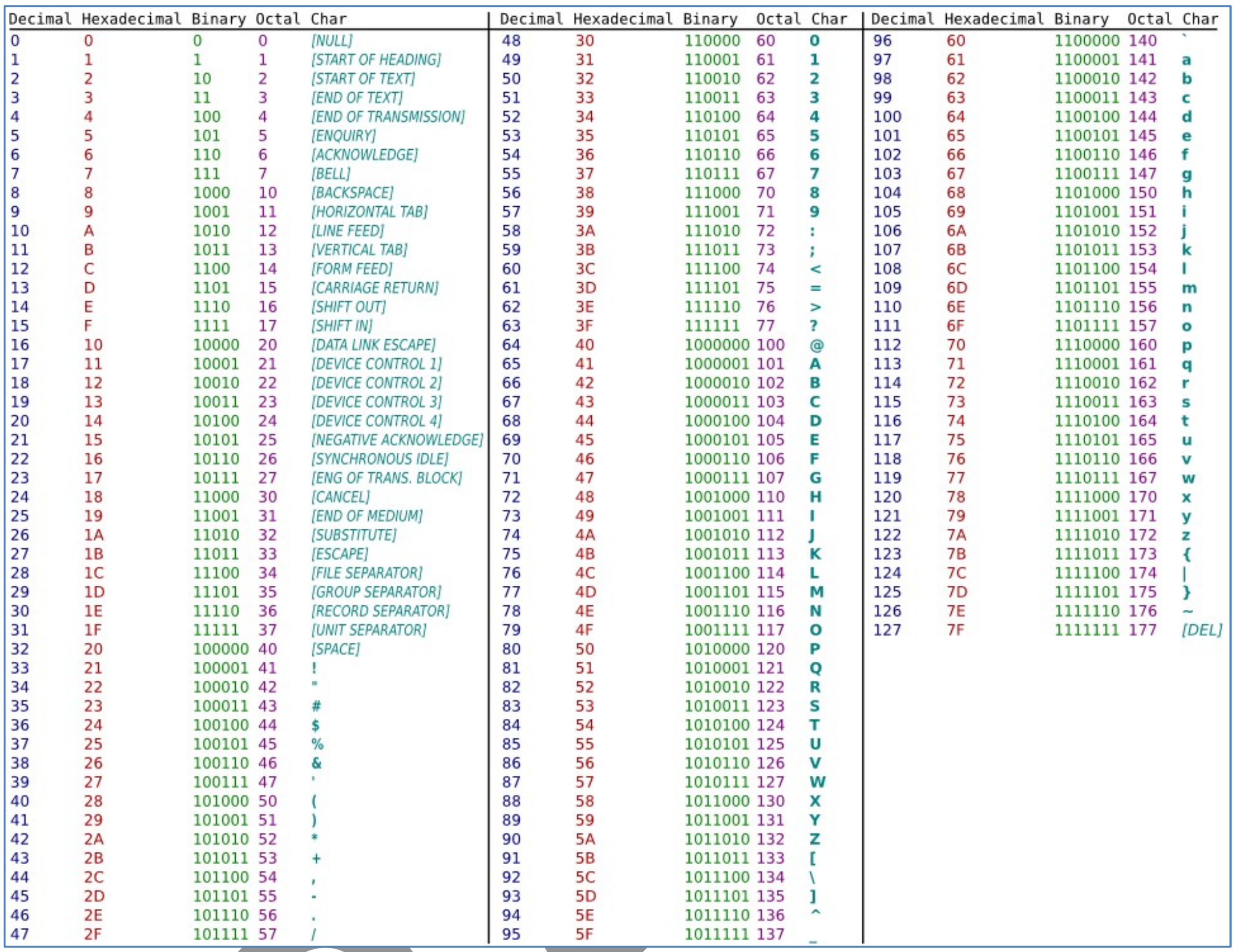
-10V	hex 8000, unsigned decimal 32768 / signed decimal -32768
0V	hex 0000, decimal 0
+10V	hex 7FFFF, (un)signed decimal 32767

When working with analogue inputs or outputs, always consult the vendor documentation about the representation of values.

7. Characters

Characters are usually coded according to the ASCII standard. This is so old a standard that I have never seen a device that doesn't support it[25]. Note that this ASCII is the same as in Modbus/ASCII which codes its network messages according to this standard, using the characters 0123456789ABCDEF: and CarriageReturn CR and LineFeed LF.

[25] *To be true: there **are** other standards invented to decode characters, such as EBCDIC (used in IBM mainframes), but I have never seen this.*



Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

The ASCII –table, showing the binary, decimal and hexadecimal coding for all characters (32..126) and the control-codes (0..31 and 127). Source: Wikimedia.

Since a character is only 7 or 8 bits in size, the least significant byte of a (16-bit) register usually contains the character's value. When using Unicode[26], a standard to encode all special characters not available in English (such as Ä and Õ̃) and all other alphabets in the world, one such a 16-bit character can be stored in a register. Luckily, Unicode is a superset of ASCII, so when a device wants Unicode it is easily possible to use it for ASCII data as well (these have their highest 8 bits all set to binary 00000000).

8. Strings

Strings of characters are also not officially supported in Modbus. But in many cases it is practical to be able to read/write strings, i.e. for error messages, or for putting texts on a display, or for getting text from an input device, etc. So we *do* see Modbus devices with strings on the market.

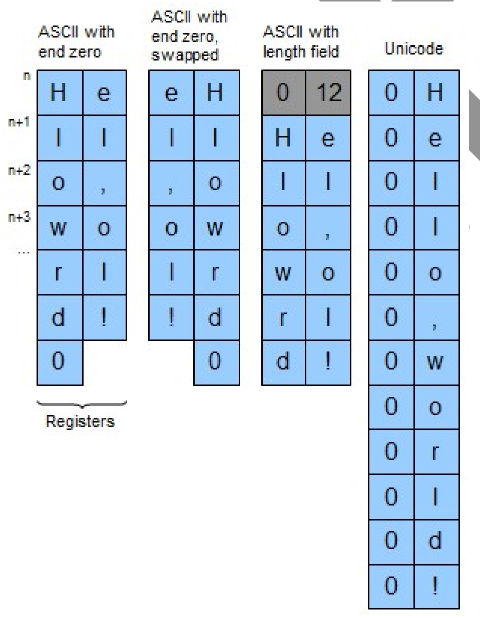
[26] *There are several other standards to decode characters of all sorts of alphabets, but it is unlikely you ever encounter them in Modbus equipment. Start with reading https://en.wikipedia.org/wiki/Unicode for an introduction to Unicode, UTF-8, UTF-16 and many, many more…*

Again, due to the absence of any specification on how strings must be stored in a device, there is a lot of variation to be found. Usually, strings are stored in consecutive groups of registers. Each register can then store two ASCII characters, or one Unicode character.

When there are ASCII-two characters per register, it is in the style of Modbus MSB/LSB sequencing that the first character is stored in the MSB of a register, and the second character in the LSB. But it *can* also be done differently (see the example below).

How long can a string be? Some vendors allow for dynamic texts, but then the length of the string must then be known. A popular implementation (as in the C programming language) is to indicate the end of the string by a ‘zero’ (binary 00000000) byte.

But other implementations keep a separate value for the string’s length, i.e. the number 12 for a string with 12 characters. The length is often stored in a register before the string’s data. This is practical for writing, but not so practical for reading, since first the length register must be read, and in a second command the actual number of characters.



Note that support for writing strings in a slave must be done with care. If a terminating zero is needed, always one byte **extra** is to be written. But strings can have an odd or even number of characters, when the string length is even a whole extra register (two bytes) must be written.

Strings should not overwrite other strings. Is it allowed to write in the middle of the string? If so, no terminating zero may be needed (because there is already one). Can strings be shortened or extended? Carefully check the vendor documentation.

The author of the software on a slave must also exercise caution when modifying the contents of registers containing string data, as the master can read them at the same time as they are modified. The software must modify a string (and its terminating zero, or the length field) in an atomic action.

9. Time

Depending on the vendor of a device, this can either mean the current ‘time-of-day’ (number of seconds since midnight), or the number of seconds or milliseconds since a certain epoch, for example 1-1-1970, 1-1-1900 or the start of the Gregorian calendar.

Dates can be represented in a standardized string format, i.e. “2018-07-13T19:10:12.450”, but also in many other formats such as “DDMMYYYY”, “MMDDYYYY”, “YYYYMMDD”, months as numeric value or as string (“Jan”, “Feb”), years with or without the century, etc. Consult the vendor documentation for more information.

CHAPTER 8. Protocol conversion

Because there are so many Modbus variants, sometimes it so happens that in one application a device with a different Modbus protocol variant is to be integrated. As we have seen in previous chapters, it is not possible to connect Modbus/ASCII to /RTU devices (and vice-versa), because the datalink protocol is different. Even when these devices have the same physical layer (RS232 or RS485), when the datalink protocol doesn't match, no communication is possible.

The same holds when connecting Modbus/TCP or ASCII or RTU devices; Ethernet is a completely different physical layer than RS232 or RS485.

As this happens often in daily practice, many vendors have developed so-called "protocol converters" or "gateways" that convert the physical layers and/or the datalink protocols from one in to the other. With such devices it is possible to have a Modbus/TCP device talk to a Modbus/ASCII or /RTU device, or vice-versa. Technically speaking, a gateway does protocol translation (from different protocol A to B, and vice-versa B to A), but of course it *can* be argued that Modbus/ASCII and / RTU are not different protocols.

1. Modbus/ASCII to /RTU and vice-versa

Because Modbus/ASCII and Modbus/RTU are quite similar, translating between them is not difficult but still needs a CPU for the message format conversion. The translation can be so quickly (relative to the speed of the serial line) that the delay of is hardly noticeable.

As soon as a complete message has been received and checked for correct checksum / CRC, the contents of the outgoing message can be calculated and its transmission can start. A clever gateway/converter will calculate the corresponding checksum / CRC during the transmission of the first characters of the outgoing message (unless the processor is very slow).

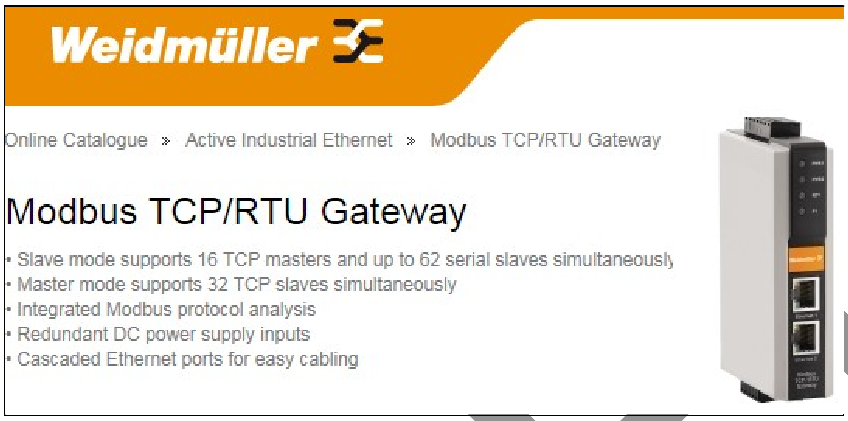
On the network master, care should be taken that the transmission time for the request and the response message is almost doubled, so a timeout should not be set too short.

In practice, Modbus/ASCII to /RTU conversions (or vice-versa) are hardly seen anymore.

2. Modbus/TCP to Modbus/RTU

Gateways that convert from Modbus/TCP to Modbus/RTU are relatively easy to make, as the protocols are quite alike. From an incoming Modbus/TCP request, the 6-byte header is stripped, the CRC must be calculated and added, and then the Modbus/RTU message can be transmitted. Upon receiving the response, the CRC is first checked, and when OK the 6-byte header is sent followed by the data part.

As the 6-byte header contains a “Transaction ID” which is not sent over Modbus/RTU, the gateway must store this field for use in the header of the response message. This makes that a gateway can handle only one request per Modbus/RTU network. As this is identical to what a Modbus/RTU master can do, it enforces no extra restrictions.


Weidmüller

Online Catalogue » Active Industrial Ethernet » Modbus TCP/RTU Gateway

Modbus TCP/RTU Gateway

- Slave mode supports 16 TCP masters and up to 62 serial slaves simultaneously
- Master mode supports 32 TCP slaves simultaneously
- Integrated Modbus protocol analysis
- Redundant DC power supply inputs
- Cascaded Ethernet ports for easy cabling

Some gateway vendors allow access to more than one Modbus/RTU network per gateway. The internal implementation of the gateway must be able to differentiate between the traffic for all these networks. One example of how to do this is by using more than one TCP-port. Whereas the default Modbus/TCP port is 502, gateways can use the higher ports too: port 502 = RTU network 1, port 503 = RTU network 2, 504 = RTU network 3, etc. This also allows all networks to be used concurrently; from the Modbus/RTU point-of-view this is not an issue since they are completely separate networks.

3. Modbus/RTU to Modbus/TCP

Gateways that operate the other way around are more difficult to implement, due to the way TCP/IP works. Such a gateway will be a Modbus/RTU slave, and this protocol is not connection-oriented as TCP/IP is. So upon reception of the first function block from the master, the gateway must set up a TCP/IP connection first to a Modbus/TCP server. But using which IP-address? This information is not available in Modbus/RTU, so it must be configured on the gateway (i.e., via an embedded webserver). After the connection is set up, the first (and all following function blocks) can be sent to the Modbus/TCP server.

As the setting up of the connection in TCP can sometimes be quite slow, the master should take care that any timeout for the *first* function block is not too short. As soon as the connection is established, the timeout can be set to a shorter interval. However many Modbus/RTU masters do not allow this.

A gateway can also implement that it sets up the connection to the Modbus/TCP server as soon as the power is turned on. This way, no special handling of the first function block is needed.

4. Ethernet to Modbus/RTU or Modbus/ASCII

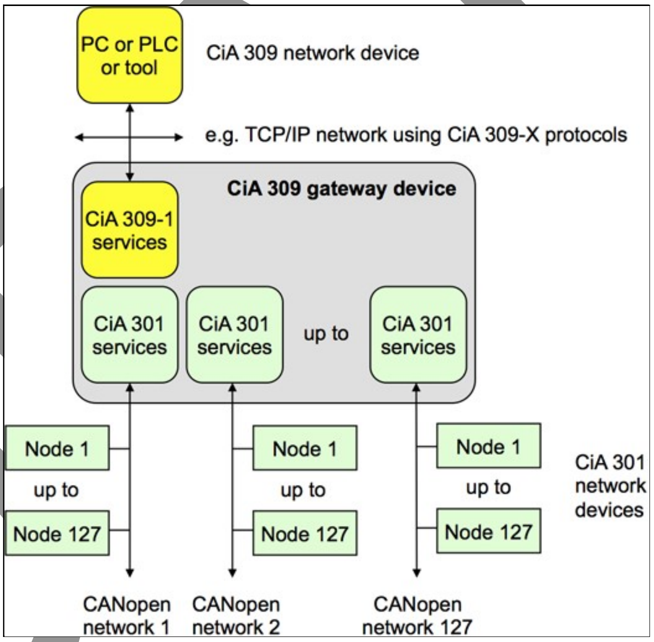
These boxes look very similar to a Modbus/TCP gateway to the serial Modbus versions, but there is a big difference: they have **no** Modbus knowledge. What comes in on Ethernet is transmitted 1:1 over the serial line; what comes in over the serial line is transmitted 1:1 on Ethernet.

So, when the serial communication is Modbus/ASCII, you'll get Modbus/ASCII-on-Ethernet, this is **not** Modbus/TCP. When the serial communication is Modbus/RTU, you still don't get Modbus/TCP, although it may look very recognizable. But remember that Modbus/TCP has an extra header, and no CRC. The converter box will not add the header and not strip the CRC.

Nevertheless, it can be a good solution in certain systems due to its simplicity.

5. Modbus to CAN/Open

The CAN in Automation User's Group (CiA) has developed suite of specifications all describing TCP-to-CAN/Open gateways. Part 2, formally labelled "CiA 309-2" specifically describes the Modbus/TCP protocol.



The CiA 309 TCP-to-CAN gateway (source: CAN in Automation)

By means of the CiA 309 protocols, an application program can access every device in all connected CAN/Open networks. This is mainly useful for remote configuration, remote diagnostics, and relatively low-speed applications. A first version 1.1.0 of the CiA 309-2 was released in 2006; its specification is publicly available from the CiA website. As of 2015, version 1.3.0 of the CiA 309-2 was released, adding support for Modbus extended exceptions. This specification is not publicly available (for members only).

On Modbus/TCP, function code 43 is reserved for the "CAN/Open General Reference". Basically, it uses the data part of the function code to give commands to the gateway what to do.

CiA 309 does **not** allow for CAN/Open "PDO" commands[27]. The "SDO" commands are supported, as are "NMT" (network management) and gateway management commands. Because the function 43 messages are limited by the Modbus specification to contain no more than 253 data bytes, larger chunks of data can be read/written by a sequence of "Extended Requests" and "Extended Responses" which are repeated until a larger chunk of data is transferred. CiA 309-2 follows what the CAN/Open protocol itself does when transferring large chunks of data via CAN itself (where a CAN-message can only contain 8 data bytes).

6. Modbus to SEMI

The Modbus User's Group has developed a specification for the conversion of Modbus/TCP to the SEMI sensor bus. SEMI is the association of vendors active in semiconductor manufacturing equipment, who has written specifications for how to use all popular industrial network protocols in the semiconductor machinery market.

The specification is freely available from the www.modbus.org website. Version 1.1 was released in 2004. It describes two methods of communication; the first using function code 91, the second using the (existing) function code 3 and 16.

7. Ethercat with Modbus/TCP

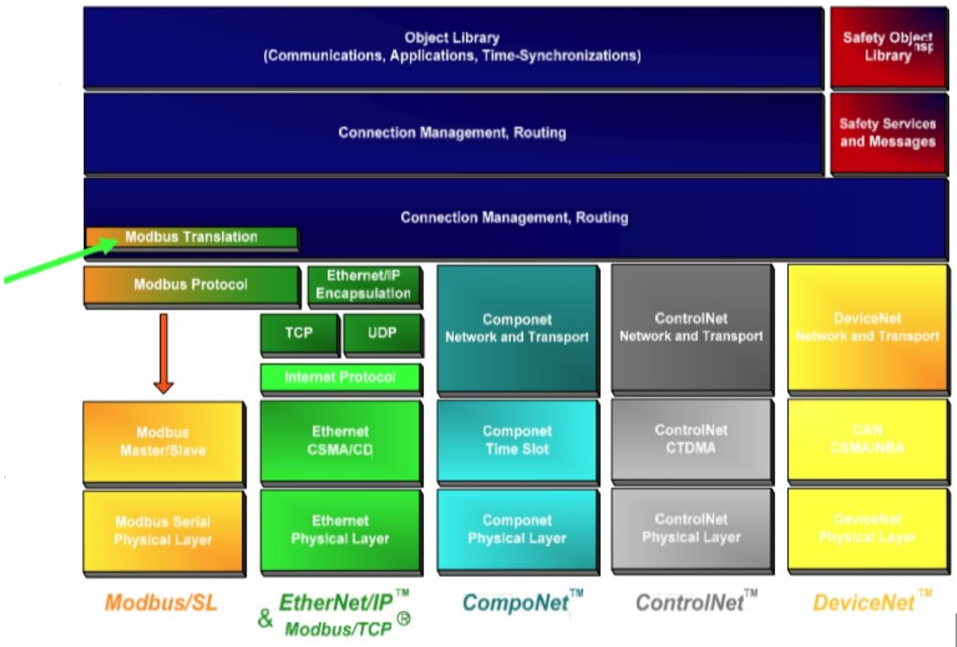
Ethercat is a very high-speed industrial Ethernet variant originally developed by Beckhoff, but now maintained by the ETG User's Group (Ethercat Technology Group, www.ethercat.org). Although it is not possible to convert from Ethercat to Modbus/TCP due to the dedicated Ethercat-controller chips this requires, it is possible to **tunnel** Modbus/TCP communication over Ethercat.

Every network cycle, following the handling of the real-time I/O data, a (user-configurable) timeslot is available for transport of non-real time data. Somewhere in the Ethercat network an Ethernet-gateway can be connected, and the Modbus/TCP messages are then sent out over an Ethernet, on which the Modbus/TCP server can be connected.

8. Ethernet/IP to Modbus/TCP

Ethernet/IP is an industrial Ethernet variant developed by the ODVA User's Group (Open DeviceNet Vendors Association, www.odva.org). Although Modbus/TCP was not initially part of Ethernet/IP, it was added when Schneider became a member of the board of directors of ODVA.

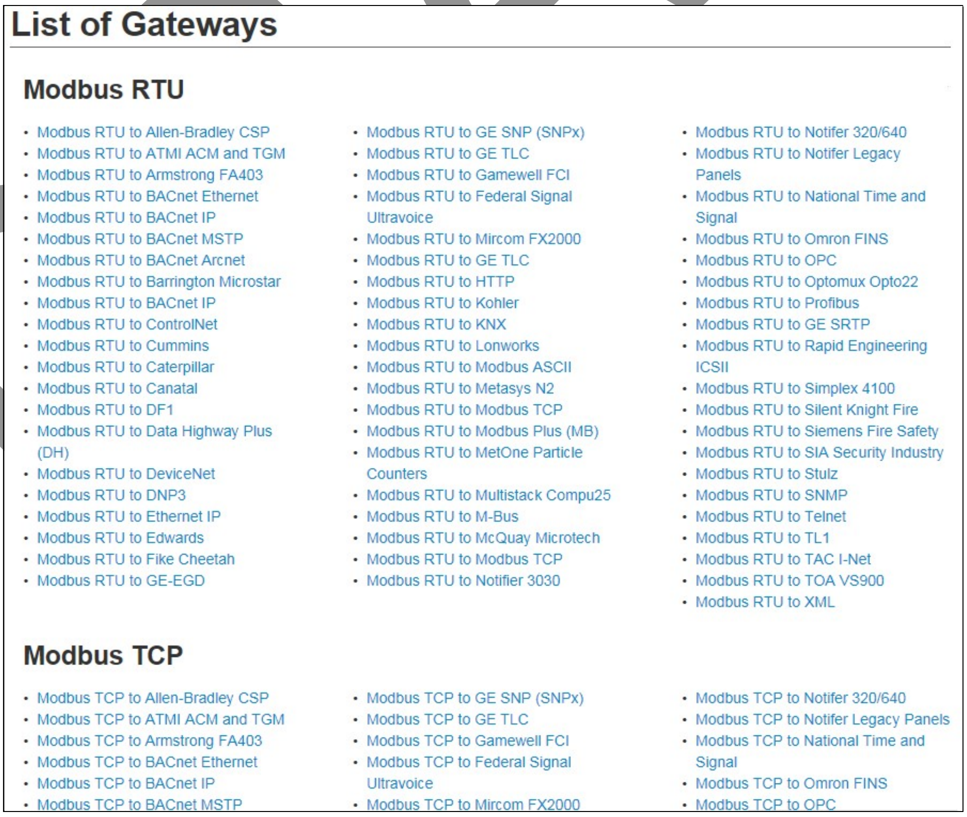
[27] *In CAN/Open, a "Protocol Data Object" is used for fast communication between devices. There can be maximum of 8 per device, they can be send per broadcast at high rates, can contain up to 8 bytes data (as limited by CAN), and have no extra protocol overhead. The "Service Data Object" is slower, has more overhead, can only be sent point-to-point, but they can be very large in size and there can be several thousand per device.*


Object Library (Communications, Applications, Time-Synchronizations)
Safety Object Library
Connection Management, Routing
Safety Services and Messages
Connection Management, Routing
Modbus Translation
Modbus Protocol
Ethernet/IP Encapsulation
TCP
UDP
Internet Protocol
Componet Network and Transport
ControlNet Network and Transport
DeviceNet Network and Transport
Modbus Master/Slave
Ethernet CSMA/CD
Componet Time Slot
ControlNet CTDMA
CAN CSMA/NBA
Modbus Serial Physical Layer
Ethernet Physical Layer
Componet Physical Layer
ControlNet Physical Layer
DeviceNet Physical Layer
Modbus/SL
EtherNet/IP™ & Modbus/TCP®
CompoNet™
ControlNet™
DeviceNet™

Architectural diagram of the CIP family of products and the location of Modbus (source: ODVA).

9. Anything else to/from Modbus

Many other existing protocols can be converted to / from one of the Modbus variants. A quick search on internet reveals many vendors. A subset of the products offered by Chipkin:



List of Gateways

Modbus RTU

- Modbus RTU to Allen-Bradley CSP
- Modbus RTU to ATMI ACM and TGM
- Modbus RTU to Armstrong FA403
- Modbus RTU to BACnet Ethernet
- Modbus RTU to BACnet IP
- Modbus RTU to BACnet MSTP
- Modbus RTU to BACnet Arcnet
- Modbus RTU to Barrington Microstar
- Modbus RTU to BACnet IP
- Modbus RTU to ControlNet
- Modbus RTU to Cummins
- Modbus RTU to Caterpillar
- Modbus RTU to Canatal
- Modbus RTU to DF1
- Modbus RTU to Data Highway Plus (DH)
- Modbus RTU to DeviceNet
- Modbus RTU to DNP3
- Modbus RTU to Ethernet IP
- Modbus RTU to Edwards
- Modbus RTU to Fike Cheetah
- Modbus RTU to GE-EGD
- Modbus RTU to GE SNP (SNPx)
- Modbus RTU to GE TLC
- Modbus RTU to Gamewell FCI
- Modbus RTU to Federal Signal Ultravoice
- Modbus RTU to Mircom FX2000
- Modbus RTU to GE TLC
- Modbus RTU to HTTP
- Modbus RTU to Kohler
- Modbus RTU to KNX
- Modbus RTU to Lonworks
- Modbus RTU to Modbus ASCII
- Modbus RTU to Metasys N2
- Modbus RTU to Modbus TCP
- Modbus RTU to Modbus Plus (MB)
- Modbus RTU to MetOne Particle Counters
- Modbus RTU to Multistack Compu25
- Modbus RTU to M-Bus
- Modbus RTU to McQuay Microtech
- Modbus RTU to Modbus TCP
- Modbus RTU to Notifier 3030
- Modbus RTU to Notifer 320/640
- Modbus RTU to Notifer Legacy Panels
- Modbus RTU to National Time and Signal
- Modbus RTU to Omron FINS
- Modbus RTU to OPC
- Modbus RTU to Optomux Opto22
- Modbus RTU to Profibus
- Modbus RTU to GE SRTP
- Modbus RTU to Rapid Engineering ICSII
- Modbus RTU to Simplex 4100
- Modbus RTU to Silent Knight Fire
- Modbus RTU to Siemens Fire Safety
- Modbus RTU to SIA Security Industry
- Modbus RTU to Stulz
- Modbus RTU to SNMP
- Modbus RTU to Telnet
- Modbus RTU to TL1
- Modbus RTU to TAC I-Net
- Modbus RTU to TOA VS900
- Modbus RTU to XML

Modbus TCP

- Modbus TCP to Allen-Bradley CSP
- Modbus TCP to ATMI ACM and TGM
- Modbus TCP to Armstrong FA403
- Modbus TCP to BACnet Ethernet
- Modbus TCP to BACnet IP
- Modbus TCP to BACnet MSTP
- Modbus TCP to GE SNP (SNPx)
- Modbus TCP to GE TLC
- Modbus TCP to Gamewell FCI
- Modbus TCP to Federal Signal Ultravoice
- Modbus TCP to Mircom FX2000
- Modbus TCP to Notifer 320/640
- Modbus TCP to Notifer Legacy Panels
- Modbus TCP to National Time and Signal
- Modbus TCP to Omron FINS
- Modbus TCP to OPC

CHAPTER 9. Performance calculations

In almost all industrial applications, the system must be able to react on external events within a maximum time. When an industrial network is used, the speed of the network also becomes relevant. The bitrate on the cable is an important parameter; a network running at 115.2 Kbit/s is faster than one running at 9.6 Kbit/s. But: is it fast enough? Before we can say 'yes', calculations need to be done taking into account:

- The number of devices on the network;
- The bitrate;
- The overhead in the network protocol;
- The speed of the software.

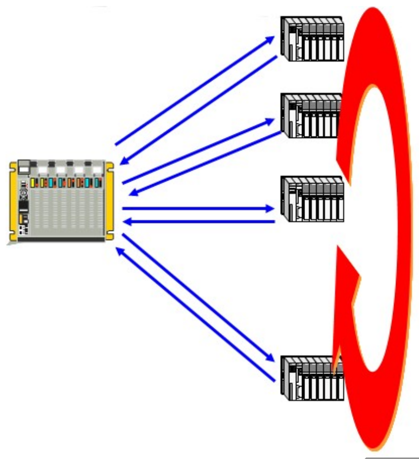
For Modbus, this is no different than for any other network protocol, except for the Modbus-specific details. We will now first discuss a basic performance calculation model, and then fill in the details for Modbus/ASCII and /RTU.

1. Remote I/O scanning

Most remote I/O system, including those using 1st generation industrial network protocols (like AS-Interface, Profibus/DP, Interbus, etc.) and including those using 2nd generation protocols (like ProfiNet, Ethercat, etc.) there is a very simple way of working. The networks have a central controller (i.e. a PLC) running the application software, and all I/O is connected to a group of remote I/O modules.

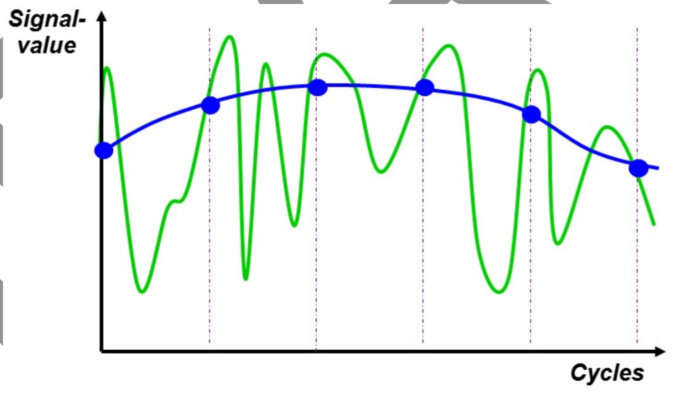
The controller is almost always the master of the network, and the I/O modules are the slaves. Although master/slave network protocols do not allow communication between slaves, this is not a problem for remote I/O systems: inputs do not communicate with other inputs, and outputs not with other outputs (and inputs). The outputs are sent from the master to the I/O modules, and these send the current status of their inputs back to the master.

The master has a list of remote I/O modules (as configured by the user), and one by one the master communicates with each slave: it is sent a network message with new settings for the outputs, and the slave respond with a network message containing the actual values of the inputs. When the master has processed all slaves, one "I/O Cycle" is finished: all outputs have been transferred to the slaves, and the inputs have been read. The time needed for this is the so-called "cycle time". When a cycle is finished, the master immediately starts with the next cycle.



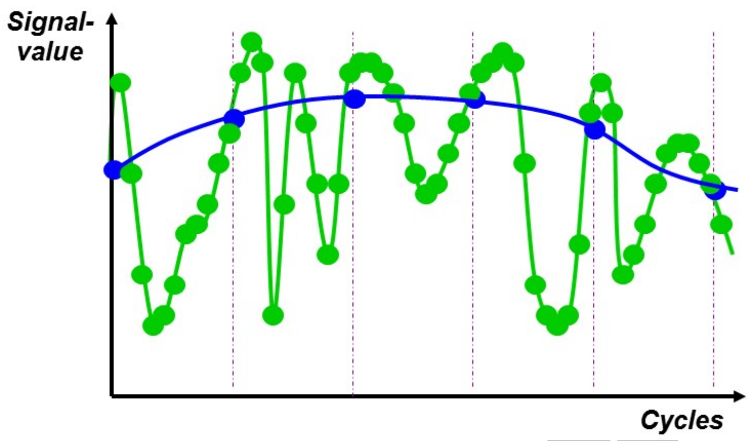
Obviously, the smaller the cycle time, the better: inputs arrive sooner at the controller, so they can be acted upon; outputs are set quicker. A smaller cycle time allows the controller to react quicker, giving a machine or production line more products per hour, or a higher accuracy, or a better quality, or less deviation in mechanical tolerances, etc. It is no wonder that vendors strive to have industrial networks allowing a better[28] (smaller) cycle time than the systems of their competitors.

The value of the cycle time is an important parameter of any industrial network with remote I/O. Is the network fast enough? In every cycle, any input or output is handled exactly **once**. Any input whose value changes quicker than the cycle time is lost; an output can only be set once per cycle time.



A slowly changing signal can be correctly sampled with a given cycle time (blue dots). But the same cycle time is much too slow for the quickly-changing green signal; many important signal changes are not seen.

[28] *The cycle time is **not** directly related to the bitrate of a network. Usually, a higher bitrate will also cause a shorter (better) cycle time. But this is not automatically true; some networks need a higher bitrate just to compensate for (excessive) overhead in their network messages. The result is that efficient industrial protocols can achieve a short cycle time on a low bitrate, and competing protocols need a high bitrate to achieve the same cycle time. For application software, the only thing of importance is the cycle time. Sales persons trying to impress you with high bitrates in their products probably don't know what they're selling.*



When the cycle time becomes ten times faster (green dots), the quickly-changing green signal is now also properly read. In most remote I/O networks, all signals are sampled at the same frequency, so the signal that changes the most determines the cycle time.

Some remote I/O networks allow different cycle times per I/O modules. Fast-changing signals are sampled quicker; slow-changing signals are sampled slower. This doesn't waste bandwidth for network messages that contain the same data over and again. The drawback is that such networks are more difficult to configure, and it is more difficult to calculate the cycle time. We will not go into more detail about these types of remote I/O networks.

Modbus allows for both types of implementations. Consult the vendor documentation of the master.

It is therefore imperative to calculate what the cycle time will be. Luckily, this can usually be calculated in advance, as the behaviour of a remote system is completely known:

- The number of I/O modules is known;
- The number of inputs and outputs on every I/O module is known;
- The network protocol is known;
- The speed (bitrate) of the network is known;
- The behaviour of the master is known.

It can thus be calculated in advance exactly how many network messages are sent each I/O cycle (2 per I/O module), how long these are (in bits), and what their transmission time will be (bits/bitrate). This gives a first-order approximation of the cycle time, which we will discuss in the next section (2). More refinement is possible when also taking software overhead into account (section 3). Even more refinement is possible when we also take into account how a specific network protocol works; we will do this for Modbus/ASCII (section 4) and Modbus/RTU (section 5).

The outcomes of the calculation(s) determine whether the network is suitable for use in the application to be made, i.e. when the cycle time is short enough. When the cycle time is too long, the network must be redesigned, i.e. using less devices, running at a higher speed (bitrate), etc., or: you should perhaps use another network protocol.

2. Simple performance model

A simple performance model [1] for **any** I/O system is given by the formula:

$$Tc = \frac{n \times 8 \times (2 \times Overhead + Inputs + Outputs)}{Bitrate}$$

Where:

Tc	The calculated cycle time (in seconds).
n	The number of I/O modules (slaves) on the network. How many there are is known from the application requirements.
8	The number of bits per byte.
2	One request message (from master to a slave), and one response message (from slave back to master).
Overhead	The number of bytes of overhead in each network message. This information can be found in the network protocol specification.
Inputs	The average number of bytes in a network message containing input data, usually the number of inputs bit divided by 8 and rounded up to the next full byte. How many inputs there are is known from the application requirements. Usually any digital signal counts for 1 bit; an analogue signal for 16 bits.
Outputs	The average number of bytes in a network message containing output data, usually the number of output bits divided by 8 and rounded up to the next full byte. How many outputs there are is known from the application requirements. Usually any digital signal counts for 1 bit; an analogue signal for 16 bits.
Bitrate	The transmission speed of the network in bits/seconds. This value is normally dependent on the length of the network; the longer the network, the lower the bitrate must be. Consult the network protocol documentation on the relation between length and bitrate.

Despite its simplicity, it gives a reasonable first order approximation if the network consists of devices which are similar in nature (processing speed, number of inputs and outputs, etc.).

3. Performance model

In the previous model we made some assumptions about the I/O modules: that they are all similar. This is not realistic; it does not take into account:

- A difference in I/O channels on every device;
- A difference in the overhead for network messages for inputs and outputs;
- Any (software) processing delays between the messages for inputs and outputs.

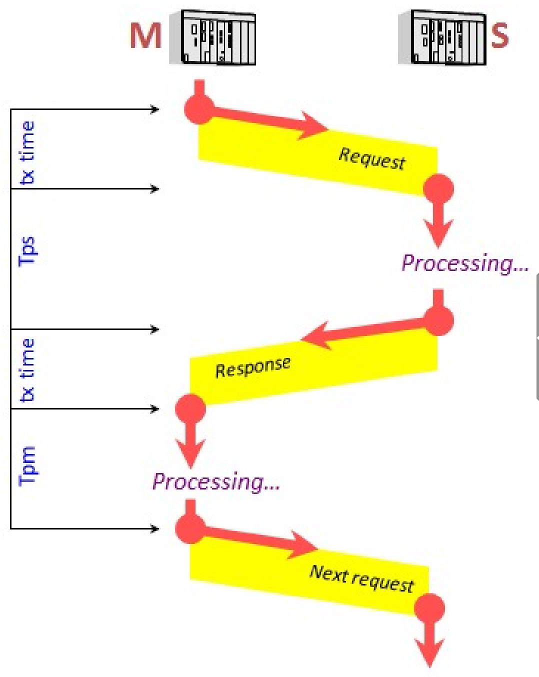
If we take these factors into account, formula [2] is getting more detailed:

$$Tc = \sum_{i=1}^{n} \left(\frac{Overhead_i + Inputs_i}{Bitrate} + Tps_i + \frac{Overhead_o + Outputs_o}{Bitrate} + Tpm \right)$$

Here we have:

$Overhead_i$	Overhead for the network message with input data (in bits) for module i. This information can be found in the network protocol specification
$Overhead_o$	Idem, but for the network message for the outputs (in bits) for module i. The overhead can be the same as for the inputs, but can also be different.
$Inputs_i$	The total number of bits used for inputs on device i. How many bits there are is known from the application requirements. Usually any digital signal counts for 1 bit; an analogue signal for 16 bits.
$Outputs_i$	Idem, but for the outputs on device i.
Tps_i	Pause time between two network messages (in seconds) on the **slave**, usually the processing time for incoming telegrams, and any application overhead. This time will differ per device, but it is very difficult to get the exact value, as vendors seldom publish information about it. In practice the only way to get to know this value is: set up a network, connect the device, connect a network-analyser, and measure the time between the two messages.
Tpm	Pause time between two network messages (in seconds) on the **master**, usually the processing time for incoming responses, application processing, and the time needed to decide on the next request message. This time will probably differ per device. It is the sum of the processing time of the Modbus protocol stack and the application software. It is therefore not possible to ask a vendor how long this time will be, as the vendor does not know about your application software. It is the experience of the author that in many cases software programmers blame the protocol stack for long processing times, while in fact it is often their own application software which is slow!

Graphically we can depict this as:



The reason that the transmission of the request and response message is drawn slightly slanted is because of the transmission delay. When the first bit is sent, it doesn't arrive immediately. The propagation speed of electrons in copper is around 200000 km/s, which sounds enormous but this is equivalent to 5 nanoseconds per meter of cable. For very high-speed industrial networks (like Ethercat) this is of importance, but for most Modbus networks the 5 ns/m is negligible (we will not take it into account in the calculations in the next sections).

4. Modbus/ASCII calculation

Formula [2] given above is valid for most (remote I/O) networks. They send one (request) message containing the new values for the outputs, and the (response) message contains the current values for the inputs.

Modbus however does **not** support this behaviour with a single function code. Instead, **two** function codes must be used, one for writing the output data and one for reading the input data; this makes that in total four network messages are sent. In the example below we assume that function codes 3 and 16 are used, as these are the most common[29]. When using other function codes, the values for the overhead must be changed. We now also add the overhead of the serial transmission format (start bit, parity bit and stop bit) which may seem negligible but is still 3 bits on every 7 bits.

[29] *Some masters have the capability to use function code 23, which allows data to be written **and** read from a slave with a single command. This is considerable more efficient than using function codes 3 and 16. Unfortunately, function code 23 is seldom implemented – and if it is, remember: both the master **and** a slave must support it.*

This gives [3]:

$$Tc = \sum_{i=1}^{n}\left(\frac{(1+d+p+s)\times\left(12+2\times(11+2\times Inputs_{\mathrm{i}}+15+2\times Outputs_{\mathrm{i}})\right)}{bitrate}+2\times Tps_{\mathrm{i}}+2\times Tpm\right)$$

In which:

d	The number of bits per byte in the Modbus/ASCII serial transmission format, which is normally 7 (some vendors use 8 data bits).
1	One start bit.
p	Has the value 1 when a parity bit is used, or else the value 0 when no parity bit is used.
s	The number of stopbits: usually 1, sometimes 2.
12	Four times 3 bytes overhead per network message (start delimiter ":", carriage return CR , line feed LF). These are not expanded from two nibbles to two bytes on the network, so should not be multiplied by 2 as is done for the remainder of the network message.
2	In Modbus/ASCII each byte of application data expands to two bytes on the network.
11	A total of 11 bytes overhead: 7 bytes for the request (slave address, function code 3, starting address H and L, number of registers H and L, checksum), and 4 bytes for the response message (slave address, function code 3, byte count, checksum).
2	A register is 2 bytes.
15	A total of 15 bytes overhead: 8 bytes for the request (slave address, function code 16, starting address H and L, number of registers H and L, byte count, checksum) and 7 bytes for the response message (slave address, function code 16, starting address H and L, number of registers H and L, checksum).
$Inputs_i$	The number of registers that are read on slave i.
$Outputs_i$	The number of registers that are written on slave i
Tps_i	See the description in the previous section. Note: it is assumed that the Tps is identical for both function codes used here.

As can be seen in this formula, Modbus/ASCII has a *lot* of overhead. Each byte of data (8 bits) is converted to 18 or 20 bits on the network cable. More than half of the bandwidth is thus lost on each network message. It is no surprise that Modbus/ASCII is hardly used in moderns system (in favour of Modbus/RTU, which we describe below).

5. Modbus/RTU calculation

Modbus/RTU is much more efficient than Modbus/ASCII as it does not split each byte in two. This gives the following formula for calculating the cycle time [4]:

$$Tc = \sum_{i=1}^{n}\left(\frac{(1+d+p+s)\times(14+13+2\times Inputs_{\mathrm{i}}+17+2\times Outputs_{\mathrm{i}})}{bitrate}+2\times Tps_{\mathrm{i}}+2\times Tpm\right)$$

In which (for the others see above):

14	The 3.5 character delay after each message, times 4 (for the four messages exchanged).
13	Total amount of overhead: 8 in the function code 03 request, and 5 bytes in the function code response (one byte more per message than with Modbus/ASCII, because the CRC is 2 bytes in size while the checksum is only 1 byte).
17	Total amount of overhead: 9 in the function code 16 request and 8 in the function code response (one byte more per message than with Modbus/ASCII, because the CRC is 2 bytes in size while the checksum is only 1 byte).

Because of the smaller transmission time of network messages, the values of Tpsi and Tpm are relatively more important.

6. Modbus/TCP calculation

Although on first thought the calculations for Modbus/TCP will resemble those of Modbus/RTU due to the similarity of the protocol, this is not entirely so: the behaviour of TCP must also be taken into account. This is not so easy to quantify, since it depends on the CPU speed, and the efficiency of the TCP/IP protocol stack implementation[30].

Sometimes one runs into strange surprises. For example, on Windows: in order to have a responsive user-interface (keyboard, mouse), the user-interface has priority for the CPU, and the protocol support comes second. This may result in very varying processing times of incoming network message; the author has experienced with his own software that this can vary from sub-milliseconds to over two hundred milliseconds.

Additionally, the overhead of Ethernet must also be taken into account, plus the overhead of IP, TCP and that of all switches the message(s) pass through.

Any Ethernet message has some overhead: 64 bits synchronisation, 48 bits for the destination network address (“MAC” address), 48 bits for the own network address and 16 bits for the protocol ID. Then comes the data field (see next section), 32 bits for a CRC, and finally each message has a pause time of 96 bits is needed to allow a receiving device to process incoming messages.

Data field

The data field in any Ethernet message must have a length of **at least** 46 bytes. When a message contains less than this amount of data, it must be padded with zero bytes. So, whatever it takes, an Ethernet message is always *at least* 64+48+48+16+46*8+32+96 = 672 bits long.

Inside the data field, the TCP/IP fields are stored. Both protocols have their own overhead, which is:

- 20 bytes(at least) for IP, and
- 24 bytes (at least) for TCP.

[30] *Sometimes one runs into strange surprises. For example, on Windows: in order to have a responsive user-interface (keyboard, mouse), the user-interface has priority for the CPU, and the protocol support comes second. This may result in very varying processing times of incoming network message; the author has experienced with his own software that this can vary from sub-milliseconds to over two hundred milliseconds.*

Luckily, by chance (or by design ?) this fits snugly in the 46 byte Ethernet minimum data field, so in this case using TCP/IP adds **no** overhead to any Ethernet message. There are even 2 bytes left for other use, but this is not enough for Modbus/TCP.

Following the TCP/IP header comes the 5-byte Modbus/TCP header, which is itself followed by the function-code specific data. For example, a function code 3 request will add another 5 bytes. So the total length of the Ethernet message is 64+48+48+16+(20+24+5+5)*8+32+96=736 bits. This is much more than the same function code uses in Modbus/ASCII or Modbus/RTU, but in Ethernet the transmission speed is much higher. At 100 Mbit/s, the 736 bits cost only 7.36 µseconds transmission time, probably negligible in comparison to the software overhead in the client and/or the server.

However, the best 'win' is the enormous raw speed of Ethernet, from 10 to 100 to 1000 Mbit/s and sometimes even 10000 Mbit/s. In practice we see that the speed of Modbus/TCP is therefore greatly dominated by the software implementations on the client and the server, and the operating systems then run on.

7. Notes on performance problems

Although Modbus is generally not used in applications requiring high-speed performance (i.e. with millisecond cycle-times), sometimes it can still be too slow, even in non-demanding applications. What can you do to increase to increase the speed of the network? Formulas 3 and 4 give insight.

1. The easiest solution seems to be: increase the bitrate. This decreases the transmission time of network messages. However, the Tpm and Tps do not decrease, so the gain in cycle time is not proportional with the increase in bitrate. Also, on RS485 networks an increased bitrate might mean that you exceed the maximum network length, and that the network is more susceptible to errors if the quality of the cabling and connectors is bad.
   When using Ethernet, increasing 'the' bitrate will probably help very little, as modern Ethernet devices usually (automatically) choose the highest bitrate possible (via the "autonegotiation" feature) per device. And even then, Ethernet is so fast that the transmission times are probably negligible in comparison to the Tpm and Tps.

2. Decrease the amount of data to be transferred. This helps a bit, but not much in relation to the overhead of the Modbus protocol.

3. Use Modbus/RTU instead of Modbus/ASCII. This helps a lot, especially on low bitrates. But note that **all** devices must be able to run Modbus/RTU (you cannot mix Modbus/ASCII and RTU on the same network).

4. Decrease the Tpm[31] and Tps[32]. This is one of the most effective ways of decreasing the cycle time, but usually impossible for ordinary users unless you have access to the source-code of

[31] *The author once worked with a master with had a Tpm of more than 70 milliseconds. After seriously complaining to the vendor and ignoring their complaints "What you want is impossible" they managed to bring this back this to less than a millisecond, giving a big boost in application performance.*

the master and (all the) slaves.

5. Use a network with fewer devices. For example, splitting a network in two smaller networks, each with half the devices of the original network, will decrease the cycle time by 50%, at the expense of having to buy an additional Modbus master.

[32] *A customer once had a Modbus/RTU slave that could only process 3 commands per seconds, despite a high bitrate. After investigation it turned out that the slave had a Tps processing-time of 300 milliseconds per command, this immediately explains why only 3 commands per second could be given (remember: the master* ***must*** *wait for the answer from the slave before it can continue).*

CHAPTER 10. Implementing Modbus

Using Modbus is something entirely different than *implementing* it. In this chapter I provide some tips for fellow programmers, based on my own experiences in making Modbus protocol stacks. When I started doing this, there was no internet and no open-source. Nowadays, first look at what is available on internet (start on: www.modbus.org). Perhaps somebody already made something that you can use directly. But perhaps there's nothing there, or not possible in combination with your application, or the licensing terms are too costly… then you still have to start programming yourself.

1. Serial I/O or Ethernet?

A serial port is cheap, and easy to program. At low bitrates (say, <= 38.4 Kbit/s) it should pose no problem for any modern CPU. Things to watch out for:

- With Modbus/ASCII, 7 bits characters must be allowed, or else 8 bits for Modbus/RTU.
- The parity bit should be programmable (odd, even, none).
- The number of stop bits should be programmable (1 or 2).

When transmitting network messages (a string of characters), care should be taken that the characters are transmitted with no intermediate pause between them (the so-called 'back to back' transmission). The exact amount of time often depends on the speed of the (transmit) interrupt handler of the serial driver, which in turn depends on the proficiency of the programmer that wrote the code. This becomes especially visible at the higher bitrates.

The problem can be alleviated when the serial port electronics (a so-called "UART" – Universal Asynchronous Receiver/Transmitter) has a built-in buffer in which much data can be stored. The UART autonomously empties this buffer, without any software interfering. On PC's, the so-called "16550" chips have a 16-byte transmit buffer. Have the Modbus software fill this buffer with one software command, instead of with many commands with only one byte.

When using serial chips controlled over USB, be aware that the USB-driver software is also active, this may cause larger pauses between characters.

When the pauses become too large, the serial receiver at the other end of the cable may conclude that the network message has ended, and that a new one starts after the pause. As the first part is not a complete Modbus message, and the second part does not look like Modbus at all, both parts are silently ignored by the receiver. The master will get no response back, and report an error.

Modbus/RTU intermessage pause

In Modbus/RTU, the maximum pause time is set (by the specification) to 3.5 times the transmission time for a single character. It is a fixed value, which cannot be configured differently. So the transmitter must make sure that there is never such a large transmission delay between two consecutive characters.

Unfortunately, the 3.5 character time limit is seldom enforced. First, the .5 part is very difficult to implement, so many vendors set the limit to 4, or higher. This no problem due to the half-duplex way of working of Modbus; when a receiver needs more time before it decides that it has received a complete network message, it is just responding a little bit slower.

2. TCP/IP

Any implementation of Modbus/TCP requires a TCP/IP. Nowadays, these are very common and probably out-of-the-box available in any embedded platform, with a standard “sockets” API (Application Programmers Interface). Sockets are the standard way of working with TCP/IP for some 30 years, and although there are vendors that offer their own API (i.e. Microsoft with WinSock) I’d recommend using use those API’s only when needed.

Because socket interfaces are so stable, a lot of documentation and example code can be found on internet. It is not difficult to comprehend, if you have basic knowledge of TCP/IP. Because little has changed in these 30 years, the “old” examples still work and they may server as a good starting point for your own implementation.

Note that any Modbus/TCP device requires an IP-address to work, this in contract to serial Modbus masters which do not need a network address. The IP-address can be either dynamically set (via DHCP) or statically; it is up to you to decide which one is needed.

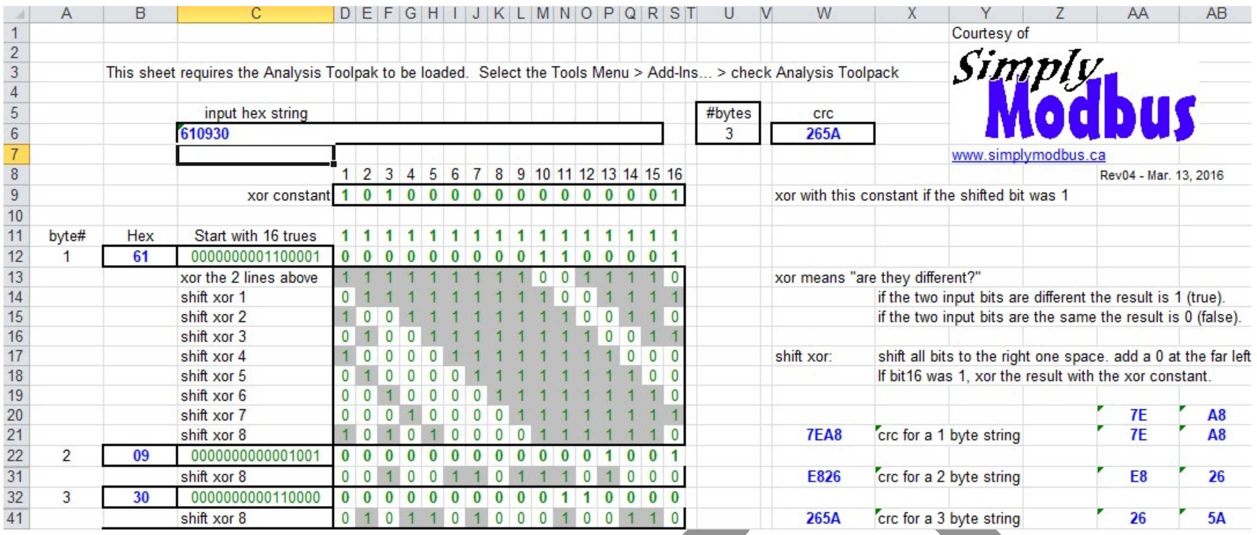
For industrial applications, often statically allocated IP-addresses are used; how this is to be implemented is up to you (it is not mentioned in the Modbus/TCP specification). I’d recommend to always allowing a customer the choice between having statically or dynamically allocated IP-addresses. Note that when a device is configured to use a dynamically allocated IP-address, the network protocol software cannot communicate before the IP-address has been assigned.

3. CRC Implementation

When implementing Modbus/RTU, each message must have the CRC (Cyclic Redundancy Check) calculated on each message:

- For received messages: to detect whether it is not damaged while in-transit.
- For messages to be sent away: to append the calculated CRC to the end of the data.

Calculating the CRC must be done according to the algorithm specified by Modbus, a 16-bit CRC. In [MBUS300] an example of an implementation in C is given; on internet more examples can be found for other programming languages.



This sheet requires the Analysis Toolpak to be loaded. Select the Tools Menu > Add-Ins... > check Analysis Toolpack

Courtesy of Simply Modbus www.simplymodbus.ca

Rev04 - Mar. 13, 2016

input hex string	#bytes	crc
610930	3	265A

byte#	Hex		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
		xor constant	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
		Start with 16 trues	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	61	0000000001100001	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1
		xor the 2 lines above	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	0
		shift xor 1	0	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1
		shift xor 2	1	0	0	1	1	1	1	1	1	1	1	0	0	1	1	0
		shift xor 3	0	1	0	0	1	1	1	1	1	1	1	1	0	0	1	1
		shift xor 4	1	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0
		shift xor 5	0	1	0	0	0	0	1	1	1	1	1	1	1	1	0	0
		shift xor 6	0	0	1	0	0	0	0	1	1	1	1	1	1	1	1	0
		shift xor 7	0	0	0	1	0	0	0	0	1	1	1	1	1	1	1	1
		shift xor 8	1	0	1	0	1	0	0	0	0	1	1	1	1	1	1	0
2	09	0000000000001001	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
		shift xor 8	0	0	1	0	0	1	1	0	1	1	1	0	1	0	0	0
3	30	0000000000110000	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
		shift xor 8	0	1	0	1	1	0	1	0	0	0	1	0	0	1	1	0

xor with this constant if the shifted bit was 1

xor means "are they different?"
if the two input bits are different the result is 1 (true).
if the two input bits are the same the result is 0 (false).

shift xor: shift all bits to the right one space. add a 0 at the far left
If bit16 was 1, xor the result with the xor constant.

		7E	A8
7EA8	crc for a 1 byte string	7E	A8
E826	crc for a 2 byte string	E8	26
265A	crc for a 3 byte string	26	5A

Calculation of a CRC is a CPU-intensive task, reason to look for an efficient implementation. The example in [MBUS300] uses a (pre-calculated) lookup table, this is very efficient, but at the expense of needing more memory. Implementations which do not need much memory also exist, but are slower.

The 2 bytes of the calculated CRC must be appended to the message in the right (MSB, LSB) order. In [MBUS300] this is explicitly mentioned. But depending on the endianness of the CPU your implementation is using, the bytes might accidentally get swapped. The receiver of your messages will then assume the message is damaged, and ignore it. It is the experience of the author that it is *very* easy to goof up here. I always check my implementations against publicly availably tools on internet[33].

4. Master or slave?

It is probably very early known whether a master (client) is to be implemented, or a slave (server). The distinction is important, because the implementation of a master is considerably more complex than the implementation of a slave. If your device operates as a slave, it cannot communicate with other slaves. But when it is a master, there can be only one on the network.

Usually, it is not a problem to determine what the role of a device should be:

- PC's, SCADA systems, PLC's, and other intelligent devices that are to be programmed by the user will have the "master / client" role
- All others (usually I/O) will have a "slave / server" role.

[33] *You do **not** want to check your master's implementation against your own slave's implementation, because if both implementations are wrong, they accept out each other's wrong CRC and everything will work – but not with any other Modbus implementation in the world.*

5. Memory map of slaves / servers

Before starting any implementation of a slave / server, the network map of the device needs to be specified:

- Which "coils" are made available?
- Which "inputs"? In many devices there are no. Note that digital and analogue inputs can also be represented as data in (read-only) holding registers; this saves the implementation of several function codes.
- Which "outputs"? In many devices there are none.
- Which "holding registers"?

Since this is all very application dependent, not much general advice can be given.

Performance optimization

For optimum performance, it is optimal to use only sequential coils/inputs/outputs/holding register, i.e. no "hole" in them. This is because Modbus does only support reading / writing data of a consecutive group. When this is not so and there is a hole in the data, the master must issue 2 read / write commands.

Additionally, put the most commonly used coils/inputs/outputs/holding registers together. For example, registers 1 and 50 can be read with one function code, but all the intermediate registers must then be read as well. It is better to put these two in registers 1 and 2, so they can be read with the command to read just these 2, instead of reading all 50 and then discarding the data for registers 2..49. On Modbus/TCP this could possibly be acceptable given the high speed of Ethernet, but on the much slower serial links it could be a performance problem in Modbus/RTU and even more so in Modbus/ASCII.

Next, develop the architecture of this data:

- What happens when a "holding register" is written - what should the application software *do* ?

Given the memory map of the device, it is now also known which function codes need to be supported. Additionally, it can be decided to implemented additional function codes, i.e. 08 for diagnostic purposes.

Everything in the holding registers

It may also be worthwhile to represent the coils, inputs and outputs somewhere in the holding register memory map too. This sound strange, why do this when Modbus specifically has separate function codes for them? This is because some masters do not support the associated function codes, meaning that the application software on those masters cannot access your device's coils, inputs and/or outputs.

But access to the holding registers with function codes 3 and 16 is almost always available. So what you can then do (for example), is to define a holding register to 'shadow' 16 coils. A coil can then be set via using function codes 6 or 16, and also via the function codes 5 and 15. Building in such versatility in a slave/server helps the application programmer a lot!

The same trick can be used for the inputs and outputs. For Modbus this does not make any difference, it does not know where data comes from or where it goes to – that is completely up to the device software.

6. Which function codes ?

As we have discussed earlier, the Modbus specification gives considerable freedom in deciding which function codes are supported in any given device. The minimum set of function codes to be supported are 3 and 16, always implement these as this gives the best chance that your device can communicate with others.

The other function codes can be added later. Once you have 3 and 16, adding others is not much extra work.

7. Parallel communication

In Modbus/ASCII and Modbus/RTU, it is not possible to communicate in parallel with multiple slaves – a master **must** wait for the response from the first slave *before* it may send a request to the next slave. So if the first slave is slow in answering, it holds up the communication with the others. Unfortunately there is nothing that can be done about that. So a master implementation can simply handle each application request one by one, for example via a FIFO queue.

In Modbus/TCP, this *can* be done differently, but this complicates the design of the master, as it must be able to keep track of the communication status of multiple servers.

CHAPTER 11. In practice

1. Common mistakes

When implementing a Modbus network with new equipment for the first time, most people run into problems: it just doesn't work! The master gives errors, no devices seem to communicate, and if they do, they send/receive the wrong data, or are too slow, etc.

Network problems can arise in many places, and they all seem to have the same outcome: no communication! Haphazard searching for the root-cause doesn't help much. Instead, follow a structured fault-finding methodology, in principle following the OSI-model bottom-up: physical layer (wiring), datalink layer (Modbus protocol, serial communication), and application layer (Modbus function codes).

Physical layer

- Correct cross-connection of TX and RX signals on RS232.
- One device supports RS232 while the other(s) RS485.
- Devices with 2-wire RS485 and 4-wire RS485.
- Correct wiring modem of control signals (when applicable). Use a breakout box to easily check this. When the TX LED on the breakout box is blinking, it indicates that data is being transmitted.
- Presence of two sets of terminators on RS485. Missing terminators are a major source of network problems, especially at high speeds and/or long networks.
- Is the cable plugged in?
- Is the device powered and is the network protocol software running?

Datalink layer

- Identical bitrate on all devices.
- Identical number of data bits, stopbits, parity bit settings on all devices
- Identical protocol (ASCII, RTU) on all devices.
- Unique network address on all devices set.
- When using an own implementation of the protocol: correct implementation of the checksum / CRC algorithms (when in doubt, this is best verified with a Modbus test tool running on a PC. There are many such tools to be found on internet, some for free.

Presentation layer

- Master doesn't understand data format(s) of a slave.
- Slave doesn't understand data format(s) of the master.

Application layer

- Using the correct slave addresses.
- Network port available and properly configured.

- Application software running.
- The master uses a function code not supported by a slave.
- The master uses broadcast which is not supported by a slave.
- Error messages showing no exceptions, retries, etc. But: if you see exceptions coming back, this shows that the physical layer and datalink layer are OK, since the exception is sent by a slave.
- The timeout setting on the master is too short.
- More than 1 master.
- No master.

Another trick that sometimes helps to find where the problem resides is: use another master, for example a Modbus test tool, where you can manually control which function codes are sent, and when. If this works, it is proof that the slave is functioning, the wiring is correct, as are the datalink settings. If the test tool is then replaced by the normal master, any failure in communication is most likely caused by the master.

2. Visual diagnostics on serial interfaces

Since [MBUSSERIAL] it is mandatory for vendors to have at least one LED, but preferably two LEDs, to aid the user in visually diagnosing the status of a device. But as there was no such requirement in [MBUS300], many Modbus devices have no LEDs, or if they have one, they behave differently than specified in [MBUSSERIAL].

A yellow LED should be switched on during network message reception or sending. It is also allowed to have two yellow LEDs, one for message reception and the other for message sending.

A red LED should be switched on when the device has an internal fault. It may be flashing for communication faults or configuration errors.

Additionally, an optional green LED should be switched on when the device is powered.

3. The “off by one” problem

Many Modbus-users encounter this: a programmer wants to read register (or coil, or input, or output) “X”, but never is the correct data returned. It is always the data of register “X+1” (or “X-1”). The same holds for writing: one writes to register “X”, but the data shows up in register “X+1” (or “X-1”). After a lot of frustration the programmer modifies his software to access register “X-1” in order to access the intended register “X”[34]. Why this behaviour?

Remember that the Modicon memory maps always start at 1; there is no coil / input / output / register 0. But on the network (in the messages) the counting starts at 0. So, many vendors have the

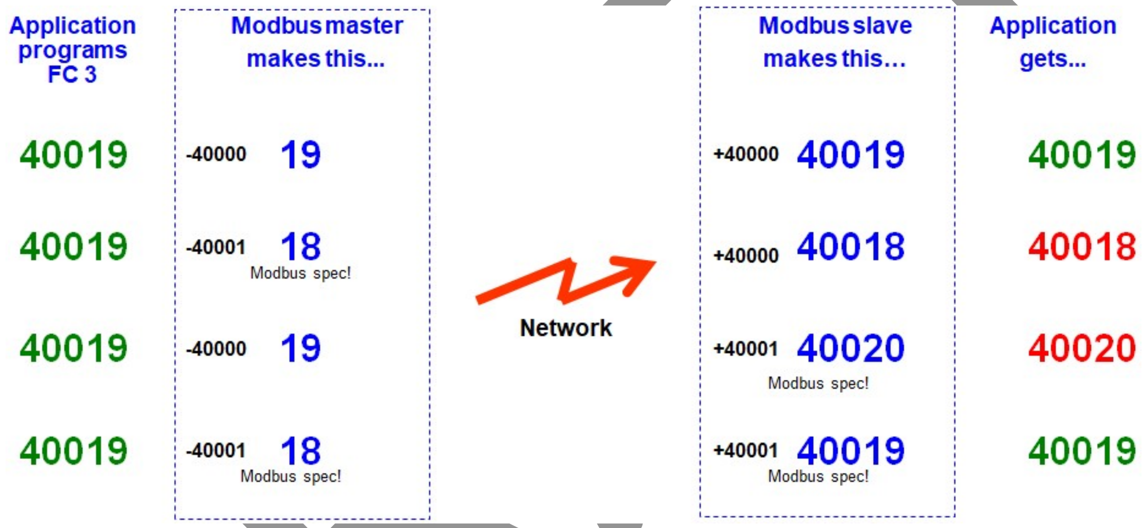
[34] *And then later comes a colleague, who thinks there is a bug in the software and modifies it to use value X again. To his surprise the application now doesn’t work anymore.*

software subtract 1 for you automatically. For example, a program may want to access register 40019, and then on the network you see 40018. The slave / server gets the 40018, adds the 1, and then we access register 40019 again. So if all is OK the -1 / +1 is completely transparent for the user.

Now the problems begin with vendors that do **not** subtract the 1 before transmission. On the network one sees (same example) register 40019. If the slave still adds the 1, it will now access register 40020.

There also vendors that do **not** add the +1. If you now have a master / client that subtracts the 1, but the slave / server does not add +1, you are now accessing register 40018!

You could be lucky, with a master/client that doesn't subtract the 1 and a client/server that doesn't add the +1. In this case everything works as expected.



So we have 4 possible combinations of devices, those that do/don't subtract 1, and those that do/don't add 1. It is on the master / client where this becomes visible, and application software can become real ugly if both types of client/servers are present in the network.

When using JBus

In case you are using JBus devices on a Modbus network, note that JBus does **not** subtract 1 and add 1.

4. Register 9999 and higher

In the original Modbus specification [MBUS300] the maximum coil / input / output / register is 9999. Some vendors of masters / client take this limit *very* literally, and do not allow a program to access anything beyond the 9999. Although for 99% of the applications this is not an issue (as a system with 9999 registers is quite large) sometimes it is. For example, on modular equipment, it could be so programmed that extension module 'n' on such a device has registers 'n * 1000' and up.

In the Modbus protocol, there is no reason to put a limit at 9999, and it has disappeared from the current specification [MBUSAPPL]. The fields in the network messages are all 16 bits, allowing access to coils / inputs / outputs / registers numbered from 0 up to 65535. Many vendors allow full access to this range, but you are warned: some don't. There is no way to circumvent this limit as it is probably hardcoded in the software.

5. Sender too fast

Sometimes you run into a (Modbus/RTU or Modbus/ASCII) device which has occasional loss of messages because it cannot handle the pace with which characters are coming in. This shouldn't happen, after all the vendor has specified that this bitrate could be used with his device (or perhaps the application load on the CPU is too high).

The solution usually chosen is to have the network and all its equipment run on a lower bitrate. But the next lower bitrate is often *substantially* lower than what you're using now, i.e. from 38400 to 28800, or 19200 to 9600. This means that the application software also has to wait longer for its data, or slaves will respond slower to devices.

But there's another trick possible: use an extra stop bit. When the devices now use one stop bit *but* all of them can be configured to use **two** stop bits, do this! The extra stop bit will cause 10% (one extra bit on 10) extra transmission time per character, which might just be enough delay to allow the slow device to handle the incoming characters.

6. Timeout handling

When a Modbus master has a timeout setting on a request, it is sometimes seen that the timeout is too short. An error is given for nothing, as the response will arrive a little bit later (but will be ignored). The only way to be sure that a timeout is not too short, is to measure (for example with a network analyser) the time spent between a request and a response, and the variation in these values. This is usually accurate enough, as the master's internal processing time is negligible in most systems.

Note that on many master protocol stacks there is only **one** setting of the timeout value, even though it may differ per slave. In such a system the slowest slave determines the timeout value for all others as well.

The length of a timeout may also differ on the number of registers or bits to be processed. In the first place the slave has more work to do internally. In the second place the network telegrams are (much) longer. It is not uncommon to see that when an application program is modified to process larger blocks of data, suddenly timeout values may occasionally appear to be too short.

7. Polling / scanning

In PLC's, it is custom to "poll" or "scan" all devices on a network for their inputs and outputs. This is a very simple way of accessing the I/O of any remote I/O module, no application programming is needed and just configuration needs to be set.

In Modbus, this is not so easy as it sounds, due to the enormous diversity in Modbus equipment, and the lack of standardization regarding implemented function codes, registers, and data types (as we have discussed earlier). This means that "plug & play" running of a network is not possible, unless the master can be configured to handle all existing variation in equipment[35] (or can only handle the Modbus devices of the vendor itself).

Because a Modbus slave may never autonomously send data to a master, the only way for a master to remain up-to-date with the state of each slave is to regularly "poll" it – ask the slave for its status, actual input settings, any error, or any diagnostic data. How often this must be done depends on the application software: every second? Or once per minute? Or as fast as possible? The master must be able to handle these application wishes.

For a programmer, it is convenient to cyclically poll all slaves in the same rate, for example 1-2-3-4-1-2-3-4-1-2-3…. But it can be done differently, for example poll one slaves at a higher rate because its data is needed faster: 1-2-1-3-1-4-1-2, etc. Modbus doesn't care[36] how the application on the master wants to control its slaves!

Polling / scanning of slaves / servers has its consequences regarding the rate with which information can be sent to the master / server. See chapter 9 for the performance calculations.

8. Maximum number of masters

The maximum number of masters on Modbus/ASCII and /RTU networks is simple: just 1. It is a gross error to have more than one; this will lead to very unpredictable results as the masters do not expect each other and will disturb each other's network-traffic in (seemingly) random moments. Also, the slaves may not be able to process a second incoming command while they are still processing the first command. Additionally, both masters receive responses from slave which they don't expect.

There is one exception to the single-master rule: when both masters **never** work simultaneous. For example, in a high-availability application, one of the masters can be 'active' and the second one in 'standby'. The standby master doesn't do anything, except keeping an eye on the first master, and taking over its duties when it fails.

[35] The author once attempted to implement this. And then, when I thought I could handle all existing devices in the world, yet another way of implementing Modbus showed up at the next customer.
[36] This is contrast with many modern fieldbus protocols like Profibus/DP and AS-i; when a slave doesn't hear anything from the master within a certain (configured) interval, it assumes the master is no longer operational, and then the slaves set their outputs to a safe state (usually: 0).

When using Modbus/TCP, an Ethernet-network may support multiple Modbus/TCP clients. This is because TCP/IP separates the network-traffic from these clients from each other; they do not disturb each other's messages as on the serial Modbus networks. However it may not be guaranteed that when two clients communicate with the **same** server, it still works.
But there should be no problem when two (or more) clients communicate only with their own servers, even when they are all on the same network.

It is even possible to run two (or more) Modbus/TCP clients on the same device (i.e., a PC with Windows or Linux), also because TCP/IP separates the network traffic. This also allows a device to be both a Modbus/TCP client **and** Modbus/TCP server simultaneously! Beautiful as it sounds, most implementations of Modbus/TCP do not support his.

9. Maximum number of slaves

The maximum number of slaves on any Modbus-network is determined by the lowest value of:

a) The electrical implementation: 1 when using RS232, 31 when using RS485 without repeaters, 32+n*30 when using RS485 with 'n' repeaters, or infinite (that is, 2^48) when using Ethernet.
b) The number of possible network addresses: 247 (or 255, see below) when using Modbus/ASCII or /RTU (due to usage of 8 bit network addresses), or 2^32 when using Modbus/TCP (due to TCP/IP using 32 bit network addresses).
c) The support for a number of slaves in the software running on a PC, or in a PLC or other (industrial) controller. Check the documentation of the equipment for the limitations.

Furthermore, you should calculate whether the network is fast enough to support the intended number of slaves. Application software usually has a limit on the time needed to refresh the status of all slaves, the time needed to respond on certain events, etc. On networks where speed is of importance, calculate in advance whether the application requirements are met.

Surprisingly, with networks running at a high bitrate and well-written implementations in the master and the slaves, Modbus can be very fast. However, usually this doesn't come "out of the box" as with modern industrial networks like Profibus and Ethercat; a certain amount of engineering effort will be needed.

Now what is the real limit: 247 or 255?

As stated above, there may be up to 247 slaves on a Modbus/ASCII or Modbus/RTU network (specified as such in [MBUS300] and also in [MBUSSERIAL]. But since a complete byte is reserved in the Modbus messages for the address, values 1…255 are theoretically possible (the 256th value, 0, is used for broadcasts). There is nothing in the Modbus protocol forbidding the use of network addresses 247…255, and indeed many vendors allow this.

10. Using broadcasts

In [MBUSSERIAL] it is specifically stated: “All slave nodes **must** recognize the broadcast address”. This is an improvement on [MBUS300], which did mention the usage of broadcast addresses, but only “… which all slaves recognize”.

Mixed operation

Apparently the author only had Modicon PLC’s in mind, because a lot of Modbus/ASCII or Modbus/RTU devices do **not** support broadcast. Theoretically, this means that a master would not be able to send messages to the broadcast address and you would have to send the same message to each slave individually (causing a big performance hit).

Luckily, there is an escape which can sometimes be used. Remember that when a slave does not support broadcasts, it will ignore any message sent to this address. But if there other slaves on the network that **do** support broadcasts, they will still react on the message. That some others don’t is completely irrelevant, all slaves operate individually. So this means that you can still send a message to the broadcast address: those devices that support broadcast will react on it, those devices that don’t support broadcasts will ignore it[37].

Processing time

When a message is sent per broadcast, no response is sent back by the slaves. This is done to prevent that multiple response messages are transmitted simultaneously over RS485, destroying each other. But how then does the master know that all his slaves are ready processing the message? This depends on the processing speed of each slave, which may differ considerably.

The master should make an “educated guess” about the processing speed of the slowest slave, and only after that period may a new command be sent. If the master is consistently too fast, the slowest slave’s memory might fill up with unprocessed commands and at a certain moment memory runs out and command may get lost[38].

Modbus/TCP

TCP does not support broadcasts, so this functionality is not available in Modbus/TCP. Sometimes this is a problem when a system is upgraded from Modbus/ASCII or Modbus/RTU to Modbus/TCP, then the application may have to be modified (partially rewritten) if it used broadcasts.

11. Limits on the message length and the data

Modbus messages cannot contain an arbitrary amount of application data. This is limited by:

- The amount mentioned in the Modbus specification [MBUS300] or [MBUSSERIAL]
- The presence of a 1-byte “NumberOfBytesThatFollow” field in several Modbus messages, with a possible value in the range 1..254.

[37] Technically this is called a “multicast”- one message to a selected group of devices.

[38] This is not a typical Modbus problem; this occurs on any network using too many broadcasts. Use them sparingly!

- Any (lower) limits set by the vendor, this is usually caused by having only limited resources present on a CPU (such as memory, processing power). Several examples for Modicon PLC's are given in [MBUS300, Appendix B], but should always be documented by the vendor.

For example, the number of coils / inputs/ outputs / registers than can be read / written with one command must fit in 240 bytes (= 120 inputs / outputs / registers, or 1920 coils). These values are those as mentioned in [MBUS300], and even though they are related to the capabilities of no longer existing Modicon PLC's, somehow they stuck and many vendors use the same limitations.

When using JBus (or with some Modbus vendors!), the limits are slightly larger:

- Read coils:	2000	(250 bytes)
- Read inputs / outputs / registers:	125	(250 bytes)
- Write coils:	1968	(246 bytes)
- Write outputs / registers:	123	(246 bytes)

Where do these higher limits come from, and why are they not *much* higher? This is because they are related to the maximum length of a network message, which is 256 bytes[39], which in turn is enforced by the usage of an 8-bit checksum in Modbus/ASCII (note that the starting colon : and the CRLF are not taken into account in the 256 bytes, because the checksum is not calculated over these 3 bytes).

Subtract from this the sizes of the message header fields (address, function code, checksum), and the remainder[40] can be used for application data.

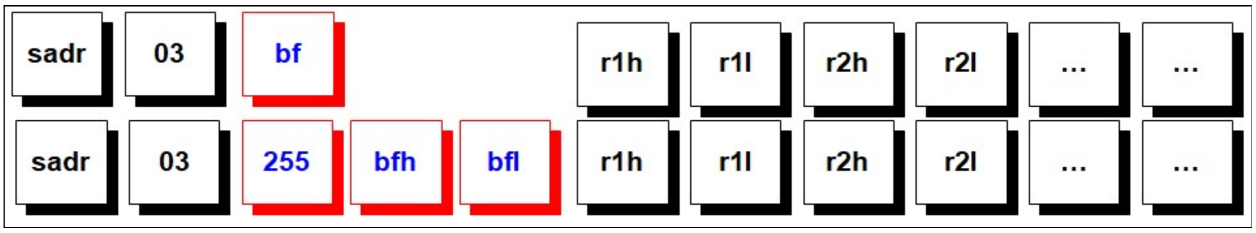
The usage of a 16-bit CRC in Modbus/RTU would have allowed for much larger messages, likely this was not done for compatibility reasons, as it would result in two completely different types[41] of serial Modbus.

When two devices on a network are going to communicate with each other, the smallest maximum message length of the two must be enforced. This must be programmed / configured as such on the master / client. Check the vendor documentation to see whether there are lower limits than allowed by the Modbus protocol.

Unofficial extension to Modbus/TCP

Some Modbus/TCP vendors support an **unofficial extension** to function code 3 (Read Multiple Registers), allowing reading much larger amount of data, theoretically up to 64 Kbytes per command. This is beneficial on Ethernet, which allows the transfer of such large amounts given the high bitrates of 100 Mbit/s or 1 Gbit/s, giving a big gain in performance.

[39] *This limit is specifically mentioned in [MBUS300] in a few places.*
[40] *It is unclear why Modbus/ASCII has set the limit to 240 bytes, perhaps the original designers thought that this number would be easier to remember than 246 or 250, also depending on the direction the data goes.*
[41] *This is a guess, the real reason is lost in the fog of history.*



The extension makes use of the fact that the field “NumberOfBytesThatFollow” (bf) must normally always have an even value (because the number of bytes is twice that of the number of registers). The value 255 is now used to indicate that the *next two bytes* are now going to contain the real number of bytes that is going to follow. With 16 bits, values up to 65534 are possible, allowing for 32767 registers to be read with a single command.

The advantage of this way of working is that it is fully compatible with the existing function code 3 usage for small amount of data (< = 254 bytes). Again, this is an **unofficial** extension to Modbus/TCP, consult the vendor documentation!

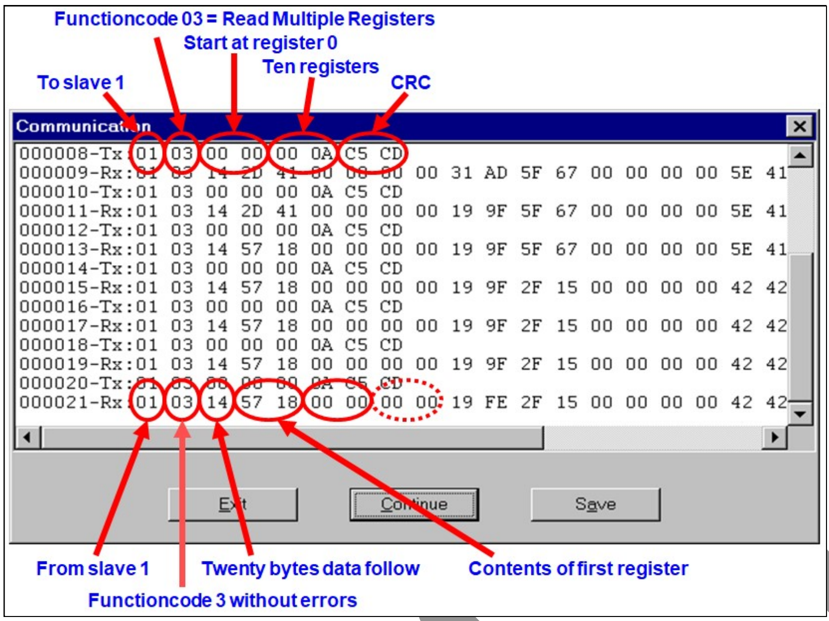
12. Network analysis

Modbus/ASCII and Modbus/RTU

Serial monitoring software can be bought from various vendors on internet; however such software is less commonly available than it used to be, because serial ports are less used than in the past. Most serial monitors just show the data sent back and forth, usually in hexadecimal, and you’ll have to do the decoding manually.

When monitoring RS232 on a device which is not itself a master or slave, be aware that you need **two** RS232 ports in order to be able to ‘see’ all traffic. This is because an RS232 link has separate RX and TX lines, and they need to be connected to 2 RX-lines on the monitoring PC. If there is only one RS232 port available, you can monitor only the traffic from the master *or* from the slave.

An example of a serial monitor catching Modbus/TCP traffic in both directions (with annotations by the author):



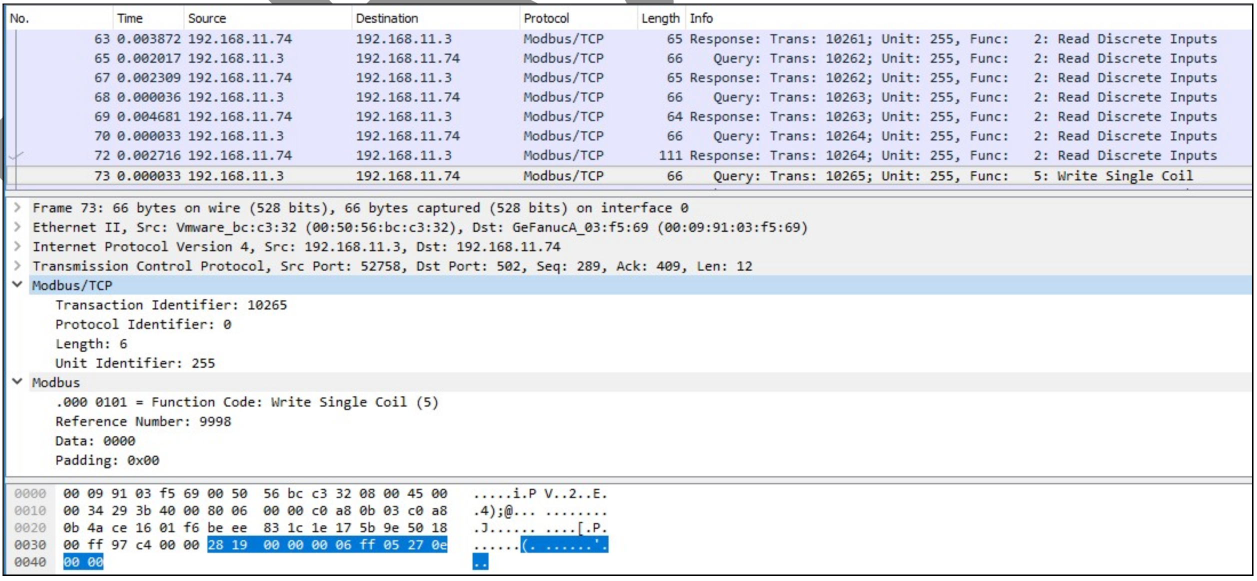
```
000008-Tx:01 03 00 00 00 0A C5 CD
000009-Rx:01 03 14 2D 41 00 00 00 00 31 AD 5F 67 00 00 00 00 5E 41
000010-Tx:01 03 00 00 00 0A C5 CD
000011-Rx:01 03 14 2D 41 00 00 00 00 19 9F 5F 67 00 00 00 00 5E 41
000012-Tx:01 03 00 00 00 0A C5 CD
000013-Rx:01 03 14 57 18 00 00 00 00 19 9F 5F 67 00 00 00 00 5E 41
000014-Tx:01 03 00 00 00 0A C5 CD
000015-Rx:01 03 14 57 18 00 00 00 00 19 9F 2F 15 00 00 00 00 42 42
000016-Tx:01 03 00 00 00 0A C5 CD
000017-Rx:01 03 14 57 18 00 00 00 00 19 9F 2F 15 00 00 00 00 42 42
000018-Tx:01 03 00 00 00 0A C5 CD
000019-Rx:01 03 14 57 18 00 00 00 00 19 9F 2F 15 00 00 00 00 42 42
000020-Tx:01 03 00 00 00 0A C5 CD
000021-Rx:01 03 14 57 18 00 00 00 00 19 FE 2F 15 00 00 00 00 42 42
```

Modbus/TCP

When using Ethernet for Modbus/TCP, other equipment is needed: a "network tap" and "network analyser" software. The tap is inserted between an Ethernet-switch and the device to be analysed. It copies all network traffic from/to the device to the network analyser (usually a laptop). The analyser software running on the laptop then translates all messages to a human-readable format.

The best-known analyser software is called "Wireshark", available from www.wireshark.org. It is open-source, free, well-maintained, and has support for hundreds of Ethernet-based network protocols, including Modbus/TCP. It is installed within a few minutes[42].

An example of Wireshark's dissection of a Modbus/TCP message for function code 05:

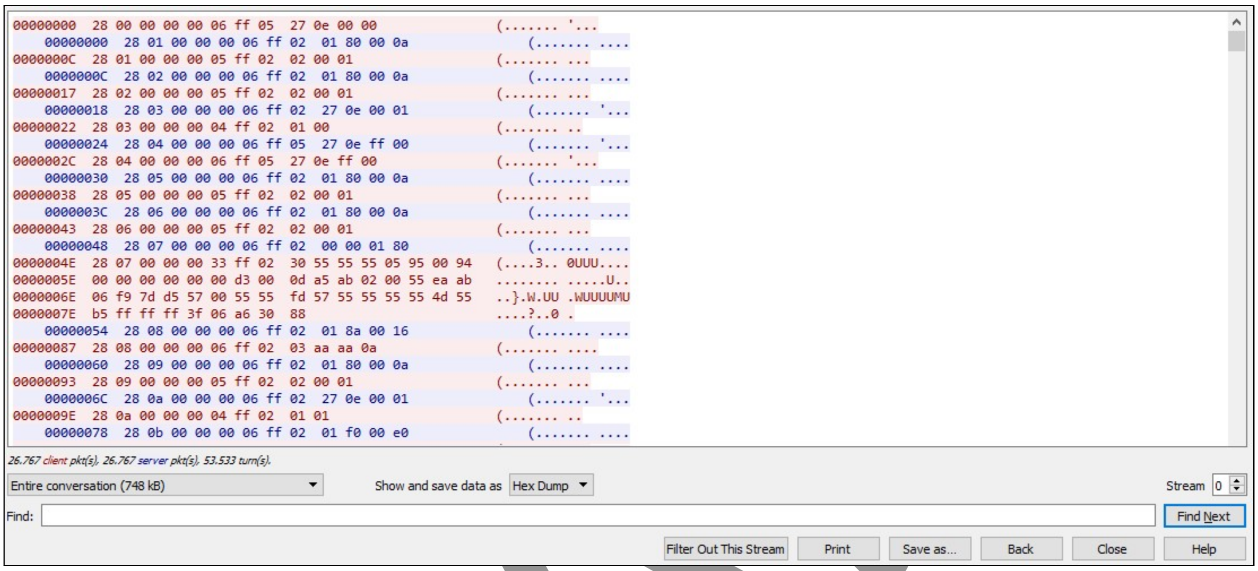


```
No.  Time      Source         Destination    Protocol   Length Info
  63 0.003872 192.168.11.74  192.168.11.3   Modbus/TCP     65 Response: Trans: 10261; Unit: 255, Func:   2: Read Discrete Inputs
  65 0.002017 192.168.11.3   192.168.11.74  Modbus/TCP     66    Query: Trans: 10262; Unit: 255, Func:   2: Read Discrete Inputs
  67 0.002309 192.168.11.74  192.168.11.3   Modbus/TCP     65 Response: Trans: 10262; Unit: 255, Func:   2: Read Discrete Inputs
  68 0.000036 192.168.11.3   192.168.11.74  Modbus/TCP     66    Query: Trans: 10263; Unit: 255, Func:   2: Read Discrete Inputs
  69 0.004681 192.168.11.74  192.168.11.3   Modbus/TCP     64 Response: Trans: 10263; Unit: 255, Func:   2: Read Discrete Inputs
  70 0.000033 192.168.11.3   192.168.11.74  Modbus/TCP     66    Query: Trans: 10264; Unit: 255, Func:   2: Read Discrete Inputs
  72 0.002716 192.168.11.74  192.168.11.3   Modbus/TCP    111 Response: Trans: 10264; Unit: 255, Func:   2: Read Discrete Inputs
  73 0.000033 192.168.11.3   192.168.11.74  Modbus/TCP     66    Query: Trans: 10265; Unit: 255, Func:   5: Write Single Coil

> Frame 73: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
> Ethernet II, Src: Vmware_bc:c3:32 (00:50:56:bc:c3:32), Dst: GeFanucA_03:f5:69 (00:09:91:03:f5:69)
> Internet Protocol Version 4, Src: 192.168.11.3, Dst: 192.168.11.74
> Transmission Control Protocol, Src Port: 52758, Dst Port: 502, Seq: 289, Ack: 409, Len: 12
v Modbus/TCP
    Transaction Identifier: 10265
    Protocol Identifier: 0
    Length: 6
    Unit Identifier: 255
v Modbus
    .000 0101 = Function Code: Write Single Coil (5)
    Reference Number: 9998
    Data: 0000
    Padding: 0x00

0000  00 09 91 03 f5 69 00 50  56 bc c3 32 08 00 45 00   .....i.P V..2..E.
0010  00 34 29 3b 40 00 80 06  00 00 c0 a8 0b 03 c0 a8   .4);@... ........
0020  0b 4a ce 16 01 f6 be ee  83 1c 1e 17 5b 9e 50 18   .J...... ....[.P.
0030  00 ff 97 c4 00 00 28 19  00 00 00 06 ff 05 27 0e   ......(. ......'.
0040  00 00                                              ..
```

[42] *Apart from the troubleshooting purposes for which it is made, it is also a good tool to learn how network-protocols work.*

At the top, we see a list of the messages that Wireshark captured, with at the right a summary of the contents. In the middle there is a dissection of a message, per protocol layer (Ethernet, IP, TCP, Modbus/TCP header and function code specific contents). At the bottom there is hexadecimal dump of the complete network message.

Wireshark can also show you how two devices communicate; this is done by "Following the TCP stream". Shown in red and blue are the messages that both devices send to each other.



```
00000000  28 00 00 00 00 06 ff 05  27 0e 00 00                 (....... '...
    00000000  28 01 00 00 00 06 ff 02  01 80 00 0a                 (....... ....
0000000C  28 01 00 00 00 05 ff 02  02 00 01                    (....... ...
    0000000C  28 02 00 00 00 06 ff 02  01 80 00 0a                 (....... ....
00000017  28 02 00 00 00 05 ff 02  02 00 01                    (....... ...
    00000018  28 03 00 00 00 06 ff 02  27 0e 00 01                 (....... '...
00000022  28 03 00 00 00 04 ff 02  01 00                       (....... ..
    00000024  28 04 00 00 00 06 ff 05  27 0e ff 00                 (....... '...
0000002C  28 04 00 00 00 06 ff 05  27 0e ff 00                 (....... '...
    00000030  28 05 00 00 00 06 ff 02  01 80 00 0a                 (....... ....
00000038  28 05 00 00 00 05 ff 02  02 00 01                    (....... ...
    0000003C  28 06 00 00 00 06 ff 02  01 80 00 0a                 (....... ....
00000043  28 06 00 00 00 05 ff 02  02 00 01                    (....... ...
    00000048  28 07 00 00 00 06 ff 02  00 00 01 80                 (....... ....
0000004E  28 07 00 00 00 33 ff 02  30 55 55 55 05 95 00 94   (....3.. 0UUU....
0000005E  00 00 00 00 00 00 d3 00  0d a5 ab 02 00 55 ea ab   ........ .....U..
0000006E  06 f9 7d d5 57 00 55 55  fd 57 55 55 55 55 4d 55   ..}.W.UU .WUUUUMU
0000007E  b5 ff ff ff 3f 06 a6 30  88                        ....?..0 .
    00000054  28 08 00 00 00 06 ff 02  01 8a 00 16                 (....... ....
00000087  28 08 00 00 00 06 ff 02  03 aa aa 0a                 (....... ....
    00000060  28 09 00 00 00 06 ff 02  01 80 00 0a                 (....... ....
00000093  28 09 00 00 00 05 ff 02  02 00 01                    (....... ...
    0000006C  28 0a 00 00 00 06 ff 02  27 0e 00 01                 (....... '...
0000009E  28 0a 00 00 00 04 ff 02  01 01                       (....... ..
    00000078  28 0b 00 00 00 06 ff 02  01 f0 00 e0                 (....... ....
```

26.767 client pkt(s), 26.767 server pkt(s), 53.533 turn(s).

Entire conversation (748 kB) Show and save data as Hex Dump Stream 0

Find:

Wireshark has much more functionality than we can discuss here, but ample information about it can be found on internet and in book stores, and there are many courses for it.

13. TCP programming mistakes

Reading data

Anyone who first starts programming applications using the TCP protocol seems to make the same mistakes. The problem is that the software usually runs fine on the development PC. But as soon as the software is put 'in production', it appears that TCP works a little bit different than expected. What one sees is that parts of the TCP message seem to be missing. But this is not really so; the 'missing data' is really there (but needs to be retrieved by the application software).

The 'problem' is caused by an optimization in TCP that it doesn't have to send large chunks of data as a whole to the destination. Depending on the memory space available in the destination, it may send only a part of the data immediately, and the rest at a later moment. The software on the destination must be aware of this: that it can get less data than expected. This is not a bug in TCP; the other data *will* arrive, not now but a little later. So the software on the destination must continue to collect data from TCP until it has all the data it needs.

For example, when a sender transmits 50 bytes, TCP may:

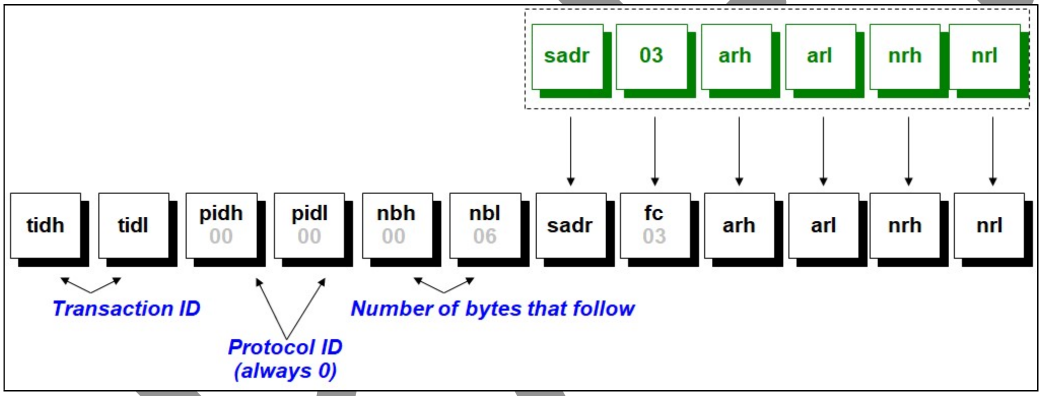
- Deliver all 50 bytes at the same time. This is what usually happens.

- Deliver 40 bytes immediately, and 10 bytes later.
- Deliver 20 bytes immediately, then another 20, and then the last 10.
- Or ***any other*** combination of chunks of data totalling 50 bytes.

For example, when using the "socket" API, a programmer would have to code in C something like[43]:

```
int nrbytes = 50;
char buf[nrbytes]; // To store the data in
int startbyte = 0; // To remember what we already have
while nrbytes > 0 do
        int nrreceived = receive(socket, &buf[startbyte], nrbytes)
        nrbytes = nrbytes – nrreceived;
        startbyte = startbyte + nrreceived;
end
```

The next step would be to know: **how many** bytes[44] am I to handle? This information is provided in the Modbus/TCP header, fields "nbh" and "nbl" (from the earlier show example of a message for function code 03):



The implementation should thus:

- Read the first six bytes;
- Take the values in byte 5 and 6 to calculate how much more data is coming;
- Read the calculated amount of data (from the previous step).

Only now we have all the data that belongs together, which can now be processed further.

Writing data

TCP as an optimization feature, originating at a time when network bandwidths were still very low. TCP tries to coalesce as much data as possible in one TCP packet, because one (large) packet with much data has much less overhead than many small packets each with a little bit of data.

Nice as this feature sounds, it may have one big drawback: your data may be held up in the sending device for a fraction of a second. For the purpose for which this feature was developed (terminal

[43] *For simplicity's sake no error handling is show.*
[44] *It could be argued that the "nbh/nbl" fields are not strictly necessary, as one could read the byte containing the function code, and then the (known) fields following it. But then the software would need to know every function code's message format.*

I/O), this "fraction of a second" was no problem. But for industrial communication, this can be a big[45] problem, severely limiting communication speed.

The TCP-feature is called "Nagle's algorithm" which we will not further discuss here, as there is a lot of information available on it on internet (i.e. en.wikipedia.org/wiki/TCP_delayed_acknowledgment). It is enabled by default on any TCP protocol stack.

The relation to Modbus/TCP is that the Nagle delay can be easily triggered when the software executes a "write, write, read" sequence. The double write is often (unknowingly) triggered by a Modbus/TCP programmer who makes his code first 'write' (= offer for transmit) the 6-byte Modbus/TCP header, immediately followed by the 'write' of the remaining Modbus/TCP data, and then the 'read' for accepting the answer.

Luckily, the Nagle algorithm can be turned off. If you write your own software this requires one line of code; if you just buy equipment it is to be hoped that the vendor does it in his software. Note that this must be done at **both** devices. Alternatively, if you write your own software, do not send Modbus/TCP messages in two chunks – the code is a *little* bit more complicated, but not very much.

14. TCP/IP port numbers

Modbus/TCP servers should use TCP port 502 for **incoming** connections, as mentioned in the specification. Some suppliers also support other ports, i.e. 503 and/or 504. This is sometimes done to allow multiple incoming connections from more than one client in parallel. However, the port number to be used must also be configured in the client, which is not always possible (oftentimes, only 502 is available and this cannot be changed).

Although it is useful when a server is able to support multiple clients, there is no good reason for using other ports than 502. This is probably due to a misunderstanding of how TCP/IP works. Port 502 is **only** necessary to set up a connection between a client and a server, and TCP then reassigns another port (somewhere in the range 1024..65535) to the connection for all other network traffic between the client and the server. Port 502 is then 'free' again to accept new incoming connections, either from the same client or any other client. But the software on the server must be specifically written to support this, as is normally done so when writing TCP/IP servers. On simple embedded devices, using slower CPU's and with little support for

There is one good reason for a server to use other ports than 502, and that is for use on Ethernet/serial converters with multiple RS232/485 ports (say COM1, COM2, etc.). There must be a way for the client to specify which serial port is to be accessed. One way to do this is by hard-coding a logical connection: port 502 goes to COM1, port 503 to COM2, port 504 to COM3, etc.

[45] *When the author first encountered this, his software could only handle 5 messages per second on a 1-Gbyte network with a 2,5 GHz PC.*

15. Useful function code 08 diagnostics

When a slave supports function code 08 **and** subfunction 11..18 (0x0B..0x12), it may be able to inform the master about the current values of its statistics counters:

Bus Message Count
Bus Communication Error Count
Slave Exception Error Count
Slave Message Count
Slave No Response Count
Slave NAK Count
Slave Busy Count
Bus Character Overrun Count

Since this functionality is seldom implemented in slaves, we will not discuss it further. See [MBUSSERIAL] Appendix A for a detailed description and flowcharts on when each counter is incremented.

Personally, I find the lack of diagnostics on many modern networks[46] a problem. Without further equipment, there is no way to know how well the network is performing. For example, retries may help to solve transmission problems, but in principle there should be **no** retries, especially on new networks. If there are, this is an indication of bad quality cabling, wrong cabling, etc. and the system should not pass the acceptance tests.

By regularly monitoring diagnostics counters it is sometimes possible to predict future problems, for example when the number of errors slowly rises. The system may still perform well, but at a certain moment the retries can no longer solve the transmission problems, and the application software will get an error. And long before that, you'll notice that network communication goes slower and slower.

[46] *Not only on Modbus this is a problem, but on most industrial networks. The use of managed switches on (industrial) Ethernet will help us a lot! (but then someone must still read the diagnostics data).*

CHAPTER 12. Cybersecurity

Since the rise of “industrial Ethernet”, Modbus/TCP has gained a reputation to be one of the most vulnerable protocols for hackers. Technically this is nothing new; the (serial) Modbus/ASCII and /RTU are just as vulnerable as Modbus/TCP, but the difference lies in the connection of Modbus/TCP devices to Internet. Whereas in the past a hacker had to travel to be in the physical vicinity of a Modbus/ASCII or Modbus/RTU in order to hack it, a Modbus/TCP can basically be accessed from anywhere on the internet, if no proper precautions are taken.

1. Weaknesses

The weaknesses of a Modbus/TCP device (usually a server) can be any of the following:

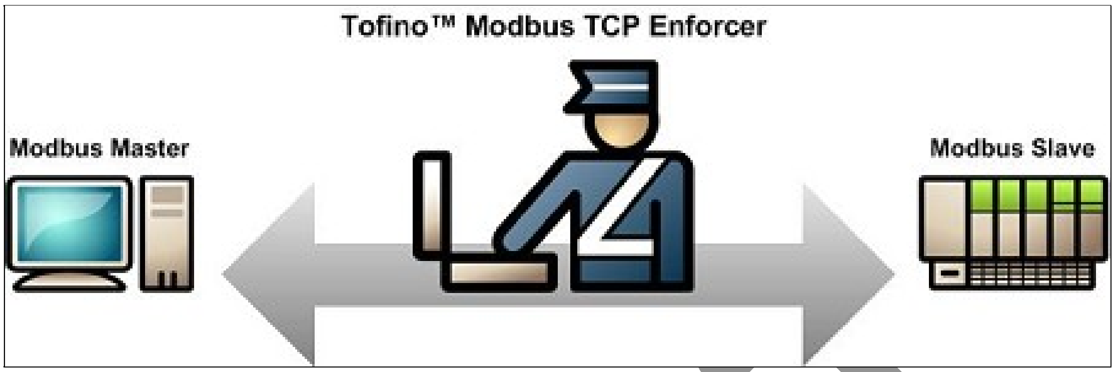
- The lack of an authentication mechanism in Modbus, accepting incoming commands from any client, either normal or fake.
- Implementations not resistant to TCP/IP protocol misuse (for example, a “ping of death”)
- Implementations not resistant to purposely wrong formatted messages;
- Implementations not resistant to not-supported function codes;
- Implementations not capable of handling a high level of network traffic;
- Allowing any bit or register to be modified by any client;
- Allowing registers to be overwritten with a value outside the normal range;
- Allowing registers with interesting data to be read by a fake client;
- Etc.

Because a Modbus/TCP client is easily written, it is not difficult for a hacker to attempt an attack on a device. According to the search-engine Shodan (which unlike Google does not search websites but devices), Modbus was the most-searched protocol in 2015.

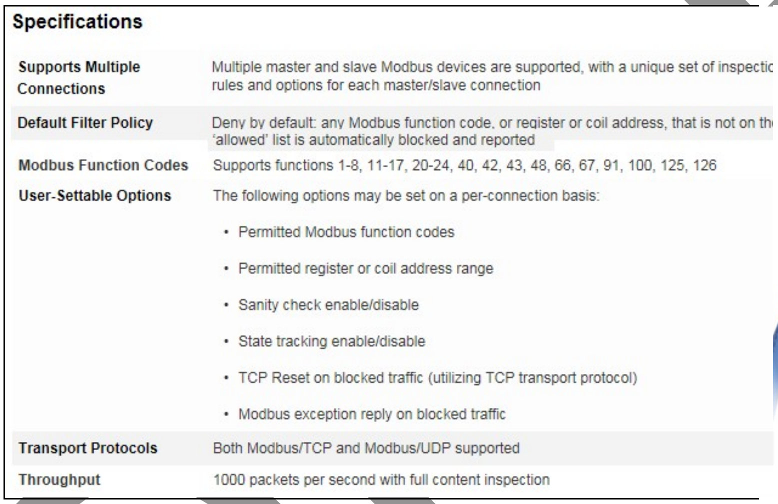
Protection against hackers is not possible in Modbus without substantial changes to the protocol. The only way to protect a Modbus/TCP network today is to use a perimeter defence, for example via a special firewall, which intercepts and checks all incoming external traffic. But Modbus/TCP is not a protocol that most firewall vendors recognize, because it is not commonly used in IT environments. Usually the only “protection” mechanism that can be set is to forbid Modbus/TCP completely, by blocking TCP port 502.

2. Modbus firewalls

To fill this gap various vendors have developed industrial firewalls, especially built for watching the use of Modbus/TCP, checking the contents of all Modbus/TCP network messages and disallowing all that is dangerous or unwanted[47]. One of the first such firewalls that came in the market was Belden's "Tofino", which is brand-labelled by many other vendors active in process automation.



The Tofino is installed in the signal path between the Modbus/TCP client, and the Modbus/TCP server. All network messages between them are inspected by the Tofino, upon user-settable filtering criteria. Messages that do not pass these criteria are discarded by the Tofino, and thus never arrive at the server.



Specifications

Supports Multiple Connections	Multiple master and slave Modbus devices are supported, with a unique set of inspectic rules and options for each master/slave connection
Default Filter Policy	Deny by default: any Modbus function code, or register or coil address, that is not on th 'allowed' list is automatically blocked and reported
Modbus Function Codes	Supports functions 1-8, 11-17, 20-24, 40, 42, 43, 48, 66, 67, 91, 100, 125, 126
User-Settable Options	The following options may be set on a per-connection basis: • Permitted Modbus function codes • Permitted register or coil address range • Sanity check enable/disable • State tracking enable/disable • TCP Reset on blocked traffic (utilizing TCP transport protocol) • Modbus exception reply on blocked traffic
Transport Protocols	Both Modbus/TCP and Modbus/UDP supported
Throughput	1000 packets per second with full content inspection

Standard firewalls usually do not recognize Modbus/TCP, because it is not a real IT-protocol. Usually you can only block it completely by disallowing access over TCP port 502.

Industrial firewalls like Palo Alto's PA220R recognize Modbus, and many other common industrial protocols.

3. Intrusion detection systems

Whereas a firewall blocks unwanted devices and protocols in a network, but only for the network traffic that passes through the firewall, an "intrusion detection system" (IDS) never blocks any traffic, but can monitor all traffic on a network when connected to a switch "span" or "mirror" port.

[47] *One wonders why the vendors of Modbus slaves/servers do not implement sanity checks in their own software.*

IDS’s exist in all sorts (even as open-source), but recognition of industrial protocols is often lacking. For this one needs an IDS specifically built for industrial systems, such as “SilentDefense” from the (Dutch) company SecurityMatters[48].

4. New developments

In 2018, Schneider released the specification for “Modbus/TCP Security” on the www.modbus.org website. It adds very useful features to Modbus/TCP, such as authentication of client and server, role-based validation, and encryption of data. At the moment this publication is written (September 2018) no implementations of Modbus/TCP Security were publicly available on the market, so there is little experience available.

One issue (that also pops up with other network protocols) is the usefulness of encryption. It is true that hackers now have no visibility into the application. But the same is true for the user, and for firewalls / IDS’s. When a hacker is able to ‘break’ into the client or server, these systems will see nothing unusual. So whether we gain, or loose, by encryption is to be determined.

[48] *I am employed by them.*

Appendix A: RS232

1. Origin

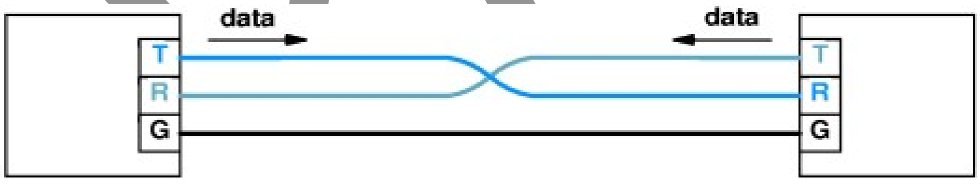
RS232 ("Recommended Standard 232") was launched in 1969. It specifies an "interface" designed for connecting terminals to modems, with which one could remotely login on a company or university mainframe. In the following decades, the specification was updated time and again, giving us versions RS232A, B, C, D, E and F.

For almost two decades RS232C stayed the latest version, until RS232D was released in 1986. The difference is the timing of a few signals, making it identical to CCITT V.24 which is the telecommunication industry's version of RS232. RS232D is backwards compatible with RS232C. In 1991 RS232E was released, the difference being a chance in the way handshake signals work. It also undid a few of the changes of RS232D. In 1997, the RS232 specification was taken over by the ANSI/EIA/TIA organisation, so formally "RS" no longer exists; its new official name is ANSI/EIA/TIA-232. In the F version[49], again a harmonisation with CCITT V.24 was undertaken.

So, RS232C is the most commonly mentioned version in datasheets. If you encounter a product with RS232D, E or F, it will probably work fine in combination with any other RS232 product.

2. The physical link

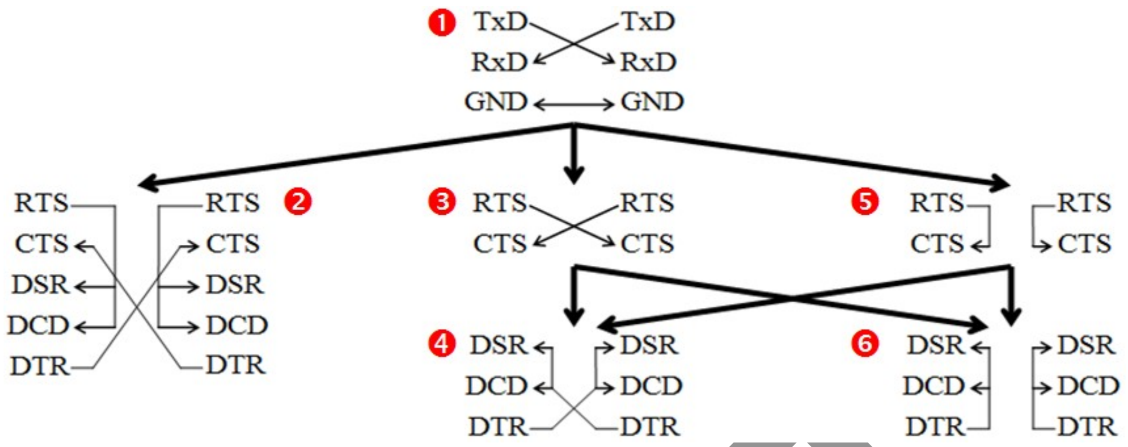
The simplest RS232 links consists of only three wires:



But the telephone-line legacy of RS232 is still often seen. The modem-control signals may come into play, but how they must be wired is always a surprise, due to the large amount of variation (and often a lack of documentation). Even when there are almost no modems used nowadays, a lot of software still controls the modem-signals of RS232 and expects these signals to work. When connecting two new devices for the first time, it is often a time-consuming job[50] to get this right, as there are no standards here. A few examples:

[49] *The differences between RS232C and the D, E, and F versions are so small or irrelevant for ordinary use that most companies seem to have missed them. The author has encountered only* ***once*** *in 20 years that a company mentioned to support RS232F in a product.*

[50] *The experience of the author is that the wiring setup of devices with which you have never worked before, and of which the documentation is not well-written, may easily take half a day (spent in trial-and-error setups).*



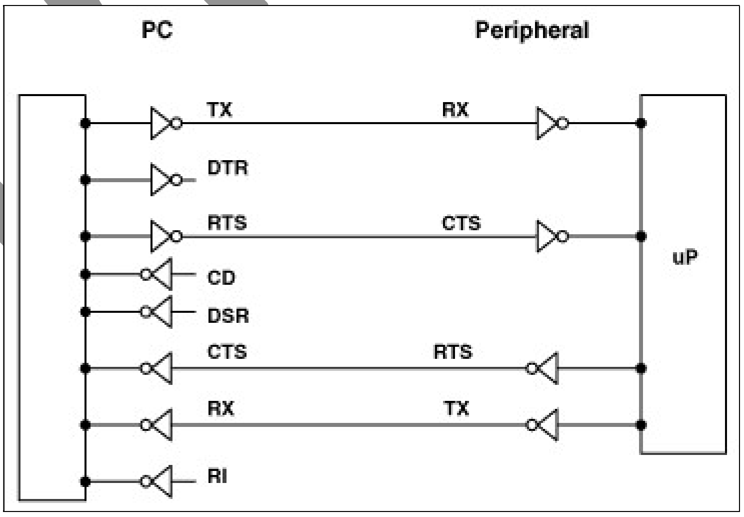
Always begin at (1), this is of course always necessary. Next, several variations exist on how the modem control signals are to be wired. In practice, we see the following combinations:

(1)
(1) + (2)
(1) + (3)
(1) + (3) + (4)
(1) + (3) + (6)
(1) + (5)
(1) + (5) + (4)
(1) + (5) + (6)

This list is certainly not complete! There are more ways to connect two RS232 devices together, the picture above assumes that two devices are to be wired symmetrically, but this does not always have to be so, especially when using devices from different vendors. For example, we can have:

(1) + (5 on one device) + (4 on the other device)

As an example:



Example of wiring RS232 according to (1)+(3) (Source: Maxim)

Phoenix Contact, in its manual for the Trusted Wireless radio module, describes several wiring variants:

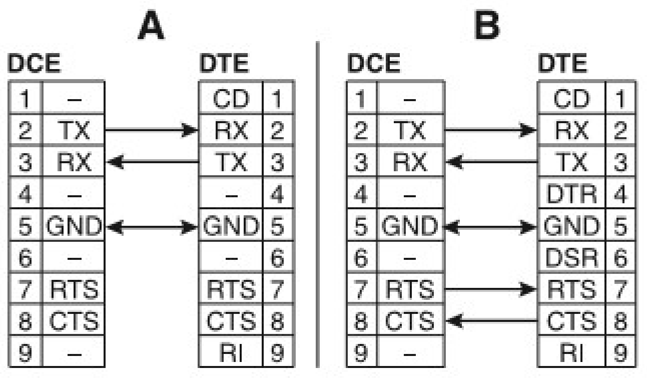


Figure 4-10 9-pos. D-SUB straight-through cable pinouts for 3-wire (A) and 5-wire (B)

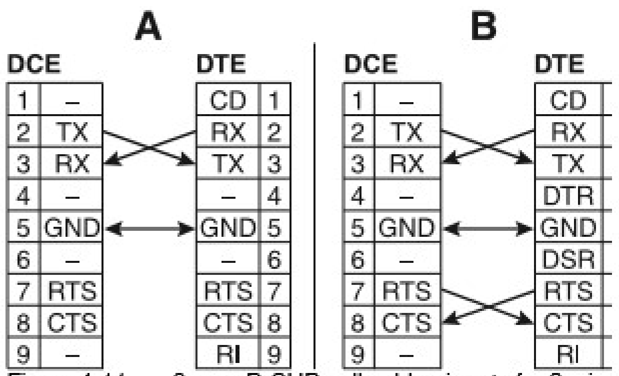


Figure 4-11 9-pos. D-SUB null cable pinouts for 3-wire (A) and 5-wire (B)

3. The cable

RS232 does not specify a type of cable to be used, but only mentions the electrical characteristics that should be adhered to. This gives considerable freedom in choosing the cable, also due to its relatively low speed RS232 is not very demanding.

The standard mentions that the total capacitance (in Farad) of the cable and the receiver electronics, should be less than 2500 pF (picoFarad). Subtract from this number the value for the receiver electronics (say, 100 pF), and then divide 2400 by the cable's capacitance per meter (say, 100 pF/meter), giving a maximum length of 24 meters. Many vendors set the limit at 15 meter if the capacitance of the cable is unknown.

In reverse, when using low-capacitance cable[51] (say, 30 pF/meter) the maximum length can be 80 meters (this fact is little known!).

[51] *Such cable exists, but may have the drawback that it must **never** get wet.*

A problem with using RS232 over longer distance is that both devices should have a good common ground. When this is not available, there will be a lot of communication problems, or even equipment might burn out[52]. The website of www.robustdc.com gives a lot of good advice on how to properly use RS232 in an industrial environment.

4. The connector

Originally, RS232 defined a 25-pin D-connector. The functions of each pin are clearly defined, but the name of a pin varies depending on the norm. Next to this we have an unofficial set of names which have become a de-facto standard in everyday use; isn't it much easier to read "TX" instead of "BA"?

Pin nr (25 pins)	CCITT V24	EIA RS232	DIN 66-020	Name	Function
1	101	AA	E1	GND	Protective ground (chassis)
2	103	BA	D1	TX	Transmitted data
3	104	BB	D2	RX	Received data
4	105	CA	S2	RTS	Request to Send
5	106	CB	M2	CTS (RFS)	Clear To Send (Ready For Sending)
6	107	CC	M1	DSR	Data Set Ready
7	102	AB	E2	SGND	Signal ground
8	109	CF	M5	CD	Carrier Detect
9					
10					
11	126	CK	S5	STF	Select Transmit Frequency
12	122	SCF	HM5	BCCD	Backward channel CD
13	121	SCB	HM2	BCR	Backward channel ready
14	118	SBA	HD1	BTX	Backward channel TX
15	114	DB	T2	TSET	Transmitter signal element timing (DTE)
16	119	SBB	HD2	BRX	Backward channel RX
17	115	DD	T4	RSET	Receiver signal element timing (DCE)
18	141		PS3	LL	Local Loopback
19	120	SCA	HS2	TBCLS	Transmit backward channel line signal
20	108.1 108.2	 CD	S1.1 S1.2	CDSL DTR	Connect data set to line Data Terminal Ready
21	110 140	CG	M6 PS2	DSQD RL	Data signal Quality Detect Remote Loopback
22	125	CE	M3	RI	Calling Indicator
23	111	CH	S4	DSRS	Data Signal Rate Selector

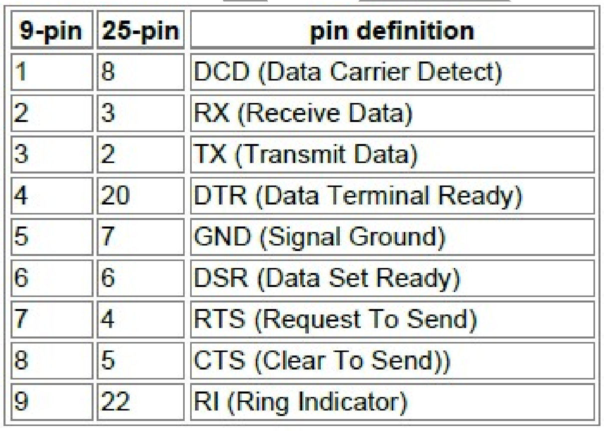
[52] *I once had a burnt-out PC which was connected to a PLC; both were fed via their own power-outlet only 5 meters separated from each other. There was an 110V voltage difference between both earth pins. A shocking experience if I would have touched both devices simultaneously. Now only the PC motherboard fried out.*

	112	CI	M4	(idem)	(idem)
24	113	DA	T1	TSET	Transmitter Signal Element Timing
25	142		PM1	TI	Test Indicator

Two pins (9 and 10) have never been given a function. Additionally, one CCITT V.24 signal number "133" (Ready For Receiving) has not been given a pin. If present, usually pin 9, 18 or 25 is used instead.

The large number of signals is almost never used. For most applications the signals RX, TX, DCD, DTR, DSR, RTS, CTS and RI suffice. For this a 9-pin D-sub connector suffices, and due to the usage of this connector on PC's it has almost replaced the 25-pin connector (which is nowadays seldom seen, usually on older products).

Which signal is used on which pin is different, see the following table:



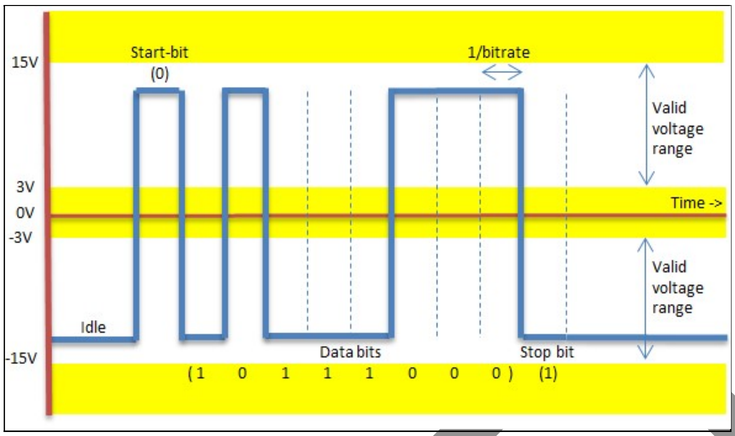
9-pin	25-pin	pin definition
1	8	DCD (Data Carrier Detect)
2	3	RX (Receive Data)
3	2	TX (Transmit Data)
4	20	DTR (Data Terminal Ready)
5	7	GND (Signal Ground)
6	6	DSR (Data Set Ready)
7	4	RTS (Request To Send)
8	5	CTS (Clear To Send))
9	22	RI (Ring Indicator)

*Note that vendors may decide to **not** support a certain pin on their equipment. This means: the pin is present in the connector, but electrically it is not connected to the device's electronics.*

5. Voltages

The electrical voltages of the signals are specified by RS232 to be in the range -15V..-3V or +3V..+15V. Voltages in the range -3V..+3V are invalid. Modern equipment often uses -12V / +12V or even -9V / +9V (laptops). This is still OK according to the RS232 specification, but note that this influences your signal margin on long cables. It could very well happen that a PLC with a Modbus master is able to communicate with a slave a long distance away, but then that PLC is replaced by a

Modbus master running on a laptop (i.e. for troubleshooting purposes, or remote configuration), no communication is possible[53].



Peculiar is that when a RS232 link is idle (i.e. not transmitting data), the voltage at rest is negative (-15V). This corresponds to a binary '1'. This also holds for a stop bit. A binary '0' and a start-bit correspond to a positive voltage (+15V).

6. Bitrate / baudrate

RS232 does not define with which bitrate / baud rate data is to be transmitted. So any possible value between 0 and 115200 bit/s is possible. In practice, only a very small subset of all possible bitrates is used, due to the capabilities of the electronic circuits used for RS232.

So we only see:

50, 110, 150, 300, 600, 1200, 2400, 4800 and 9600 bits/s
19.2, 28.8, 38.4, 57.6 and 115.2 Kbit/s.

Most Modbus devices do not support all these bitrates, especially the simpler and cheaper devices. One often sees the following supported bitrates:

9.6, 19.2, 38.4 and 57.6 Kbit/s.
On PC's: 115.2 Kbit/s.

In order for devices on network to communicate, ***all*** devices on the same cable must be configured to the same bitrate. A device using a different bitrate will not be able to communicate with anyone else. Exactly how the bitrate is configured on a device depends on the vendor (consult the device's documentation). On a network with 'n' devices from various vendors you may well end up with 'n' different configuration methods.

[53] *The author once experienced this with software developed on a desktop PC, which ran fine with a Modbus slave. But the customer installed my software on his laptop, and it didn't work with the same slave.*

7. Data transmission

RS232 does not define how data (0's and 1's) is to be transmitted. It goes no further than defining the voltage levels for a '0' and '1', but does not define how large amounts of data (= more than 1 bit) are to be transmitted. Actually it cannot do this, as an OSI-layer 1 specification does not have this knowledge; this is defined in OSI-layer 2 (datalink).

In practice, data transmission over RS232 is always done according to the "UART" format, where data is transmitted character-per-character, each preceded by a start bit and followed by a stop bit. The UART format is described in the next section "Serial Transmission Format".

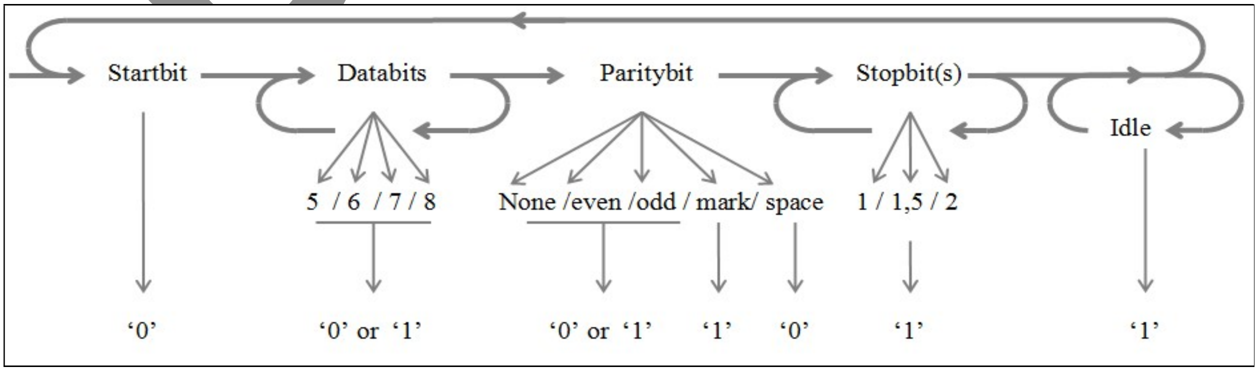
Multiple consecutive characters, forming a "network telegram", are defined by the network protocol. This is of course different per network protocol; the Modbus way of working is described in chapters 5 and 6.

8. Serial Transmission Format

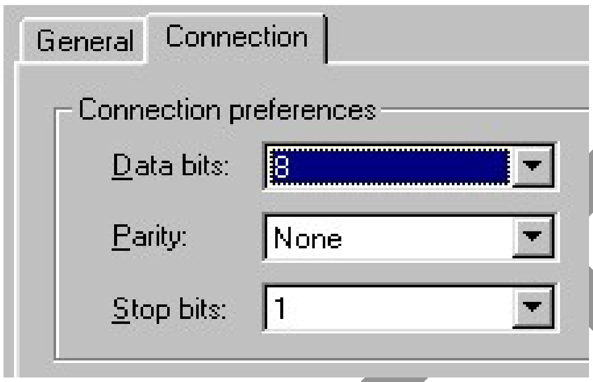
Each character is sent according to a standard format:

- When no transmission is active, the line has the value '1'.
- One 'start bit', which always has the value '0'.
- 5, 6, 7 or 8 data bits. Modbus only uses 7 data bits (for Modbus/ASCII) or 8 data bits (for Modbus/RTU).
- An optional bit, which can be used for odd or even parity, or have a fixed value 0 or a fixed value 1, or be completely left out. The "even" parity is to be mandatory supported in all Modbus devices. It is recommended that support for "no parity" is also implemented, for maximum compatibility.
- 1, 1.5 or 2 stop bits, which always have the value '1'. The number of stop bits is 1 when an (odd or even) parity is used or 2 when no parity is used. Note that it is the experience of the author that many devices still allow the combination of (no parity, one stop bit), as this saves up to 10% in network bandwidth! (at the expense of less detection of transmission errors).
- When there are more characters to send, the sequence can repeat itself (with the next start bit, followed by the data, etc.). When all data has been sent, the line remains in idle value '1'.

The following picture shows the transmission of one character graphically:



The configuration settings for the number of data bits, the parity bit and the stop bits must be identical for **all** devices on the network. How exactly this is configured on a device depends on the vendor (consult the device documentation for this).



Data bits: 8
Parity: None
Stop bits: 1

9. Parity

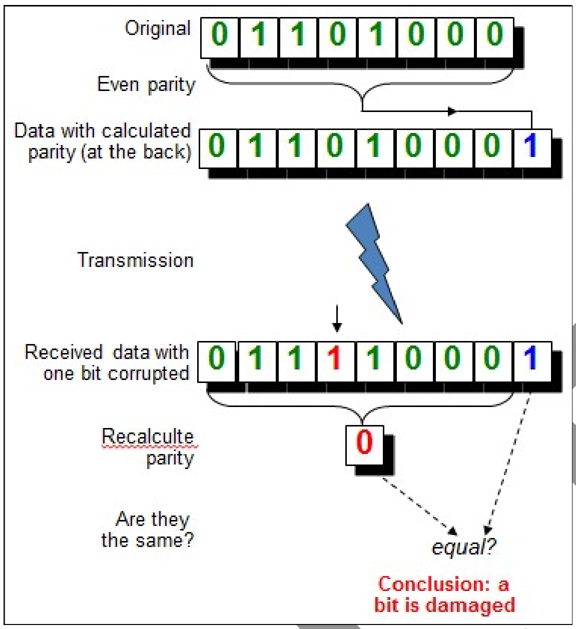
"Parity" is a very simple algorithm to detect damaged bits in a character:

- A '0' is sent, but is received as a '1'
- A '1' is sent, but is received as a '0'

No application wants to work with damaged data, so any network protocol, including Modbus, employs defensive measures: error detection and error repair. Parity is a simple method to detect damaged data. Error **repair** is not possible with parity; the application software must take care of this.

Whenever a character (7 or 8 bits) is sent, one additional bit is added: the "parity bit". The value of the parity bit, 0 or 1, is to be calculated according to one of the following algorithms:

- With "even parity", the total number of '1' bits (7+1 or 8+1) must be even.
- With "odd parity", the total number of '1' bits (7+1 or 8+1) must be odd.



After the calculation of the parity bit, all 7+1 or 8+1 bits are transmitted over the network. A device that receives these bits does the same (even or odd) calculation on the 7 or 8 data bits.

If no data was damaged while in transit, the value of the parity bit as calculated by the receiving device must be the same as the parity bit calculated by the sending device. If there are *not* the same, some error has occurred in the 7 or 8 data bits. Unfortunately, it is unknown which bit(s) was (were) damaged, so the damaged bits cannot be repaired. That is why the parity algorithm is only “error detection”.

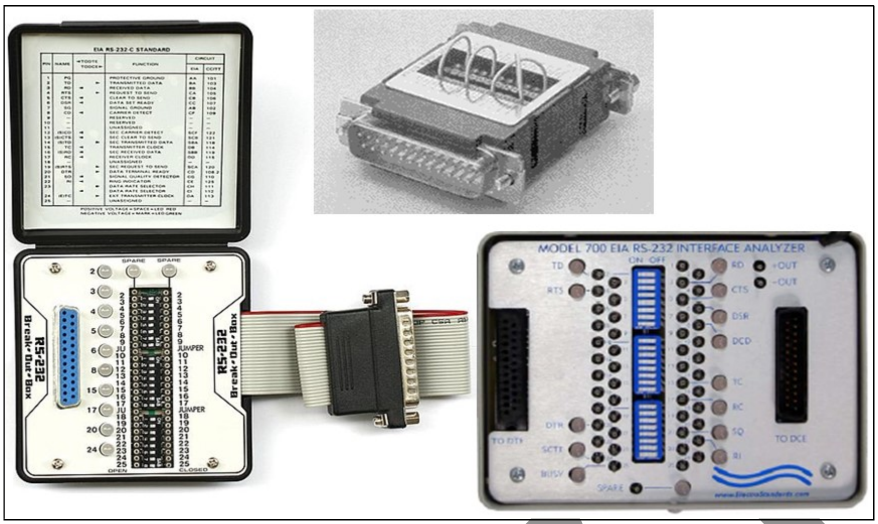
The receiver of the data must now discard the damaged character, and as a consequence the complete network message. In Modbus, any “repair action” must now be done by the network master, after a timeout has expired.

10. Programming

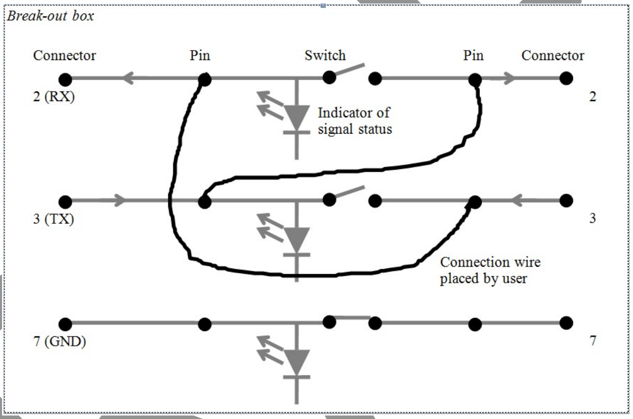
Not much can be added to what it already written down in: en.wikibooks.org/wiki/Serial_Programming.

11. Troubleshooting RS232

Due to the large number of signals, it is often difficult to find out how all the modem control signals operate on two devices. A “breakout box” helps to see what is going on, and allows signals to be connected to each other in a few seconds (or to be disconnected).



Electrically, a breakout box is very simple. It helps to determine the 'right' way to connect two devices together, without needing any soldering.



A breakout box does not help to find problems with the bitrate, parity settings or stop bits. This can only be done with a "serial monitor" or a "sniffer".

Appendix B: RS485

1. Origin

RS485 became a standard in 1983, developed by the Electronics Industry Association (EIA). The letters "RS" stand for "Recommended Standard" or "Radio Standard". Officially RS485 doesn't exist anymore today, since the EIA has transferred it to the "Telecommunications Industry Association". The official name is now "ANSI TIA/EIA-485".Nevertheless, many vendors (and users) still name it RS485.

RS422 and RS485 are often confused with each other, where RS485 is often seen as the 'full duplex' (bidirectional) version of RS422. There are also many vendors who name any 2-wire network 'RS422' and a 4-wire network 'RS485'. Both are **not** correct. There are electrical differences between RS422 and RS485; RS485 can be used in RS422 networks, but not the other way around.

2. The physical link

RS485 specifies a physical link using two signals, called[54]:

A[55], TxD/RxD+, TXD0/RXD0, TD/RD+, +, D0 or D+

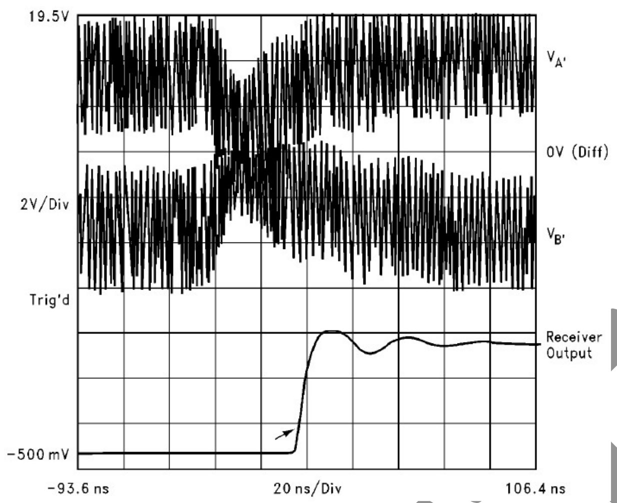
B, TxD/RxD-, TXD1/RXD1, TD/RD-, -, D1 or D-

Optionally there may be a C, G, GND, REF or SC, the common voltage reference connection, usually called "Ground".

With these two signals, a so-called "differential" or "balanced" transmission line is created which his very resistant to electronic noise. This makes RS485 ideal for use in industrial environments. The voltages on the A and B signals are inverted with respect to each other. The voltage difference between A and B determines whether a '0' bit or '1 ' bit is transmitted. When electric noise affects the A and B signals, this will change the voltage, but the voltage difference remains the same, in effect cancelling out the noise.

[54] *One often wonders what the purpose of a standard is, if nobody follows it.*

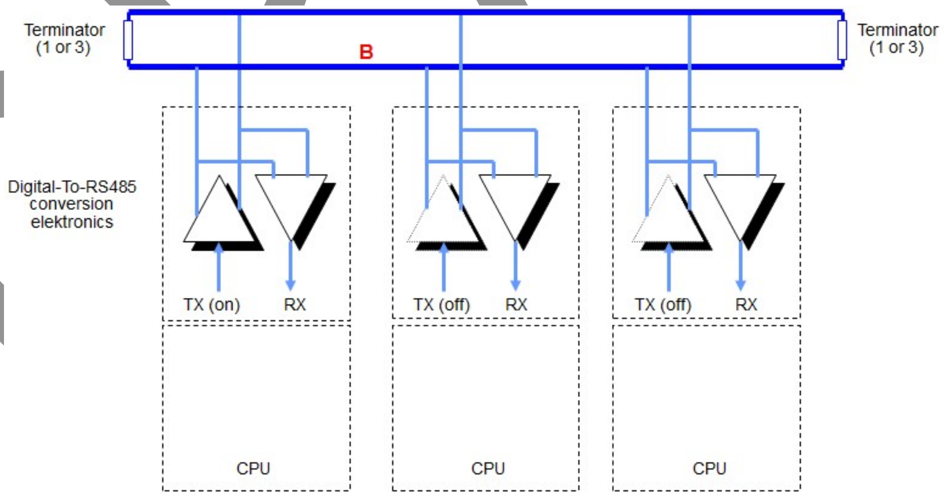
[55] *According to Lynn Linse of Digi, on a thread in the control.com website (control.com/thread/1026208408), there are two different ways of labelling A and B. The first vendor who made RS485 chips, Texas Instrument, did it wrong, but before this was known many other vendors copied what TI did. Later, vendors did it better, but now there are two systems in use and you never know whether one vendor's A must be connected to another vendor's A, or perhaps to the B. Luckily, connecting the wrong wires together brings to damage.*



The heavily disturbed A and B signals (top) do not pass an
RS485 receiver, which delivers a clean signal (bottom)
changing from a ‘0’ to a ‘1’ (source: National Semiconductor AN-1047)

An RS485 network consist of two long wires, carrying the A and B signal, often called the “trunk”. Additionally, there is a third wire, being the common ground. Devices can be attached to the trunk, either directly or via a short cable (called “stub”).

All devices share the same cable, which is why it is sometimes called a “party line network”. The official designation is “bus”. On the bus, only **one** device may have an active transmission at any time; all others may receive only. RS485 does not describe how this is to be done; it is a task of the higher OSI-protocol layers to implement this. In Modbus, this is taken care of via the master/slave way of working.



Electrical view of an RS485 network, with the trunk at the top and three devices connected via stubs. The electronics in each device split the outgoing and incoming data streams. Only one transmitter may be active at any time.

Termination resistors
At both ends of the trunk, so-called “termination resistors” must be connected. There are two systems for this, the first uses **one** resistor between both wires, the second also has **three**: one resistor between both wires, but also a pull-up and a pull-down resistor. Which of the two is needed depends on the vendor; nowadays most vendors use one resistor, but you might encounter older equipment requiring three.

Maximum number of devices
The maximum number of devices on a RS485 link is often given as: 32. This is the default maximum, based on the electrical output power of each transmitting device (32 “unit loads”) and the power consumption of each receiver (1 “unit load”). But there are also RS485 electronic circuits of which the receiver consumes only ½ unit load (UL), or ¼ UL. In such cases up to 64 or 128 devices can be connected to the same physical link. Unfortunately vendors hardly ever mention the UL factor of their equipment’s receiving electronics, which is why it is safest to always assume 1 UL.

3. The cable
RS485 does not specify which type of cable must be used. This is because it is related to the electronics in the devices, which should match with the characteristics of the cable. Normally the vendors specify this.

But whatever you do, **always** use twisted-pair cable. The twists are there to cancel out a lot of electric noise and other disturbances. When putting a connector on the cable, let the twists continue as far as possible.

Another measure to keep disturbances out of the way is to use shielded cable, but then you should consider where both ends of the shield are to be connected to. It is the experience of the author (and personal frustration…) that I get lots of different advices; some say both ends must be grounded, others warn me to *not* do that; some vendors say that no current may run over the shield, others say a small current is fine.

Both the best way to keep disturbances out of your network is to keep the sources of the disturbances away from the network cable: high-voltage power lines, electric motors, robots, welding equipment, frequency converters, etc.

All measures taken together give a 6-line of defence against data corruption:

- Keep disturbance sources away
- Use shielded cable
- Use twisted-pair cable
- Usage of parity, checksum and CRC’s
- Executing retries
- Application checks.

The first three can be done by you, Modbus does the 4th and (usually also the) 5th, and finally the 6th line of defence is again up to you (in the application software).

4. The connector

RS485 does not define a standard connector to be used. Often the 9-pin sub-D is used, either the male or female version, but others can be seen as well. Additionally, which pin is used for what signal is also undefined.

Note that even when two devices have the same type of connector, it is not guaranteed that the signals are on the same pin!

5. Voltages

RS485 does not specify the voltages to be used. In practice, they are often around 5V.

This usually suffices, except in very noisy environments. A solution could be to use higher voltages (i.e. with a pair of converter devices), or to use fibre optic cables.

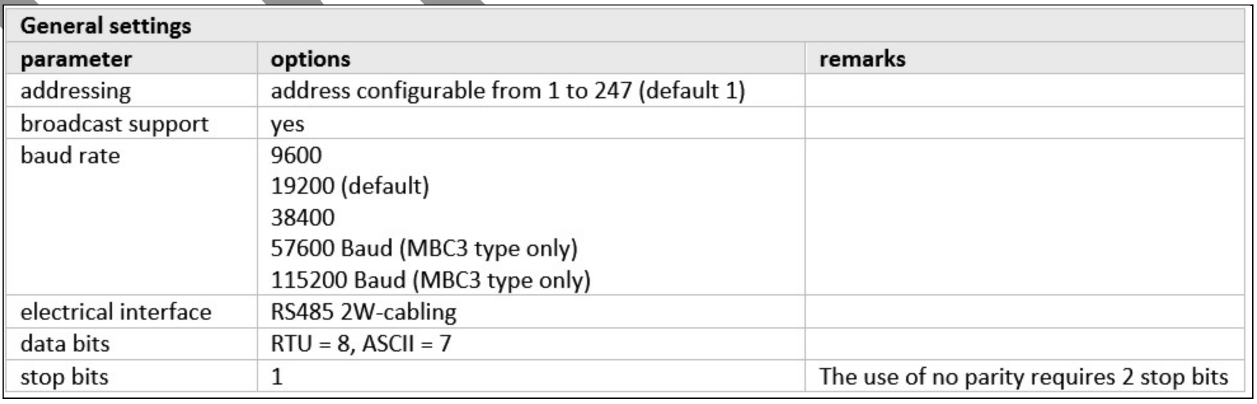
6. Bitrate / baudrate

Just as with RS232, RS485 does not define with which bitrate / baud rate data is to be transmitted. So any possible value between 0 and 10 Mbit/s is possible. In practice, only a very small subset of all possible bitrates is used, due to the capabilities of the electronic circuits used for RS485.

So we usually only see:

50, 110, 150, 300, 600, 1200, 2400, 4800 and 9600 bits/s
19.2, 28.8, 38.4, 57.6 , 115.2, 187.5, 375 and 500 Kbit/s.
1.0, 1.5, 3.0, 6.0 and 10 Mbit/s.

Theoretically, RS422/RS485 cannot operate beyond 10 Mbit/s, but due to Profibus/DP's means of extending RS485 technology speeds up to 12 Mbit/s are possible. However this is not encountered when using Modbus together with RS485.



General settings

parameter	options	remarks
addressing	address configurable from 1 to 247 (default 1)	
broadcast support	yes	
baud rate	9600 19200 (default) 38400 57600 Baud (MBC3 type only) 115200 Baud (MBC3 type only)	
electrical interface	RS485 2W-cabling	
data bits	RTU = 8, ASCII = 7	
stop bits	1	The use of no parity requires 2 stop bits

Excerpt from a vendor's manual showing which bitrates / baud rates are supported, and also that a 2-wire RS485 cabling system is used. Furthermore it also shows that the device supports usage of broadcast. Source: Bronkhorst Hi-Tech.

7. Data transmission

This is identical to RS232 (see appendix A).

8. Serial transmission format

This is identical to RS232 (see appendix A).

9. Parity

This is identical to RS232 (see appendix A).

10. Troubleshooting RS485

Troubleshooting RS485 is sometimes easier than with RS232, as there are no modem control signals that can be wired wrong. But troubleshooting can also be more difficult, because there can be a lot of devices on a network, each of which can be the source of a problem. Due to the bus way of wiring, this device can cause problems for other devices, which is sometimes difficult to detect (especially on longer networks).

Typical issues:

- The wrong type of cable is used
- The cable (trunk) is too long (for the bitrate used)
- The stubs are too long (for the bitrate used)
- There is more than one device per stub
- The termination resistors are missing
- There are too many termination resistors
- Not all devices are working on the same bitrate
- The A and B wires are swapped
- Two- and four-wire RS485 devices are mixed
- There are too many devices on the network

INDEX