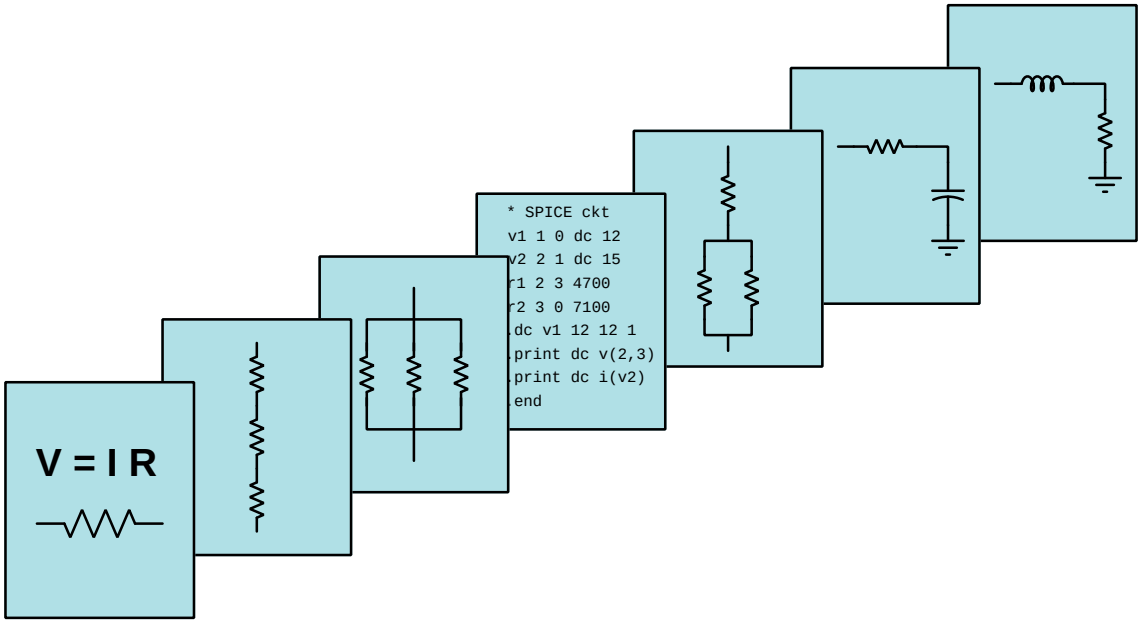
# Modular Electronics Learning (ModEL) project




## Modbus Networks



Last update = 19 February 2025

# Contents

# Chapter 1

# Introduction

## 1.1 Recommendations for students

The first industrial programmable logic controller (PLC) was manufactured by the Modicon company in the United States, and very soon after the development of this revolutionary control computer that same manufacturer developed a digital communication network called *Modbus* designed to allow multiple Modicon PLCs to communicate data between each other over simple two- or three-conductor network cables. The development of Modbus happened in 1979, and for better or for worse this same communication protocol is still in widespread use at the time of this writing (2019).

One factor instrumental in the wide adoption of Modbus throughout American industry was that it was *opened* to other manufacturers. In other words, the data-formatting standard defining Modbus as a communication protocol was published for anyone to study and use. The result of opening this standard is that many manufacturers adopted it for use within their own products, and fairly soon multiple manufacturers' products became *interoperable* because they "spoke" the same digital "language". Now it is common to find Modbus "spoken" by not only industrial control products but also personal computers (with the appropriate software libraries installed), commercial heating and cooling control products, variable-frequency AC motor drives (VFDs), and laboratory instrumentation, and even hobbyist-level electronic devices.

This module introduces the Modbus protocol along with practical applications for its use.

Important concepts related to Modbus include **digital** versus **analog** signaling, digital memory **reading** versus **writing**, memory **addresses**, **serial protocols**, the **OSI Reference model**, **master** versus **slave** network devices, **data frames**, **error checking**, **encapsulation**, **endianness**, and digital **codes**.

Here are some good questions to ask of yourself while studying this subject:

- How might an experiment be designed and conducted to measure the latency (time lag) within a Modbus control network? What hypothesis (i.e. prediction) might you pose for that experiment, and what result(s) would either support or disprove that hypothesis?
- What is the purpose of a VFD?

- What is the purpose of a PLC?
- How is a PLC able to differentiate between different Modbus slave devices (e.g. VFDs) connected to it?
- What are some advantages and also disadvantages to using Modbus rather than analog signals for applications such as electric motor control?
- How does Modbus relate to various serial data protocols such as EIA/TIA-232?
- What are some of the limitations of Modbus inherent to its data frame format?
- How does ASCII differ from RTU in Modbus communications?
- How is encapsulation (sometimes) used in Modbus communications?
- How is it possible to communicate data longer than 16 bits (e.g. 32-bit floating-point values) in Modbus?
- What does it mean to say that Modbus is a "layer-7" protocol?
- How are Modbus data registers addressed within the target device?
- What does it mean if data is encoded into binary in *big-endian* order versus *little-endian* order?
- What does it mean if data is *byte-swapped*? What does it mean if data is *word-swapped*?

## 1.2 Challenging concepts related to Modbus

The following list cites concepts related to this module's topic that are easily misunderstood, along with suggestions for properly understanding them:

- **Encapsulation** – the notion of one protocol's data frame being contained as payload within another's is somewhat confusing, but is made clear by inspection of actual communications using packet-sniffing software (e.g. `Wireshark`).
- **Network arbitration** – the same is true for channel arbitration techniques such as CSMA/CD, master-slave, and others. It is helpful to imagine a set of devices all requiring access to a common channel of communication, and stepping through each of the protocols to see how they manage this access without having multiple devices "talk over" one another. This, like so many other things, simply takes time to digest and cannot be rushed.
- **Relative addressing** – Modbus is a legacy protocol, and it definitely shows its age by features such as decimal-oriented address ranges, absolute addresses starting with "1", and relative addresses starting with "0".

## 1.3 Recommendations for instructors

This section lists realistic student learning outcomes supported by the content of the module as well as suggested means of assessing (measuring) student learning. The outcomes state what learners should be able to do, and the assessments are specific challenges to prove students have learned.

- **Outcome** – Demonstrate effective technical reading and writing

  Assessment – Students present their outlines of this module's instructional chapters (e.g. Case Tutorial, Tutorial, Historical References, etc.) ideally as an entry to a larger Journal document chronicling their learning. These outlines should exhibit good-faith effort at summarizing major concepts explained in the text.

- **Outcome** – Decode Modbus messages

  Assessment – Interpret all aspects of a Modbus message given the message frame; e.g. pose problems in the form of the "Interpreting an ASCII message frame" Quantitative Reasoning question.

- **Outcome** – Independent research

  Assessment – Locate Modbus-compatible device datasheets and properly interpret some of the information contained in those documents including physical-layer communication standard (e.g. serial, Ethernet, etc.), Modbus memory maps, etc.

# Chapter 2

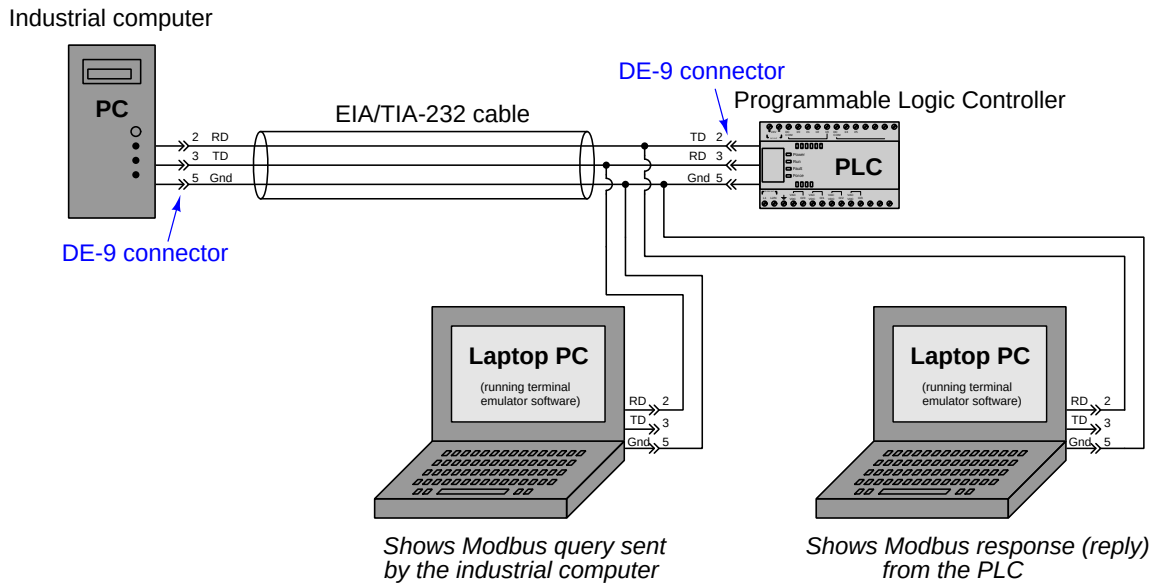
# Case Tutorial

The idea behind a *Case Tutorial* is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as a scientific experimenter. Hopefully you will find these cases illuminating, and a good supplement to text-based tutorials.

These examples also serve well as challenges following your reading of the other Tutorial(s) in this module – can you explain *why* the circuits behave as they do?

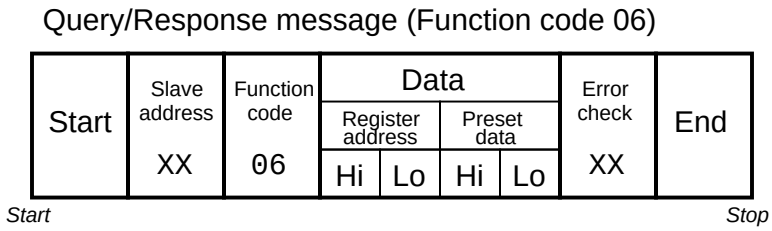
## 2.1 Example: Modbus ASCII "Write One Register" exchange

In the following system an industrial computer sends a Modbus query to a programmable logic controller (PLC), which in turn replies with a Modbus response. Both devices use Modbus ASCII to communicate, which allows us to use a pair of portable laptop computers to display each message in human-readable form:




ASCII message sent by the industrial computer = `:050610010200E2`

ASCII message sent in response by the PLC = `:050610010200E2`



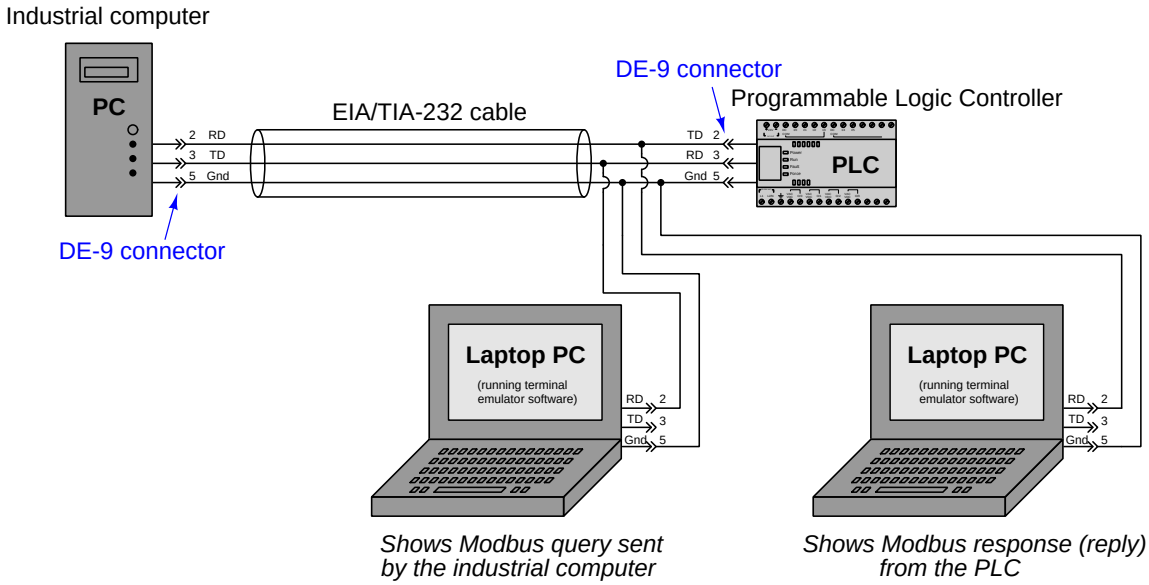

**Analysis of query :** `05 06 1001 0200 E2`

- `:` is the starting character
- `05` is the slave address of the PLC
- `06` is the function code (06 = "Write One Register")
- `1001` is the register's relative address (relative address `0x1001` = 4097 decimal = absolute address 44098 decimal)
- `0200` is the data to be written to register 44098
- `E2` is the message checksum (LRC)

The PLC's response to this message is to simply echo it verbatim so that the industrial computer will be able to verify its receipt.

## 2.2 Example: Modbus ASCII "Read Register" exchange

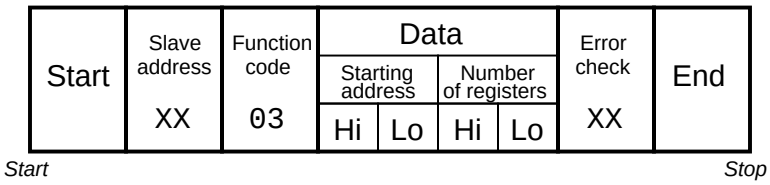
In the following system an industrial computer sends a Modbus query to a programmable logic controller (PLC), which in turn replies with a Modbus response. Both devices use Modbus ASCII to communicate, which allows us to use a pair of portable laptop computers to display each message in human-readable form:




ASCII message sent by the industrial computer = `:050310000002E6`

ASCII message sent in response by the PLC = `:050304FF0600648B`

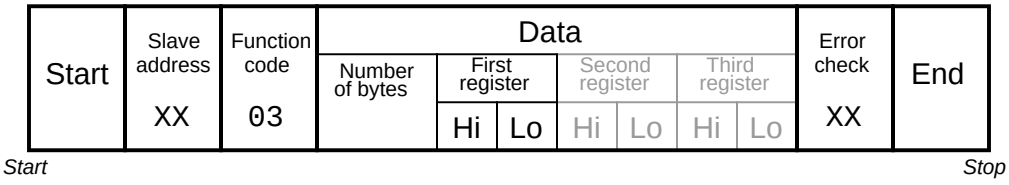
Query message (Function code 03)




**Analysis of query :** `05 03 1000 0002 E6`

- `:` is the starting character
- `05` is the slave address of the PLC
- `03` is the function code (03 = "Read Register")
- `1000` is the starting address (relative address `0x1000` = 4096 decimal = absolute address 44097 decimal)
- `0002` is the number of 16-bit registers to be read (two)
- `E6` is the message checksum (LRC)

Response message (Function code 03)




**Analysis of response :** `05 03 04 FF06 0064 8B`

- `:` is the starting character
- `05` is the slave address of the PLC
- `03` is the function code (03 = "Read Register")
- `04` is the number of bytes returned (four bytes = two 16-bit registers)
- `FF06` is the value stored in register 44097
- `0064` is the value stored in register 44098
- `8B` is the message checksum (LRC)

# Chapter 3

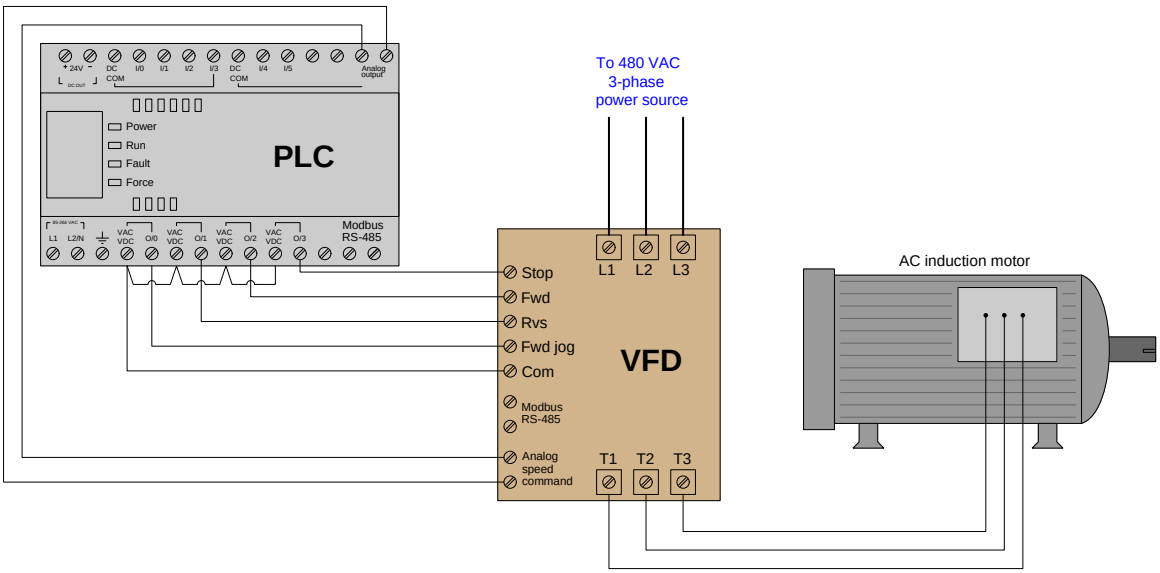
# Tutorial

*Modbus* is a very popular data communication protocol found in a variety of industrial and commercial systems, often associated with electric motor controls. This Tutorial will explore the basic concept of networked control for industrial devices, and then explore details of the Modbus protocol itself.

One of the most common forms of computer used in industrial control is the *Programmable Logic Controller*, or *PLC*. These devices are similar in function to microcontrollers, but designed to be programmed using languages much simpler than assembly or C in order to allow technical personnel with limited programming experience to configure these controllers to perform useful automation tasks.

## 3.1 Discrete motor control

We may begin our exploration of Modbus by first considering an example of a PLC-controlled motor system that does *not* employ Modbus. Here, the PLC sends individually-wired Forward, Reverse, and Stop, and speed-control command signals to a variable-frequency drive (VFD) which then sends three-phase power of varying frequency to an AC induction motor to do some useful task:

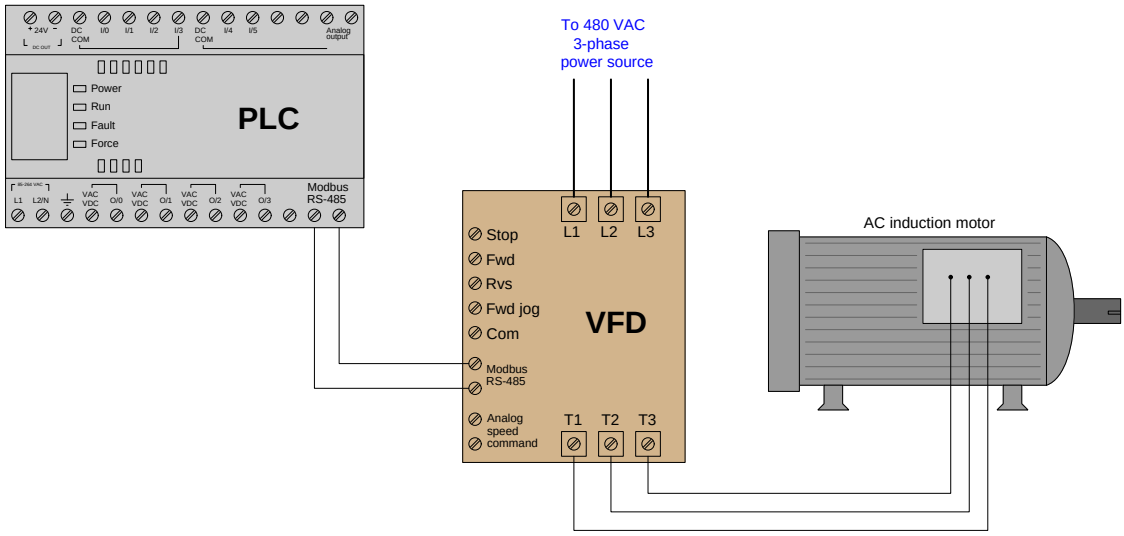



The discrete commands (e.g. Stop, Forward, Reverse) are nothing more than on/off contact closures provided by the PLC's output channels to the VFD's input terminals. When the PLC commands the VFD to run in the Reverse direction, it simply activates output channel `O/1` which closes a relay contact inside the PLC to connect the VFD's "Rvs" terminal to the VFD's "Com" terminal. The VFD detects this electrical continuity, and responds by running the motor in its reverse direction. Motor speed is commanded by an analog voltage signal (typically 0 to 10 Volts DC) output by the PLC, with 0 Volts representing zero speed and 10 Volts representing full speed. The VFD receives this analog voltage signal and responds to it by outputting the appropriate frequency of three-phase AC power to the induction motor.

While this system is certainly functional, it does not represent the *only* way for the PLC to issue commands to the VFD to control the motor. Instead of using discrete conductors for each motor function, it is possible to connect the PLC and VFD together with a digital network cable and issue commands as digital codes to do the same. One such digital network standard is *Modbus*, which we will see applied in the next section.

## 3.2 Networked motor controls

Now consider this updated motor control system, where the only connecting wires between the PLC and VFD is a single two-conductor cable between the Modbus/RS-485 terminals[1] of both devices. The PLC functions as a Modbus master device while the VFD functions as a Modbus slave:

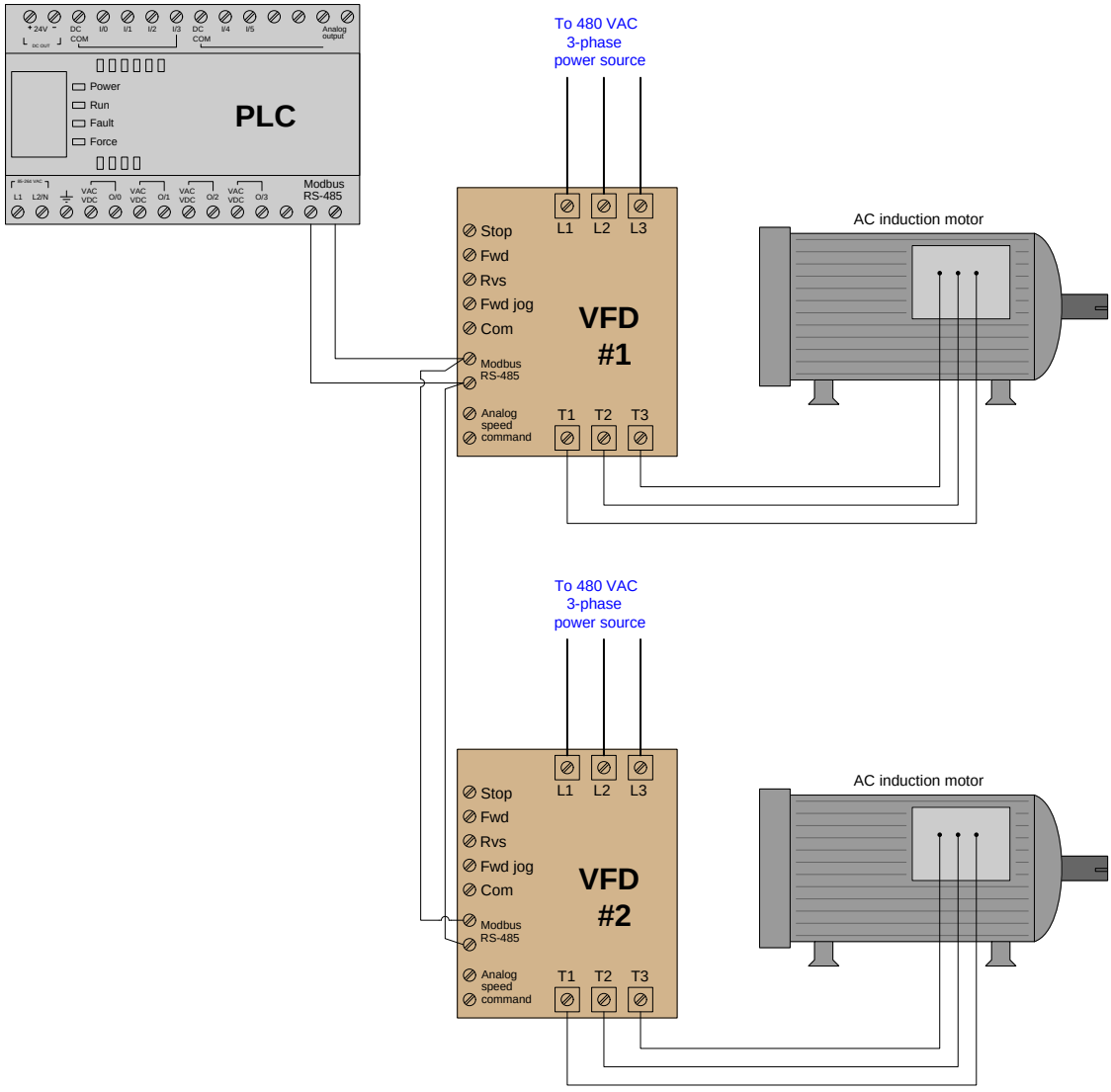



By using appropriate Modbus commands transmitted to the VFD, the PLC is able to issue all the same commands (e.g. Stop, Forward, Reverse, speed control) as before but using far fewer wires. For example Modbus command code `05` writes a single bit of data to the receiving device, allowing the PLC to send discrete-signal commands to the VFD one at a time. When the PLC commands the VFD to run in the Reverse direction, it issues a `05` command followed by a "1" data bit addressed to the appropriate memory location inside the VFD reserved for the "Reverse" command bit. When the PLC commands the VFD to change motor speed, it issues an `06` Modbus code ("write register") followed by a 16-bit number representing the desired motor speed and the appropriate address within the VFD reserved for speed command.

Not only can the PLC issue all the same commands as before, but it may also *read* data from the VFD which it could not do before. For example, if the VFD provides a memory location for storing fault codes (e.g. motor overcurrent, bus undervoltage, etc.), the PLC may be programmed to issue an `03` Modbus code to read a single register (16 bit binary number) from that memory location within the VFD, and thereby monitor the status of the VFD to alert human technicians of potential problems, and/or to modify its own supervisory control of the motor.

[1] RS-485 is one standard for *serial* data communication using Non-Return-to-Zero (NRZ) encoding of bits. Two un-grounded conductors convey a pulsed voltage signal between the connected devices with one polarity of voltage representing a "0" bit and the other polarity representing a "1" bit.

Another advantage of the Modbus communication standard is that it is designed to address multiple devices on the same network. This means our hypothetical PLC is not limited to controlling and monitoring just one motor, but *up to 247 separate Modbus slave devices* on the same two-wire communication cable! The following illustration shows how this might work for multiple motors:




Each VFD is given its own Modbus network *slave address*, so that the PLC is able to distinguish between the two drives when communicating on the same wire pair. Every Modbus code transmitted by the PLC contains this address as a single byte (8 bits) of data in order to make the receiving VFD aware that the code applies to it and not to any other Modbus device on the network. In

this example, we may wish to address VFD #1 with Modbus address 1, and VFD #2 with Modbus address 2. The Modbus standard provides a "broadcast address" of 0 which addresses *all* devices on the network simultaneously. For example, if the PLC needed to start all motors in the same direction at once, it could issue a Modbus code `05` (write a single bit) to the same address inside each VFD representing the command bit for the correct direction of motor rotation. So long as the VFDs are identically configured, the data will be received and interpreted by each VFD identically which will cause them to both start up in the same direction.

The only disadvantages to using Modbus as opposed to dedicated wires for each sensing and control function are *speed* and *reliability*. Modbus is necessarily slower than dedicated wire control because the PLC cannot simultaneously issue different commands on the network. For example, if the PLC needed to tell a VFD to begin turning its motor in the forward direction at 1050 RPM, the Modbus-based system would need to issue two separate Modbus codes whereas the individually-wired system could issue these commands all at once. This disadvantage, however, is hardly worth considering if the Modbus network communicates at reasonably high speed (thousands of bits per second). The disadvantage of reliability may be readily perceived if we consider how each system would respond to a wire fault (e.g. one wire coming loose and disconnected from a screw terminal). In the individually-wired system, one wire fault disables that one motor-control function but not necessarily any of the other functions. In the Modbus-based system, one wire fault disables *everything* because any Modbus communication requires full function of that two-conductor communication cable. The problem is even larger when multiple devices are controlled by the same Modbus cable: if a fault occurs between the controlling PLC and all the field devices, the PLC will lose control (and monitoring) for every one of those field devices! This is a factor worth considering when deciding whether or not to use any digital communication method for monitoring and control of multiple devices.

Modbus, especially when implemented over simple serial networks such as EIA/TIA-232[2] and EIA/TIA-485[3], is a rather primitive protocol. The seemingly arbitrary decimal codes used to issue commands and specify addresses is antiquated by modern standards. For better or for worse, though, a great many digital industrial devices "speak" Modbus, even if they are also capable of communicating via other network protocols. Using Modbus to communicate with modern control equipment is therefore an act of homage to 1970's-era telecommunications: all participating devices in a Modbus network essentially behave the same as a 1970's vintage Modicon PLC for the sake of exchanging information, even if their processing capabilities enable communications far more sophisticated than the Modbus protocol. A Modbus device querying another Modbus device does not "know" how modern or antiquated that other device is, because the basic Modbus standard has remained fixed for all this time.

[2] EIA/TIA-232 is also known by its older title *RS-232*. It is a Non-Return-to-Zero (NRZ) serial data protocol using ground-referenced voltage signals to represent "0" and "1" bits. These bits are transmitted one at a time at some constant bit rate, and interpreted by the receiving device(s) before being assembled into whole digital words.

[3] Also known by its older title *RS-485*.

## 3.3 Modbus history

Developed by the Modicon company (the original manufacturer of the *Programmable Logic Controller*, or *PLC*) in 1979 for use in its industrial control products, *Modbus* is a protocol designed specifically for exchanging process data between industrial control devices. The Modbus standard does not specify any details of physical networking, and thus may be deployed on many different types of physical networks. In other words, Modbus primarily falls within layer 7 of the OSI Reference Model (the so-called "Application Layer") and therefore is compatible[4] with any lower-level communication protocols including EIA/TIA-232, EIA/TIA-485, Ethernet (the latter via TCP/IP), and a special token-passing network also developed by Modicon called *Modbus Plus*. The Modbus standard primarily defines the *meaning* of various Modbus commands, the addressing scheme used to place data within devices, and the formatting of the data.

Modbus consists of a set of standardized digital codes intended to read data from and write data to industrial devices. A Modbus-compliant industrial device has been programmed to understand these codes and respond to them appropriately when received. The simplest Modbus codes read and write single bits of data in the device's memory, for example the status of a PLC input channel, PLC output channel, or status bit within a PLC program. Other Modbus codes operate on 16-bit words of data, useful for reading and writing counter and timer accumulated values, operands for mathematical instructions, converted analog signals, etc.

Early implementations of Modbus used EIA/TIA-485 as the network physical layer, which is strictly a layer 1 protocol. This meant that Modbus needed to specify a channel arbitration scheme in order to negotiate communications with multiple devices on a network. The arbitration chosen was master/slave, where one PLC functioned as the master Modbus device and all other devices functioned as Modbus slaves.

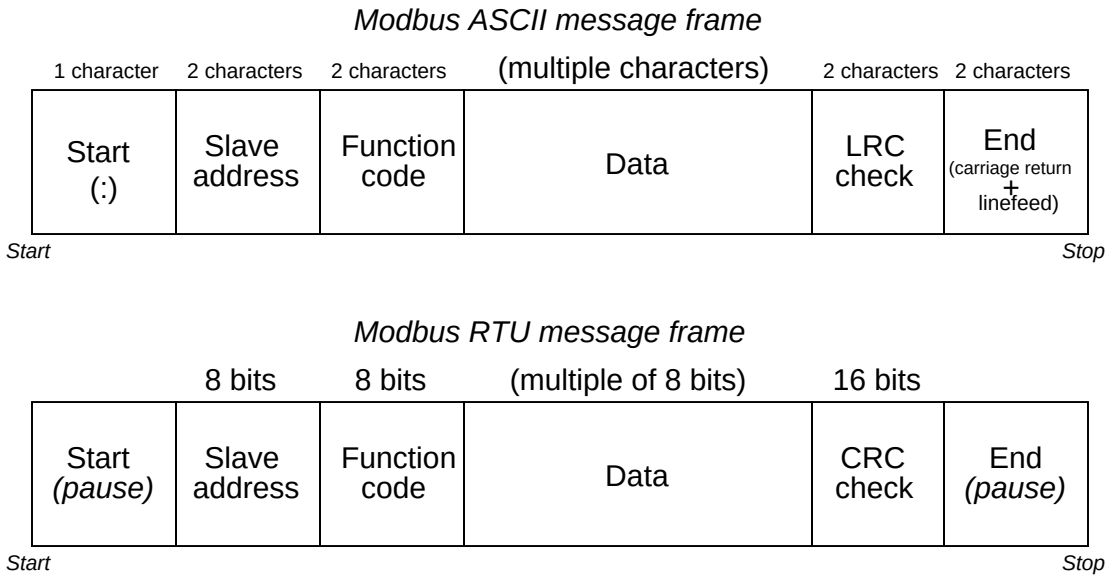
Interestingly, this vestige of master/slave arbitration survives to this day, even when Modbus commands are communicated via networks with their own differing arbitration methods. For example, Modbus commands communicated over Ethernet still reference "slave" addresses even though the Ethernet network those messages are sent over uses CSMA/CD arbitration. In other words, there is a hint of OSI layer 2 embedded within Modbus messages that still dictates which Modbus devices may issue commands and which must obey commands.

[4] These Modbus data frames may be communicated directly in serial form, or "wrapped" in TCP segments and IP packets and Ethernet frames, or otherwise contained in any form of packet-based protocol as needed to transport the data from one device to another. Thus, Modbus does not "care" how the data is communicated, just what the data means for the end-device.

## 3.4 Serial Modbus data frames

The Modbus communication standard defines a set of commands for reading (receiving) and writing (transmitting) data between a master device and one or more slave devices connected to the network. Each of these commands is referenced by a numerical code, with addresses of the master and slave devices' internal registers (data sources and data destinations) specified along with the function code in the Modbus frame.

Two different formats are specified in the Modbus standard for simple serial-based networks: *ASCII* and *RTU*. The difference between these two modes is how addresses, function codes, data, and error-checking bits are represented. In Modbus ASCII mode, all slave device addresses, function codes, and data are represented in the form of ASCII characters (7 bits each), which may be read directly by any terminal program (e.g. `minicom`, `Hyperterminal`, `kermit`, etc.) intercepting the serial data stream. This makes troubleshooting easier: to be able to directly view the Modbus data frames in human-readable form. In Modbus RTU mode, all slave device addresses, function codes, and data are expressed in raw binary form. Different error-checking techniques are used for ASCII and RTU modes as well. The following diagram compares data frames for the two Modbus modes:




As you can see from a comparison of the two frames, ASCII frames require nearly twice[5] the

[5] Recall that each ASCII character requires 7 bits to encode. This impacts nearly every portion of the Modbus data frame. Slave address and function code portions, for example, require 14 bits each in ASCII but only 8 bits each in RTU. The data portion of a Modbus ASCII frame requires one ASCII character (7 bits) to represent each hexadecimal symbol that in turn represents just 4 bits of actual data. The data portion of a Modbus RTU frame, by contrast, codes the data bits directly (i.e. 8 bits of data appear as 8 bits within that portion of the frame). Additionally, RTU data frames use quiet periods (pauses) as delimiters, while ASCII data frames use three ASCII characters in total to mark the start and stop of each frame, at a "cost" of 21 additional bits. These additional delimiting bits do serve a practical purpose, though: they format each Modbus ASCII data frame as its own line on the screen of a terminal program.

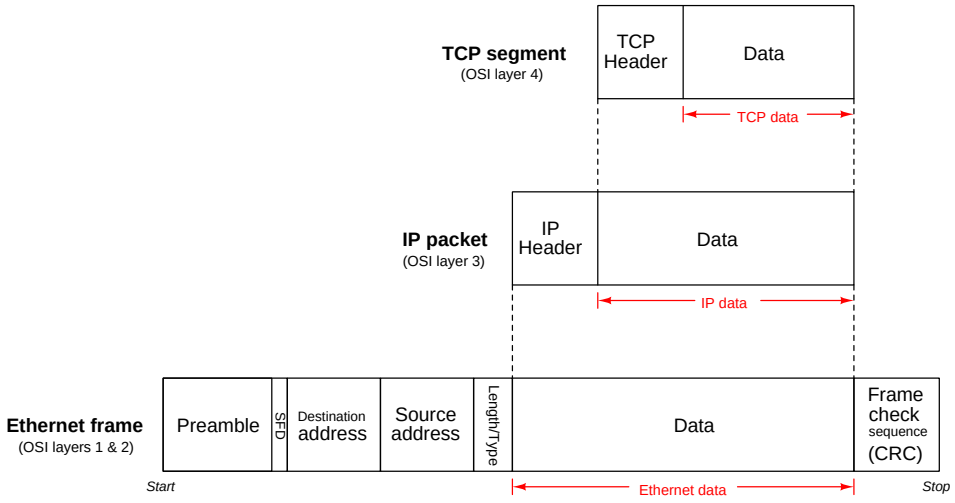
number of bits as RTU frames, making Modbus ASCII slower than Modbus RTU for any given data rate (bits per second).

The contents of the "Data" field vary greatly depending on which function is invoked, and whether or not the frame is issued by the master device or from a slave device. More details on Modbus "Data" field contents will appear in a later subsection. Also, you may consult section 5.1 beginning on page 36 which lists the data frame format for multiple Modbus function codes.

## 3.5 Modbus/TCP data frames

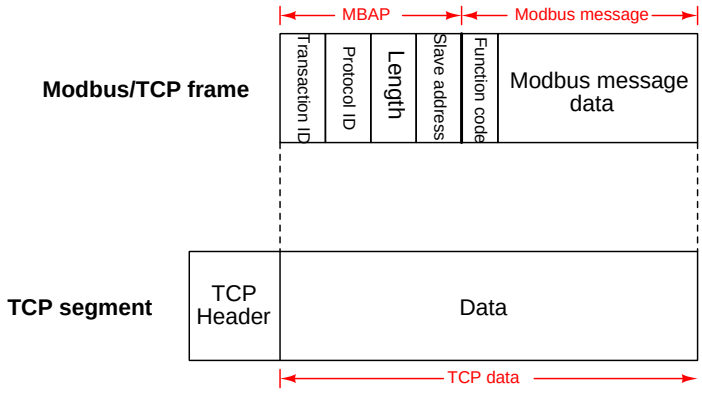
Since Modbus is strictly a "layer 7" protocol, these message frames may be embedded within other data frames specified by lower-level protocols. *Modbus/TCP* is a popular example of this, where individual Modbus data frames are encapsulated within TCP/IP segments/packets, which are then (usually) encapsulated again as Ethernet frame data payloads to arrive at the destination device. This "multi-layered" approach inherent to Modbus being such a high-level protocol may seem cumbersome, but it offers great flexibility in that Modbus frames may be communicated over nearly any kind of virtual and physical network type.

The following illustration shows how digital data is encapsulated within the Transmission Control Protocol (TCP) by the prepending of a TCP "header" containing digital codes directing any TCP-compliant networking device with instructions specific to TCP. Next, we see how that TCP "segment" is further encapsulated into an IP "packet" by the prepending of more digital information, this time instructing any IP-compliant device what to do with that packet. Lastly, this IP packet is further encapsulated as the payload of an Ethernet frame by "wrapping" it with a header full of addressing and other information necessary for Ethernet at the front and with a "frame check sequence" at the end for error-checking:




This process of repeated encapsulation of data-within-data is how *any* digital data must be prepared for transmission across a network using TCP, IP, and Ethernet protocols.

In Modbus/TCP the Modbus message is itself encapsulated with a special header unique to Modbus/TCP before becoming the payload (data) of a TCP segment. This header is called a *Modbus Application Protocol* or *MBAP*. Details of the MBAP header are shown in this next illustration showing the encapsulation of an entire Modbus/TCP frame within a TCP segment:




Transmission Control Protocol (TCP) requires that virtual *ports* be numerically specified on both ends of the network connection for each segment, and for Modbus/TCP the standardized port number is *502*.

It should be noted that data fields contained within the MBAP such as the Transaction ID, the Protocol ID, etc. are all encoded *big-endian*[6] which means all their bits and bytes read directly in the order received over time, from most-significant bit to least-significant bit. The same is true for data fields within the serial-based Modbus frames (i.e. RTU and ASCII). However, it is important to note that any data written to or read from a Modbus device via Modbus registers will follow the "endianness" prescribed by the manufacturer of that device, which may be big-endian or little-endian at the manufacturer's discretion. It is even common to find some manufacturers ordering their Modbus data registers to be byte-swapped[7] or word-swapped[8], so one must pay close attention to the manufacturer's technical documentation when programming Modbus-based communication systems.

[6]The designators "big-endian" and "little-endian" refer to whether or not a digital word is transmitted in order of MSB to LSB (i.e. the "big end" first) or LSB to MSB ("little end" first). For example, the number one hundred seventy-five encoded as a single byte (8 binary bits) would be `0b10101111` if written in big-endian order. However, that same number would be `0b11110101` if written in little-endian order.

[7]This is where the two bytes of data comprising a single 16-bit Modbus register are reversed in order even if the bits within each of those bytes is big-endian. For example the 16-bit word `0x3BA9` in a system using byte-swapped words will need to be interpreted as `0xA93B`.

[8]This is where two consecutive 16-bit words representing a single 32-bit field of data swap the order of those 16-bit words. For example, in a system using word-swapping but no byte swapping, the two consecutive words `0x57C1` and `0xD028` actually represent a 32-bit word `0xD02857C1`. If the system uses word-swapping *and* byte-swapping, the proper way to interpret those same two consecutive 16-bit registers would be `0x28D0C157`. Confusing, no? The real fun begins when the manufacturer fails to document any of this, leaving you to figure it out on your own.

## 3.6 Modbus function codes and addresses

A listing of commonly-used Modbus function codes appears in the following table:

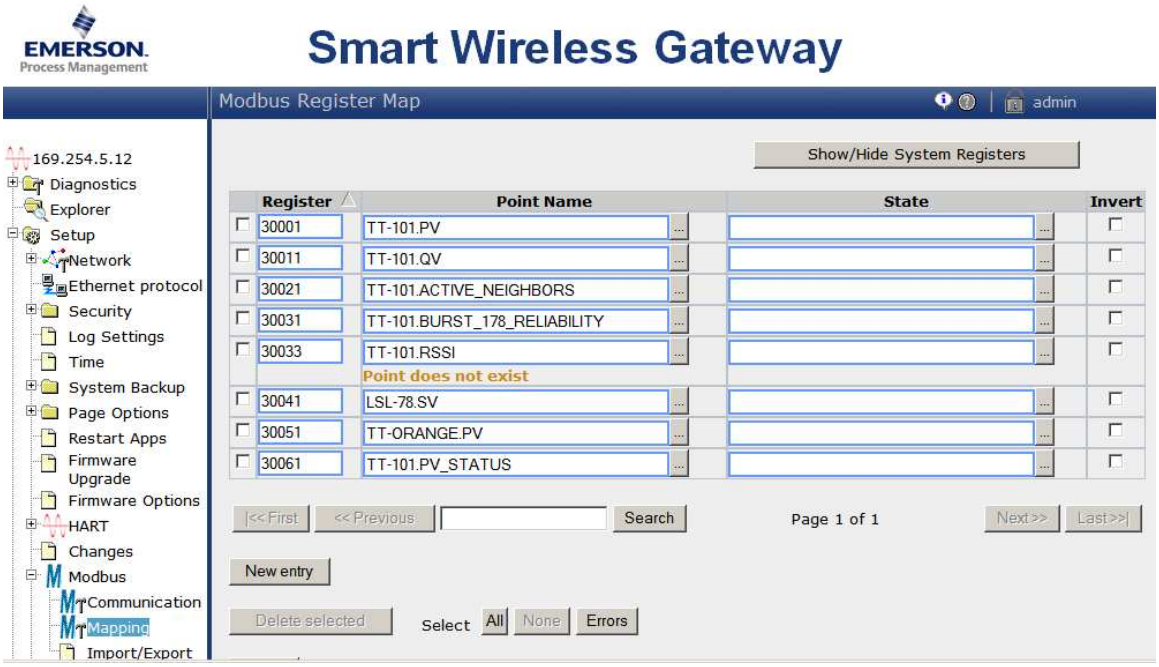
| **Modbus code** (decimal) | **Function** |
|---|---|
| 01 | Read one or more PLC output "coils" (1 bit each) |
| 02 | Read one or more PLC input "contacts" (1 bit each) |
| 03 | Read one or more PLC "holding" registers (16 bits each) |
| 04 | Read one or more PLC analog input registers (16 bits each) |
| 05 | Write (force) a single PLC output "coil" (1 bit) |
| 06 | Write (preset) a single PLC "holding" register (16 bits) |
| 15 | Write (force) multiple PLC output "coils" (1 bit each) |
| 16 | Write (preset) multiple PLC "holding" registers (16 bits each) |

Live data inside of any digital device is always located at some *address* within that device's random-access memory (RAM). The Modbus "984" addressing standard defines sets of fixed numerical addresses where various types of data may be found in a PLC or other control device. The absolute address ranges (according to the Modbus 984 scheme) are shown in this table, with each address holding 16 bits of data:

| **Modbus codes** | **Address range** | **Purpose** |
|---|---|---|
| 01, 05, 15 | 00001 to 09999 | Discrete outputs ("coils"), *read/write* |
| 02 | 10001 to 19999 | Discrete inputs ("contacts"), *read-only* |
| 04 | 30001 to 39999 | Analog input registers, *read-only* |
| 03, 06, 16 | 40001 to 49999 | "Holding" registers, *read/write* |

Note how all the Modbus address ranges begin at the number one, not zero as is customary for so many digital systems. For example, a PLC with sixteen analog input channels numbered 0 through 15 by the manufacturer may "map" those input registers to Modbus addresses 30001 through 30016, respectively.

While this fixed addressing scheme was correct for the original PLCs developed by Modicon, it almost never corresponds directly to the addresses within a modern Modbus master or slave device. Manufacturer's documentation for Modbus-compatible devices normally provide Modbus "mapping" references so technicians and engineers alike may determine which Modbus addresses refer to specific bit or word registers in the device. In some cases the configuration software for a Modbus-compatible device provides a utility where you may assign specific device variables to standard Modbus register numbers. An example of a Modbus variable mapping page appears in this screenshot taken from the configuration utility for an Emerson Smart Wireless gateway, used to "map" data from variables within *Wireless*HART radio-based industrial field instruments to Modbus registers within the gateway device where other devices on a wired network may read that data:




As you can see here, the primary variable within temperature sensor TT-101 (`TT-101.PV`) has been mapped to Modbus register 30001, where any Modbus master device on the wired network will be able to read it. Likewise, the secondary variable within level switch LSL-78 (`LSL-78.SV`) has been mapped to Modbus register 30041.

It is important to note that Modbus registers are 16 bits each, which may or may not exactly fit the bit width of the device variable in question. If the device variable happens to be a 32-bit floating point number, then *two* contiguous Modbus registers must be used to hold that variable, only the first of which will likely appear on the Modbus mapping page (i.e. the Modbus map will only show the *first* Modbus register of that pair). If the device variable happens to be a boolean (single bit), then it is likely only one bit within the 16-bit Modbus register will be used, the other 15 bits being "wasted" (unavailable) for other purposes. Details such as this may be documented in the manual for the device performing the Modbus mapping (in this case the Emerson Smart Wireless Gateway), or you may be forced to discover them by experimentation.

## 3.7 Modbus relative addressing

An interesting idiosyncrasy of Modbus communication is that the address values specified within Modbus data frames are *relative* rather than *absolute*. Since each Modbus read or write function only operates on a limited range of register addresses, there is no need to specify the entire address in the data frame. For example, Modbus function code 02 reads discrete input registers in the device with an absolute address range of 10001 to 19999 (i.e. all the addresses beginning with the digit "1"). Therefore, it is not necessary for the "read" command function 02 to specify the first digit of the register address. Instead, the read command only needs to specify a four-digit "relative address" specifying how far up from the beginning of the address range (in this case, from 10001) to go.

An analogy to aid your understanding of relative addressing is to envision a hotel building with multiple floors. The first digit of every room number is the same as the floor number, so that the first floor only contains rooms numbered in the 100's, the second floor only contains rooms numbered in the 200's, etc. With this very orderly system of room numbers, it becomes possible to specify a room's location in more than one way. For example, you could give instructions to go to room 314 (an *absolute* room number), or alternatively you could specify that same room as "number 14 (a *relative* room number) on the third floor". To a hotel employee who only works on the third floor, the shortened room number might be easier to remember.

In Modbus, relative addresses are just a little bit more complicated than this. Relative addresses actually span a range beginning at zero, while absolute addresses begin with "1" as the least-significant digit. This means there is an additional offset of 1 between a Modbus relative address and its corresponding absolute address. Returning to the hotel analogy, imagine the very first room on the third floor was room 301 (i.e. there was no room 300) and that the relative address represented the number of rooms *past* that first room. In this unintuitive scheme, room 314 could be specified as "the 13th room after the starting room on the third floor". If this seems needlessly confusing, you are not alone. *Welcome to Hotel Modbus.*

A few examples are given here for illustration:

- Read the content of contact register 12440: *Modbus read function 02; relative address 2439*
- Read the content of analog input register 30050: *Modbus read function 04; relative address 49*
- Read the content of holding register 41000: *Modbus read function 03; relative address 999*
- Write multiple output coils in register 00008: *Modbus write function 15; relative address 7*

In each case, the pattern is the same: the relative address gets added to the first address of that range in order to arrive at the absolute address within the Modbus device. Referencing the first example shown above: 2439 (relative address) + 10001 (first address of register range) = 12440 (absolute address).

Thankfully, the only time you are likely to contend with relative addressing is if you program a computer using some low-level language such as assembly or C++. Most high-level industrial programming languages such as Function Block or Ladder Diagram make it easy for the end-user by allowing absolute addresses to be directly specified in the read and write commands. In a typical PLC program, for example, you would read contact register 12440 by simply specifying the number 12440 within the address field of a "read 02" instruction.

The following listing shows code (written in the C language) utilizing the open-source `libmodbus` function library instructing a computer to access 16-bit integer data from four Modbus "holding" registers (absolute addresses 49001 through 49004) via Modbus/TCP. The device's IP address is `192.169.0.10` and port `502` is used for the TCP connection:

```
#include <stdio.h>
#include <modbus.h>

modbus_t *Device;

int main (void)
{
  int read_count;
  uint16_t inreg_word[4];

  Device = modbus_new_tcp ("192.168.0.10", 502);
  modbus_set_error_recovery (Device, MODBUS_ERROR_RECOVERY_LINK);

  read_count = modbus_read_registers (Device, 9000, 4, inreg_word);

  printf("Number of registers read = %i \n", read_count);
  printf("Value of register 49001 = %i \n", inreg_word[0]);
  printf("Value of register 49002 = %i \n", inreg_word[1]);
  printf("Value of register 49003 = %i \n", inreg_word[2]);
  printf("Value of register 49004 = %i \n", inreg_word[3]);

  modbus_close (Device);
  modbus_free (Device);

  return read_count;
}
```

Note how the starting address passed to the read function is specified in relative form (9000), when in fact the desired absolute starting address inside the device is 49001. The result of running this code is shown here, the Modbus device in question being an Emerson Smart Wireless Gateway at 4:00 PM (i.e. 16:00 military time) on March 22, 2016. These four registers (49001 through 49004) happen to contain date and time information (year, month, day, and hour) stored in the device:

```
Number of registers read = 4
Value of register 49001 = 2016
Value of register 49002 = 3
Value of register 49003 = 22
Value of register 49004 = 16
```

This next listing shows similar code (also written in the C language[9]) accessing 16-bit integer data from three Modbus "analog input" registers (absolute addresses 30015 through 30017) via Modbus/TCP from the same device as before:

```
#include <stdio.h>
#include <modbus.h>

modbus_t *Device;

int main (void)
{
  int read_count;
  uint16_t inreg_word[3];

  Device = modbus_new_tcp ("192.168.0.10", 502);
  modbus_set_error_recovery (Device, MODBUS_ERROR_RECOVERY_LINK);

  read_count = modbus_read_input_registers (Device, 14, 3, inreg_word);

  printf("Number of registers read = %i \n", read_count);
  printf("Value of register 30015 = %i \n", inreg_word[0]);
  printf("Value of register 30016 = %i \n", inreg_word[1]);
  printf("Value of register 30017 = %i \n", inreg_word[2]);

  modbus_close (Device);
  modbus_free (Device);

  return read_count;
}
```

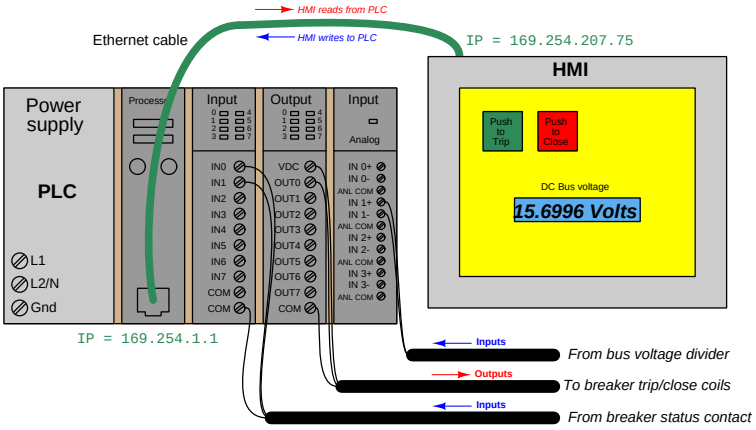
Note once again how the relative starting address specified in the code (14) maps to the absolute Modbus register address 30015, since analog input registers begin with the address 30001 and relative addresses begin at 0.

[9]This C-language code is typed and saved as a plain-text file on the computer, and then a *compiler* program is run to convert this "source" code into an "executable" file that the computer may then run. The compiler I use on my Linux-based systems is `gcc` (the GNU C Compiler). If I save my Modbus program source code to a file named `tony_modbus.c`, then the command-line instruction I will need to issue to my computer instructing GCC to compile this source code will be `gcc tony_modbus.c -lmodbus`. The argument `-lmodbus` tells GCC to "link" my code to the code of the pre-installed libmodbus library in order to compile a working executable file. By default, GCC outputs the executable as a file named `a.out`. If I wish to rename the executable something more meaningful, I may either do so manually after compilation, or invoke the "outfile" option of `gcc` and specify the desired executable filename: (e.g. `gcc -o tony.exe tony_modbus -lmodbus`). Once compiled, the executable file many be run and the results of the Modbus query viewed on the computer's display.

When using the `libmodbus` C/C++ library, the distinction between reading "analog input" registers (address range 30001 to 39999) and "holding" registers (address range 40001 to 49999) is made by the particular `libmodbus` function called. To read "analog input" registers in the 3XXXX address range, you use the `modbus_read_input_registers()` function. To read "holding" registers in the 4XXXX address range, you use the `modbus_read_registers()` function. This subtle difference in function names is important. Refer back to the two previous code examples to verify for yourself which function call is used in each of the register-reading applications.

## 3.8 Analyzing a Modbus/TCP message

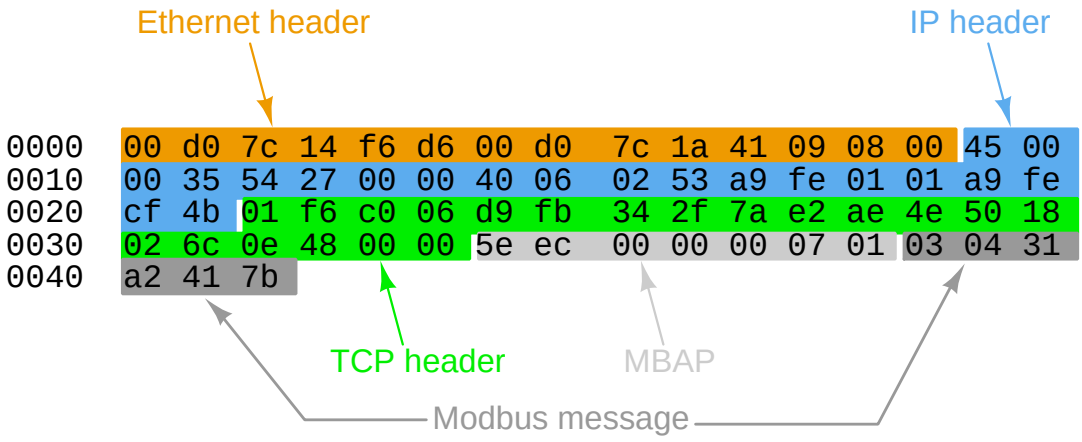
In this section we will analyze a single Modbus message communicated over an Ethernet network between a PLC and a *Human-Machine Interface* display unit also known as an *HMI*. These two industrial computers are used to monitor DC voltage in a solar power system as well as display and control the status of one of the circuit breakers within that power system:




Each device has its own unique IP address: the PLC's being `169.254.1.1` and the HMI's being `169.254.207.75`. Periodically the HMI sends data read requests to the PLC in order to poll for updated DC voltage measurements and breaker status bits. The DC voltage is sensed by one analog input channel on the PLC and scaled into units of Volts by the PLC, the result stored as a 32-bit floating-point number following the ANSI/IEEE standard 754 floating-point format.

The data exchange we are about to analyze is the HMI responding to the PLC's poll for measured DC voltage. In this particular brand and model of PLC, scaled analog measurement values are stored as two consecutive 16-bit "holding" registers, and so must be accessed by Modbus function code 03. The read command requests two registers to be transmitted back to the PLC, and in this case that response was captured using a network-monitoring application called `Wireshark`.

Here is a *hex dump* of the entire data frame, consisting of the Modbus message prepended with a MBAP header, encapsulated within a TCP segment, encapsulated within a single IP packet, encapsulated within one Ethernet frame. Each of these encapsulating elements is shown by color-highlighting:




Breaking down these portions byte-by-byte:

**Ethernet header**

- `00 d0 7c 14 f6 d6` is the destination device MAC address, with the first three bytes reserved for Koyo Electric (the HMI manufacturer in this case)
- `00 d0 7c 1a 41 09` is the source device MAC address, again with the first three bytes reserved for Koyo Electric (the PLC manufacturer in this case)
- `08 00` specifies the Ethernet payload type as IP version 4

**IP header**

- `45` specifies IP version 4 and a header length of 20 bytes (code 5)
- `00` declares the differentiated services field as "standard" (code 0)
- `00 35` specifies the total length of the IP packet
- `54 27` is the IP packet identification number
- `00 00` IP flags and fragment offset (zero since there is no fragmentation in this short packet)
- `40` specifies the time-to-live (64 seconds)
- `06` specifies the IP payload type as TCP (code 6)

- `02 53` is the header checksum, for error-detection purposes
- `a9 fe 01 01` is the source IP address (169.254.1.1 for the PLC)
- `a9 fe cf 4b` is the destination IP address (169.254.207.75 for the HMI)

**TCP header**

- `01 f6` specifies the source port number, in this case 502 which is reserved for Modbus/TCP
- `c0 06` specifies the destination port number, in this case 49158
- `d9 fb 34 2f` is the sequence number for this TCP segment, which is important for lengthy messages requiring segmenting into multiple TCP segments but is not necessary for this short Modbus message
- `7a e2 ae 4e` is the acknowledgment number for this TCP segment
- `5` specifies the total length of the TCP header
- `018` indicates PSH and ACK flag statuses
- `02 6c` specifies the window number
- `0e 48` is the checksum, for error-detection purposes
- `00 00` points to the last byte of any "urgent" data, nonexistent in this example

**MBAP**

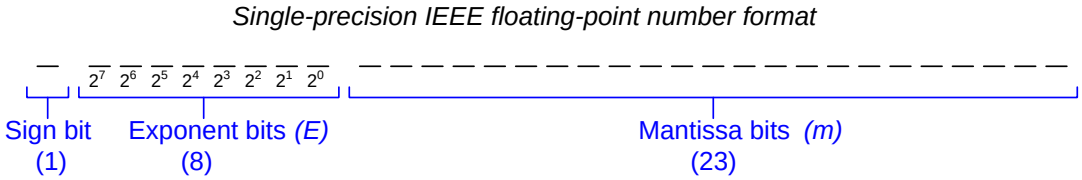
- `5e ec` is the Transaction Identifier for this Modbus message
- `00 00` is the Protocol Identifier, 0 by default
- `00 07` specifies the length (in bytes) of all remaining data fields
- `01` is the Unit Identifier, equivalent to the traditional Modbus slave address number

**Modbus message**

- `03` specifies the Modbus function code, in this case 03 (read "holding" register)
- `04` is the byte count for the registers requested
- `31 a2` is the contents of Register 0 (the first one read back from the PLC)
- `41 7b` is the contents of Register 1 (the second one read back from the PLC)

In order to properly interpret the 32-bit floating-point value representing DC bus voltage in the solar power system being monitored by the PLC, it helps to know that this particular brand and model of PLC (Koyo Electric "CLICK") happens to use word-swapped order for its floating-point numbers. Thus, the register data shown in the hex dump as `0x31A2417B` actually needs to be interpreted as `0x417B31A2`.

The ANSI/IEEE 754 floating-point number standard assigns the following meanings to the 32 bits:




Translating the word-swap-corrected `0x417B31A2` data word from the PLC into binary numeration instead of hexadecimal:

```
Raw binary for 0x417B31A2:
0100 0001 0111 1011 0011 0001 1010 0010
```

Next, populating the ANSI/IEEE-754 template with these bits:

```
Sign    Exponent                Mantissa
0       1000 0010      1111 0110 0110 0011 0100 010
```

The sign bit of 0 tells us this is a positive and not a negative quantity. The `0b10000010` exponent has a decimal value of 130, from which we must subtract the standard bias value of 127 to yield 3, which is the power-of-two we will apply to the number's mantissa. In addition to the −127 bias value specified in the ANSI/IEEE-754 standard, this standard also assumes a 1 prepended to the 23-bit mantissa field to give a 24-bit binary value. If we prepend that 1 bit to the mantissa shown, we get:

```
1.1111 0110 0110 0011 0100 010
```

Lastly, we must apply the exponent value of 3, which means $2^3$, or shifting the binary point three places to the right. This is the DC voltage value represented in fixed-point binary format:

```
1111.1011 0011 0001 1010 0010
```

Converting this into decimal[10], we get 15.699617385864258 Volts DC which was the DC bus voltage at the time of this Modbus query.

[10] The four "1" bits to the left of the binary point simply represent 15 (8 + 4 + 2 + 1). Bits to the right of the binary point have fractional place-weight values, the first bit representing one-half ($2^{-1}$), the next bit one-quarter, etc. I used Python as a manual calculator to convert this fixed-point binary value into decimal: `>>> 15 + pow(2,-1) + pow(2,-3) + pow(2,-4) + pow(2,-7) + pow(2,-8) + pow(2,-12) + pow(2,-13) + pow(2,-15) + pow(2,-19)`

# Chapter 4

# Historical References

This chapter is where you will find references to historical texts and technologies related to the module's topic.

Readers may wonder why historical references might be included in any modern lesson on a subject. Why dwell on old ideas and obsolete technologies? One answer to this question is that the initial discoveries and early applications of scientific principles typically present those principles in forms that are unusually easy to grasp. Anyone who first discovers a new principle must necessarily do so from a perspective of ignorance (i.e. if you truly *discover* something yourself, it means you must have come to that discovery with no prior knowledge of it and no hints from others knowledgeable in it), and in so doing the discoverer lacks any hindsight or advantage that might have otherwise come from a more advanced perspective. Thus, discoverers are forced to think and express themselves in less-advanced terms, and this often makes their explanations more readily accessible to others who, like the discoverer, comes to this idea with no prior knowledge. Furthermore, early discoverers often faced the daunting challenge of explaining their new and complex ideas to a naturally skeptical scientific community, and this pressure incentivized clear and compelling communication. As James Clerk Maxwell eloquently stated in the Preface to his book *A Treatise on Electricity and Magnetism* written in 1873,

> It is of great advantage to the student of any subject to read the original memoirs on that subject, for science is always most completely assimilated when it is in its nascent state . . . `[page xi]`

Furthermore, grasping the historical context of technological discoveries is important for understanding how science intersects with culture and civilization, which is ever important because new discoveries and new applications of existing discoveries will always continue to impact our lives. One will often find themselves impressed by the ingenuity of previous generations, and by the high degree of refinement to which now-obsolete technologies were once raised. There is much to learn and much inspiration to be drawn from the technological past, and to the inquisitive mind these historical references are treasures waiting to be (re)-discovered.

## 4.1 Big-endians and Little-endians

A fictional novel published in 1726 entitled *Gulliver's Travels Into Several Remote Nations of the World* contains a reference to a dispute between two nations, that of Lilliput and of Blefuscu. The dispute is over the most trivial of matters, namely which end of an egg should be broken prior to eating. Clearly a satire by Swift on the religious controversies of his day, the schism between the "Big-endians" and "Little-endians" served as a convenient reference for computer scientists to describe the differing ways in which digital data could be organized within a digital system.

Without further adieu, I present to you the passage from Swift's famous book introducing the term "Big-endian" into the English lexicon:

> . . . our histories of six thousand moons make no mention of any other regions than the two great empires of Lilliput and Blefuscu. Which two mighty powers have, as I was going to tell you, been engaged in a most obstinate war for six-and-thirty moons past. It began upon the following occasion. It is allowed on all hands, that the primitive way of breaking eggs, before we eat them, was upon the larger end; but his present majesty's grandfather, while he was a boy, going to eat an egg, and breaking it according to the ancient practice, happened to cut one of his fingers. Whereupon the emperor his father published an edict, commanding all his subjects, upon great penalties, to break the smaller end of their eggs. The people so highly resented this law, that our histories tell us, there have been six rebellions raised on that account; wherein one emperor lost his life, and another his crown. These civil commotions were constantly fomented by the monarchs of Blefuscu; and when they were quelled, the exiles always fled for refuge to that empire. It is computed that eleven thousand persons have at several times suffered death, rather than submit to break their eggs at the smaller end. Many hundred large volumes have been published upon this controversy: but the books of the Big-endians have been long forbidden, and the whole party rendered incapable by law of holding employments. During the course of these troubles, the emperors of Blefusca did frequently expostulate by their ambassadors, accusing us of making a schism in religion, by offending against a fundamental doctrine of our great prophet Lustrog, in the fifty-fourth chapter of the Blundecral (which is their Alcoran). This, however, is thought to be a mere strain upon the text; for the words are these: "that all true believers break their eggs at the convenient end." And which is the convenient end, seems, in my humble opinion to be left to every man's conscience, or at least in the power of the chief magistrate to determine. Now, the Big-endian exiles have found so much credit in the emperor of Blefuscu's court, and so much private assistance and encouragement from their party here at home, that a bloody war has been carried on between the two empires for six-and-thirty moons, with various success; during which time we have lost forty capital ships, and a much a greater number of smaller vessels, together with thirty thousand of our best seamen and soldiers; and the damage received by the enemy is reckoned to be somewhat greater than ours. However, they have now equipped a numerous fleet, and are just preparing to make a descent upon us; and his imperial majesty, placing great confidence in your valour and strength, has commanded me to lay this account of his affairs before you.

One of those computer scientists referencing Jonathan Swift's satirical novel was Danny Cohen, in a document appropriately dated on April Fool's Day (April 1), 1980. The tone of Cohen's document

is quite humorous, and definitely worth reading especially for those interested in the architectures of early computing hardware such as the Motorola 68000 microprocessor IC; Digital Equipment Corporation's PDP10, PDP11/45, VAX computers; and the IBM model 360 computer. He makes extensive reference of Swift's story while describing the fundamental decision of how to organize and communicate data words in a digital system.

Cohen's document concludes neatly with the following three sentences, which I include for your edification:

> It may be interesting to notice that the point which Jonathan Swift tried to convey in Gulliver's Travels in exactly the opposite of the point of this note.
>
> Swift's point is that the difference between breaking the egg at the little-end and breaking it at the big-end is trivial. Therefore, he suggests, that everyone does it in his own preferred way.
>
> We agree that the difference between sending eggs with the little- or the big-end first is trivial, but we insist that everyone must do it in the same way, to avoid anarchy. Since the difference is trivial we may choose either way, but a decision must be made.

At the time of this writing (2023), more than four decades after Cohen's missive, the state of digital anarchy he described remains alive and well, with little-endian and big-endian formats commonly found coexisting across digital data networks. While the problem of which bit to send first in a serial (i.e. one bit at a time) communication channel seems to be settled[1] within each of the various network standards (e.g. Ethernet, EIA/TIA-232, etc.), the problem of byte and word order for large data segments remains. It is not uncommon, for example, to find manufacturers of industrial data equipment arbitrarily using different 16-bit word orders for storing 32-bit binary numbers, so that when a 32-bit binary number is received by a digital device of different manufacture, swapping of word or byte orders may be necessary in order to preserve the meaning of that 32-bit number. This is particularly common in Modbus-based communication systems where the Modbus administrative data fields (e.g. address numbers, function codes, length descriptors, etc.) are all ordered big-endian, but the hardware-specific data words may be big-endian, little-endian, byte-swapped, word-swapped, or any combination thereof!

[1] For example, all manufacturers of EIA/TIA-232 serial data communication hardware have agreed to transmit the LSB first followed by bits of increasing order. Thus, we do not encounter anarchy when connecting one manufacturer's 232-compliant modem to another manufacturer's 232-compliant modem. Ditto for the interoperability of all Ethernet communication hardware. This is a Very Good Thing.

# Chapter 5

# Derivations and Technical References

This chapter is where you will find mathematical derivations too detailed to include in the tutorial, and/or tables and other technical reference material.

## 5.1 Modbus function command formats

Every Modbus data frame, whether ASCII or RTU mode, has a field designated for "data." For each Modbus function, the content of this "data" field follows a specific format. It is the purpose of this subsection to document the data formats required for common Modbus functions, both the "Query" message transmitted by the Modbus master device to a slave device, and the corresponding "Response" message transmitted back to the master device by the queried slave device.

Since each Modbus data frame is packaged in multiples of 8 bits (RTU), they are usually represented in text as individual bytes (two hexadecimal characters). For example, a 16-bit "word" of Modbus data such as `1100100101011011` would typically be documented as `C9 5B` with a deliberate space separating the "Hi" (`C9`) and "Lo" (`5B`) bytes. The fact that all Modbus words appear in this order means that Modbus follows the "big-endian" standard.

### 5.1.1 Function code 01 – Read Coil(s)

This Modbus function reads the statuses of slave device discrete outputs ("coils") within the slave device, returning those statuses in blocks of eight (even if the "number of coils" specified in the query is not a multiple of eight!). Relevant Modbus addresses for this function range from 00001 to 09999 (decimal) but the starting address is a hexadecimal number representing the $(n-1)^{th}$ register from the beginning of this range (e.g. absolute decimal address 00100 would be specified as hexadecimal `0x0063`, as a high-order byte `00` immediately followed by a low-order byte `63`).

Query message (Function code 01)

| Start | Slave address | Function code | Data | | | | Error check | End |
|---|---|---|---|---|---|---|---|---|
| | | | Starting address | | Number of coils | | | |
| | XX | 01 | Hi | Lo | Hi | Lo | XX | |
| *Start* | | | | | | | | *Stop* |

Response message (Function code 01)

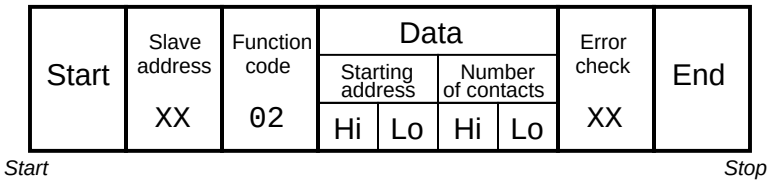
| Start | Slave address | Function code | Data | | | | Error check | End |
|---|---|---|---|---|---|---|---|---|
| | | | Number of bytes | First byte (8 coils) | Second byte (8 coils) | Third byte (8 coils) | | |
| | XX | 01 | | | | | XX | |
| *Start* | | | | | | | | *Stop* |

Note that the second and third bytes representing coil status are shown in grey, because their existence assumes more than one byte worth of coils has been requested in the query.

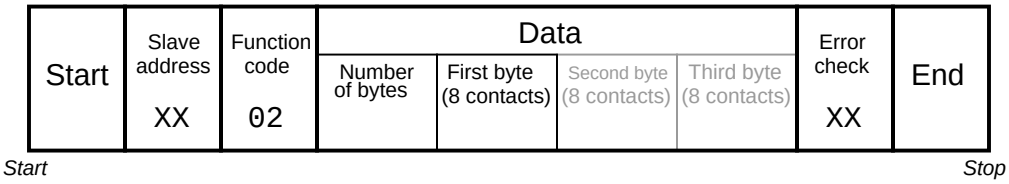
### 5.1.2 Function code 02 – Read Contact(s)

This Modbus function reads the statuses of slave device discrete inputs ("contacts") within the slave device, returning those statuses in blocks of eight (even if the "number of contacts" specified in the query is not a multiple of eight!). Relevant Modbus addresses for this function range from 10001 to 19999 (decimal), but the starting address is a hexadecimal number representing the $(n-1)^{th}$ register from the beginning of this range (e.g. absolute decimal address 10256 would be specified as hexadecimal `0x00FF`: the "Hi" byte being `00` and the "Lo" byte being `FF`).

Query message (Function code 02)




Response message (Function code 02)



### 5.1.3 Function code 03 – Read Holding Register(s)

This Modbus function reads the statuses of "holding" registers within the slave device, with the size of each register assumed to be two bytes (16 bits). Relevant Modbus addresses for this function range from 40001 to 49999 (decimal), but the starting address is a hexadecimal number representing the $(n-1)^{th}$ register from the beginning of this range (e.g. absolute decimal address 40980 would be specified as hexadecimal `0x03D3`, "Hi" byte `03` and "Lo" byte `D3`.).

Query message (Function code 03)

| Start | Slave address | Function code | Data | | | | Error check | End |
|---|---|---|---|---|---|---|---|---|
| | | | Starting address | | Number of registers | | | |
| | XX | 03 | Hi | Lo | Hi | Lo | XX | |
| *Start* | | | | | | | | *Stop* |

Response message (Function code 03)

| Start | Slave address | Function code | Data | | | | | | Error check | End |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Number of bytes | First register | | Second register | | Third register | | |
| | XX | 03 | | Hi | Lo | Hi | Lo | Hi | Lo | XX | |
| *Start* | | | | | | | | | | *Stop* |

Note that since the query message specifies the number of registers (each register being two bytes in size), and the response message replies with the number of *bytes*, the response message's "number of bytes" field will have a value twice that of the query message's "number of registers" field. Note also that the maximum number of registers which may be requested in the query message (65536) with "Hi" and "Lo" byte values grossly exceeds the number of bytes the response message can report (255) with its single byte value.

### 5.1.4 Function code 04 – Read Analog Input Register(s)

This Modbus function is virtually identical to 03 (Read Holding Registers) except that it reads "input" registers instead: addresses 30001 through 39999 (decimal). As with all the Modbus relative addresses, the starting address specified in both messages is a hexadecimal number representing the $(n-1)^{th}$ register from the beginning of this range (e.g. absolute decimal address 32893 would be specified as hexadecimal `0x0B4C`, "Hi" byte `0B` followed by "Lo" byte `4C`).

Query message (Function code 04)

| Start | Slave address | Function code | Data | | | | Error check | End |
|---|---|---|---|---|---|---|---|---|
| | | | Starting address | | Number of registers | | | |
| | XX | 04 | Hi | Lo | Hi | Lo | XX | |

*Start* *Stop*

Response message (Function code 04)

| Start | Slave address | Function code | Data | | | | | | Error check | End |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Number of bytes | First register | | Second register | | Third register | | |
| | XX | 04 | | Hi | Lo | Hi | Lo | Hi | Lo | XX | |

*Start* *Stop*

Note that since the query message specifies the number of registers (each register being two bytes in size), and the response message replies with the number of *bytes*, the response message's "number of bytes" field will have a value twice that of the query message's "number of registers" field. Note also that the maximum number of registers which may be requested in the query message (65536) with "Hi" and "Lo" byte values grossly exceeds the number of bytes the response message can report (255) with its single byte value.

### 5.1.5 Function code 05 – Write (Force) Single Coil

This Modbus function writes a single bit of data to a discrete output ("coil") within the slave device. Relevant Modbus addresses for this function range from 00001 to 09999 (decimal) but the starting address is a hexadecimal number representing the $(n-1)^{th}$ register from the beginning of this range (e.g. absolute decimal address 07200 would be specified as hexadecimal `0x1C1F`, the "Hi" byte being `1C` and the "Lo" byte being `1F`).

Query/Response message (Function code 05)

| Start | Slave address | Function code | Data | | | | Error check | End |
|---|---|---|---|---|---|---|---|---|
| | | | Coil address | | Force data | | | |
| | XX | 05 | Hi | Lo | Hi | Lo | XX | |

*Start* *Stop*

The "force data" for a single coil consists of either `00 00` (force coil off) or `FF 00` (force coil on). No other data values will suffice – anything other than `00 00` or `FF 00` will be ignored by the slave device.

A normal response message will be a simple echo (verbatim repeat) of the query message.

### 5.1.6 Function code 06 – Write (Preset) Single Holding Register

This Modbus function writes data to a single "holding" register within the slave device. Relevant Modbus addresses for this function range from 40001 to 49999 (decimal) but the starting address is a hexadecimal number representing the $(n-1)^{th}$ register from the beginning of this range (e.g. absolute decimal address 40034 would be specified as hexadecimal `0x0021`, "Hi" byte `00` followed by "Lo" byte `21`).

Query/Response message (Function code 06)

| Start | Slave address | Function code | Data | | | | Error check | End |
|---|---|---|---|---|---|---|---|---|
| | | | Register address | | Preset data | | | |
| | XX | 06 | Hi | Lo | Hi | Lo | XX | |

*Start* *Stop*

A normal response message will be a simple echo (verbatim repeat) of the query message.

### 5.1.7 Function code 15 – Write (Force) Multiple Coils

This Modbus function writes multiple bits of data to a set of discrete outputs ("coils") within the slave device. Relevant Modbus addresses for this function range from 00001 to 09999 (decimal) but the starting address is a hexadecimal number representing the $(n-1)^{th}$ register from the beginning of this range (e.g. absolute decimal address 03207 would be specified as hexadecimal `0x0C86`, as a "Hi" byte `0C` and a "Lo" byte `86`).

Query message (Function code 15)

| Start | Slave address | Function code | Data | | | | | | | | Error check | End |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Starting address | | Number of coils | | Number of bytes | Force data first word | | Force data second word | | |
| | XX | 0F | Hi | Lo | Hi | Lo | | Hi | Lo | Hi | Lo | XX |

*Start* *Stop*

Response message (Function code 15)

| Start | Slave address | Function code | Data | | | | Error check | End |
|---|---|---|---|---|---|---|---|---|
| | | | Starting address | | Number of coils | | | |
| | XX | 0F | Hi | Lo | Hi | Lo | XX | |

*Start* *Stop*

Note that the query message specifies both the number of coils (bits) and the number of bytes.

### 5.1.8 Function code 16 – Write (Preset) Multiple Holding Register

This Modbus function writes multiple words of data to a set of "holding" registers within the slave device. Relevant Modbus addresses for this function range from 40001 to 49999 (decimal) but the starting address is a hexadecimal number representing the $(n-1)^{th}$ register from the beginning of this range (e.g. absolute decimal address 47441 would be specified as hexadecimal `0x1D10`, a "Hi" byte of `1D` followed by a "Lo" byte of `10`).

Query message (Function code 16)

| Start | Slave address | Function code | Data | | | | | | | | Error check | End |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Starting address | | Number of registers | | Number of bytes | Preset data first register | | Preset data second register | | |
| | XX | 10 | Hi | Lo | Hi | Lo | | Hi | Lo | Hi | Lo | XX | |
| *Start* | | | | | | | | | | | | *Stop* |

Response message (Function code 16)

| Start | Slave address | Function code | Data | | | | Error check | End |
|---|---|---|---|---|---|---|---|---|
| | | | Starting address | | Number of registers | | | |
| | XX | 10 | Hi | Lo | Hi | Lo | XX | |
| *Start* | | | | | | | | *Stop* |

Note that the query message specifies both the number of registers (16-bit words) and the number of bytes, which is redundant (the number of bytes must *always* be twice the number of registers, given that each register is two bytes[1] in size). Note also that the maximum number of registers which may be requested in the query message (65536) with "Hi" and "Lo" byte values grossly exceeds the number of bytes the response message can report (255) with its single byte value.

[1] Even for devices where the register size is less than two bytes (e.g. Modicon M84 and 484 model controllers have 10 bits within each register), data is still addressed as two bytes' worth per register, with the leading bits simply set to zero to act as placeholders.

## 5.2 The OSI Reference Model

| | |
|---|---|
| Layer 7<br>**Application** | This is where digital data takes on practical meaning in the context of some human or overall system function.<br>*Examples: HTTP, FTP, HART, Modbus* |
| Layer 6<br>**Presentation** | This is where data gets converted between different formats.<br>*Examples: ASCII, EBCDIC, MPEG, JPG, MP3* |
| Layer 5<br>**Session** | This is where "conversations" between digital devices are opened, closed, and otherwise managed for reliable data flow.<br>*Examples: Sockets, NetBIOS* |
| Layer 4<br>**Transport** | This is where complete data transfer is handled, ensuring all data gets put together and error-checked before use.<br>*Examples: TCP, UDP* |
| Layer 3<br>**Network** | This is where the system determines network-wide addresses, ensuring a means for data to get from one node to another.<br>*Examples: IP, ARP* |
| Layer 2<br>**Data link** | This is where basic data transfer methods and sequences (frames) are defined within the smallest segment(s) of a network.<br>*Examples: CSMA/CD, Token passing, Master/Slave* |
| Layer 1<br>**Physical** | This is where data bits are equated to electrical, optical, or other signals. Other physical details such as cable and connector types are also specified here.<br>*Examples: EIA/TIA-232, 422, 485, Bell 202* |

# Chapter 6

# Questions

This learning module, along with all others in the ModEL collection, is designed to be used in an inverted instructional environment where students independently read[1] the tutorials and attempt to answer questions on their own *prior* to the instructor's interaction with them. In place of lecture[2], the instructor engages with students in Socratic-style dialogue, probing and challenging their understanding of the subject matter through inquiry.

Answers are not provided for questions within this chapter, and this is by design. Solved problems may be found in the Tutorial and Derivation chapters, instead. The goal here is *independence*, and this requires students to be challenged in ways where others cannot think for them. Remember that you always have the tools of *experimentation* and *computer simulation* (e.g. SPICE) to explore concepts!

The following lists contain ideas for Socratic-style questions and challenges. Upon inspection, one will notice a strong theme of *metacognition* within these statements: they are designed to foster a regular habit of examining one's own thoughts as a means toward clearer thinking. As such these sample questions are useful both for instructor-led discussions as well as for self-study.

[1] Technical reading is an essential academic skill for any technical practitioner to possess for the simple reason that the most comprehensive, accurate, and useful information to be found for developing technical competence is in textual form. Technical careers in general are characterized by the need for continuous learning to remain current with standards and technology, and therefore any technical practitioner who cannot read well is handicapped in their professional development. An excellent resource for educators on improving students' reading prowess through intentional effort and strategy is the book textitReading For Understanding – How Reading Apprenticeship Improves Disciplinary Learning in Secondary and College Classrooms by Ruth Schoenbach, Cynthia Greenleaf, and Lynn Murphy.

[2] Lecture is popular as a teaching method because it is easy to implement: any reasonably articulate subject matter expert can talk to students, even with little preparation. However, it is also quite problematic. A good lecture always makes complicated concepts seem easier than they are, which is bad for students because it instills a false sense of confidence in their own understanding; reading and re-articulation requires more cognitive effort and serves to verify comprehension. A culture of teaching-by-lecture fosters a debilitating dependence upon direct personal instruction, whereas the challenges of modern life demand independent and critical thought made possible only by gathering information and perspectives from afar. Information presented in a lecture is ephemeral, easily lost to failures of memory and dictation; text is forever, and may be referenced at any time.

General challenges following tutorial reading

- Summarize as much of the text as you can in one paragraph of your own words. A helpful strategy is to explain ideas as you would for an intelligent child: as simple as you can without compromising too much accuracy.
- Simplify a particular section of the text, for example a paragraph or even a single sentence, so as to capture the same fundamental idea in fewer words.
- Where did the text make the most sense to you? What was it about the text's presentation that made it clear?
- Identify where it might be easy for someone to misunderstand the text, and explain why you think it could be confusing.
- Identify any new concept(s) presented in the text, and explain in your own words.
- Identify any familiar concept(s) such as physical laws or principles applied or referenced in the text.
- Devise a proof of concept experiment demonstrating an important principle, physical law, or technical innovation represented in the text.
- Devise an experiment to disprove a plausible misconception.
- Did the text reveal any misconceptions you might have harbored? If so, describe the misconception(s) and the reason(s) why you now know them to be incorrect.
- Describe any useful problem-solving strategies applied in the text.
- Devise a question of your own to challenge a reader's comprehension of the text.

General follow-up challenges for assigned problems

- Identify where any fundamental laws or principles apply to the solution of this problem, especially before applying any mathematical techniques.
- Devise a thought experiment to explore the characteristics of the problem scenario, applying known laws and principles to mentally model its behavior.
- Describe in detail your own strategy for solving this problem. How did you identify and organized the given information? Did you sketch any diagrams to help frame the problem?
- Is there more than one way to solve this problem? Which method seems best to you?
- Show the work you did in solving this problem, even if the solution is incomplete or incorrect.
- What would you say was the most challenging part of this problem, and why was it so?
- Was any important information missing from the problem which you had to research or recall?
- Was there any extraneous information presented within this problem? If so, what was it and why did it not matter?
- Examine someone else's solution to identify where they applied fundamental laws or principles.
- Simplify the problem from its given form and show how to solve this simpler version of it. Examples include eliminating certain variables or conditions, altering values to simpler (usually whole) numbers, applying a limiting case (i.e. altering a variable to some extreme or ultimate value).
- For quantitative problems, identify the real-world meaning of all intermediate calculations: their units of measurement, where they fit into the scenario at hand. Annotate any diagrams or illustrations with these calculated values.
- For quantitative problems, try approaching it qualitatively instead, thinking in terms of "increase" and "decrease" rather than definite values.
- For qualitative problems, try approaching it quantitatively instead, proposing simple numerical values for the variables.
- Were there any assumptions you made while solving this problem? Would your solution change if one of those assumptions were altered?
- Identify where it would be easy for someone to go astray in attempting to solve this problem.
- Formulate your own problem based on what you learned solving this one.

General follow-up challenges for experiments or projects

- In what way(s) was this experiment or project easy to complete?
- Identify some of the challenges you faced in completing this experiment or project.

- Show how thorough documentation assisted in the completion of this experiment or project.
- Which fundamental laws or principles are key to this system's function?
- Identify any way(s) in which one might obtain false or otherwise misleading measurements from test equipment in this system.
- What will happen if (component $X$) fails (open/shorted/etc.)?
- What would have to occur to make this system unsafe?

## 6.1 Conceptual reasoning

These questions are designed to stimulate your analytic and synthetic thinking[3]. In a Socratic discussion with your instructor, the goal is for these questions to prompt an extended dialogue where assumptions are revealed, conclusions are tested, and understanding is sharpened. Your instructor may also pose additional questions based on those assigned, in order to further probe and refine your conceptual understanding.

Questions that follow are presented to challenge and probe your understanding of various concepts presented in the tutorial. These questions are intended to serve as a guide for the Socratic dialogue between yourself and the instructor. Your instructor's task is to ensure you have a sound grasp of these concepts, and the questions contained in this document are merely a means to this end. Your instructor may, at his or her discretion, alter or substitute questions for the benefit of tailoring the discussion to each student's needs. The only absolute requirement is that each student is challenged and assessed at a level equal to or greater than that represented by the documented questions.

It is far more important that you convey your *reasoning* than it is to simply convey a correct answer. For this reason, you should refrain from researching other information sources to answer questions. What matters here is that *you* are doing the thinking. If the answer is incorrect, your instructor will work with you to correct it through proper reasoning. A correct answer without an adequate explanation of how you derived that answer is unacceptable, as it does not aid the learning or assessment process.

You will note a conspicuous lack of answers given for these conceptual questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your conceptual answers, where applicable, is to use circuit simulation software to explore the effects of changes made to circuits. For example, if one of these conceptual questions challenges you to predict the effects of altering some component parameter in a circuit, you may check the validity of your work by simulating that same parameter change within software and seeing if the results agree.

[3] *Analytical* thinking involves the "disassembly" of an idea into its constituent parts, analogous to dissection. *Synthetic* thinking involves the "assembly" of a new idea comprised of multiple concepts, analogous to construction. Both activities are high-level cognitive skills, extremely important for effective problem-solving, necessitating frequent challenge and regular practice to fully develop.

### 6.1.1 Reading outline and reflections

*"Reading maketh a full man; conference a ready man; and writing an exact man"* – Francis Bacon

Francis Bacon's advice is a blueprint for effective education: reading provides the learner with knowledge, writing focuses the learner's thoughts, and critical dialogue equips the learner to confidently communicate and apply their learning. Independent acquisition and application of knowledge is a powerful skill, well worth the effort to cultivate. To this end, students should read these educational resources closely, journal their own reflections on the reading, and discuss in detail their findings with classmates and instructor(s). You should be able to do all of the following after reading any instructional text:

☑ Briefly SUMMARIZE THE TEXT in the form of a journal entry documenting your learning as you progress through the course of study. Share this summary in dialogue with your classmates and instructor. Journaling is an excellent self-test of thorough reading because you cannot clearly express what you have not read or did not comprehend.

☑ Demonstrate ACTIVE READING STRATEGIES, including verbalizing your impressions as you read, simplifying long passages to convey the same ideas using fewer words, annotating text and illustrations with your own interpretations, working through mathematical examples shown in the text, cross-referencing passages with relevant illustrations and/or other passages, identifying problem-solving strategies applied by the author, etc. Technical reading is a special case of problem-solving, and so these strategies work precisely because they help solve any problem: paying attention to your own thoughts (metacognition), eliminating unnecessary complexities, identifying what makes sense, paying close attention to details, drawing connections between separated facts, and noting the successful strategies of others.

☑ Identify IMPORTANT THEMES, especially GENERAL LAWS and PRINCIPLES, expounded in the text and express them in the simplest of terms as though you were teaching an intelligent child. This emphasizes connections between related topics and develops your ability to communicate complex ideas to anyone.

☑ Form YOUR OWN QUESTIONS based on the reading, and then pose them to your instructor and classmates for their consideration. Anticipate both correct and incorrect answers, the incorrect answer(s) assuming one or more plausible misconceptions. This helps you view the subject from different perspectives to grasp it more fully.

☑ Devise EXPERIMENTS to test claims presented in the reading, or to disprove misconceptions. Predict possible outcomes of these experiments, and evaluate their meanings: what result(s) would confirm, and what would constitute disproof? Running mental simulations and evaluating results is essential to scientific and diagnostic reasoning.

☑ Specifically identify any points you found CONFUSING. The reason for doing this is to help diagnose misconceptions and overcome barriers to learning.

### 6.1.2 Foundational concepts

Correct analysis and diagnosis of electric circuits begins with a proper understanding of some basic concepts. The following is a list of some important concepts referenced in this module's full tutorial. Define each of them in your own words, and be prepared to illustrate each of these concepts with a description of a practical example and/or a live demonstration.

Programmable Logic Controller (PLC)

Variable Frequency Drive (VFD)

Analog signal

Discrete signal

Serial versus Parallel data

Master-slave arbitration

Data frame

OSI model

Memory address

Frame check sequence

ASCII

Encapsulation

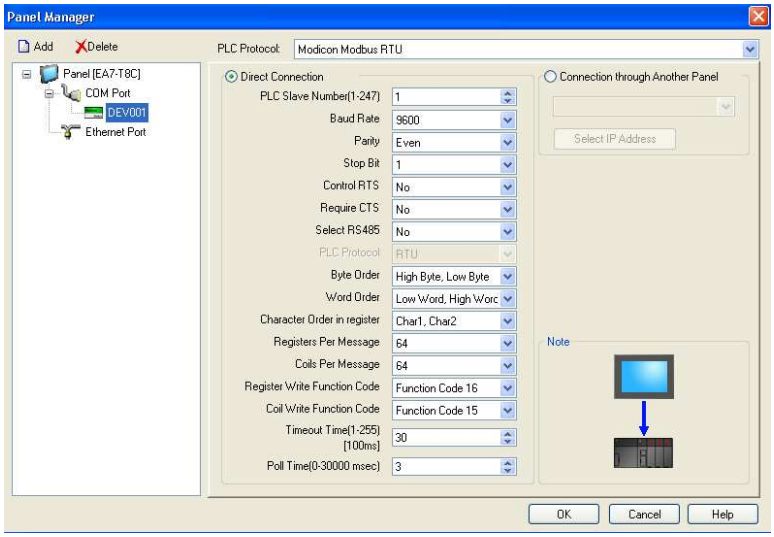
Ethernet

Internet Protocol (IP)

Transmission Control Protocol (TCP)

### 6.1.3 Display panel configuration

The following screen capture shows the configuration window for an industrial display panel called an *HMI* (Human-Machine Interface). This particular configuration window sets communication parameters for it to exchange data with a Modbus device:




Identify the purpose for the "Control RTS" and "Require CTS" parameters, both of which happen to be de-activated (set to "No").

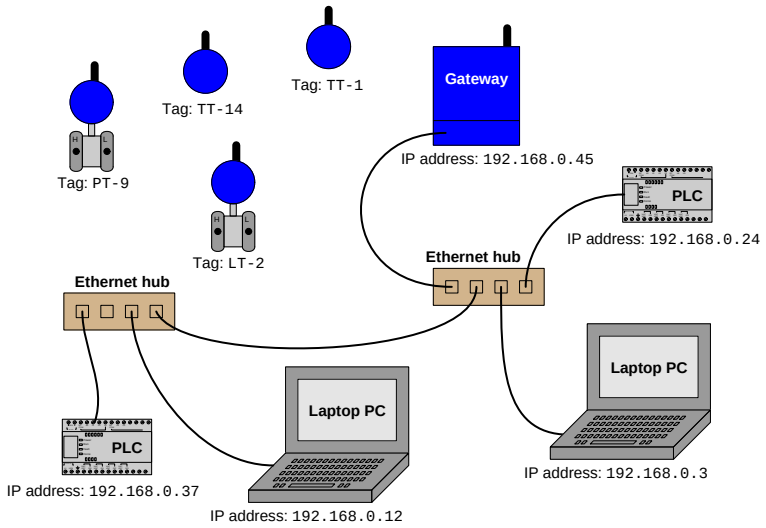
Two options exist for Modbus Register Write function codes: 06 and 16. Likewise, two options exist for Modbus Coil Write function codes: 05 and 15. Explain the difference between each option, and why one setting might be more useful than the other.

Challenges

- Can we tell what type of physical-layer network this panel will use?
- Is the byte order little-endian or big-endian?
- Is the word order little-endian or big-endian?

### 6.1.4 Wireless gateway system

An industrial radio protocol called *Wireless*HART exchanges digital data between wired and wireless sensor devices using a *gateway*. This "gateway" acts as a radio transceiver and also a Modbus slave device for any wired computers seeking data from it. The following diagram shows how two PLCs and two laptop computers may connect to one of these gateways via Ethernet, which then allows data to be exchanged between these devices and the radio-based sensors (each sensor having its own unique "tag" name):




Suppose you are tasked with building an industrial measurement system using several *Wireless*HART sensors. The central component of the wireless network, of course, is the *gateway* device. Your particular network gateway provides Modbus access to the data received from all those transmitters via an RS-485 (serial) network port.

The challenge is this: five programmable logic controllers (PLCs) require access to the measurement variables of five different *Wireless*HART sensors. Each PLC has an RS-485 serial port, meaning they may be all "daisy-chained" on one RS-485 wired network connecting also to the wireless network gateway device. What is not so easy is figuring out how all five of the PLCs will be able to read data from the gateway, since Modbus is fundamentally a *master/slave* protocol (one "master" device sending data to and receiving data from multiple "slave" devices). In our system, we need five different PLCs to get data from the one gateway, which itself is a "slave" device.

Explain how it is possible to configure a single PLC as the Modbus "master" device, and still have the other four PLCs receive data from the wireless network gateway ("slave") device.

Determine whether or not it is possible to configure *all* five of the PLCs as Modbus "master" devices on the RS-485 network, yet avoid "bus contention" issues as they all attempt to query the wireless gateway ("slave") device. Explain why this proposed solution is (or is not!) possible!

Challenges

- Which of these devices may be queried using the `ping` utility, and which of these cannot?

## 6.2 Quantitative reasoning

These questions are designed to stimulate your computational thinking. In a Socratic discussion with your instructor, the goal is for these questions to reveal your mathematical approach(es) to problem-solving so that good technique and sound reasoning may be reinforced. Your instructor may also pose additional questions based on those assigned, in order to observe your problem-solving firsthand.

Mental arithmetic and estimations are strongly encouraged for all calculations, because without these abilities you will be unable to readily detect errors caused by calculator misuse (e.g. keystroke errors).

You will note a conspicuous lack of answers given for these quantitative questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. My advice is to use circuit simulation software such as SPICE to check the correctness of quantitative answers. Refer to those learning modules within this collection focusing on SPICE to see worked examples which you may use directly as practice problems for your own study, and/or as templates you may modify to run your own analyses and generate your own practice problems.

Completely worked example problems found in the Tutorial may also serve as "test cases[4]" for gaining proficiency in the use of circuit simulation software, and then once that proficiency is gained you will never need to rely[5] on an answer key!

[4] In other words, set up the circuit simulation software to analyze the same circuit examples found in the Tutorial. If the simulated results match the answers shown in the Tutorial, it confirms the simulation has properly run. If the simulated results disagree with the Tutorial's answers, something has been set up incorrectly in the simulation software. Using every Tutorial as practice in this way will quickly develop proficiency in the use of circuit simulation software.

[5] This approach is perfectly in keeping with the instructional philosophy of these learning modules: *teaching students to be self-sufficient thinkers*. Answer keys can be useful, but it is even more useful to your long-term success to have a set of tools on hand for checking your own work, because once you have left school and are on your own, there will no longer be "answer keys" available for the problems you will have to solve.

### 6.2.1 Miscellaneous physical constants

Note: constants shown in **bold** type are *exact*, not approximations. Values inside of parentheses show one standard deviation ($\sigma$) of uncertainty in the final digits: for example, the magnetic permeability of free space value given as $1.25663706212(19) \times 10^{-6}$ H/m represents a center value (i.e. the location parameter) of $1.25663706212 \times 10^{-6}$ Henrys per meter with one standard deviation of uncertainty equal to $0.0000000000019 \times 10^{-6}$ Henrys per meter.

Avogadro's number ($N_A$) = **6.02214076** $\times 10^{23}$ **per mole** ($mol^{-1}$)

Boltzmann's constant ($k$) = **1.380649** $\times 10^{-23}$ **Joules per Kelvin** (J/K)

Electronic charge ($e$) = **1.602176634** $\times 10^{-19}$ **Coulomb** (C)

Faraday constant ($F$) = **96,485.33212...** $\times 10^{4}$ **Coulombs per mole** (C/mol)

Magnetic permeability of free space ($\mu_0$) = $1.25663706212(19) \times 10^{-6}$ Henrys per meter (H/m)

Electric permittivity of free space ($\epsilon_0$) = $8.8541878128(13) \times 10^{-12}$ Farads per meter (F/m)

Characteristic impedance of free space ($Z_0$) = 376.730313668(57) Ohms ($\Omega$)

Gravitational constant ($G$) = $6.67430(15) \times 10^{-11}$ cubic meters per kilogram-seconds squared ($m^3/kg\text{-}s^2$)

Molar gas constant ($R$) = **8.314462618... Joules per mole-Kelvin** (J/mol-K) = 0.08205746(14) liters-atmospheres per mole-Kelvin

Planck constant ($h$) = **6.62607015** $\times 10^{-34}$ **joule-seconds** (J-s)

Stefan-Boltzmann constant ($\sigma$) = **5.670374419...** $\times 10^{-8}$ **Watts per square meter-Kelvin**$^4$ ($W/m^2 \cdot K^4$)

Speed of light in a vacuum ($c$) = **299,792,458 meters per second** (m/s) = 186282.4 miles per second (mi/s)

Note: All constants taken from NIST data "Fundamental Physical Constants – Complete Listing", from `http://physics.nist.gov/constants`, National Institute of Standards and Technology (NIST), 2018 CODATA Adjustment.

### 6.2.2 Introduction to spreadsheets

A powerful computational tool you are encouraged to use in your work is a *spreadsheet*. Available on most personal computers (e.g. Microsoft Excel), *spreadsheet* software performs numerical calculations based on number values and formulae entered into cells of a grid. This grid is typically arranged as lettered columns and numbered rows, with each cell of the grid identified by its column/row coordinates (e.g. cell `B3`, cell `A8`). Each cell may contain a string of text, a number value, or a mathematical formula. The spreadsheet automatically updates the results of all mathematical formulae whenever the entered number values are changed. This means it is possible to set up a spreadsheet to perform a series of calculations on entered data, and those calculations will be re-done by the computer any time the data points are edited in any way.

For example, the following spreadsheet calculates average speed based on entered values of distance traveled and time elapsed:

| | A | B | C | D |
|---|---|---|---|---|
| 1 | Distance traveled | 46.9 | Kilometers | |
| 2 | Time elapsed | 1.18 | Hours | |
| 3 | Average speed | = B1 / B2 | km/h | |
| 4 | | | | |
| 5 | | | | |

Text labels contained in cells `A1` through `A3` and cells `C1` through `C3` exist solely for readability and are not involved in any calculations. Cell `B1` contains a sample distance value while cell `B2` contains a sample time value. The formula for computing speed is contained in cell `B3`. Note how this formula begins with an "equals" symbol (`=`), references the values for distance and speed by lettered column and numbered row coordinates (`B1` and `B2`), and uses a forward slash symbol for division (`/`). The coordinates `B1` and `B2` function as *variables*[6] would in an algebraic formula.

When this spreadsheet is executed, the numerical value `39.74576` will appear in cell `B3` rather than the formula `= B1 / B2`, because 39.74576 is the computed speed value given 46.9 kilometers traveled over a period of 1.18 hours. If a different numerical value for distance is entered into cell `B1` or a different value for time is entered into cell `B2`, cell `B3`'s value will automatically update. All you need to do is set up the given values and any formulae into the spreadsheet, and the computer will do all the calculations for you.

Cell `B3` may be referenced by other formulae in the spreadsheet if desired, since it is a variable just like the given values contained in `B1` and `B2`. This means it is possible to set up an entire chain of calculations, one dependent on the result of another, in order to arrive at a final value. The arrangement of the given data and formulae need not follow any pattern on the grid, which means you may place them anywhere.

[6] Spreadsheets may also provide means to attach text labels to cells for use as variable names (Microsoft Excel simply calls these labels "names"), but for simple spreadsheets such as those shown here it's usually easier just to use the standard coordinate naming for each cell.

Common[7] arithmetic operations available for your use in a spreadsheet include the following:

- Addition (`+`)
- Subtraction (`-`)
- Multiplication (`*`)
- Division (`/`)
- Powers (`^`)
- Square roots (`sqrt()`)
- Logarithms (`ln()` , `log10()`)

Parentheses may be used to ensure[8] proper order of operations within a complex formula. Consider this example of a spreadsheet implementing the *quadratic formula*, used to solve for roots of a polynomial expression in the form of $ax^2 + bx + c$:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

| | A | B |
|---|---|---|
| 1 | x_1 | = (-B4 + sqrt((B4^2) - (4*B3*B5))) / (2*B3) |
| 2 | x_2 | = (-B4 - sqrt((B4^2) - (4*B3*B5))) / (2*B3) |
| 3 | a = | 9 |
| 4 | b = | 5 |
| 5 | c = | -2 |

This example is configured to compute roots[9] of the polynomial $9x^2 + 5x - 2$ because the values of 9, 5, and $-2$ have been inserted into cells `B3`, `B4`, and `B5`, respectively. Once this spreadsheet has been built, though, it may be used to calculate the roots of *any* second-degree polynomial expression simply by entering the new $a$, $b$, and $c$ coefficients into cells `B3` through `B5`. The numerical values appearing in cells `B1` and `B2` will be automatically updated by the computer immediately following any changes made to the coefficients.

[7] Modern spreadsheet software offers a bewildering array of mathematical functions you may use in your computations. I recommend you consult the documentation for your particular spreadsheet for information on operations other than those listed here.

[8] Spreadsheet programs, like text-based programming languages, are designed to follow standard order of operations by default. However, my personal preference is to use parentheses even where strictly unnecessary just to make it clear to any other person viewing the formula what the intended order of operations is.

[9] Reviewing some algebra here, a *root* is a value for $x$ that yields an overall value of zero for the polynomial. For this polynomial ($9x^2 + 5x - 2$) the two roots happen to be $x = 0.269381$ and $x = -0.82494$, with these values displayed in cells `B1` and `B2`, respectively upon execution of the spreadsheet.

Alternatively, one could break up the long quadratic formula into smaller pieces like this:

$$y = \sqrt{b^2 - 4ac} \qquad z = 2a$$

$$x = \frac{-b \pm y}{z}$$

| | A | B | C |
|---|---|---|---|
| 1 | x_1 | = (-B4 + C1) / C2 | = sqrt((B4^2) - (4*B3*B5)) |
| 2 | x_2 | = (-B4 - C1) / C2 | = 2*B3 |
| 3 | a = | 9 | |
| 4 | b = | 5 | |
| 5 | c = | -2 | |

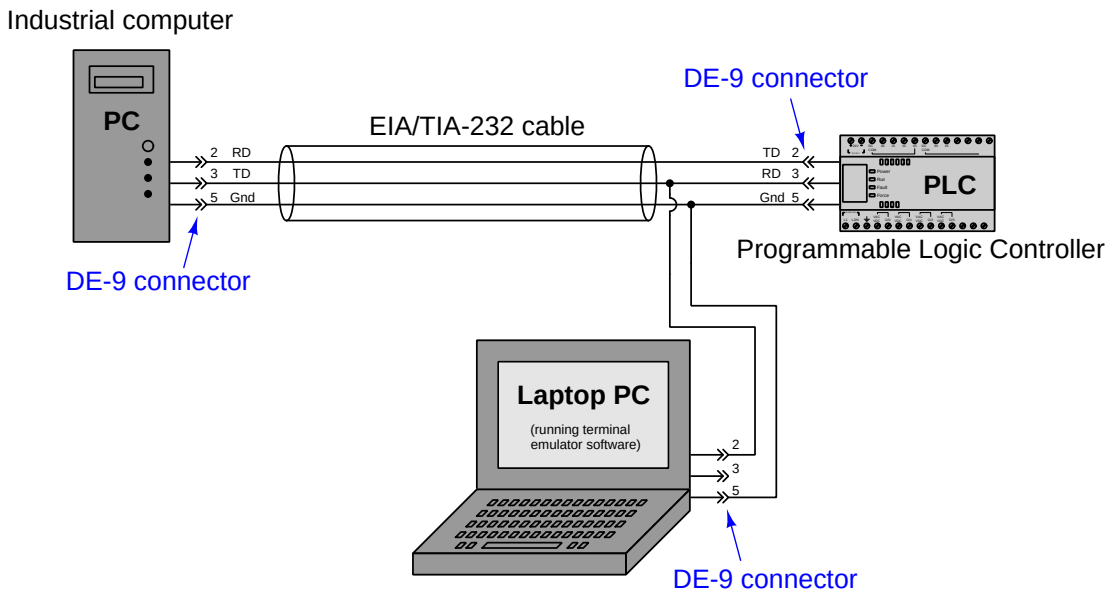
Note how the square-root term ($y$) is calculated in cell C1, and the denominator term ($z$) in cell C2. This makes the two final formulae (in cells B1 and B2) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary[10] – all that matters is that they properly reference each other in the formulae.

Spreadsheets are particularly useful for situations where the same set of calculations representing a circuit or other system must be repeated for different initial conditions. The power of a spreadsheet is that it automates what would otherwise be a tedious set of calculations. One specific application of this is to simulate the effects of various components within a circuit failing with abnormal values (e.g. a shorted resistor simulated by making its value nearly zero; an open resistor simulated by making its value extremely large). Another application is analyzing the behavior of a circuit design given new components that are out of specification, and/or aging components experiencing drift over time.

[10] My personal preference is to locate all the "given" data in the upper-left cells of the spreadsheet grid (each data point flanked by a sensible name in the cell to the left and units of measurement in the cell to the right as illustrated in the first distance/time spreadsheet example), sometimes coloring them in order to clearly distinguish which cells contain entered data versus which cells contain computed results from formulae. I like to place all formulae in cells below the given data, and try to arrange them in logical order so that anyone examining my spreadsheet will be able to figure out *how* I constructed a solution. This is a general principle I believe all computer programmers should follow: *document and arrange your code to make it easy for other people to learn from it.*

### 6.2.3 Interpreting an ASCII message frame

Using a serial terminal connected to an EIA/TIA-232 communication cable, you are able to intercept Modbus ASCII messages and display them as text:




Suppose you receive the following message:

```
:05030A5D000889
```

Identify the *slave address*, *function code*, *data*, and *LRC check* portions of this Modbus ASCII message.

Then, based on the function code, determine what the message is requesting from or sending to the recipient.

Challenges

- Why is pin 3 of the laptop computer's EIA/TIA-232 serial port left unconnected, while the two Modbus devices use all three pins?

### 6.2.4 Modbus ASCII message exchange

The following computer screenshot shows Modbus ASCII test software used to transmit a Modbus ASCII message to a slave device[11] and also show the received response from that slave device:


Explain the meanings of both the sent data and the received data.

Challenges

- Identify a practical use for this kind of test software.

[11] In this particular case, an Automation Direct SOLO model temperature controller.

## 6.3 Diagnostic reasoning

These questions are designed to stimulate your deductive and inductive thinking, where you must apply general principles to specific scenarios (deductive) and also derive conclusions about the failed circuit from specific details (inductive). In a Socratic discussion with your instructor, the goal is for these questions to reinforce your recall and use of general circuit principles and also challenge your ability to integrate multiple symptoms into a sensible explanation of what's wrong in a circuit. Your instructor may also pose additional questions based on those assigned, in order to further challenge and sharpen your diagnostic abilities.

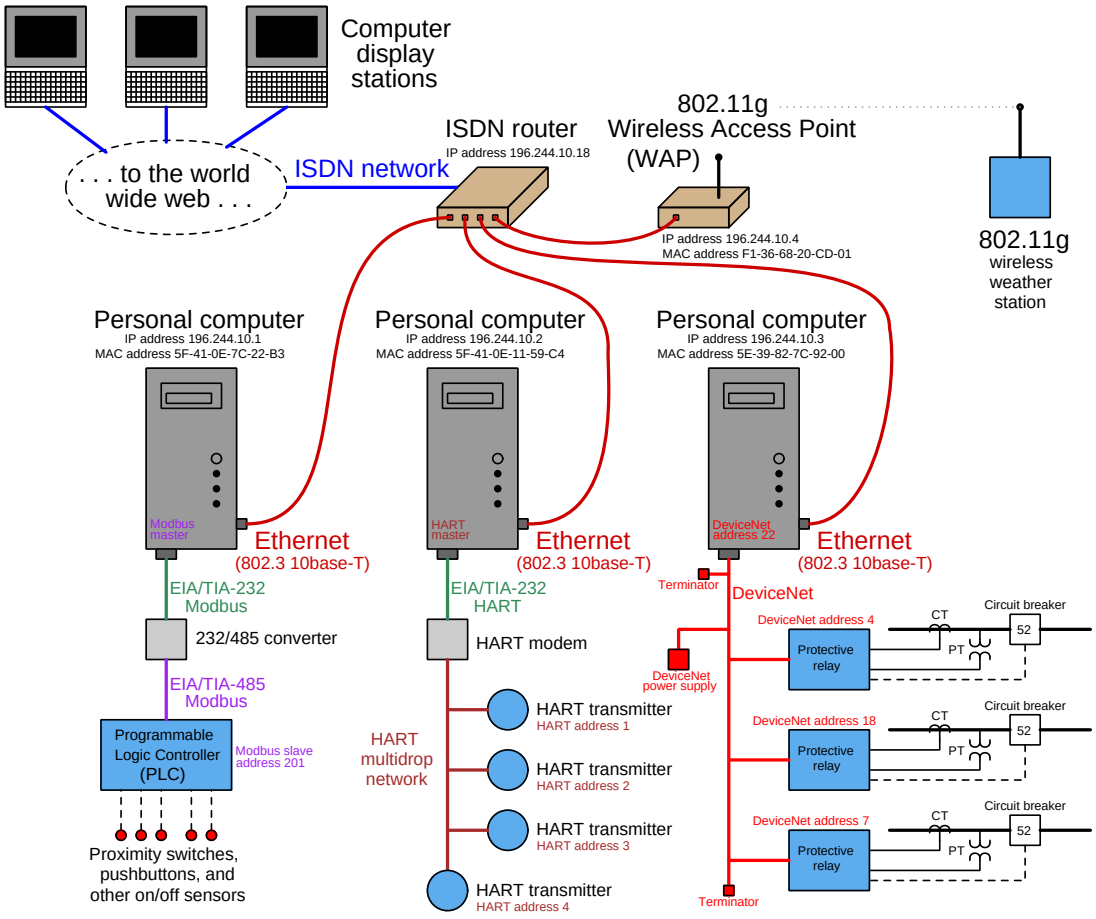
As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of the relevant principles and analytical processes in order to ensure you have a strong foundation upon which to build further understanding.

You will note a conspicuous lack of answers given for these diagnostic questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your diagnostic answers, where applicable, is to use circuit simulation software to explore the effects of faults placed in circuits. For example, if one of these diagnostic questions requires that you predict the effect of an open or a short in a circuit, you may check the validity of your work by simulating that same fault (substituting a very high resistance in place of that component for an open, and substituting a very low resistance for a short) within software and seeing if the results agree.

### 6.3.1 SCADA system fault

Suppose the industrial SCADA (Supervisory Control And Data Acquisition) system shown below develops a fault, preventing sensor data from the PLC from being viewed by internet display stations (upper left portion of diagram). The sensor data appearing on these display stations is "frozen" at old values and not updating over time as they should:




Diagnosing this problem, you notice that both the *Tx* and *Rx* LEDs on the 232/485 converter blink at regular intervals. Knowing this, identify possible fault locations preventing live sensor data from being viewed on the display stations.

Also identify components within this system that you believe to be functioning properly.

Finally, identify diagnostic steps you could take to isolate the nature and/or location of the fault.

Challenges

- Does the PLC communicate with Modbus serial or Modbus/TCP, or is this even possible to discern?
- Does the PLC communicate with Modbus RTU or Modbus ASCII, or is this even possible to discern?
- Would it be worthwhile to attempt to ping the PLC?

# Appendix A

# Problem-Solving Strategies

The ability to solve complex problems is arguably one of the most valuable skills one can possess, and this skill is particularly important in any science-based discipline.

- Study principles, not procedures. Don't be satisfied with merely knowing how to compute solutions – learn *why* those solutions work.
- Identify what it is you need to solve, identify all relevant data, identify all units of measurement, identify any general principles or formulae linking the given information to the solution, and then identify any "missing pieces" to a solution. Annotate all diagrams with this data.
- Sketch a diagram to help visualize the problem. When building a real system, always devise a plan for that system and analyze its function *before* constructing it.
- Follow the units of measurement and meaning of every calculation. If you are ever performing mathematical calculations as part of a problem-solving procedure, and you find yourself unable to apply each and every intermediate result to some aspect of the problem, it means you don't understand what you are doing. Properly done, every mathematical result should have practical meaning for the problem, and not just be an abstract number. You should be able to identify the proper units of measurement for each and every calculated result, and show where that result fits into the problem.
- Perform "thought experiments" to explore the effects of different conditions for theoretical problems. When troubleshooting real systems, perform *diagnostic tests* rather than visually inspecting for faults, the best diagnostic test being the one giving you the most information about the nature and/or location of the fault with the fewest steps.
- Simplify the problem until the solution becomes obvious, and then use that obvious case as a model to follow in solving the more complex version of the problem.
- Check for exceptions to see if your solution is incorrect or incomplete. A good solution will work for *all* known conditions and criteria. A good example of this is the process of testing scientific hypotheses: the task of a scientist is not to find support for a new idea, but rather to *challenge* that new idea to see if it holds up under a battery of tests. The philosophical

principle of *reductio ad absurdum* (i.e. disproving a general idea by finding a specific case where it fails) is useful here.

- Work "backward" from a hypothetical solution to a new set of given conditions.
- Add quantities to problems that are qualitative in nature, because sometimes a little math helps illuminate the scenario.
- Sketch graphs illustrating how variables relate to each other. These may be quantitative (i.e. with realistic number values) or qualitative (i.e. simply showing increases and decreases).
- Treat quantitative problems as qualitative in order to discern the relative magnitudes and/or directions of change of the relevant variables. For example, try determining what happens if a certain variable were to increase or decrease before attempting to precisely calculate quantities: how will each of the dependent variables respond, by increasing, decreasing, or remaining the same as before?
- Consider limiting cases. This works especially well for qualitative problems where you need to determine which direction a variable will change. Take the given condition and magnify that condition to an extreme degree as a way of simplifying the direction of the system's response.
- Check your work. This means regularly testing your conclusions to see if they make sense. This does *not* mean repeating the same steps originally used to obtain the conclusion(s), but rather to use some other means to check validity. Simply repeating procedures often leads to *repeating the same errors* if any were made, which is why alternative paths are better.

# Appendix B

# Instructional philosophy

*"The unexamined circuit is not worth energizing"* – Socrates (if he had taught electricity)

These learning modules, although useful for self-study, were designed to be used in a formal learning environment where a subject-matter expert challenges students to digest the content and exercise their critical thinking abilities in the answering of questions and in the construction and testing of working circuits.

The following principles inform the instructional and assessment philosophies embodied in these learning modules:

- The first goal of education is to enhance clear and independent thought, in order that every student reach their fullest potential in a highly complex and inter-dependent world. Robust reasoning is *always* more important than particulars of any subject matter, because its application is universal.
- Literacy is fundamental to independent learning and thought because text continues to be the most efficient way to communicate complex ideas over space and time. Those who cannot read with ease are limited in their ability to acquire knowledge and perspective.
- Articulate communication is fundamental to work that is complex and interdisciplinary.
- Faulty assumptions and poor reasoning are best corrected through challenge, not presentation. The rhetorical technique of *reductio ad absurdum* (disproving an assertion by exposing an absurdity) works well to discipline student's minds, not only to correct the problem at hand but also to learn how to detect and correct future errors.
- Important principles should be repeatedly explored and widely applied throughout a course of study, not only to reinforce their importance and help ensure their mastery, but also to showcase the interconnectedness and utility of knowledge.

These learning modules were expressly designed to be used in an "inverted" teaching environment[1] where students first read the introductory and tutorial chapters on their own, then individually attempt to answer the questions and construct working circuits according to the experiment and project guidelines. The instructor never lectures, but instead meets regularly with each individual student to review their progress, answer questions, identify misconceptions, and challenge the student to new depths of understanding through further questioning. Regular meetings between instructor and student should resemble a Socratic[2] dialogue, where questions serve as scalpels to dissect topics and expose assumptions. The student passes each module only after consistently demonstrating their ability to logically analyze and correctly apply all major concepts in each question or project/experiment. The instructor must be vigilant in probing each student's understanding to ensure they are truly *reasoning* and not just *memorizing*. This is why "Challenge" points appear throughout, as prompts for students to think deeper about topics and as starting points for instructor queries. Sometimes these challenge points require additional knowledge that hasn't been covered in the series to answer in full. This is okay, as the major purpose of the Challenges is to stimulate analysis and synthesis on the part of each student.

The instructor must possess enough mastery of the subject matter and awareness of students' reasoning to generate their own follow-up questions to practically any student response. Even completely correct answers given by the student should be challenged by the instructor for the purpose of having students practice articulating their thoughts and defending their reasoning. Conceptual errors committed by the student should be exposed and corrected not by direct instruction, but rather by reducing the errors to an absurdity[3] through well-chosen questions and thought experiments posed by the instructor. Becoming proficient at this style of instruction requires time and dedication, but the positive effects on critical thinking for both student and instructor are spectacular.

An inspection of these learning modules reveals certain unique characteristics. One of these is a bias toward thorough explanations in the tutorial chapters. Without a live instructor to explain concepts and applications to students, the text itself must fulfill this role. This philosophy results in lengthier explanations than what you might typically find in a textbook, each step of the reasoning process fully explained, including footnotes addressing common questions and concerns students raise while learning these concepts. Each tutorial seeks to not only explain each major concept in sufficient detail, but also to explain the logic of each concept and how each may be developed

[1] In a traditional teaching environment, students first encounter new information via *lecture* from an expert, and then independently apply that information via *homework*. In an "inverted" course of study, students first encounter new information via *homework*, and then independently apply that information under the scrutiny of an expert. The expert's role in lecture is to simply *explain*, but the expert's role in an inverted session is to *challenge*, *critique*, and if necessary *explain* where gaps in understanding still exist.

[2] Socrates is a figure in ancient Greek philosophy famous for his unflinching style of questioning. Although he authored no texts, he appears as a character in Plato's many writings. The essence of Socratic philosophy is to leave no question unexamined and no point of view unchallenged. While purists may argue a topic such as electric circuits is too narrow for a true Socratic-style dialogue, I would argue that the essential thought processes involved with scientific reasoning on *any* topic are not far removed from the Socratic ideal, and that students of electricity and electronics would do very well to challenge assumptions, pose thought experiments, identify fallacies, and otherwise employ the arsenal of critical thinking skills modeled by Socrates.

[3] This rhetorical technique is known by the Latin phrase *reductio ad absurdum*. The concept is to expose errors by counter-example, since only one solid counter-example is necessary to disprove a universal claim. As an example of this, consider the common misconception among beginning students of electricity that voltage cannot exist without current. One way to apply *reductio ad absurdum* to this statement is to ask how much current passes through a fully-charged battery connected to nothing (i.e. a clear example of voltage existing without current).

from "first principles". Again, this reflects the goal of developing clear and independent thought in students' minds, by showing how clear and logical thought was used to forge each concept. Students benefit from witnessing a model of clear thinking in action, and these tutorials strive to be just that.

Another characteristic of these learning modules is a lack of step-by-step instructions in the Project and Experiment chapters. Unlike many modern workbooks and laboratory guides where step-by-step instructions are prescribed for each experiment, these modules take the approach that students must learn to closely read the tutorials and apply their own reasoning to identify the appropriate experimental steps. Sometimes these steps are plainly declared in the text, just not as a set of enumerated points. At other times certain steps are implied, an example being assumed competence in test equipment use where the student should not need to be told *again* how to use their multimeter because that was thoroughly explained in previous lessons. In some circumstances no steps are given at all, leaving the entire procedure up to the student.

This lack of prescription is not a flaw, but rather a feature. Close reading and clear thinking are foundational principles of this learning series, and in keeping with this philosophy all activities are designed to *require* those behaviors. Some students may find the lack of prescription frustrating, because it demands more from them than what their previous educational experiences required. This frustration should be interpreted as an unfamiliarity with autonomous thinking, a problem which must be corrected if the student is ever to become a self-directed learner and effective problem-solver. Ultimately, the need for students to read closely and think clearly is more important both in the near-term and far-term than any specific facet of the subject matter at hand. If a student takes longer than expected to complete a module because they are forced to outline, digest, and reason on their own, so be it. The future gains enjoyed by developing this mental discipline will be well worth the additional effort and delay.

Another feature of these learning modules is that they do not treat topics in isolation. Rather, important concepts are introduced early in the series, and appear repeatedly as stepping-stones toward other concepts in subsequent modules. This helps to avoid the "compartmentalization" of knowledge, demonstrating the inter-connectedness of concepts and simultaneously reinforcing them. Each module is fairly complete in itself, reserving the beginning of its tutorial to a review of foundational concepts.

This methodology of assigning text-based modules to students for digestion and then using Socratic dialogue to assess progress and hone students' thinking was developed over a period of several years by the author with his Electronics and Instrumentation students at the two-year college level. While decidedly unconventional and sometimes even unsettling for students accustomed to a more passive lecture environment, this instructional philosophy has proven its ability to convey conceptual mastery, foster careful analysis, and enhance employability so much better than lecture that the author refuses to ever teach by lecture again.

Problems which often go undiagnosed in a lecture environment are laid bare in this "inverted" format where students must articulate and logically defend their reasoning. This, too, may be unsettling for students accustomed to lecture sessions where the instructor cannot tell for sure who comprehends and who does not, and this vulnerability necessitates sensitivity on the part of the "inverted" session instructor in order that students never feel discouraged by having their errors exposed. *Everyone* makes mistakes from time to time, and learning is a lifelong process! Part of the instructor's job is to build a culture of learning among the students where errors are not seen as shameful, but rather as opportunities for progress.

To this end, instructors managing courses based on these modules should adhere to the following principles:

- Student questions are always welcome and demand thorough, honest answers. The only type of question an instructor should refuse to answer is one the student should be able to easily answer on their own. Remember, *the fundamental goal of education is for each student to learn to think clearly and independently.* This requires hard work on the part of the student, which no instructor should ever circumvent. Anything done to bypass the student's responsibility to do that hard work ultimately limits that student's potential and thereby does real harm.

- It is not only permissible, but encouraged, to answer a student's question by asking questions in return, these follow-up questions designed to guide the student to reach a correct answer through their own reasoning.

- All student answers demand to be challenged by the instructor and/or by other students. This includes both correct and incorrect answers – the goal is to practice the articulation and defense of one's own reasoning.

- No reading assignment is deemed complete unless and until the student demonstrates their ability to accurately summarize the major points in their own terms. Recitation of the original text is unacceptable. This is why every module contains an "Outline and reflections" question as well as a "Foundational concepts" question in the Conceptual reasoning section, to prompt reflective reading.

- No assigned question is deemed answered unless and until the student demonstrates their ability to consistently and correctly apply the concepts to *variations* of that question. This is why module questions typically contain multiple "Challenges" suggesting different applications of the concept(s) as well as variations on the same theme(s). Instructors are encouraged to devise as many of their own "Challenges" as they are able, in order to have a multitude of ways ready to probe students' understanding.

- No assigned experiment or project is deemed complete unless and until the student demonstrates the task in action. If this cannot be done "live" before the instructor, video-recordings showing the demonstration are acceptable. All relevant safety precautions must be followed, all test equipment must be used correctly, and the student must be able to properly explain all results. The student must also successfully answer all Challenges presented by the instructor for that experiment or project.

Students learning from these modules would do well to abide by the following principles:

- No text should be considered fully and adequately read unless and until you can express every idea *in your own words, using your own examples.*
- You should always articulate your thoughts as you read the text, noting points of agreement, confusion, and epiphanies. Feel free to print the text on paper and then write your notes in the margins. Alternatively, keep a journal for your own reflections as you read. This is truly a helpful tool when digesting complicated concepts.
- Never take the easy path of highlighting or underlining important text. Instead, *summarize* and/or *comment* on the text using your own words. This actively engages your mind, allowing you to more clearly perceive points of confusion or misunderstanding on your own.
- A very helpful strategy when learning new concepts is to place yourself in the role of a teacher, if only as a mental exercise. Either explain what you have recently learned to someone else, or at least *imagine* yourself explaining what you have learned to someone else. The simple act of having to articulate new knowledge and skill forces you to take on a different perspective, and will help reveal weaknesses in your understanding.
- Perform each and every mathematical calculation and thought experiment shown in the text on your own, referring back to the text to see that your results agree. This may seem trivial and unnecessary, but it is critically important to ensuring you actually understand what is presented, especially when the concepts at hand are complicated and easy to misunderstand. Apply this same strategy to become proficient in the use of *circuit simulation software*, checking to see if your simulated results agree with the results shown in the text.
- Above all, recognize that learning is hard work, and that a certain level of frustration is unavoidable. There are times when you will struggle to grasp some of these concepts, and that struggle is a natural thing. Take heart that it will yield with persistent and varied[4] effort, and never give up!

Students interested in using these modules for self-study will also find them beneficial, although the onus of responsibility for thoroughly reading and answering questions will of course lie with that individual alone. If a qualified instructor is not available to challenge students, a workable alternative is for students to form study groups where they challenge[5] one another.

To high standards of education,

Tony R. Kuphaldt

[4] As the old saying goes, "Insanity is trying the same thing over and over again, expecting different results." If you find yourself stumped by something in the text, you should attempt a different approach. Alter the thought experiment, change the mathematical parameters, do whatever you can to see the problem in a slightly different light, and then the solution will often present itself more readily.

[5] Avoid the temptation to simply share answers with study partners, as this is really counter-productive to learning. Always bear in mind that the answer to any question is far less important in the long run than the method(s) used to obtain that answer. The goal of education is to empower one's life through the improvement of clear and independent thought, literacy, expression, and various practical skills.

# Appendix C

# Tools used

I am indebted to the developers of many open-source software applications in the creation of these learning modules. The following is a list of these applications with some commentary on each.

You will notice a theme common to many of these applications: a bias toward *code*. Although I am by no means an expert programmer in any computer language, I understand and appreciate the flexibility offered by code-based applications where the user (you) enters commands into a plain ASCII text file, which the software then reads and processes to create the final output. Code-based computer applications are by their very nature *extensible*, while WYSIWYG (What You See Is What You Get) applications are generally limited to whatever user interface the developer makes for you.

The `GNU/Linux` computer operating system

> There is so much to be said about Linus Torvalds' `Linux` and Richard Stallman's `GNU` project. First, to credit just these two individuals is to fail to do justice to the *mob* of passionate volunteers who contributed to make this amazing software a reality. I first learned of `Linux` back in 1996, and have been using this operating system on my personal computers almost exclusively since then. It is *free*, it is completely *configurable*, and it permits the continued use of highly efficient `Unix` applications and scripting languages (e.g. shell scripts, Makefiles, `sed`, `awk`) developed over many decades. `Linux` not only provided me with a powerful computing platform, but its open design served to inspire my life's work of creating open-source educational resources.

Bram Moolenaar's `Vim` text editor

> Writing code for any code-based computer application requires a *text editor*, which may be thought of as a word processor strictly limited to outputting plain-ASCII text files. Many good text editors exist, and one's choice of text editor seems to be a deeply personal matter within the programming world. I prefer `Vim` because it operates very similarly to `vi` which is ubiquitous on `Unix`/`Linux` operating systems, and because it may be entirely operated via keyboard (i.e. no mouse required) which makes it fast to use.

Donald Knuth's TEX typesetting system

Developed in the late 1970's and early 1980's by computer scientist extraordinaire Donald Knuth to typeset his multi-volume magnum opus *The Art of Computer Programming*, this software allows the production of formatted text for screen-viewing or paper printing, all by writing plain-text code to describe how the formatted text is supposed to appear. TEX is not just a markup language for documents, but it is also a Turing-complete programming language in and of itself, allowing useful algorithms to be created to control the production of documents. Simply put, TEX *is a programmer's approach to word processing*. Since TEX is controlled by code written in a plain-text file, this means anyone may read that plain-text file to see exactly how the document was created. This openness afforded by the code-based nature of TEX makes it relatively easy to learn how other people have created their own TEX documents. By contrast, examining a beautiful document created in a conventional WYSIWYG word processor such as Microsoft `Word` suggests nothing to the reader about *how* that document was created, or what the user might do to create something similar. As Mr. Knuth himself once quipped, conventional word processing applications should be called WYSIAYG (What You See Is *All* You Get).

Leslie Lamport's LATEX extensions to TEX

Like all true programming languages, TEX is inherently extensible. So, years after the release of TEX to the public, Leslie Lamport decided to create a massive extension allowing easier compilation of book-length documents. The result was LATEX, which is the markup language used to create all ModEL module documents. You could say that TEX is to LATEX as `C` is to `C++`. This means it is permissible to use any and all TEX commands within LATEX source code, and it all still works. Some of the features offered by LATEX that would be challenging to implement in TEX include automatic index and table-of-content creation.

Tim Edwards' `Xcircuit` drafting program

This wonderful program is what I use to create all the schematic diagrams and illustrations (but not photographic images or mathematical plots) throughout the ModEL project. It natively outputs PostScript format which is a true vector graphic format (this is why the images do not pixellate when you zoom in for a closer view), and it is so simple to use that I have never had to read the manual! Object libraries are easy to create for `Xcircuit`, being plain-text files using PostScript programming conventions. Over the years I have collected a large set of object libraries useful for drawing electrical and electronic schematics, pictorial diagrams, and other technical illustrations.

`Gimp` graphic image manipulation program

Essentially an open-source clone of Adobe's `PhotoShop`, I use `Gimp` to resize, crop, and convert file formats for all of the photographic images appearing in the ModEL modules. Although `Gimp` does offer its own scripting language (called `Script-Fu`), I have never had occasion to use it. Thus, my utilization of `Gimp` to merely crop, resize, and convert graphic images is akin to using a sword to slice bread.

`SPICE` circuit simulation program

`SPICE` is to circuit analysis as TEX is to document creation: it is a form of markup language designed to describe a certain object to be processed in plain-ASCII text. When the plain-text "source file" is compiled by the software, it outputs the final result. More modern circuit analysis tools certainly exist, but I prefer `SPICE` for the following reasons: it is *free*, it is *fast*, it is *reliable*, and it is a fantastic tool for *teaching* students of electricity and electronics how to write simple code. I happen to use rather old versions of `SPICE`, version 2g6 being my "go to" application when I only require text-based output. `NGSPICE` (version 26), which is based on Berkeley `SPICE` version 3f5, is used when I require graphical output for such things as time-domain waveforms and Bode plots. In all `SPICE` example netlists I strive to use coding conventions compatible with all `SPICE` versions.

Andrew D. Hwang's `ePiX` mathematical visualization programming library

This amazing project is a `C++` library you may link to any `C/C++` code for the purpose of generating PostScript graphic images of mathematical functions. As a completely free and open-source project, it does all the plotting I would otherwise use a Computer Algebra System (CAS) such as `Mathematica` or `Maple` to do. It should be said that `ePiX` is *not* a Computer Algebra System like `Mathematica` or `Maple`, but merely a mathematical *visualization* tool. In other words, it won't determine integrals for you (you'll have to implement that in your own `C/C++` code!), but it can graph the results, and it does so beautifully. What I really admire about `ePiX` is that it is a `C++` programming library, which means it builds on the existing power and toolset available with that programming language. Mr. Hwang could have probably developed his own stand-alone application for mathematical plotting, but by creating a `C++` library to do the same thing he accomplished something much greater.

`gnuplot` mathematical visualization software

Another open-source tool for mathematical visualization is `gnuplot`. Interestingly, this tool is *not* part of Richard Stallman's GNU project, its name being a coincidence. For this reason the authors prefer "gnu" *not* be capitalized at all to avoid confusion. This is a much "lighter-weight" alternative to a spreadsheet for plotting tabular data, and the fact that it easily outputs directly to an X11 console or a file in a number of different graphical formats (including PostScript) is very helpful. I typically set my `gnuplot` output format to default (X11 on my Linux PC) for quick viewing while I'm developing a visualization, then switch to PostScript file export once the visual is ready to include in the document(s) I'm writing. As with my use of `Gimp` to do rudimentary image editing, my use of `gnuplot` only scratches the surface of its capabilities, but the important points are that it's *free* and that it *works well*.

`Python` programming language

Both Python and C++ find extensive use in these modules as instructional aids and exercises, but I'm listing Python here as a *tool* for myself because I use it almost daily as a *calculator*. If you open a Python interpreter console and type `from math import *` you can type mathematical expressions and have it return results just as you would on a hand calculator. Complex-number (i.e. *phasor*) arithmetic is similarly supported if you include the complex-math library (`from cmath import *`). Examples of this are shown in the Programming References chapter (if included) in each module. Of course, being a fully-featured programming language, Python also supports conditionals, loops, and other structures useful for calculation of quantities. Also, running in a console environment where all entries and returned values show as text in a chronologically-ordered list makes it easy to copy-and-paste those calculations to document exactly how they were performed.

# Appendix D

# Creative Commons License

Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

**Section 1 – Definitions.**

a. **Adapted Material** means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. **Adapter's License** means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. **Copyright and Similar Rights** means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

d. **Effective Technological Measures** means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. **Exceptions and Limitations** means fair use, fair dealing, and/or any other exception or

limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.

i. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. **You** means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

**Section 2 – Scope.**

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

A. reproduce and Share the Licensed Material, in whole or in part; and

B. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures.

For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. Downstream recipients.

A. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

B. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

**Section 3 – License Conditions.**

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:

A. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

**Section 4 – Sui Generis Database Rights.**

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

**Section 5 – Disclaimer of Warranties and Limitation of Liability.**

a. Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors,

whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.

b. To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

**Section 6 – Term and Termination.**

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

**Section 7 – Other Terms and Conditions.**

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

**Section 8 – Interpretation.**

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully

be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at `creativecommons.org/policies`, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at `creativecommons.org`.

# Appendix E

# References

"Modbus Application Protocol Specification", version 1.1b, Modbus-IDA, Modbus Organization, Inc., 2006.

"Modbus Messaging on TCP/IP Implementation Guide", version 1.0b, Modbus-IDA, Modbus Organization, Inc., 2006.

"Modicon Modbus Protocol Reference Guide", (PI-MBUS-300) revision J, Modicon, Inc. Industrial Automation Systems, North Andover, MA, 1996.

Park, John; Mackay, Steve; Wright, Edwin; *Practical Data Communications for Instrumentation and Control*, IDC Technologies, published by Newnes (an imprint of Elsevier), Oxford, England, 2003.

# Appendix F

# Version history

This is a list showing all significant additions, corrections, and other edits made to this learning module. Each entry is referenced by calendar date in reverse chronological order (newest version first), which appears on the front cover of every learning module for easy reference. Any contributors to this open-source document are listed here as well.

**19 February 2025** – minor edits to the "SCADA system fault" Diagnostic Reasoning question to make it less confusing to readers.

**24 September 2024** – edited some instructor notes, and minor edits to the Tutorial.

**17 September 2024** – divided the Introduction chapter into sections, one with recommendations for students, one with a listing of challenging concepts, and one with recommendations for instructors.

**21 February 2024** – minor edits to the Tutorial, as well as added instructor notes to some of the questions.

**8 July 2023** – added a new Tutorial section analyzing the contents of a Modbus/TCP read register response, and also added a Historical Reference chapter with an entry on digital data *endianness*.

**29 November 2022** – placed questions at the top of the itemized list in the Introduction chapter prompting students to devise experiments related to the tutorial content.

**10 May 2021** – commented out or deleted empty chapters.

**18 March 2021** – corrected multiple instances of "volts" that should have been capitalized "Volts".

**22 February 2021** – added content to the Introduction, as well as minor edits to the Tutorial chapter. Also corrected a copy-and-paste error in the Case Tutorial where the function code was supposed to be 06 rather than 03.

**28 September 2020** – added a Case Tutorial chapter showing examples of Modbus ASCII message exchanges, and made minor edits to the Derivations and Technical References chapter to clarify the

meaning of "Hi" and "Lo" byte order. Also eliminated TEX-style tables for LATEX-style tables.

**14 September 2020** – significantly edited the Introduction chapter to make it more suitable as a pre-study guide and to provide cues useful to instructors leading "inverted" teaching sessions.

**27 June 2020** – added more content.

**26 June 2020** – consolidated the Simplified and Full Tutorials into a single Tutorial because their content complemented each other so well.

**14 April 2019** – document first created.

# Index