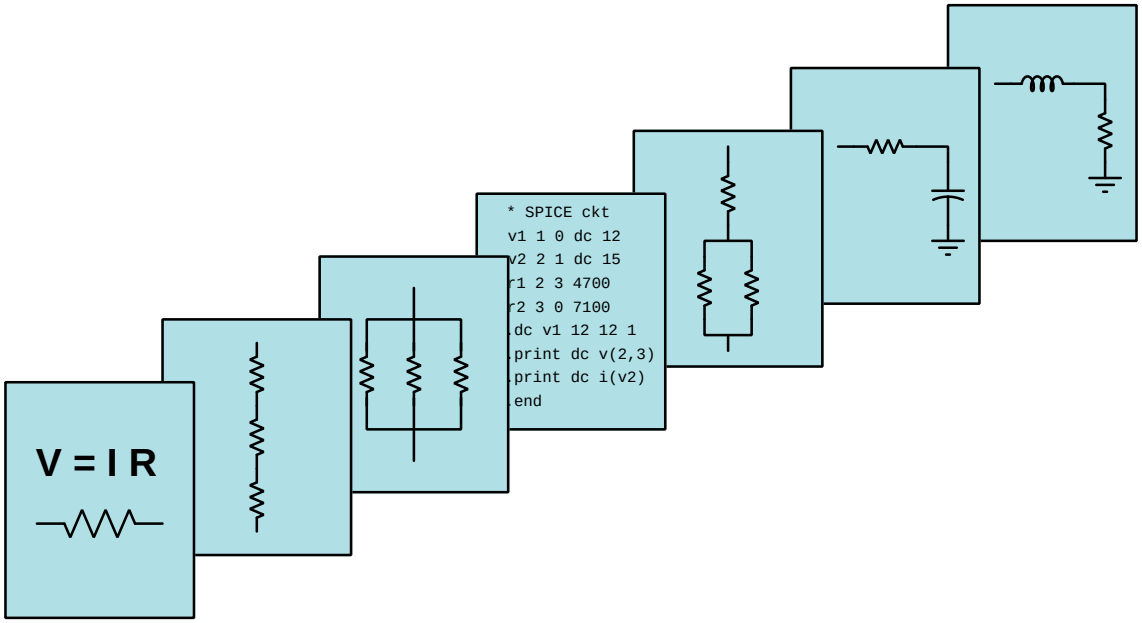
Modular Electronics Learning (ModEL) project



Serial Data Communication

© 2020-2026 by Tony R. Kuphaldt – under the terms and conditions of the Creative Commons Attribution 4.0 International Public License

Last update = 7 March 2026

This is a copyrighted work, but licensed under the Creative Commons Attribution 4.0 International Public License. A copy of this license is found in the last Appendix of this document. Alternatively, you may visit `http://creativecommons.org/licenses/by/4.0/` or send a letter to Creative Commons: 171 Second Street, Suite 300, San Francisco, California, 94105, USA. The terms and conditions of this license allow for free copying, distribution, and/or modification of all licensed works by the general public.

Contents

1 Introduction **3**
- 1.1 Recommendations for students . . . 3
- 1.2 Challenging concepts related to serial data communication . . . 5
- 1.3 Recommendations for instructors . . . 6

2 Case Tutorial **7**
- 2.1 Example: UART data frames with and without parity . . . 8
- 2.2 Example: ASCII data captured on an oscilloscope . . . 11

3 Tutorial **13**
- 3.1 Serial communication principles . . . 14
- 3.2 Physical encoding of bits . . . 17
- 3.3 Communication speed . . . 21
- 3.4 Data frames . . . 23
- 3.5 Parity . . . 26
- 3.6 Frame check sequences . . . 30
- 3.7 Flow control . . . 31
- 3.8 Channel arbitration . . . 33
  - 3.8.1 Master-slave . . . 34
  - 3.8.2 Token-passing . . . 35
  - 3.8.3 TDMA . . . 36
  - 3.8.4 CSMA . . . 37
- 3.9 The OSI Reference Model . . . 39

4 Historical References **43**
- 4.1 Ancient serial data communication . . . 44
- 4.2 The original telegraph code . . . 46

5 Derivations and Technical References **47**
- 5.1 International Morse Code . . . 48
- 5.2 ASCII character codes . . . 51
- 5.3 Hash algorithms . . . 52

6 Animations **55**
6.1 Animation of serial-in, parallel-out shift register . . . 56
6.2 Animation of parallel-in, serial-out shift register . . . 75

7 Questions **95**
7.1 Conceptual reasoning . . . 99
7.1.1 Reading outline and reflections . . . 100
7.1.2 Foundational concepts . . . 101
7.1.3 Manchester encoding of a digital word . . . 104
7.1.4 Serial data stream decoding . . . 105
7.1.5 More Manchester and FSK decoding . . . 107
7.1.6 Ambiguous Manchester data stream . . . 108
7.1.7 Serial configuration for an HMI . . . 109
7.1.8 Serial data programming in a simulated PLC . . . 110
7.1.9 Interpreting NRZ data frames . . . 111
7.1.10 Interpreting ASCII character from EIA/TIA-232 data frame . . . 112
7.1.11 Interpreting more NRZ data frames . . . 113
7.1.12 EIA/TIA-232 data frames of ASCII characters . . . 115
7.1.13 Microcontroller UART communication program . . . 120
7.1.14 Microcontroller UART text and number program . . . 123
7.2 Quantitative reasoning . . . 126
7.2.1 Miscellaneous physical constants . . . 127
7.2.2 Introduction to spreadsheets . . . 128
7.2.3 Manchester data frame with a specified bit rate . . . 131
7.3 Diagnostic reasoning . . . 132
7.3.1 Testing determinism . . . 132

A Problem-Solving Strategies **133**

B Instructional philosophy **135**
B.1 First principles of learning . . . 136
B.2 Proven strategies for instructors . . . 137
B.3 Proven strategies for students . . . 139
B.4 Design of these learning modules . . . 140

C Tools used **143**

D Creative Commons License **147**

E References **155**

F Version history **157**

Index **158**

Chapter 1

Introduction

1.1 Recommendations for students

Serial data communication is the transmission and reception of digital data one bit at a time, in sequence, and it is as old as the telegraph. This module discusses principles common to most serial networks without focusing on any one in particular.

Important concepts related to serial communication include the **encoding** of information in digital form, **delimiting** of serial data, **NRZ** encoding, **RZ** encoding, **Manchester** encoding, **FSK** encoding, **bit rate**, **baud**, the packaging of information into **frames**, **start** and **stop** bits, **parity**, **ASCII** code, signal **noise**, **frame checking**, **flow control**, **full-duplex** versus **half-duplex**, channel **arbitration** techniques, **tokens**, **polling**, data **collisions**, **determinism**, **jabber**, and the **OSI model**.

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

- How might an experiment be designed and conducted to test the bit-rate of a serial data signal? What hypothesis (i.e. prediction) might you pose for that experiment, and what result(s) would either support or disprove that hypothesis?
- How does serial communication differ from parallel communication?
- How was information communicated over early telegraph systems?
- What are "mark" and "space" states?
- What are some of the possible interpretational problems when communicating serially?
- What are some different ways we may encode digital bits of information as electrical signals?
- What is the purpose of a "frame" in serial data communication?
- How do we measure the speed of communication in a serial system?
- When is *bit rate* not equal to *baud*?

- What kinds of auxiliary information needs to be communicated in a serial system besides the data itself?
- How are streams of serial data synchronized for reliable reception?
- How do NRZ, RZ, FSK, and Manchester encoding schemes differ from one another?
- What are some common sources of errors in serial communication?
- What are some different ways to check for the presence of errors in serial communication?
- What is the basic principle of a parity bit?
- How can parity be fooled (i.e. what type(s) of error(s) can get through without being detected by a parity check)?
- How can multiple devices share the same communication channel without interfering with each other? What are some of the different strategies for accomplishing this goal?
- How could you explain each type of channel arbitration protocol by analogy to human conversations? (e.g. "Master-Slave would be like three people in a group who . . .")
- What are some weaknesses of each channel arbitration scheme?
- What is "flow control" and why is it necessary in some applications but not in others?
- Why is determinism an important quality for some communication networks?
- How does the OSI Reference Model relate to different communication systems?
- How is it possible for some layers of the OSI Reference Model to apply to a specific communication standard but not other layers?

1.2 Challenging concepts related to serial data communication

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

- **Manchester encoding** – physical encoding schemes where the rising/falling edges of a pulse convey bit information are generally more difficult to interpret than NRZ signals due to the challenge of identifying which edge(s) represent data bits versus which are merely reversals setting up for the next data bit.
- **Parity** – a common area of confusion for students is the subject of *parity bits* and error-checking in general. In order to understand these topics accurately and to avoid misconception, it is helpful to run some *thought experiments* whereby you imagine data becoming corrupted during serial transmission, and analyze how each of the error-checking techniques would be able to identify the presence of data corruption.
- **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.
- **OSI reference model** – an easy way to get confused on this subject is to try to interpret it as a standard in and of itself. It is more accurately thought of as a taxonomy of communication instead: merely a way to label and describe aspects of communications standards. No single communications embodies all seven layers of the OSI model, and many relate to only one of those layers!
- **Synchronous versus Asynchronous communication** – synchronous communication is when two or more digital devices communicate in lock-step with each other due to the simultaneous transmission of a clock signal. Asynchronous communication is when two or more digital devices use internal clocks to keep pace with each other, but rely on a "start" signal of some kind to synchronize themselves together at the start of each data frame.

The *Case Tutorial* chapter contains sections showing pulse waveforms for UART and EIA/TIA-232 which are helpful in learning to decipher serial data frames and their parity bits.

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.

  Assessment – Students present their "Conceptual Toolbox" Theory Journal entries listing important concepts applied in the Tutorial text(s). A "Conceptual Toolbox" serves as inspiration when solving problems, each concept suggesting a potential problem-solving action. Challenge students to collaboratively generate a Conceptual Toolbox for the day's topic during the class session and then use it when solving problems together, thus fostering the habit of reasoning from principles rather than recalling procedures!

- **Outcome** – Interpret serial data frames from oscillographs

  Assessment – Correctly determine the binary data word represented by NRZ, RZ, Manchester, and FSK waveforms; e.g. pose problems in the form of the "Serial data stream decoding" Conceptual Reasoning question.

  Assessment – Correctly determine the binary data word, and whether or not parity checks okay, from oscillographs of serial data transmissions; e.g. pose problems in the form of the "Interpreting NRZ data frames" and "Interpreting more NRZ data frames" Conceptual Reasoning questions.

  Assessment – Correctly determine ASCII characters from oscillographs of serial data transmissions; e.g. pose problems in the form of the "EIA/TIA-232 data frames of ASCII characters" Conceptual Reasoning question.

- **Outcome** – Independent research

  Assessment – Read and summarize in your own words reliable source documents on the subject of OSI Reference Model.

Chapter 2

Case Tutorial

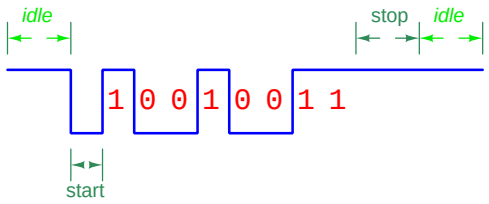
The idea behind a *Case Tutorial* is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as 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?

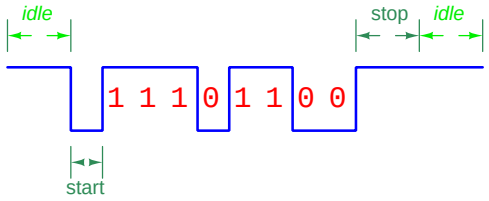
2.1 Example: UART data frames with and without parity

In the following illustrations we will show three examples of UART serial data signals representing eight-bit binary numbers, assuming the use of positive DC source voltage for "idle" and logical "1" states and zero voltage for logical "0" states[1], *no parity bit*, and two stop bits:

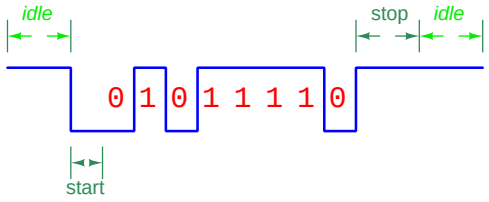
Binary number = 11001001



Binary number = 00110111



Binary number = 01111010

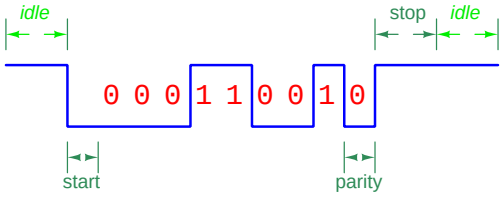


Note how UART data is always transmitted with the least-significant bit (LSB) sent first and the most-significant bit (MSB) last. This is why the pulse waveforms' high and low states seem to come in reverse order.

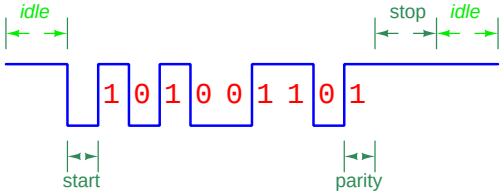
[1] Note how this "positive" logic is consistent with most digital logic devices such as logic gates, but is *opposite* of certain communication standards such as EIA/TIA-232 where a positive voltage is a 0 or "space" state and a negative voltage is a 1 or a "mark" state!

Next we will examine three more examples of UART signals using the same parameters as before (positive logic levels, 8 data bits, and 2 stop bits), but this time including a *parity bit* set for *odd parity*:

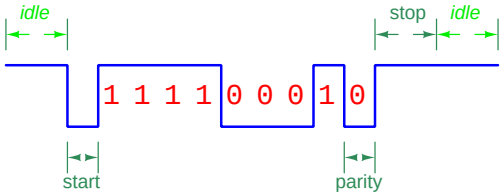
Binary number = 10011000



Binary number = 01100101



Binary number = 10001111



Note how each one of the bits' time duration is identical, the data bits each occupying the exact same amount of time on the horizontal axis as each start bit, each stop bit, and each parity bit. The duration of the "idle" time is arbitrary, depending only on how often binary numbers get communicated.

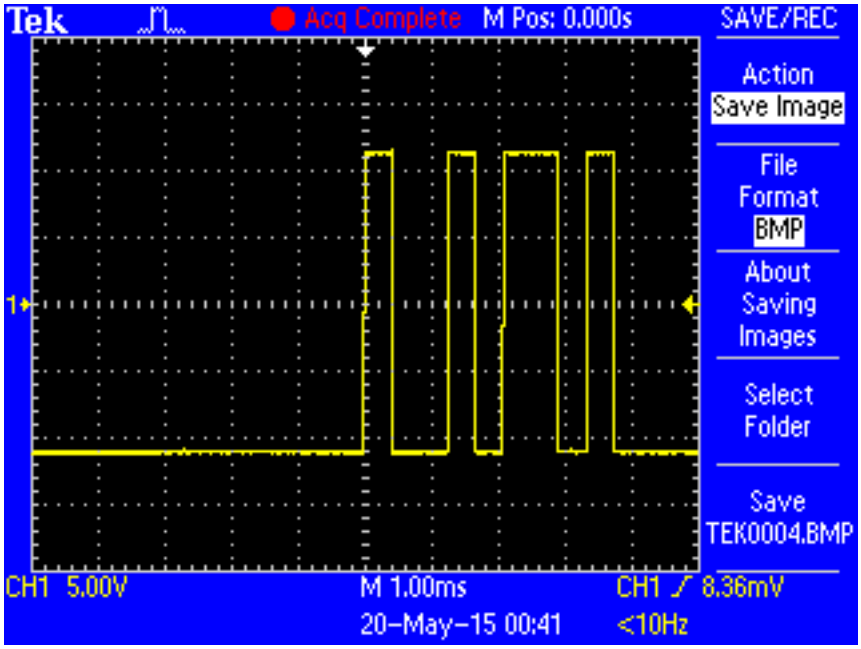
Parity bits exist for the purpose of *error detection* in serial data communication. Errors usually originate in communication systems due to excessive electrical *noise*, and so it is possible for noise to grow strong enough to corrupt one of the bits so as to be interpreted at the receiving end as the wrong logical state; i.e. a bit transmitted as a 0 being received as a 1, or vice-versa. If both transmitting and receiving devices are pre-configured to generate and interpret (respectively) a parity bit whose value is based on the number of "1" bits in the data payload, any single-bit corruption will be detectable as such when the receiver does its own parity calculation and determines a mis-match.

In the last set of UART signal examples shown, parity was configured for "odd". This meant that the transmitting device counted up the number of "1" bits in the eight-bit binary number and then made the parity bit either a "1" or a "0" as necessary to bring the total "1" bit count to an odd value. For example, the binary number `10011000` in the first parity example already had an odd number of "1" bits and so the transmitting device set the parity bit to zero because it didn't need any more "1" bits to make an odd count. However, the binary number `01100101` used in the

second parity example had an even number of “1” bits and so the transmitting device did need to set the parity but to one in order to being the total “1” bit count to an odd number. The receiving device simply has to count all the “1” bits it receives in total to see if it finds an odd number. If it happens to count an even number of “1” bits in total, it would knows something has gone wrong.

2.2 Example: ASCII data captured on an oscilloscope

This is a screen-capture from a digital oscilloscope showing the ASCII code for the capital letter "K" (binary `01001011`) communicated via EIA/TIA-232 using 8 data bits, odd parity, and 1 stop bit:



Since the ASCII code for "K" contains an even number of 1's and the communication is configured for odd parity, the parity bit is set (1) to bring the total number of 1 states to an odd value.

Chapter 3

Tutorial

3.1 Serial communication principles

One of the great benefits of digital technology is the ability to *communicate* vast amounts of information over networks. This very text you are reading was transmitted in digital form over the electronic network we call the *Internet*: a feat virtually impossible for any sort of analog technology. *Serial* data communication refers to the transmission of digital signals one bit at a time rather than multiple bits at a time. While this may seem inefficient from a perspective of speed, it is highly efficient in terms of necessary data channels: by communicating just one bit at a time, only one wire (or one radio signal, or one optical fiber) is necessary to convey that data.

The task of encoding digital data as a sequential stream of bits and then sending those bits to a receiving device requires mutually-agreed *standards* for the encoding, the "packaging" of those bits, the speed at which the bits are sent, methods for multiple devices to use a common channel, and a host of other concerns. This tutorial will delineate the major points of compatibility necessary for digital devices to communicate serially. We begin with a brief exploration of some of the standards used in early *telegraph* systems.

An early form of digital communication was *Morse Code*, used to communicate alpha-numerical information as a series of "dots" and "dashes" over telegraph[1] systems. Each letter in the alphabet, and each numerical digit (0 through 9) was represented in Morse Code by a specific series of "dot" and "dash" symbols, a "dot" being a short pulse and a "dash" being a longer pulse. Samuel Morse's original code specification relied on both "long" and "short" dashes in addition to dot symbols, and also had pauses between symbols within some of the characters, as well as an inconsistent pattern for representing numerical digits. These inconsistencies in Morse's original code were eliminated with the invention of the *Continental Code*, so named because of its widespread use in Europe. This improved code standard eventually became adopted world-wide and subsequently known as the *International Morse Code*. Samuel Morse's code became known as *American Morse*, and faded into obscurity.

As primitive as these codes were, they encapsulated many of the basic principles we find in modern digital serial communication systems. First, a system of codes was necessary in order to represent English letters and numerals by electrical pulses. Next, there needed to be some way to delineate the beginning and end of each character.

[1] I do not expect any reader of this book to have firsthand knowledge of what a "telegraph" is, but I suspect some will have never heard of one until this point. Basically, a telegraph was a primitive electrical communication system stretching between cities using a keyswitch at the transmitting end to transmit on-and-off pulses and a "sounder" to make those pulses audible on the receiving end. Trained human operators worked these systems, one at the transmitting end (encoding English-written messages into a series of pulses) and one at the receiving end (translating those pulses into English letters).

International Morse Code (English letters and Arabic numerals only)

A •—	J •———	S •••	0 —————
B —•••	K —•—	T —	1 •————
C —•—•	L •—••	U ••—	2 ••———
D —••	M ——	V •••—	3 •••——
E •	N —•	W •——	4 ••••—
F ••—•	O ———	X —••—	5 •••••
G ——•	P •——•	Y —•——	6 —••••
H ••••	Q ——•—	Z ——••	7 ——•••
I ••	R •—•		8 ———••
			9 ————•

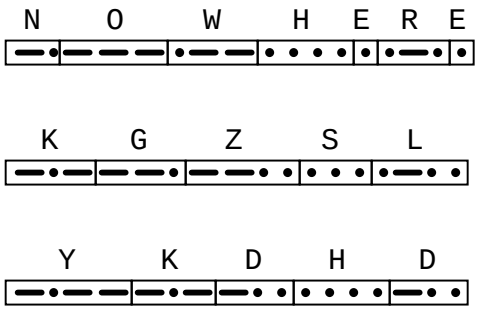
Consider the encoding for the word NOWHERE. By placing an extra pause between characters, it is easy to delineate individual characters in the message:

"NOWHERE"

—• ——— •—— •••• • •—• •
N O W H E R E

If this space between characters were not present, it would be impossible to determine the message with certainty. By removing the spaces, we find multiple non-sensical interpretations are possible for the same string of "dots" and "dashes:"

Same sequence of "dots" and "dashes," with multiple interpretations!



For that matter, it is even possible to confuse the meaning of the text string "NOWHERE" when the individual characters are properly interpreted. Does the string of characters say "NOWHERE," or does it say "NOW HERE"?

This simple example illustrates the need for *delimiting* in serial data communication. Some means must be employed to distinguish individual groups of bits (generally called *frames* or *packets*) from one another, lest their meanings be lost. In the days when human operators sent and interpreted Morse and Continental code messages, the standard delimiter was an extra time delay (pause) between characters, and between words[2]. This is not much different from the use of whitespace to delineate words, sentences, and paragraphs typed on a page. Sentenceswouldcertainlybeconfusingtoreadifnotforspaces!

In later years, when *teletype* machines were designed to replaced skilled Morse operators, the concept of frame delineation had to be addressed more rigorously. These machines consisted of a typewriter-style keyboard which marked either paper strips or pages with dots corresponding to a 5-bit code called the *Baudot code*. The paper strip or sheets were then read electrically and converted into a serial stream of on-and-off pulses which were then transmitted along standard telegraph circuit lines. A matching teletype machine at the receiving end would then convert the signal stream into printed characters (a telegram). Not only could unskilled operators use teletype machines, but the data rate far exceeded what the best human Morse operators could achieve[3]. However, these machines required special "start" and "stop" signals to synchronize the communication of each character, not being able to reliably interpret pauses like human operators could. Developed in the 1870's, the Baudot code still finds application today in *Radio Teletype* or *RTTY* communication, a digital mode of radio communication used by many amateur radio operators.

Interestingly, modern asynchronous[4] serial data communication relies on the same concept of "start" and "stop" bits to synchronize the transmission of data packets. Each new packet of serial data is preceded by some form of "start" signal, then the packet is sent, and followed up by some sort of "stop" signal. The receiving device(s) synchronize to the transmitter when the "start" signal is detected, and non-precision clocks keep the transmitting and receiving devices in step with each other over the short time duration of the data packet. So long as the transmitting and receiving clocks are close enough to the same frequency, and the data packet is short enough in its number of bits, the synchronization will be good enough for each and every bit of the message to be properly interpreted at the receiving end.

[2] According to Recommendation ITU-R M.1677 from the ITU Radiocommunication Assembly in 2004, a dash should have a time duration three times that of a dot, the time delay between dots and dashes within the same character should last as long as a single dot, the time delay between characters should last as long as a dash (three dots), and the space between two words should last as long as seven dots.

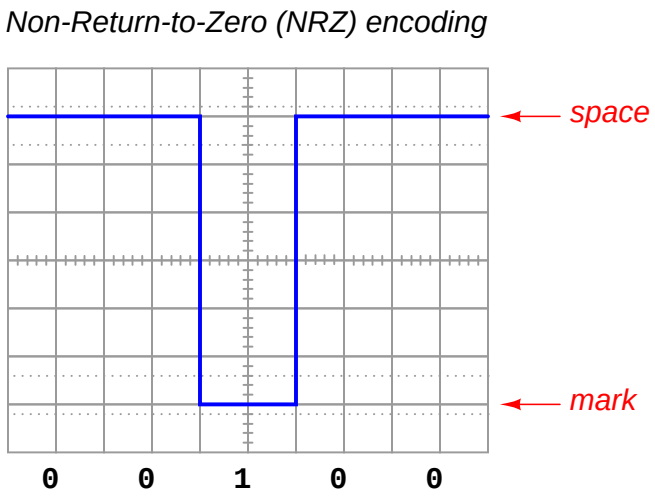
[3] A test message sent in 1924 between two teletype machines achieved a speed of 1920 characters per minute (32 characters per second), sending the sentence fragments "THE WESTERN ELECTRIC COMPANY", "FRESHEST EGGS AT BOTTOM MARKET PRICES", and "SHE IS HIS SISTER".

[4] "Asynchronous" refers to the transmitting and receiving devices not having to be in perfect synchronization in order for data transfer to occur. In synchronous serial networks, a common "clock" signal maintains transmitting and receiving devices in a constant state of synchronization, so that data packets do not have to be preceded by "start" bits or followed by "stop" bits. Synchronous data communication networks are therefore more efficient (not having to include "extra" bits in the data stream) but also more complex. Most long-distance, heavy traffic digital networks (such as the "backbone" networks used for the Internet) are synchronous for this reason.

3.2 Physical encoding of bits

Telegraph systems were Boolean in nature: representing "dots" and "dashes" by one electrical state of the telegraph line, and pauses by another. When manually-actuated keyswitches were abandoned in favor of teletype machines, and Morse code abandoned in favor of the *Baudot* (5-bit) code for representing alphanumeric characters, the electrical nature of the telegraph (at least initially[5]) remained the same. The telegraph line would either be energized or not, corresponding to *marks* or *spaces* made on the teletype paper.

Many modern digital communication standards represent binary "1" and "0" values in exactly this way: a "1" is represented by a "mark" state and a "0" is represented by a "space" state. "Marks" and "spaces" in turn correspond to different voltage levels between the conductors of the network circuit. For example, the very common EIA/TIA-232 serial communications standard (once the most popular way of connecting peripheral devices to personal computers, formerly called RS-232) defines a "mark" (1) state as −3 Volts between the data wire and ground, and a "space" (0) state as +3 Volts between the data wire and ground. This is referred to as *Non-Return-to-Zero*[6] or *NRZ* encoding:

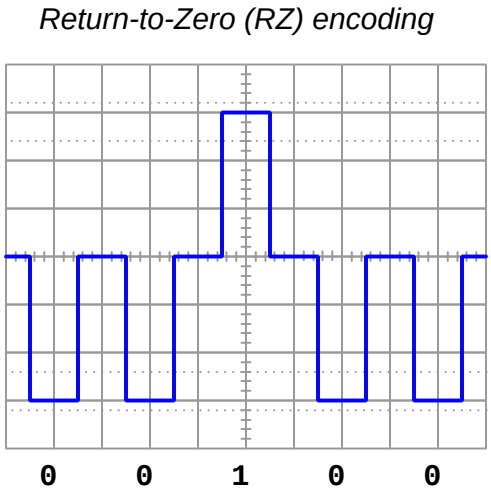


An easy way to remember the difference between a "mark" and a "space" in this scheme is to recall the operation of old telegraph printing units, specifically how they created marks and spaces on moving paper strip. When the printing unit was energized (i.e. the transmitting key was pressed, sending current through the solenoid coil of the printer, corresponding to a "1" state), the printer's iron armature would be pulled *down* to draw a mark on the paper strip. When de-energized (transmitting key released, stopping current in the telegraph line, corresponding to a "0" state), the printer's armature would spring-return *up* from the paper to leave a blank space.

[5] Later versions of teletype systems employed audio tones instead of discrete electrical pulses so that many different channels of communication could be funneled along one telegraph line, each channel having its own unique audio tone frequency which could be filtered from other channels' tones. This was a form of analog *multiplexing*.

[6] This simply refers to the fact that the signal never settles at 0 Volts.

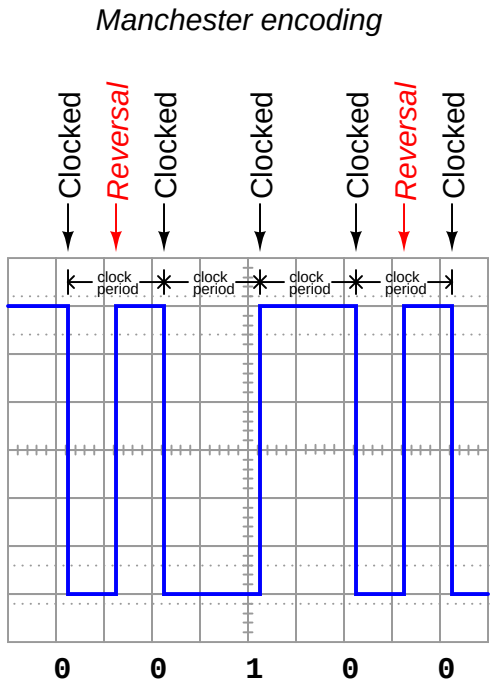
A different way to represent bits in serial fashion is to use bipolar (positive *and* negative) voltage levels to represent "1" and "0" bits respectively with zero-voltage representing resting states between bits. This general approach is known as Bipolar *Return-to-Zero* or *RZ* encoding:



This encoding is used in the ARINC 429 serial communications standard, popularly applied in aviation electronics ("avionics") systems. An advantage this offers over NRZ encoding is that with RZ each and every bit has its own distinct pulse peak which makes the bit states completely unambiguous. Another way of stating this same fact is to say that the serial data stream is *self-clocking*: i.e. the clock signal synchronizing each bit may be interpreted directly from the data stream.

Other versions of RZ encoding exist besides this one, but the basic principle of always returning to a zero-voltage state between "high" bits is the common feature.

Another method to serially represent bits is to use an oscillating (square-wave) signal, counting *up* and *down* transitions (pulse edges) at specific times to represent 1 and 0 states. This is called *Manchester encoding*, and it is used in the 10 Mbps (10 million bits per second) version of *Ethernet* as well as some specialized industrial network standards as well (e.g. *FOUNDATION Fieldbus* "H1" and *Profibus* "PA"):



Note how each binary bit (0 or 1) is represented by the *direction* of the voltage signal's transition. A low-to-high transition represents a "1" state while a high-to-low transition represents a "0" state. Extra "reversal" transitions appear in the pulse stream only to set up the voltage level as needed for the next bit-representing transitions. The representation of bits by transitions rather than by static voltage levels guarantees the receiving device can naturally detect the clock frequency of the transmitted signal[7]. Manchester data is therefore another example of a *self-clocking* encoding scheme.

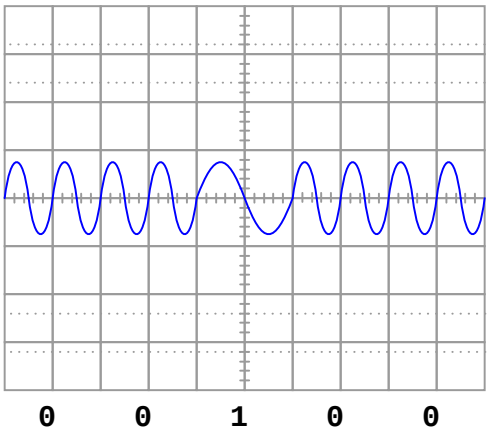
Interpreting a Manchester waveform is easier than it first appears. The key is identifying which transitions represent "clocked" bits and which transitions represent "reversals" prior to bits. If we identify the widest periods of the waveform, we know the transitions in these periods must represent real bits because there are no reversals. Another way to say this is the greatest time period found between successive transitions in a Manchester waveform *is* the clock period. Once we identify this

[7] This is most definitely *not* the case with NRZ encoding. To see the difference for yourself, imagine a continuous string of either "0" or "1" bits transmitted in NRZ encoding: it would be nothing but a straight-line DC signal. In Manchester encoding, it is *impossible* to have a straight-line DC signal for an indefinite length of time. Manchester signals *must* oscillate at a minimum frequency equal to the clock speed, thereby guaranteeing all receiving devices the ability to detect that clock speed and thereby synchronize themselves with it.

clock period, we may step along the waveform at that same period width to distinguish clocked bits from reversals.

Yet another method for encoding binary 1 and 0 states is to use sine waves of different frequencies ("tone bursts"). This is referred to as *Frequency Shift Keying*, or *FSK*, and it is the method of encoding embodied in the Bell 202 standard once used for computer modems and for communicating caller ID data over simple telephone service lines.

Frequency Shift Key (FSK) encoding



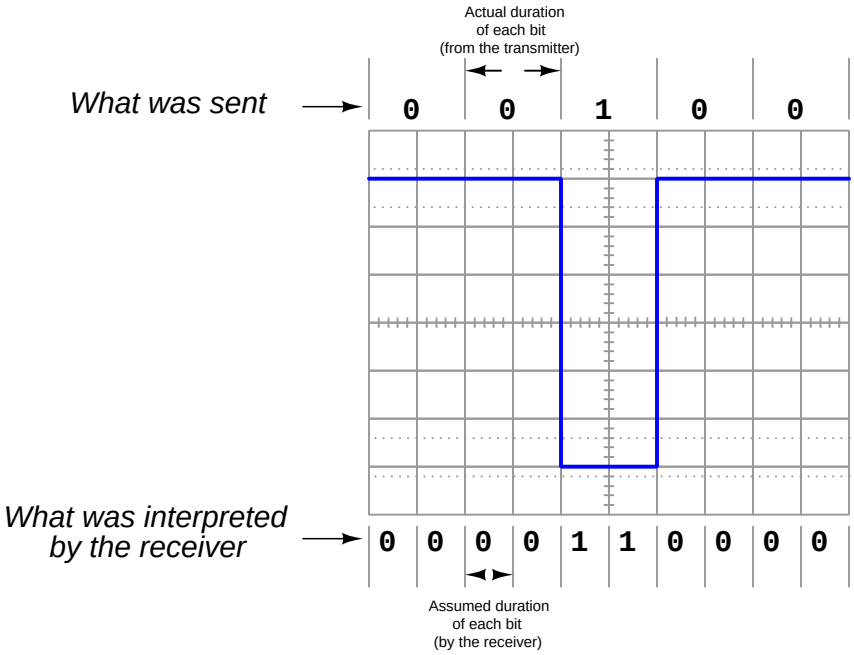
The Bell 202 standard defines two complete 2200 Hz cycles as a "0" bit (space) and one complete 1200 Hz cycle as a "1" bit (mark). This standard was invented as a way to exchange digital data over telephone networks, which were built to communicate audio-frequency AC signals and thus could not reliably communicate the square-wave (NRZ) signals associated with direct digital data. By assigning digital values to different audio frequencies, serial data could be communicated over telephone channels as a series of sine-wave tones.

Other methods exist as well for encoding digital data along network cables as well.

3.3 Communication speed

In order to successfully communicate digital data along a network, there must not only be a standard agreed upon between transmitting and receiving devices for encoding bits (NRZ, Manchester, FSK, etc.), but there must also be a standard in place for the *speed* at which those bits will be sent. This is especially true for NRZ and FSK encoding, where the "clock" speed is not explicitly present in the signal[8].

For example, consider the confusion that could arise interpreting a NRZ signal if the transmitting device sends data at half the speed assumed by the receiving device:



Thus, one of the essential parameters in a serial data communication system is the *bit rate*, measured in *bits per second* (bps). Some communications standards have fixed bit rates, such as FOUNDATION Fieldbus H1 and Profibus PA, both standardized at exactly 31.25 kbps. Some, such as Ethernet, have a few pre-defined speeds (10 Mbps, 100 Mbps, 1 Gbps) defined by the specific transmitting and receiving hardware used. Others, such as EIA/TIA-232 may be arbitrarily set by the user at speeds ranging from 300 bps to over 115 kbps.

An older term sometimes used synonymously with bit rate is *baud rate*, however "bits per second" and "baud" are actually different things. "Baud" refers to the number of voltage (or current) alternations per second of time, whereas "bits per second" refers to the actual number of binary data bits communicated per second of time. Baud is useful when determining whether or not the bandwidth (the maximum frequency capacity) of a communications channel is sufficient for a certain communications purpose. For a string of alternating bits (e.g. `010101010101`) using NRZ encoding,

[8] This is one of the advantages of RZ and Manchester encoding: they are "self-clocking" serial digital signals.

the baud rate is equivalent[9] to the bit rate: exactly one voltage transition for each bit. For a string of unchanging bits (e.g. `000000000000` or `111111111111`) using NRZ encoding, the baud rate is far less than the bit rate. In systems using Manchester encoding, the worst-case[10] baud rate will be exactly *twice* the bit rate, with two transitions (one up, one down) per bit. In some clever encoding schemes, it is possible to encode multiple bits per signal transition, such that the bit rate will actually be greater than the baud rate.

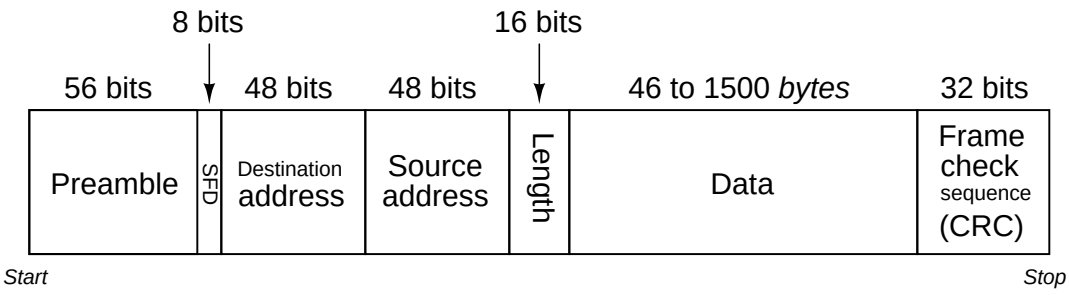
[9] This is likely why "bit rate" and "baud rate" became intermingled in digital networking parlance: the earliest serial data networks requiring speed configuration were NRZ in nature, where "bps" and "baud" are one and the same.

[10] For Manchester encoding, "worst-case" is a sequence of identical bit states, such as `111111111111`, where the signal must make an extra (down) transition in order to be "ready" for each meaningful (up) transition representing the next "1" state.

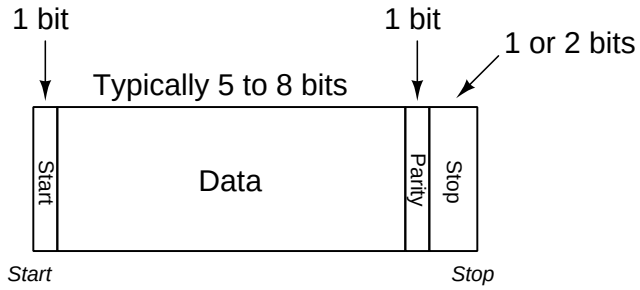
3.4 Data frames

Serial data communicated *asynchronously* means the transmitting and receiving hardware need not be in perfect synchronization to reliably send and receive digital data. In order for this to work, data must be sent in "frames" or "packets" of fixed (maximum) length, each frame preceded by a special "start" signal and concluded with a special "stop" signal. As soon as the transmitting device issues the "start" signal, the receiving device synchronizes to that start time, and runs at the pre-determined clock speed to gather the successive bits of the message until the "stop" signal is received. So long as the internal clock circuits of the transmitting and receiving devices are running at *approximately* the same speed, the devices will be synchronized closely enough to exchange a short message without any bits being lost or corrupted. There is such a thing as a *synchronous* digital network, where all transmitting and receiving devices are locked into a common clock signal so they cannot stray out of step with each other. The obvious advantage of synchronous communication is that no time need be wasted on "start" and "stop" bits, since data transfer may proceed continuously rather than in packets. However, synchronous communication systems tend to be more complex due to the need to keep all devices in perfect synchronization, and thus we see synchronous systems used for long-distance, high-traffic digital networks such as those use for Internet "backbones" and not for short-distance networks.

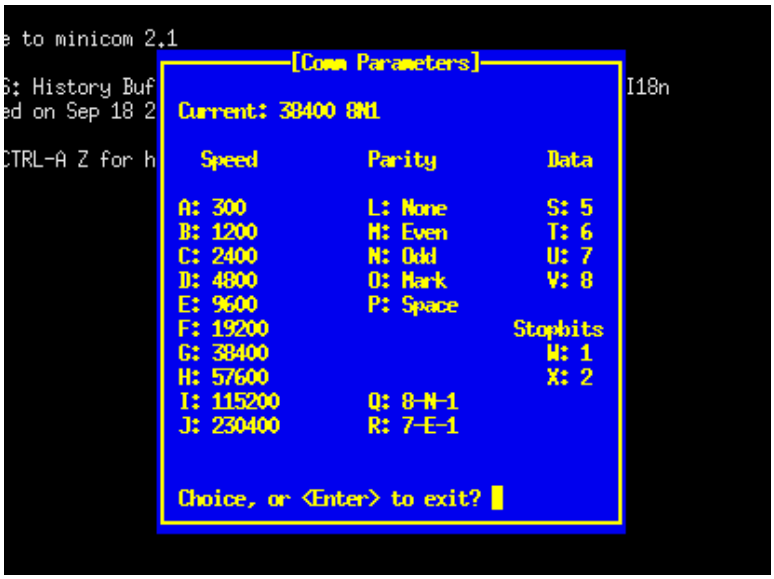
Like bit rate, the particular scheme of start and stop bits must also be agreed upon in order for two serial devices to communicate with each other. In some networks, this scheme is fixed and cannot be altered by the user. Ethernet is an example of this, where a sequence of 64 bits (an alternating string of "1" and "0" bits ending with a "1, 1"; this is the "preamble" and "start frame delimiter" or "SFD" bit groups) is used to mark the start of a frame and another group of bits specifies the length of the frame (letting the receiver know ahead of time when the frame will end). A graphic description of the IEEE 802.3 standard for Ethernet data frames is shown here, illustrating the lengths and functions of the bits comprising an Ethernet frame:



A simpler example of a data frame is that generated and received by *UART* (Universal Asynchronous Receiver-Transmitter) circuits, used in communications standards such as EIA/TIA-232, 422, and 485. A single bit marks the start of a message frame, followed by a succession of data bits comprising the "payload" of the frame, and terminated by at least one bit marking the end (stop) of the frame. An optional *parity bit* serves as a crude error-checking feature:



In many UART-based serial networks the user is free to select options for the number of data bits, parity, and stop bits. It is imperative in such systems that *all* transmitting and receiving devices within a given network be configured exactly the same, so that they will all "agree" on how to send and receive data. A screenshot from a UNIX-based serial communication terminal program (called *minicom*[11]) shows these options:

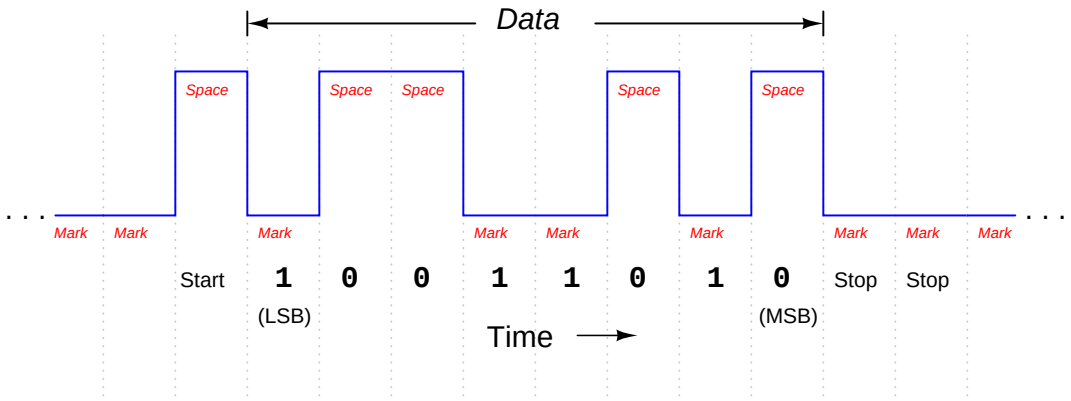


In this particular screenshot, you can see the data rate options (extending from 300 bps all the way up to 230400 bps!), the number of data bits (from 5 to 8), and the number of stop bits (1 or 2), all configurable by the user. Of course, if this program were being used for communication of

[11] An equivalent program for Microsoft Windows is *Hyperterminal*. A legacy application, available for both Microsoft Windows and UNIX operating systems, is the serial communications program called *kermit*.

data between two personal computers, *both* of those computers would need these parameters set identically in order for the communication to take place. Otherwise, the two computers would not be in agreement on speed, number of data bits, and stop bits; their respective data frames simply would not match.

To give an example of an EIA/TIA-232 data frame shown as a series of voltage states, consider this waveform communicating a string of eight bits (`01011001`), using NRZ encoding. Here, a single "start" marks the beginning of the data frame, while two successive "stop" bits end it, and in this case there is no parity bit. Also note how the bit sequence is transmitted "backwards," with the least-significant bit (LSB) sent first and the most-significant bit (MSB) sent last[12]:



Serial bitstream for the digital byte 01011001, where the least-significant bit (LSB) is sent first

Interestingly, the "mark" state (corresponding to a binary bit value of "1") is the default state of the communications channel when no data is being passed. The "start" bit is actually a space (0). This is the standard encoding scheme for EIA/TIA-232, EIA/TIA-485, and some other NRZ serial communication standards.

[12] This is standard in all UART serial communication, which EIA/TIA-232 is based on.

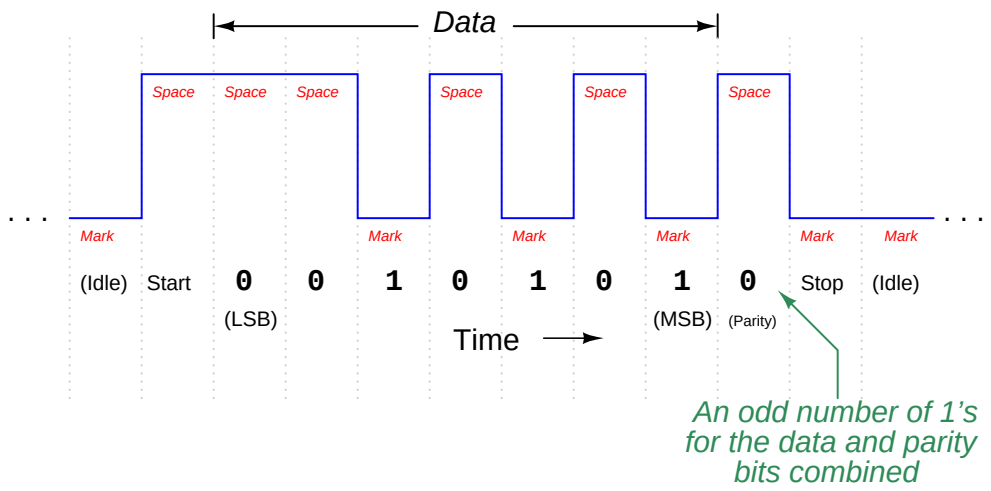
3.5 Parity

One of the options you probably noticed in the "minicom" terminal program screenshot was something called *parity*. This is a simple form of error-checking used in many serial communication standards. The basic principle is quite simple: an extra bit is added at the end of the data frame (between the data and stop bits) to force the total number of "1" states to be either odd or even. For example, in the data stream just shown (`10011010`), there is an *even* number of "1" bits. If the serial device sending this eight-bit data group were configured for "odd" parity, it would append an additional "1" to the end of that frame to make the total number of "1" bits odd rather than even. If the next data group were `11001110` instead (already having an odd number of "1" bits), the transmitting device would have to attach a "0" parity bit on to the data frame in order to maintain an odd count of "1" bits.

Meanwhile, the receiving device is programmed to count up all the "1" bits in each data frame (including the parity bit), and check to see that the total number is still odd (if the receiving device is configured for odd parity just as the transmitting device, which the two should *always* be in agreement). Unlike the transmitting device which is tasked with *creating* the parity bit state, the receiving device is tasked with *reading* all the data bits plus the parity bit to check if the count is still as it should be. If any one bit somehow gets corrupted during transmission, the received frame will not have the correct parity, and the receiving device will "know" something has gone wrong. Parity does not suggest *which* bit got corrupted, but it will indicate if there was a single-bit[13] corruption of data, which is better than no form of error-checking at all.

[13] It should take only a moment or two of reflection to realize that such a parity check cannot detect an *even* number of corruptions, since flipping the states of any *two* or any *four* or any *six* (or even all eight!) bits will not alter the evenness/oddness of the bit count. So, parity is admittedly an imperfect error-detection scheme. However, it is certainly better than no error detection at all!

The following example shows how parity-checking would work to detect a transmission error in a 7-bit data word. Suppose a digital device asynchronously transmits the character "T" using ASCII encoding ("T" = 1010100), with one start bit, one stop bit, and "odd" parity. Since the "start" bit is customarily a 0 state (space), the data transmitted in reverse order (LSB first, MSB last), the parity bit transmitted after the data's MSB, and the "stop" bit represented by a 1 state (mark), the entire frame will be the following sequence of bits: `0001010101`. Viewed on an oscilloscope display where a negative voltage represents a "mark" and a positive voltage represents a "space," the transmitted data frame will look like this:



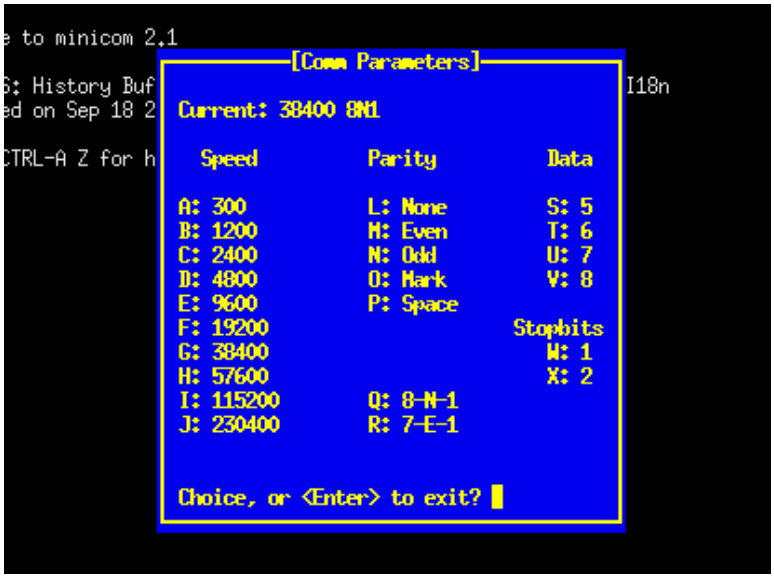
Note how the parity bit in this particular frame has been generated by the transmitting device as a 0 state, because the parity type is set for "odd," and the transmitting device realizes that the 7-bit data word already has an odd number of 1 bits in it and doesn't need another "1" for the parity bit. The pulse waveform you see above is how this data frame will be transmitted onto the network.

Now suppose this transmitted frame encounters a significant amount of electrical noise as it travels to the receiving device. If the frame reaches the receiver as shown in the next illustration, the receiving device will interpret the message incorrectly:

One of the bits has been corrupted by noise, such that the fifth transmitted data bit (which should be 1) is instead received as a 0. The receiving device, of course, has no knowledge of the noise present on the NRZ signal because all it "sees" is the "mark" or "space" states as interpreted by its input buffer circuitry. When the receiving device goes to count the number of 1 bits in the message (data plus parity bit, disregarding start and stop bits), however, it will count an even number of 1's instead of an odd number of 1's. Since the receiving device is also set for "odd" parity to match the transmitting device, it expects an odd number of 1's in the received message. Thus, it "knows" there is a problem somewhere in this transmission, because the received parity is not odd as it should be.

Parity-checking does not tell us *which* bit is corrupted, but it does indicate that *something* has gone wrong in the transmission. If the receiving device is programmed to take action on receipt of a non-matching parity, it may reply with a request for the transmitting device to re-send the data as many times as necessary until the parity is correct.

If we look at the "minicom" terminal screenshot again to analyze the parity options, we see there are several to choose from:



The five options for parity in this program include *None*, *Even*, *Odd*, *Mark*, and *Space*. "None" parity is self-explanatory: the transmitting device does not attach an extra bit for parity at all, and the receiving device does not bother to check for it. Since the inclusion of a parity bit does add to the bulk of a data frame, it has the unfortunate effect of slowing down communications (more bit "traffic" occupying the channel than would otherwise need to be), thus the option to waive parity altogether for a more compact (faster) data frame. "Even" and "Odd" parity options work as previously described, with the transmitting device adding a parity bit to each frame to bring the total "1" bit count either to an even number or to an odd number (depending on the user's configuration), and the receiving device checks for the same. "Mark" and "Space" are really of limited usefulness. In either of these two options, a parity bit is added, but the transmitting device does not bother to calculate the evenness or oddness of the data bits, rather simply making the parity bit always equal to 1 ("mark") or 0 ("space") as chosen by the user. The receiving device checks to see that the parity bit is always that value. These two options are of limited usefulness because the parity bit fails to reflect the status of the data being transmitted. The only corruption the receiving device can detect, therefore, is a corruption of the parity bit itself!

One will often find the communications parameters of a serial network such as this displayed in "shorthand" notation as seen at the top of the "minicom" terminal display: `38400 8N1`. In this case, the terminal program is configured for a bit rate of 38400 bits per second, with a data field 8 bits long, no parity bit, and 1 stop bit. A serial device configured for a bit rate of 9600 bps, with a 7-bit data field, odd parity, and 2 stop bits would be represented as `9600 7O2`.

3.6 Frame check sequences

Parity bits are not the only way to detect error, though. Some communication standards employ more sophisticated means called a *frame check sequence*, which is a collection of bits mathematically calculated by the transmitting device based on the content of the data. The mathematical calculation is called a *hash algorithm*, and it always generates an output with a fixed number of bits (called a *hash code* or a *digest* or simply a *hash*). In the IEEE 802.3 Ethernet standard, the hash algorithm is called a 32-bit Cyclic Redundancy Check, or *CRC32*[14]. Like a parity algorithm, the transmitting device processes the data bits to create the hash and then appends that hash to the end of the data field. The receiving device then takes the received data field and performs the exact same hash algorithm to see that its hash matches the one transmitted. As it is highly unlikely that any random corruption of bits during transmission would result in identical hash codes, this method is more robust for error-checking than parity.

An analogy for hashing is to envision someone sending a package to someone else in the mail, and being concerned that portions of the package's contents might be lost in transit. If the sender places the package on a precise weigh-scale and then writes the weight measurement on the package prior to sending, the receiving party will be able to weight the package on their own precise scale to see that their measurement agrees with the weight written by the sender on the package. If the receiving weight doesn't match the recorded weight of the package at sending time, something must be different. The weight measurement is the hash code, or digest, of the package. If included in the package's mailing label and used as a means of detecting loss, it would be the frame check sequence.

Like parity, frame check sequences do not indicate *where* the errors lie, and also like parity they are imperfect. A chance always exists that just the right combination of errors may occur in transmission causing the frame check sequence values at both ends to match even though the data is not identical[15], but this is highly unlikely (calculated to be one chance in 10^{14} for Ethernet's 32-bit CRC). It is certainly better than having no error detection ability at all.

If the communications software in the receiving device is configured to take action on a detection of error, it may return a "request for re-transmission" to the transmitting device, so the corrupted message may be re-sent. This is analogous to a human being hearing a garbled transmission in a telephone conversation, and subsequently requesting the other person repeat what they just said.

[14] Several web-based (online) CRC32 generators exist which you may experiment with. Also, stronger has algorithms such as SHA256 are commonly available as command-line tools in operating systems such as Linux.

[15] This is called a *hash collision*.

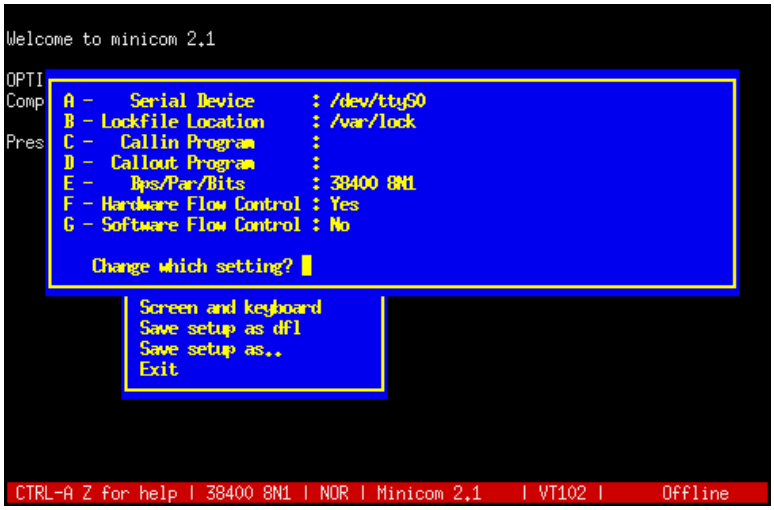
3.7 Flow control

Another option often found in serial data communications settings is something called *flow control*. In the context of digital communications, "flow control" refers to the ability of a receiving device to request a reduction in speed or even a complete cessation of data transmission if the speed of the transmitted data is too fast for the receiving device to keep pace. An example common to personal computers is that of a mechanical printer: while the computer may be able to transmit data to be printed at a very rapid pace, the printer is limited by the speed of its printing mechanism. In order to make the printing process go more smoothly, printers are equipped with *buffer memory* to store portions of the print job received from the transmitting computer that have not had time to print yet. However, these buffers are of finite size, and may become overwhelmed on large print jobs. So, if and when a printer detects its buffer near full capacity, it may issue a command to the computer to freeze serial data transmission until the printer's buffer has had some time to empty. In other words, the printer can send a message to the computer saying "Stop!" when its buffer is full, then later send another message saying "Resume" when its buffer is empty enough to resume filling. Thus, the receiving device has control over the flow of data necessary to manage its buffer resources.

Flow control in serial networks may take place in either *hardware* mode or *software* mode. "Hardware" mode refers to the existence of additional connector pins and cable conductors specifically designated for such "halt" and "resume" signals. "Software" mode refers to data codes communicated over the regular network channel telling the transmitting device to halt and resume. Software flow control is sometimes referred to as XON/XOFF in honor of the names given to these codes[16]. Hardware flow control is sometimes referred to as RTS/CTS ("Request To Send" and "Clear To Send" respectively) in honor of the labels given to the serial port pins conveying these signals. In a hardware flow-controlled system the device intending to transmit first asserts an RTS signal which is then received by the other device, that other device in turn responding with an asserted CTS signal indicating it is ready (i.e. "clear") to begin receiving data.

[16] The "XOFF" code tells the transmitting device to halt its serial data stream to give the receiving device a chance to "catch up." In data terminal applications, the XOFF command may be issued by pressing the key combination <Ctrl><S>. This will "freeze" the stream of text data sent to the terminal by the host computer. The key combination <Ctrl><Q> sends the "XON" code, enabling the host computer to resume data transmission to the terminal.

The following screen shot shows options for flow control in the "minicom" terminal program:



Here, you can see "hardware" flow control enabled and "software" flow control disabled. The enabling of "hardware" flow control means the serial communications cable must be equipped with the necessary lines to convey these handshaking signals (when needed) between devices. Software flow control tends to be the more popular option, the advantage of this of course being fewer conductors necessary in the serial data cable. The disadvantage of using software flow control over hardware is a slight inefficiency in data throughput, since the XON and XOFF commands require time to be transmitted serially over the same network as the rest of the data.

3.8 Channel arbitration

When two or more communication devices exchange data, the directions of their communication may be classified into one of two categories: *simplex* or *duplex*. A "simplex" network is one-way communication only. A sensor outputting digital data to a remotely-located indicator over a digital network would be an example of simplex communication, where the flow of information goes from sensor to indicator, and never the other direction. A public-address (PA) system is an analog example of a simplex communication system, since audio information only goes in one direction (from the person with the microphone to the audience).

"Duplex" communication refers to two-way data exchange. Voice telephony is an analog example of two-way (duplex) communication, where either person at the end of the connection can hear the other person talking. Duplex communication may be further subdivided into *half-duplex* and *full-duplex*, referring to whether or not the two-way communication may be simultaneous. In a "full-duplex" system, both devices may transmit data to each other simultaneously because they have separate channels (separate wires, or optical fibers, or radio frequencies) for their respective transmissions. In a "half-duplex" system, only one device may transmit at any time because the devices must share a common channel. A telephone system is an example of a full-duplex system, although it may be rather difficult for the people to understand each other when they are speaking over one another. A push-to-talk radio system ("walkie-talkie") is an example of a half-duplex system, where each person must take turns talking.

In half-duplex systems, there must be some way for the respective devices to "know" when they are allowed to transmit. If multiple devices sharing one communications channel attempt to transmit simultaneously, their messages will "collide" in such a way that no device on the network will be able to interpret either message. The problem is analogous to two people simultaneously pressing the "talk" buttons on their two-way radio units: neither of the talking people can hear each other, and anyone else on the same channel hears the garbled amalgam of those two peoples' superimposed transmissions. In order to avoid this scenario in a half-duplex network, there must be some strategy to coordinate transmissions so only one device "talks" at any given time. The problem of deciding "who" gets to "talk" at any given time is generally known as *channel arbitration*. Several strategies for addressing this problem have been developed in the data communications field, a few of which will be described in this subsection.

3.8.1 Master-slave

Our first method works on the principle of having only one device on the network (the "master") with permission to arbitrarily transmit data. All other devices on the network are "slaves," which may only respond in direct answer to a query from the master. If the network happens to be simplex in nature, slave devices don't even have the ability to transmit data – all they can do is "listen" and receive data from the one master device.

For example, in a half-duplex master-slave network, if one slave device has data that needs to be sent to another slave device, the first slave device must wait until it is prompted ("polled") by the master device before it is allowed to transmit that data to the network. Once the data is transmitted, any and all slave devices may receive that transmission, since they all "listen" to the same communications channel.

An example of a master-slave industrial network is a *Modbus* network connecting a programmable logic controller (PLC) to multiple variable-frequency motor drives (VFDs). The master device (the PLC) initiates all communications, with the slave devices (the motor drives) at most replying to the PLC master (and in many cases not replying at all, but merely receiving data from the PLC in simplex mode).

Master-slave arbitration is simple and efficient, but suffers from one glaring weakness: if the master device happens to fail, all communication on the network halts. This means the ability of *any* device on the network to transmit information utterly depends on the proper function of *one* device, representing a high level of dependence on that one (master) device's function.

Some master-slave networks address this problem by pre-assigning special "back-up" status to one or more slave devices. In the event that the master device fails and stops transmitting for a certain amount of time, the back-up device becomes "deputized" to act as the new master, taking over the role of the old master device by ensuring all slave devices are polled on schedule.

3.8.2 Token-passing

Another method of arbitrating which device gets to transmit on a channel in a half-duplex network is the *token-passing* method. Here, a special data message called the "token" serves as temporary authorization for each device to transmit. Any device in possession of the token is allowed to act as a master device, transmitting at will. After a certain amount of time, that device must relinquish the token by transmitting the token message on the network, complete with the address of the next device. When that other device receives the token message, it switches into master mode and transmits at will. The strategy is not unlike a group of people situated at a table, where only one of them at a time holds some object universally agreed to grant speaking authority to the holder.

Token-passing ensures only one device is allowed to transmit at any given time, and it also solves the problem inherent to master-slave networks of what happens when the master device fails. If one of the devices on a token-passing network fails, its silence will be detected after the last token-holding device transmits the token message to the failed device. After some pre-arranged period of time, the last token-holding device may re-transmit the token message to the *next* device after the one that failed, re-establishing the pattern of token sharing and ensuring all devices get to "speak" their turn once more.

Examples of token-passing networks include the general-purpose Token Ring network standard (IEEE 802.5) and the defunct Token Bus (IEEE 802.4).

Token-passing networks require a substantially greater amount of "intelligence" built into each network device than master-slave requires. The benefits, though, are greater reliability and a high level of bandwidth utilization. That being said, token-passing networks may suffer unique disadvantages of their own. For example, there is the question of what to do if such a network becomes severed, so that the one network is now divided into two segments. At the time of the break, only one device will possess the token, which means only one of the segments will possess any token at all. If this state of affairs holds for some time, it will mean the devices lucky enough to be in the segment that still has the token will continue communicating with each other, passing the token to one another over time as if nothing was wrong. The isolated segment, however, lacking any token at all, will remain silent even though all its devices are still in working order and the network cable connecting them together is still functional. In a case like this, the token-passing concept fares no better than a master-slave network. However, what if the designers of the token-passing network decide to program the devices to automatically generate a new token in the event of prolonged network silence, anticipating such a failure? If the network becomes severed and broken into multiple segments, the isolated segments will now generate their own tokens and resume communication between their respective devices, which is certainly better than complete silence as before. The problem now is, what happens if a technician locates the break in the network cable and re-connects it? Now, there will be *multiple* tokens on one network, and confusion will reign!

Another example of a potential token-passing weakness is to consider what would happen to such a network if the device in possession of the token failed before it had an opportunity to relinquish the token to another device. Now, the entire network will be silent, because no device possesses the token! Of course, the network designers could anticipate such a scenario and pre-program the devices to generate a new token after some amount of silence is detected, but then this raises the possibility of the previously-mentioned problem when a network becomes severed and multiple tokens arise in an effort to maintain communication in those isolated network segments, then at some later time the network is re-connected and now multiple tokens create data collision problems.

3.8.3 TDMA

A method of channel arbitration similar to token-passing is *TDMA*, or "Time Division Multiple Access." Here, each device is assigned an absolute "time slot" in a repeating schedule when it alone is allowed to transmit. With token-passing, permission to transmit is granted to each device by the previous device as it relinquishes the token. With TDMA, permission to transmit is granted by an appointment on a fixed time schedule. TDMA is less time-efficient than token-passing because devices with no data to transmit still occupy the same amount of time in the schedule as when they have data to transmit. However, TDMA has the potential to be more tolerant of device failure and network segmentation than token-passing because neither the failure of a device nor segmentation of the network can prevent remaining devices from communicating with each other. If a device fails (becomes "silent") in a TDMA network, that time slot simply goes unused while all other communication continues unabated. If the network becomes severed, each set of devices in the two segments will still follow their pre-programmed time schedule and therefore will still be able to communicate with each other.

An example of a TDMA network includes the legacy 2G (GSM) cellular telephone standard, which used TDMA as part of a larger strategy to manage access between multiple cell phones and cell towers. TDMA arbitration works very well for wireless (radio) networks where the communication channel is inherently unreliable due to physical obstacles. If a device on a TDMA wireless network falls out of range or becomes blocked, the rest of the network carries on without missing a step.

Practical TDMA networks are not quite as fault tolerant as the idealized vision of TDMA previously described. Real TDMA networks do depend on some "master" device to assign new time slots and also to maintain synchronization of all device clocks so that they do not "lose their place" in the schedule. If this master device fails, the TDMA network will lose the ability to accept new devices and will (eventually) lose synchronization.

In light of this fact, it might appear at first that TDMA is no better than master-slave arbitration, since both ultimately depend on one master device to manage communication between all other devices. However, TDMA does offer one significant benefit over master-slave, and that is more efficient use of time. In a master-slave network, the master must poll each and every device on the network to check if it has data to transmit. This polling requires additional network time beyond that required by the "slave" devices to report their data. In a TDMA network, the master device need only occupy time transmitting to the network when updating time-slot assignments and when broadcasting time synchronization messages. You can think of TDMA as being a "smarter" version of master-slave arbitration, where the devices need only be told once when they may transmit, rather than having to be told every single time when they may transmit.

3.8.4 CSMA

A completely different method of channel arbitration is where any and all devices have permission to transmit when the network is silent. This is generally called *CSMA*, or "Carrier Sense Multiple Access." There are no dedicated master and slave devices with CSMA, nor are devices permitted to transmit in a pre-determined order as with token-passing or in a pre-determined schedule as with TDMA. *Any* device on a CSMA network may "talk" in any order and at any time whenever the network is free. This is analogous to an informal conversation between peers, where anyone is permitted to break the silence.

Of course, such an egalitarian form of channel arbitration invites instances where two or more devices begin communicating simultaneously. This is called a *collision*, and must be addressed in some manner in order for any CSMA network to be practical.

Multiple methods exist to overcome this problem. Perhaps the most popular in terms of number of installed networks is *CSMA/CD* ("Carrier Sense Multiple Access with Collision Detection"), the strategy used in Ethernet. With CSMA/CD, all devices are not only able to sense an idle channel, but are also able to sense when they have "collided" with another transmitting device. In the event of a collision, the colliding devices both cease transmission, and set random time-delays to wait before re-transmission. The individual time delays are randomized to decrease the probability that a re-collision between the same devices will occur after the wait. This strategy is analogous to several peers in one group holding a conversation, where all people involved are equally free to begin speaking, and equally deferential to their peers if ever two or more accidently begin speaking at the same time. Occasional collisions are normal in a CSMA/CD network, and should not be taken as an indication of trouble unless their frequency becomes severe.

A different method of addressing collisions is to pre-assign to each device on the network a priority number, which determines the order of re-transmission following a collision. This is called *CSMA/BA*, or "Carrier Sense Multiple Access with Bitwise Arbitration," and it is analogous to several people of different social levels in one group holding a conversation. All are free to speak when the room is silent, but if two or more people accidently begin speaking at the same time, the person of highest "rank" is allowed to continue while the "lower-rank" person(s) must wait. This is the strategy used in CAN Bus technology, one of the more popular data networks for digital automotive subsystems.

Some CSMA networks lack the luxury of collision detection, and must therefore strive to prevent collisions rather than gracefully recover from them. Wireless digital networks are an example where collision detection is not an option, since a wireless (radio) device having a single antenna and a single channel cannot "hear" any other devices' transmissions while it is transmitting, and therefore cannot detect a collision if one were to occur. A way to avoid collisions for such devices is to pre-assign each device on the network with a priority number, which determines how long each device is forced to wait after detecting a "quiet" network before it is allowed to transmit a new message. So long as no two devices on the network have the same "wait" time, there will be no collisions. This strategy is called *CSMA/CA*, or "Carrier Sense Multiple Access with Collision Avoidance," and is the technique used for WLAN networks (the IEEE 802.11 specification). A consequence of collision avoidance, though, is unequal access to the network. Those devices with higher-priority (shorter wait times) will always have an advantage in transmitting their data over devices of lower priority. The degree of disparity in network access grows as more devices occupy the network. CSMA/CA is analogous to a group of shy people talking, each person afraid to speak at the same time as another,

and so each person waits a different amount of time following the conclusion of the last utterance before daring to speak. This sort of ultra-polite behavior may ensure no one accidently interrupts another, but it also means the shiest person will hardly ever get a chance to speak.

One characteristic distinguishing all CSMA networks from master-slave, token-passing, and TDMA networks is a lack of *determinism*. "Determinism" is the ability of a given arbitration method to guarantee every device on the network is able to transmit within a specified maximum time. That is to say, a "deterministic" network ensures no device will ever be forced to wait longer than some specified maximum amount of time before it can transmit. A master-slave or TDMA network following a repeating schedule guarantees that the wait time will never be longer than the period of the cycle, and therefore is deterministic. A token-passing network following a definite order guarantees that the wait time will never take longer than the number of devices times the maximum time each device may hold the token, and so is also deterministic. A CSMA network, however, offers no such guarantee for response time and is therefore a "non-deterministic" network. At least hypothetically, any device on a CSMA network may be prevented indefinitely from transmitting its message if it keeps being blocked by other devices' transmissions (the one highest-priority device in a CSMA/BA or CSMA/CA network being an exception of course). Determinism is important in industrial control systems where communication delays may adversely affect the stability of a closed-loop feedback control, and it is especially important in safety control systems where fast action is needed to avert catastrophe.

A potential problem in any digital network, but particularly networks employing CSMA arbitration, is something known as *jabbering*. If a network device happens to fail in such a way that it ceaselessly transmits a signal on the network, none of the other CSMA devices will ever be allowed to transmit because they continuously detect a "carrier" signal from the jabbering device[17]. Some Ethernet components sport *jabber latch* protection circuits designed to detect jabber and automatically cut off the offending device from the network, or employ "store-and-forward" buffering which is able to block jabbered data frames.

[17] I once encountered this very type of failure on the job, where a copper-to-fiber adapter on a personal computer's Ethernet port jammed the entire network by constantly spewing a meaningless stream of data. Fortunately, indicator lights on all the channels of the communications equipment clearly showed where the offending device was on the network, allowing us to take it out of service for replacement.

3.9 The OSI Reference Model

Digital data communication may be described in many ways. For example, a connection formed between two computers to exchange a text document is a multi-layered activity, involving many steps to convert human language into electrical impulses for transmission, then re-convert those electrical impulses into human language again at the receiving end. Not surprisingly, there usually exist many different ways to perform this same task: different types of networks, different encodings, different communications and presentation software, etc.

To illustrate by analogy, think of all the actions and components necessary to transport items using an automobile. In order to move furniture from an apartment to a house, for example, you would require the following:

- An appropriate vehicle
- Addresses or directions for both locations
- A driver's license and knowledge of driving rules
- Fuel for the vehicle
- Knowledge of how to safely stack furniture for transport
- Knowledge of how the furniture is to be placed in the house

These details may seem trivial to mention, as human beings familiar with the common task of moving personal belongings from one location to another, but imagine having to describe every single action and component to someone from a primitive culture ignorant of vehicles, addresses, maps, driver's licenses, fuel, etc. One way to help describe all this complexity would be to assign different people to different layers of detail. For example, an automotive engineer could discuss the details of how engines burn fuel to do mechanical work (propelling the vehicle) while a furniture loader could describe how furniture is to be loaded and taken off the vehicle. A driving instructor could then explain all the procedures of safely driving the vehicle, while a city planner could explain the organization of streets and addresses in relation to a city map. Finally, an interior decorator could wax eloquent on the proper placement of furniture in the house. Each person would be describing a different aspect of the furniture move, each one of those aspects being important to the overall goal of moving furniture from one location to another.

Moreover, for each one of the aspects described by a specialist, there may exist several different alternatives. For example, there are many different models and designs of vehicle one might use for the job, and there may be different driving rules depending on where the two locations are for the move. Addresses and directions will *certainly* vary from city to city, and even within one city there will be alternative routes between the two locations. Finally, there is virtually no end to arrangements for furniture at the destination house, each one with its own functional and esthetic merits.

By the same token, the task of transporting digital data may be divided into similar categories. In order to move and process data from one computer to another, you need the following:

- An appropriate cable (or radio link) connecting the two computers
- Standardized electrical signals to represent bit states
- Addresses for each computer on the network
- Algorithms specifying how each computer takes turns "talking" on the common network
- Algorithms specifying how to organize packets of data to be sent and received serially
- Software to format the data on the transmitting end and interpret the data on the receiving end

Each of these aspects is important to the overall goal of creating, moving, and interpreting digital data between two or more computers, and there are many alternative methods (standards) for each aspect. We may represent 0 and 1 bits using NRZ (Non-Return to Zero) encoding, Manchester encoding, FSK modulation, etc.; the signals may be electrical or they may be optical or they may even be radio waves; the options for electrical cables and connector types are many. Bits may be framed differently as they are packaged for transmission, and arbitration between devices on the network managed in a variety of different ways. How we address multiple devices on a network so messages get routed to their proper destinations is important as well.

A scheme originally intended as a formal standard, but now widely regarded as a general model to describe the portions of other standards, helps us clarify the complexity of digital communications by dividing communication functions into seven[18] distinct "layers." Developed by the *ISO* (International Organization for Standards)[19] in 1983, the *OSI Reference Model* divides communication functions into the following categories, shown in this table with examples:

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

The vast majority of digital networking standards in existence address mere portions of the 7-layer model. Any one of the various Ethernet standards, for example, applies to layers 1 and 2, but none of the higher-level layers. In other words, Ethernet is a means of encoding digital information in electronic form and packaging that data in a standard format understandable to other Ethernet devices, but it provides no functionality beyond that. Ethernet does *not* specify how data will be routed over large-area networks, how to manage data-exchange sessions between computers (opening connections, initiating data transfer, closing connections), nor how to format the data to

[18] An additional layer sometimes added to the OSI model is layer 8, representing either the human user of the network system or the physical process interfacing with the network system. If the purpose of this model is to describe all the functioning portions of a communications link in the context of a system used for some practical purpose, layer 8 represents an essential part of that system and should not be ignored.

[19] If you are thinking the acronym should be "IOS" instead of "ISO," you are thinking in terms of English. "ISO" is a non-English acronym!

represent real-world variables and media. Common serial network standards such as EIA/TIA-232 and EIA/TIA-485 don't even go that far, being limited mostly to layer 1 concerns (signal voltage levels, wiring, and in some cases types of electrical connectors). For example, EIA/TIA-485 does not specify how to address multiple devices connected to a common electrical network – all it does is specify what voltage levels represent "0" and "1" bits.

By contrast, some other digital networking standards specify nothing about lower-level layers, instead focusing on high-level concerns. Modbus, for example, is concerned only with layer 7, and not with any of the lower-level layers. This means if two or more devices on a network use "Modbus" to communicate with each other, it refers only to the high-level programming codes designed to poll and interpret data within those devices. The actual cable connections, electrical signals, and communication techniques used in that "Modbus" network may vary widely. Anything from EIA/TIA-232 to Ethernet to a wireless network such as WLAN may be used to actually communicate the high-level Modbus instructions between devices. Similarly, the ASCII standard strictly defines alphanumerical and control characters may be presented as seven-bit digital words (layer 6), but says nothing about how that data should be physically represented (layer 1), organized into frames of bits (layer 2), routed to different locations (layer 3), etc.

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 Ancient serial data communication

The representation and communication of text by discrete (on/off) signals is rooted in ancient practice. The Greek historian Polybius described one such system in his *Histories* written in the second century BCE, used to communicate information about military maneuvers. In this passage Polybius describes the problem posed by primitive fire signals, and presents an improved method:

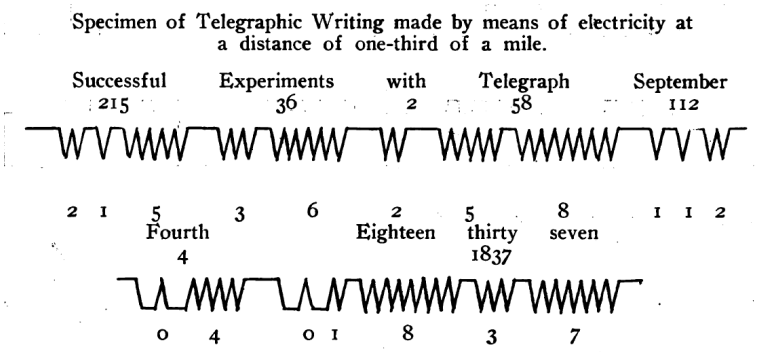
> It is evident to all that in every matter, and especially in warfare, the power of acting at the right time contributes very much to the success of enterprises, and fire signals are the most efficient of all the devices that aid us to do this. For they show what has recently occurred and what is still in the course of being done, and by means of them anyone who cares to do so even if he is at a distance of three, four, or even more days' journey can be informed. So that it is always surprising how help can be brought by means of fire messages when the situation requires it. Now in former times, as fire signals were simple beacons, they were for the most part of little use to those who used them. It was possible for those who had agreed to convey a signal that a fleet had arrived in Oreus, Peparethus, or Chalcis, but when it came to some of the citizens having changed sides or having been guilty of treachery or a massacre having taken place in town, or anything of the kind, things that often happen, but cannot all be foreseen – and it is chiefly unexpected occurrences which require instant consideration and help – all such matters defied communication by fire signal. It was quite impossible to have a preconceived code for things which there was no means of foretelling.
>
> This is the vital matter; for how can anyone consider how to render assistance if he does not know how many of the enemy have arrived, or where? And how can anyone be of good cheer or the reverse, or in fact think of anything at all, if he does not understand how many ships or how much corn has arrived from the allies?
>
> The most recent method, devised by Cleoxenus and Democleitus and perfected by myself, is quite definite and capable of dispatching with accuracy every kind of urgent messages, but in practice it requires care and exact attention. It is as follows: We take the alphabet and divide it into five parts, each consisting of five letters. Each of the two parties who signal to each other must get ready five tablets and write one division of the alphabet on each tablet. The dispatcher of the message will raise the first set of torches on the left side indicating which tablet is to be consulted; i.e., one torch if it is the first, two if it is the second, and so on. Next he will raise the second set on the right on the same principle to indicate what letter of the tablet the receiver should write down.

Note the last sentence in the first paragraph: *"It was quite impossible to have a preconceived code for things which there was no means of foretelling."* One could, for example, imagine a fire-signal code consisting of a few distinct patterns of torches, each one encoding a specific military event (e.g. enemy approaching, surrender, victory, retreat, mutiny, riot, type of weapon(s) employed). However, to be of any practical use the number of specific codes would have to be extremely large. Instead, the code described by Polybius neatly handles this problem by leveraging another pre-existing code: *written language*. This way the fire beacons need only encode characters of their alphabet, and with those characters they could describe *any* military event encoded in their written language. The only limitations then would be vocabulary and the speed at which messages could be communicated.

This compounding of codes is very common in digital communication systems. For example, the ASCII code is useful for encoding English alpha-numerical symbols as seven-bit binary words, but when transmitting digital data over long distances we must devise ways to encode the bits themselves as physical signals (e.g. pulses of voltage, current, light, sound, shifting frequencies, etc.). For the sake of argument, we could even employ Polybius' crude fire-torches as the "physical layer" of encoding for an ASCII character set! Once received and decoded, the ASCII symbols could themselves be interpreted as a code representing something other than text. The *Modbus ASCII* standard is a good example of this, where multiple ASCII characters are interpreted to represent "frames" of digital data which could be used to represent any arbitrary information, from numbers to letters to machine instructions to measurements, etc.

4.2 The original telegraph code

Samuel Morse, inventor of the telegraph, actually did not use what is known today as *Morse Code* to transmit his first messages. Instead, his original method of encoding was based on decimal numbers, with specific words represented by unique numbers. Page 29 of the book *Early History of the Electro-Magnetic Telegraph* by Alfred Vail documents a typical message communicated using Morse's system:



The *words* in the diagram were the intelligence transmitted.

The *numbers* (in this instance arbitrary) are the numbers of the words in a telegraphic dictionary.

The *points* are the markings of the register, each point being marked every time the Electric fluid passes.

The register marks but one kind of mark to wit, (V) this can be varied two ways. By intervals, thus, (V VV VVV), signifying one, two, three, etc., and by reversing, thus, (Λ). Examples of both these varieties are seen in the diagram.

The single numbers are separated by *short,* and the whole numbers by *long intervals.*

The jagged lines form an oscillograph of the telegraph circuit's digital state over time, and as you can see it used groups of successive pulses to count digits of decimal numbers. For example, the number 215 was encoded as two pulses followed by a brief rest, then one pulse followed by another rest, then a sequence of five pulses. A longer rest separated individual numbers from each other, as opposed to the shorter rests which merely separated digits of the same number from each other.

A "telegraphic dictionary" served to cross-reference common words with number values. As you can see from the example shown, the number 215 represented the word *Successful*, while 36 represented *Experiments*, 2 represented *With*, and so on. One can only imagine how many numbers would have been required to represent even a rudimentary cross-section of the English vocabulary!

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 International Morse Code

The following shows a subset of the International Morse Code, the full version containing many non-English characters and other punctuation marks. The full standard for International Morse Code is Recommendation ITU-R M.1677-1 published by the Radiocommunication Sector of the International Telecommunications Union (ITU-R).

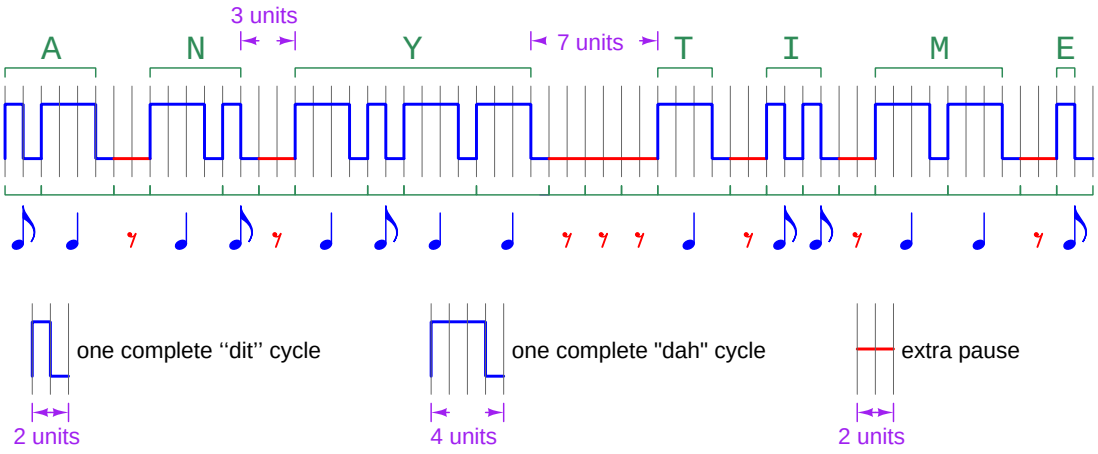
International Morse Code (English letters, Arabic numerals, some symbols)

A •—	J •———	S •••	0 —————
B —•••	K —•—	T —	1 •————
C —•—•	L •—••	U ••—	2 ••———
D —••	M ——	V •••—	3 •••——
E •	N —•	W •——	4 ••••—
F ••—•	O ———	X —••—	5 •••••
G ——•	P •——•	Y —•——	6 —••••
H ••••	Q ——•—	Z ——••	7 ——•••
I ••	R •—•		8 ———••
			9 ————•

? ••——•• *(Question mark)*	, ——••—— *(Comma)*	/ —••—• *(Slash)*	- —••••— *(Hyphen)*
& •—••• *(Ampersand)*	: ———••• *(Colon)*	. •—•—•— *(Period)*	" •—••—• *(Quotation mark)*
× —••— *(Multiplication)*	+ •—•—• *(Addition)*	= —•••— *(Equals)*	•••—•— *(End of work)*
(—•——• *(Left parenthesis)*	) —•——•— *(Right parenthesis)*	′ •————• *(Apostrophe)*	•••••••• *(Backspace delete)*

According to Recommendation ITU-R M.1677 from the ITU Radiocommunication Assembly in 2004, a dash (sounded as "dah") should have a tone duration three times that of a dot (sounded as "dit"), the time delay between dits and dahs within the same character should last as long as the audible duration of a single dit, the time delay between characters should last as long as the audible duration of a dah (three dits), and the space between two words should last as long as the audible duration of seven dits.

If we model each dit as a pulse with a 50% duty cycle (on as long as it is off) and a dah as a pulse with a 75% duty cycle (on three times as long as it is off), we see that the total on/off cycle time of a dah is exactly *twice* the duration of the total on/off cycle time of a dit. As such, we may model dits and dahs and the extra pauses demarcating letters and words using standard musical notations of eighth- and quarter-notes. Consider the following example of the phrase `ANY TIME`, where the smallest subdivision of time (marked by the thin grey vertical lines) is the audible duration of a single dit:



With practice, one's sending of Morse words and phrases will have an almost musical feel. In fact, it is possible to practice sending Morse Code transmissions to the beat of a metronome in order to help learn these precise timing relationships. Getting the pause (rest) durations correct is the most challenging part of this. In real-world practice, if one must compromise on timing accuracy while sending Morse Code, it is better to err on the side of having too much silence between letters and words than too little! The intentional practice of lengthening pause times for easier decoding of individual characters is called *Farnsworth timing*.

Lest anyone think that Morse Code is obsolete, know that this form of communication still enjoys a vibrant following within the amateur radio community for its elegant simplicity as well as for its readability even with low signal-to-noise ratios[1]. Morse Code (or "CW" as amateur radio operators refer to it) is without a doubt the most robust and bandwidth-efficient modulation mode

[1] I have personally experienced the difference between single-sideband telephony (SSB phone) and CW in applications where a voice signal was completely unintelligible but CW pulses sent on the same frequency by the same transmitter were clear and unmistakable. Also, it is routine for low-power (QRP) radio operators to communicate with other people thousands of kilometers away on HF bands using CW mode on transmitters having just 5 Watts of output power, sent through makeshift long-wire antennas!

that human beings are able to directly interpret. Transmitter and receiver hardware for CW radio communication is also far simpler than for any other modulation mode, a crude CW transmitter being little more than an RF oscillator combined with an on/off switch.

If you are interested in learning Morse Code, expect to spend about a week's worth of time learning to send all the alphanumeric characters. Beware that learning to *receive* Morse Code, especially at speed, takes longer than this!

5.2 ASCII character codes

ASCII characters consist of seven-bit digital words. The following table shows all 128 possible combinations of these seven bits, from 0000000 (ASCII "NUL" character) to 1111111 (ASCII "DEL" character). For ease of organization, this table's columns represent the most-significant three bits of the seven-bit word, while the table's rows represent the least-significant four bits. For example, the capital letter "C" would be encoded as 1000011 in the ASCII standard.

↓ LSB / MSB →	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	−	=	M	]	m	}
1110	SO	RS	.	>	N	ˆ	n	˜
1111	SI	US	/	?	O	_	o	DEL

It is worth noting that the ASCII codes for the Arabic numerals 0 through 9 are simply the four-bit binary representation of those numbers preceded by 011. For example, the number six (0110) is represented in ASCII as 0110110; the number three (0011) in ASCII as 0110011; etc. This is useful to know, for example, if you need to program a computer to convert single decimal digits to their corresponding ASCII codes: just take each four-bit numerical value and add forty-eight (0x30 in hexadecimal) to it.

5.3 Hash algorithms

A *hash algorithm* is a mathematical/logical procedure by which a large set of data generates a result of fixed-bit-width that is fairly unique. We see hash algorithms applied as error-checking mechanisms in such communication standards as IEEE 802.3 Ethernet.

Interestingly, hash algorithms are applicable to more than just transmitted data in a digital network. They may also be used to validate the integrity of *any* collection of digital data, including *files* existing within a computer's memory or stored on a digital drive. Comparisons of hash codes for such data would then reveal tampering or corruption of that data[2].

To demonstrate, we will use the same frame check sequence algorithm used by Ethernet (called *CRC32*) to process a dataset consisting of ASCII text characters from the following passage taken from Henry David Thoreau's book *Walden*:

> If I wished a boy to know something about the arts and sciences, for instance, I would not pursue the common course, which is merely to send him into the neighborhood of some professor, where anything is professed and practised but the art of life; – to survey the world through a telescope or a microscope, and never with his natural eye; to study chemistry, and not learn how his bread is made, or mechanics, and not learn how it is earned; to discover new satellites to Neptune, and not detect the motes in his eyes, or to what vagabond he is a satellite himself; or to be devoured by the monsters that swarm all around him, while contemplating the monsters in a drop of vinegar. Which would have advanced the most at the end of a month – the boy who had made his own jackknife from the ore which he had dug and smelted, reading as much as would be necessary for this – or the boy who had attended the lectures on metallurgy at the Institute in the meanwhile, and had received a Rodgers' penknife from his father? Which would be most likely to cut his fingers?... To my astonishment I was informed on leaving college that I had studied navigation! – why, if I had taken one turn down the harbor I should have known more about it. Even the poor student studies and is taught only political economy, while that economy of living which is synonymous with philosophy is not even sincerely professed in our colleges. The consequence is, that while he is reading Adam Smith, Ricardo, and Say, he runs his father in debt irretrievably.

[2] I once used an industrial control system that relied on UV-erasable EPROM memory ICs to store the code defining the system's functions. These EPROM chips were susceptible to data loss by accidental exposure to light, and so as a precautionary measure we used the programming device to generate a hash (which we called a "checksum") and hand-wrote that hash code on the piece of tape we used to cover up the glass UV-exposure window on the IC. If ever there was doubt that a particular EPROM chip had become corrupted, we could place that chip into the programmer machine's DIP socket and re-run the hash algorithm to see if the hash code it generated still matched what was written on the tape. A mis-match indicated data corruption.

Applying the CRC32 hash algorithm to this text[3] generates the following digest:

```
a1c544e8
```

Now, to demonstrate the sensitivity of the hash algorithm to even the smallest change, consider the result when we alter a single character from Thoreau's text (changing the question-mark symbol following the word *fingers* into a blank space character, this substitution representing a single-bit difference between the ASCII characters: `0100000` for the space character versus `0100001` for the exclamation point) and re-run the CRC32 hash algorithm on that edited text:

```
9719f081
```

Of course, these disparate hash codes tell us nothing about the location or extent of the data corruption, only that *some* corruption occurred and a message re-transmission is necessary.

[3]Note that CRC32 is not the only hash algorithm in existence – it just happens to be simple and fast to compute which is why it was chosen as the frame check sequence algorithm for Ethernet. The relatively high speed of Ethernet serial communication requires a hash algorithm that is simple enough to be executed for every single Ethernet frame by end-devices having limited computing power. An example of a more robust hash algorithm is *SHA1*, which is simple to test on any computer with a command-line interface and the necessary SHA1 application installed. Simply save the data to a file (e.g. `walden.txt`) and then type the command: `sha1sum walden.txt` to see the hash (digest) printed to the console. For the *Walden* passage example, the original SHA1 digest is `d48055e2065e9324fefb0ea54747d0990e855041` and the SHA1 digest for the corrupted passage is `14f0df7536e301978148d6d46ecf8cff7c6edb35` which as you can see is quite different.

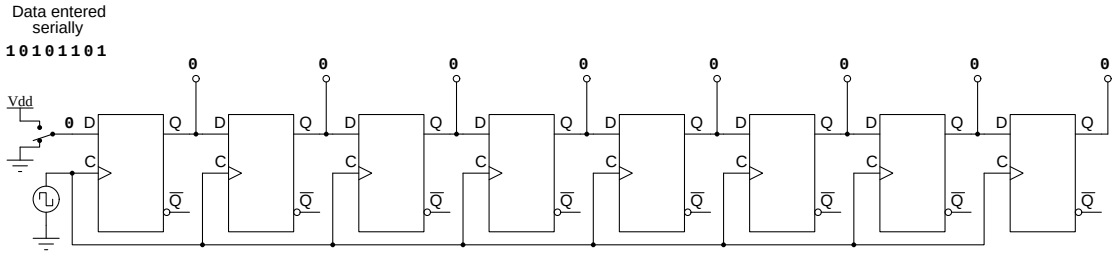
Chapter 6

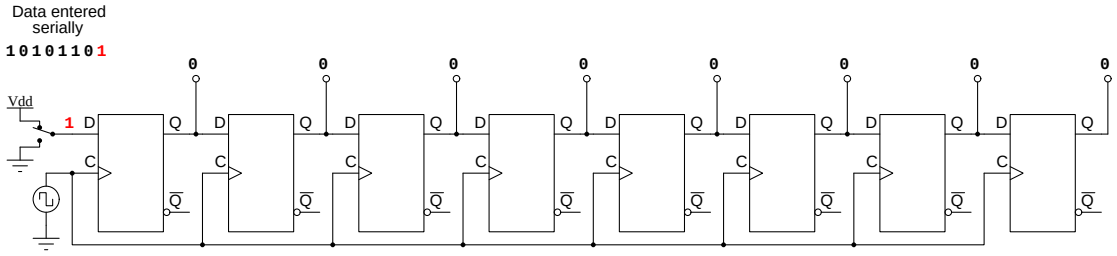
Animations

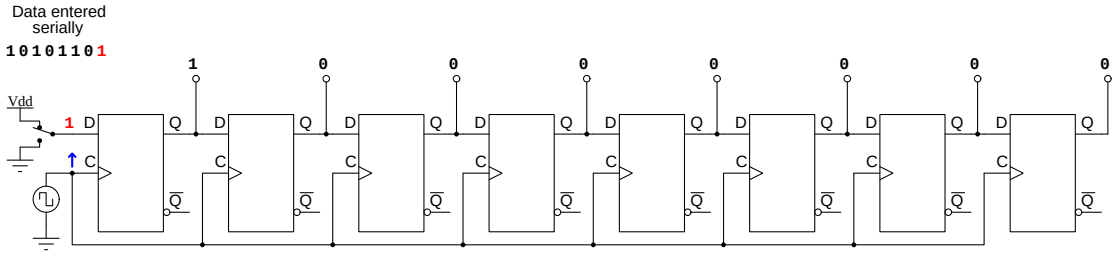
Some concepts are much easier to grasp when seen in *action*. A simple yet effective form of animation suitable to an electronic document such as this is a "flip-book" animation where a set of pages in the document show successive frames of a simple animation. Such "flip-book" animations are designed to be viewed by paging forward (and/or back) with the document-reading software application, watching it frame-by-frame. Unlike video which may be difficult to pause at certain moments, "flip-book" animations lend themselves very well to individual frame viewing.

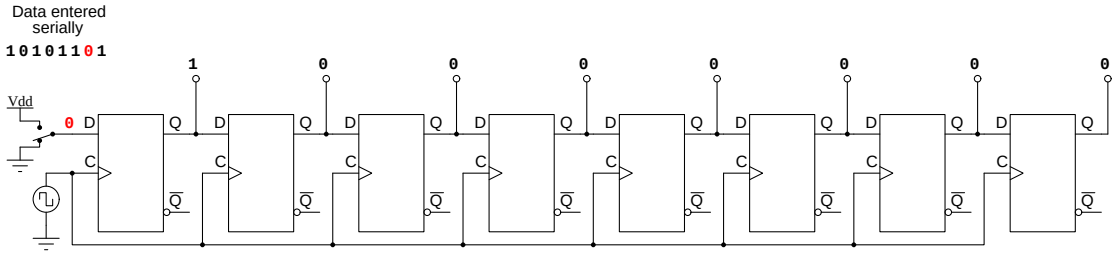
6.1 Animation of serial-in, parallel-out shift register

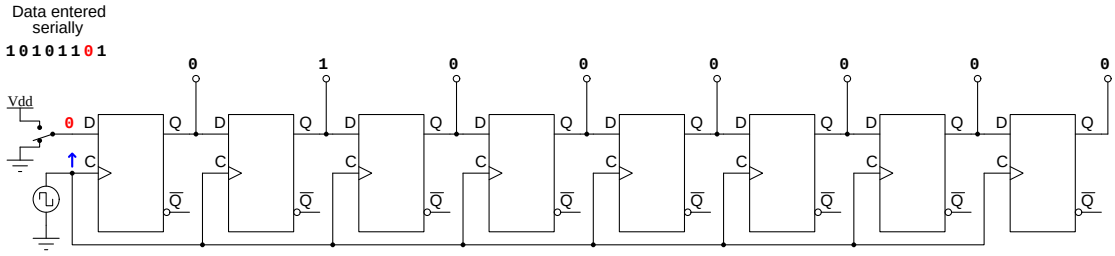
The following animation shows an eight-bit serial-in, parallel-out shift register accepting a serial stream of eight bits and one-by-one shifting them to its eight output lines for parallel reading.

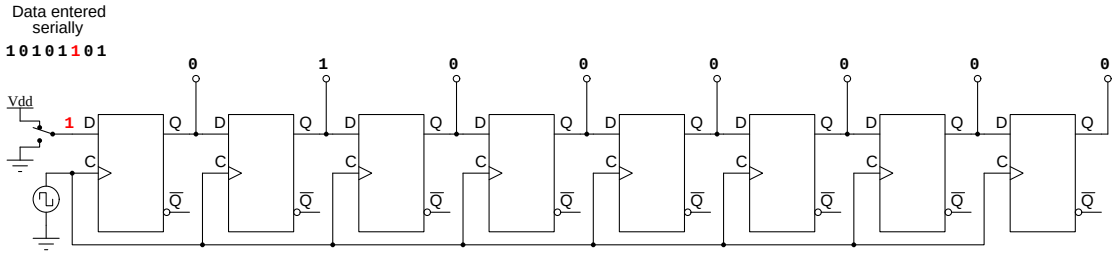

Data entered
serially
10101101
Vdd
0
D
Q
C
$\overline{Q}$

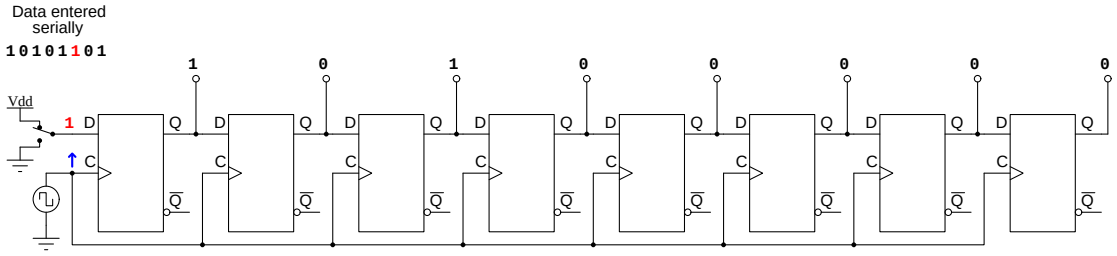

Data entered
serially
10101101
Vdd
1
D
Q
C
Q
0

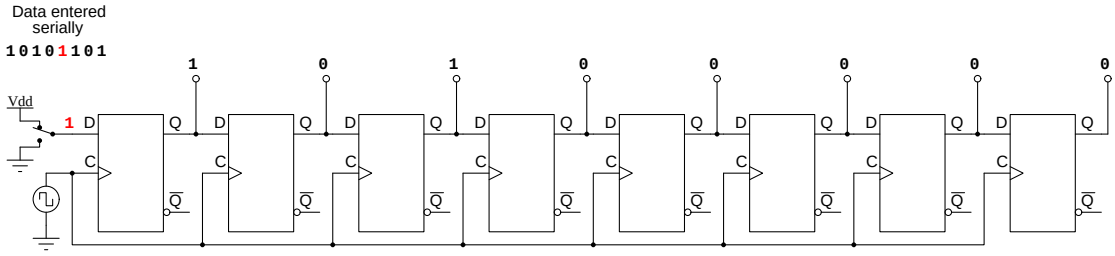

Data entered serially
10101101
Vdd
1
D
C
Q
$\overline{Q}$
1
0
0
0
0
0
0
0

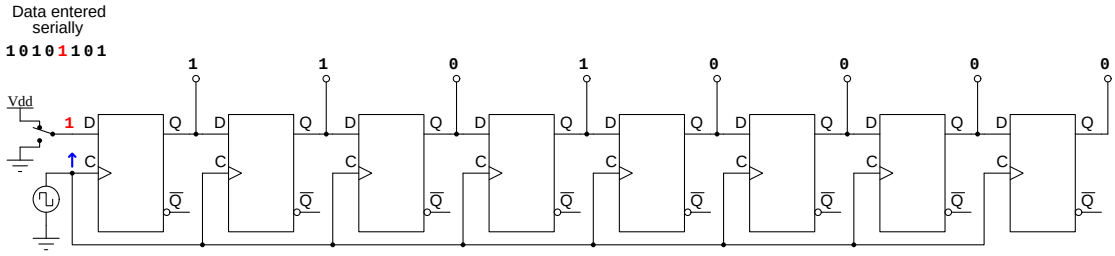

Data entered
serially
1 0 1 0 1 1 0 1
Vdd
0
D
Q
C
Q
1
0
0
0
0
0
0
0

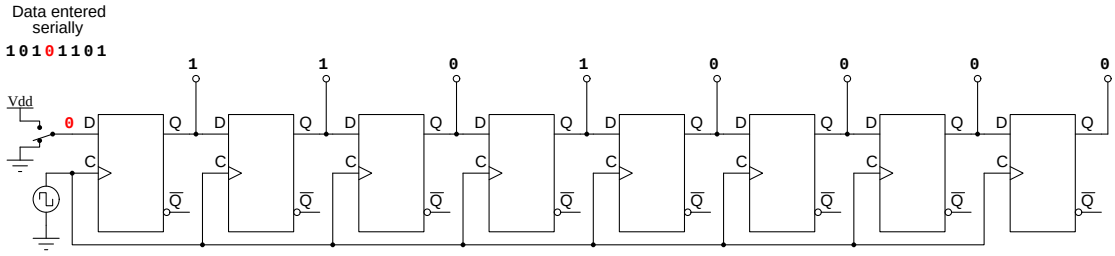

Data entered
serially
1 0 1 0 1 1 0 1
Vdd
0
D
C
Q
$\overline{Q}$
0
1
0
0
0
0
0
0

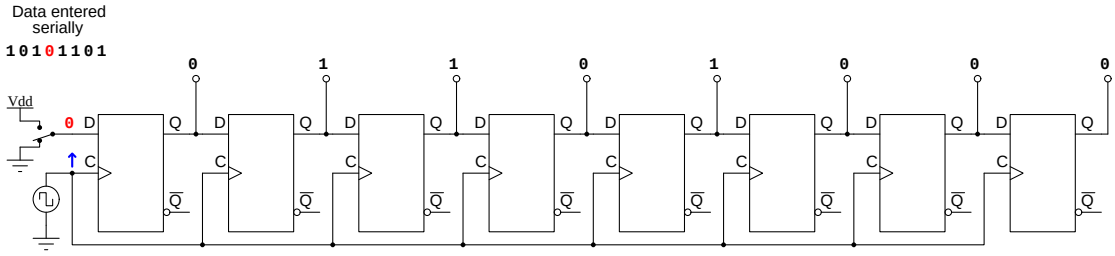

Data entered
serially
10101101
Vdd
1
D
Q
C
Q
0
1
0
0
0
0
0
0

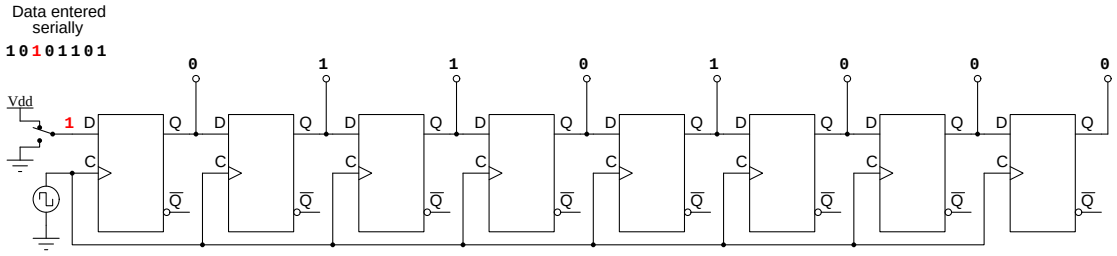

Data entered serially
10101101
Vdd
1
D
C
Q
$\overline{Q}$
1
0
1
0
0
0
0
0

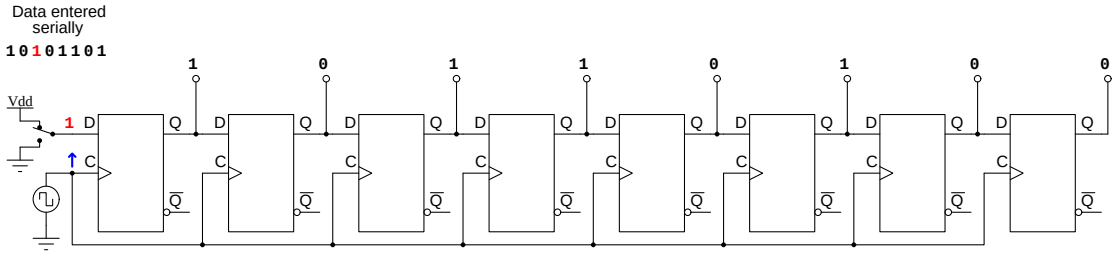

Data entered
serially
10101101
Vdd
1
D
Q
C
Q
1
0
1
0
0
0
0
0

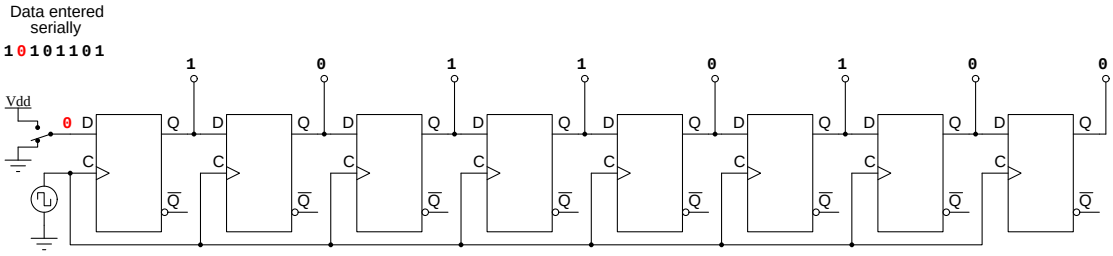

Data entered serially
1 0 1 0 1 1 0 1
Vdd
1
D
C
Q
Q̄
1
1
0
1
0
0
0
0

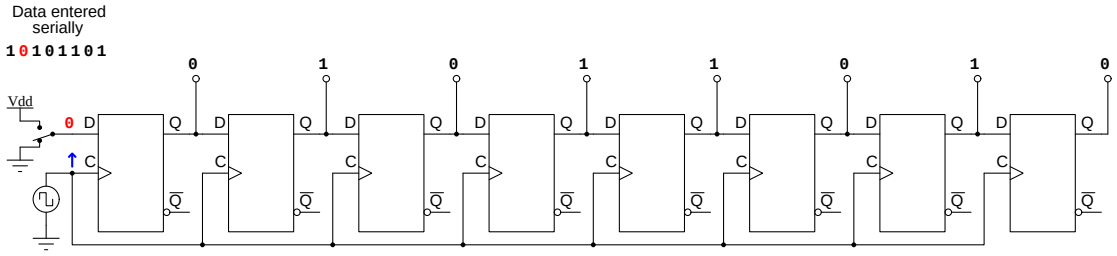

Data entered
serially
10101101
Vdd
0
D
Q
C
Q
1
1
0
1
0
0
0
0

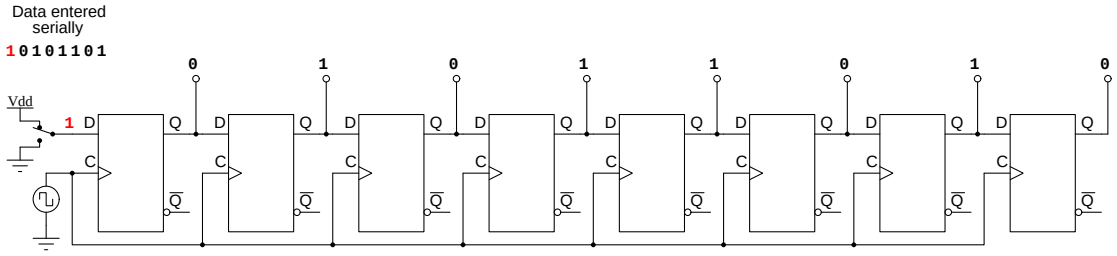

Data entered
serially
10101101
Vdd
0
D
C
Q
$\overline{Q}$
0
1
1
0
1
0
0
0

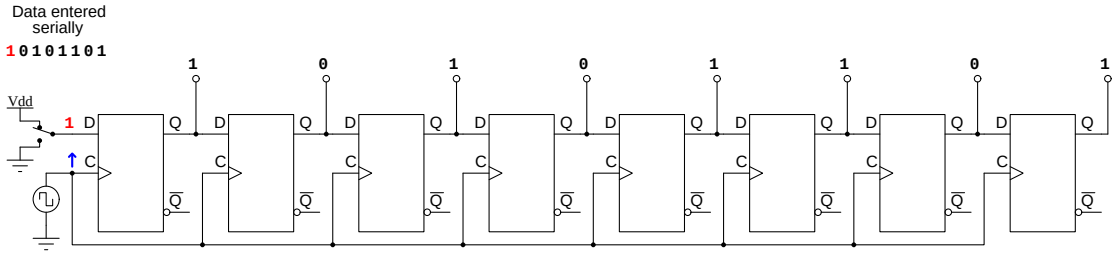

Data entered
serially
10101101
Vdd
1
D
Q
C
Q
0
1
1
0
1
0
0
0

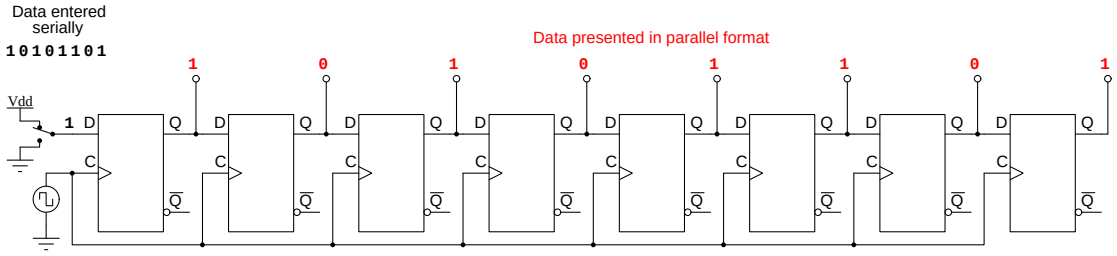

Data entered serially
10101101
Vdd
1
D
C
Q
$\overline{Q}$
1
0
1
1
0
1
0
0


Data entered
serially
10101101
Vdd
0
D
Q
C
Q
1
0
1
1
0
1
0
0


Data entered
serially
10101101
Vdd
0
D
C
Q
$\overline{Q}$
0
1
0
1
1
0
1
0

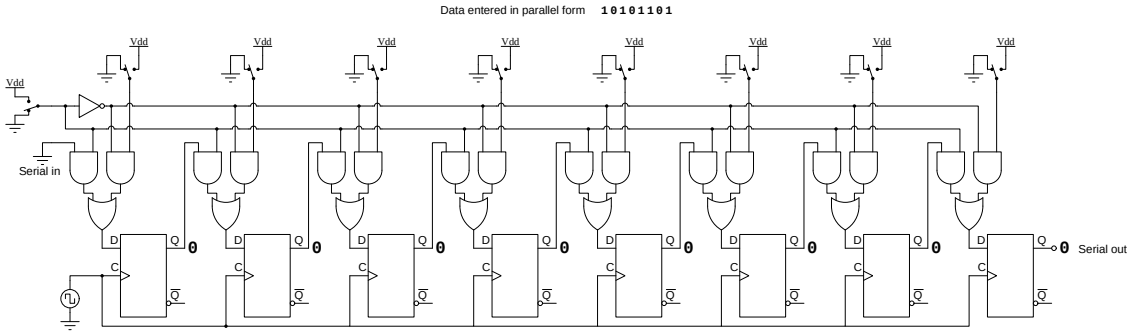

Data entered
serially
10101101
Vdd
1
D
Q
C
Q
0
1
0
1
1
0
1
0

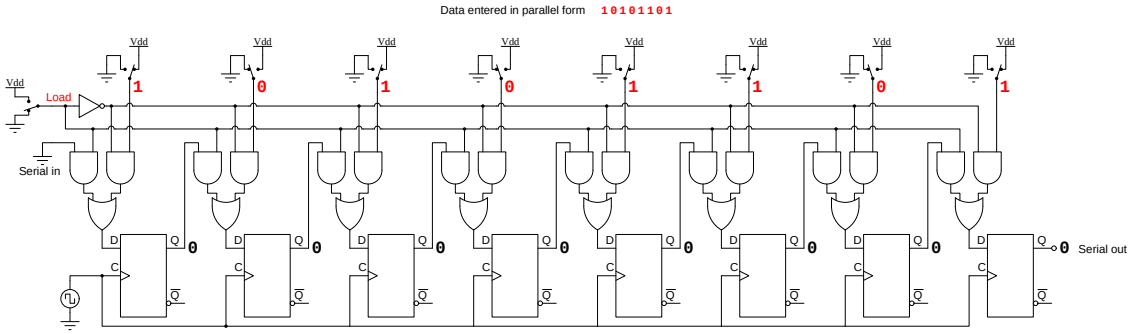

Data entered
serially
10101101
Vdd
1
D
C
Q
$\overline{Q}$
1
0
1
0
1
1
0
1

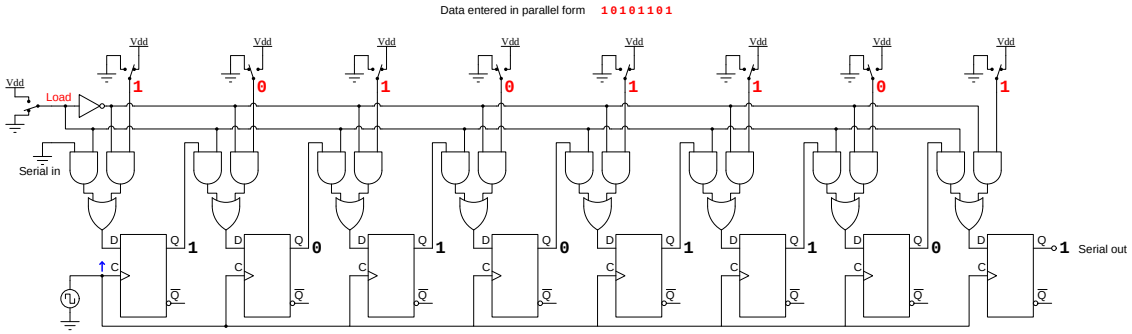

Data entered serially
10101101
Data presented in parallel format
1 0 1 0 1 1 0 1
Vdd
1
D
Q
C
Q̄

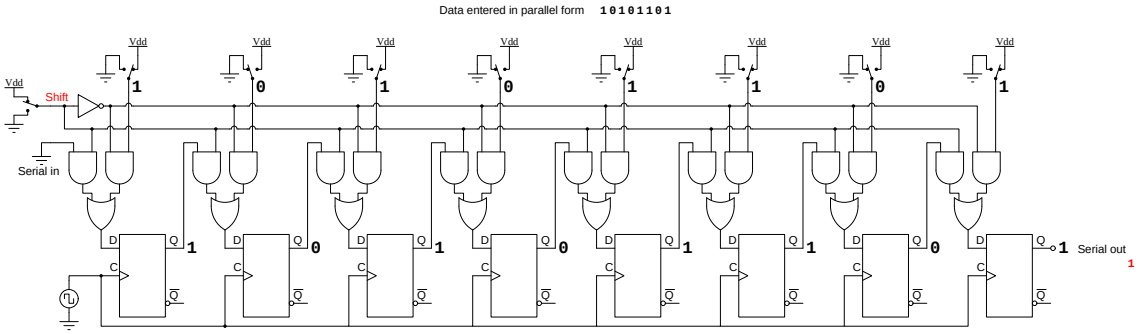
6.2 Animation of parallel-in, serial-out shift register

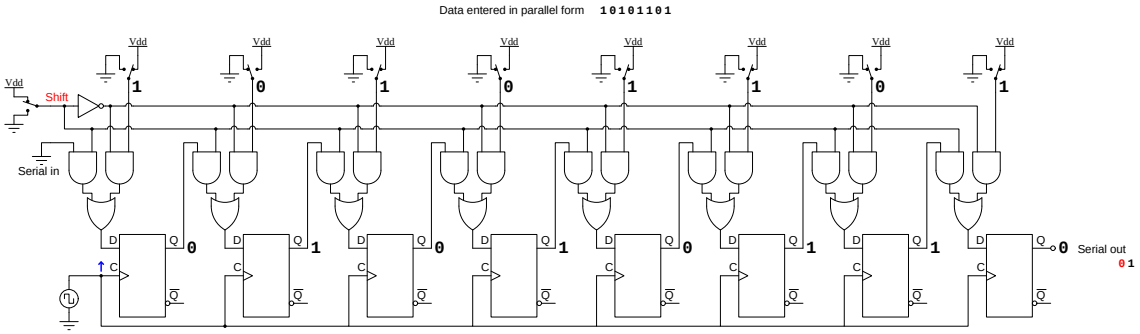
The following animation shows an eight-bit parallel-in, serial-out shift register accepting a parallel batch of of eight bits and one-by-one shifting them to its eight output lines for serial transmission to some other digital circuit.

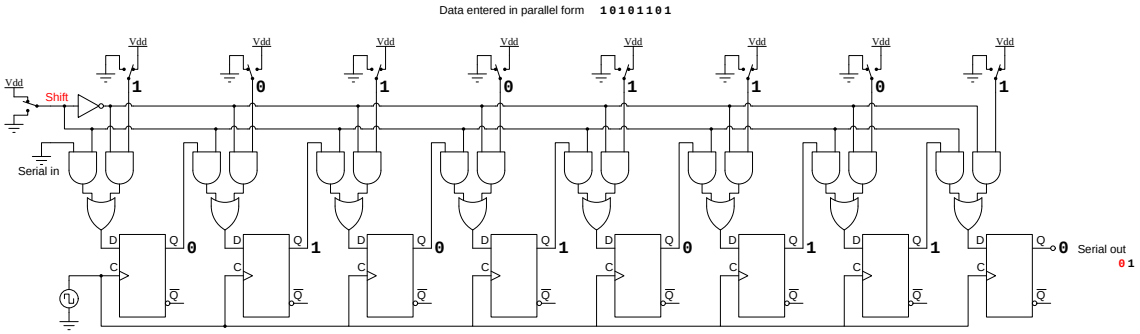

Data entered in parallel form 10101101
Vdd
Serial in
D
Q
C
Q
0
Serial out

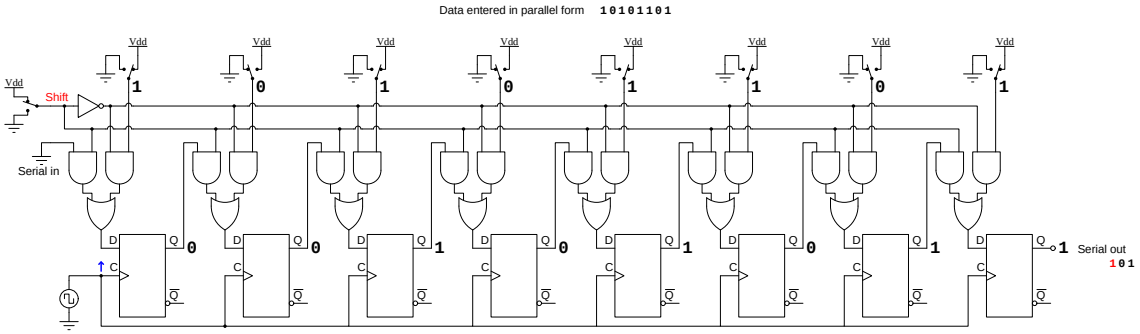

Data entered in parallel form 10101101
Vdd
Load
Serial in
Serial out
D
Q
C
1
0

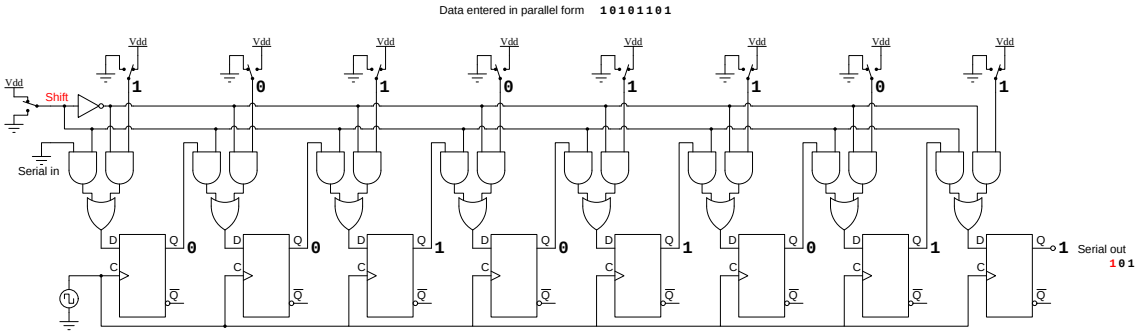

Data entered in parallel form 10101101
Vdd
Load
Serial in
Serial out
D
Q
C
1
0

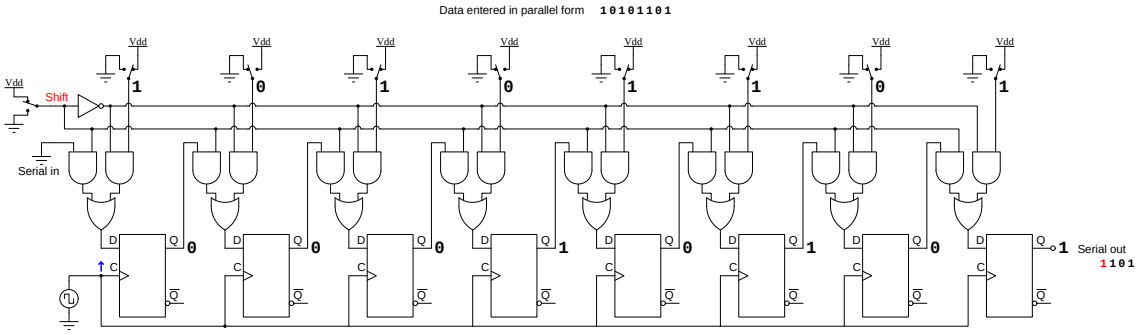

Data entered in parallel form 10101101
Vdd
Shift
Serial in
1
0
1
0
1
1
0
1
D
Q
C
Q
1
0
1
0
1
1
0
1
Serial out
1

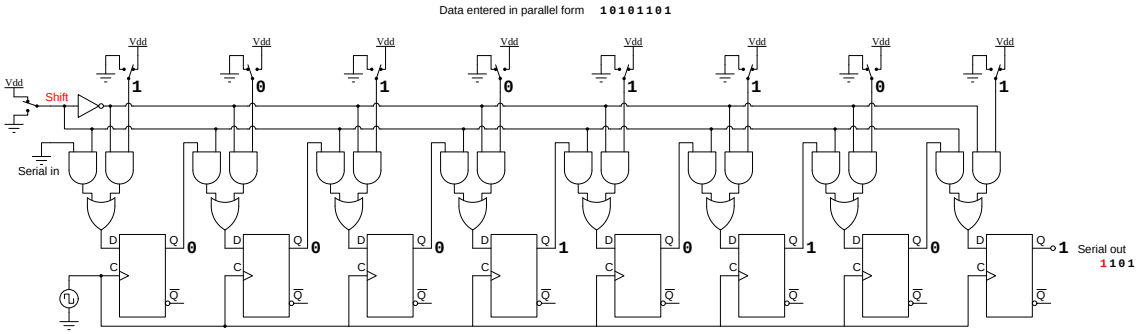

Data entered in parallel form 10101101
Vdd
Shift
Serial in
Serial out
01
D
Q
C
1
0

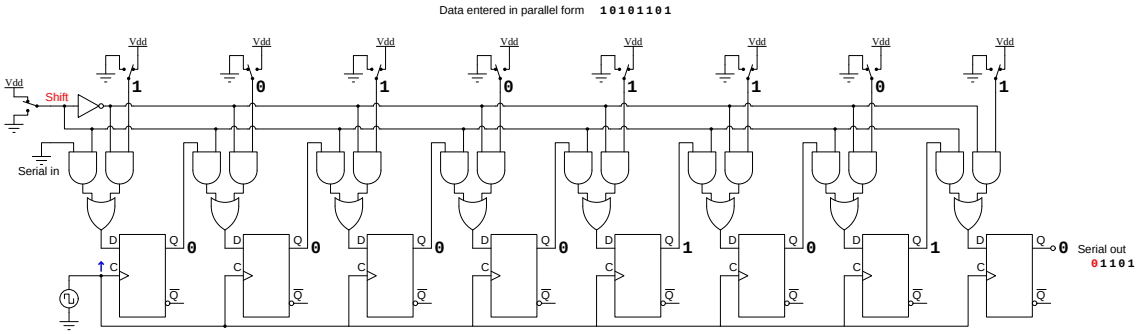

Data entered in parallel form 10101101
Vdd
Shift
Serial in
1
0
1
0
1
1
0
1
D
Q
C
Q
0
1
0
1
0
1
1
0
Serial out
01

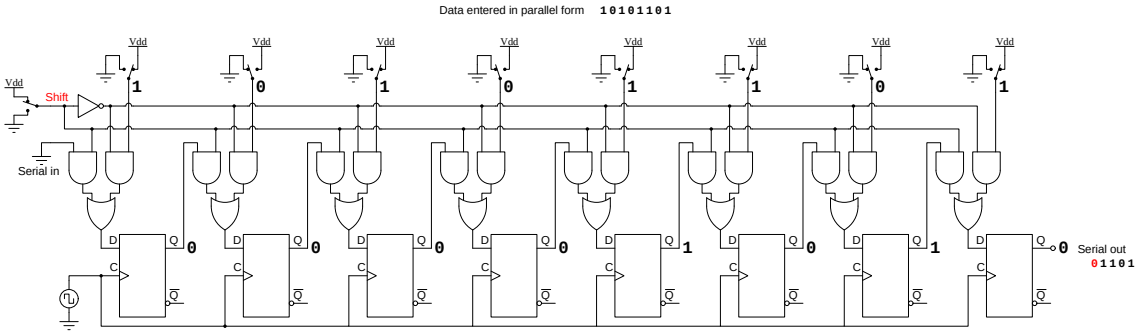

Data entered in parallel form 10101101
Vdd
Shift
Serial in
1
0
1
0
1
1
0
1
D
Q
C
Serial out
101

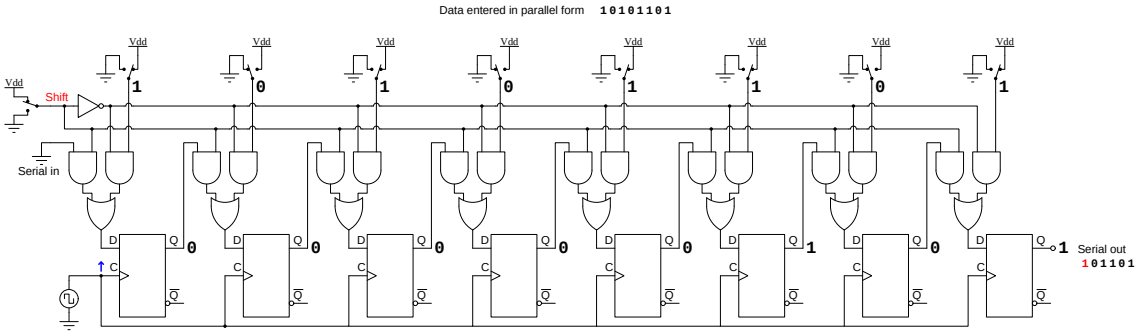

Data entered in parallel form 10101101
Vdd
Shift
Serial in
1
0
1
0
1
1
0
1
D
Q
C
Q
0
0
1
0
1
0
1
1
Serial out
101

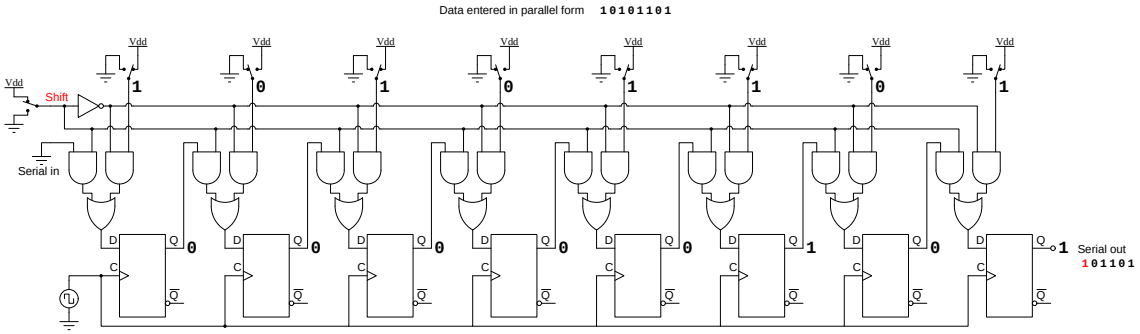

Data entered in parallel form 10101101
Vdd
Shift
Serial in
1
0
1
0
1
1
0
1
D
Q
C
Q
0
0
0
1
0
1
0
1
Serial out
1101

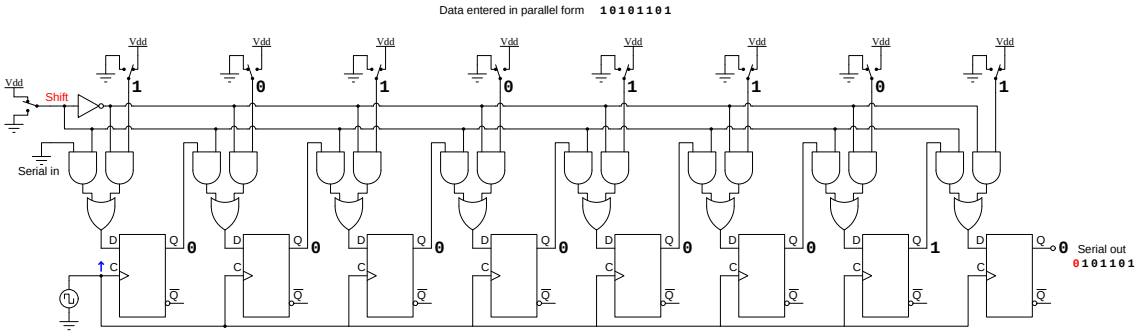

Data entered in parallel form 10101101
Vdd
Shift
Serial in
1
0
1
0
1
1
0
1
D
Q
C
Q
0
1
Serial out
1101

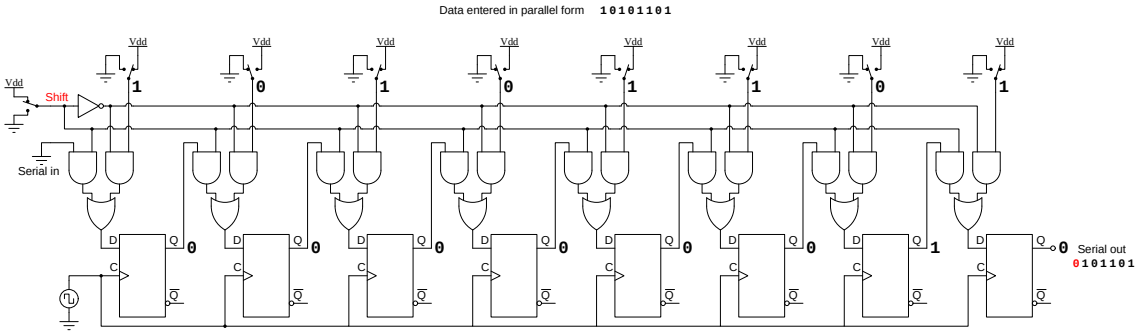

Data entered in parallel form 10101101
Vdd
Shift
Serial in
1
0
1
0
1
1
0
1
D
Q
C
Serial out
01101

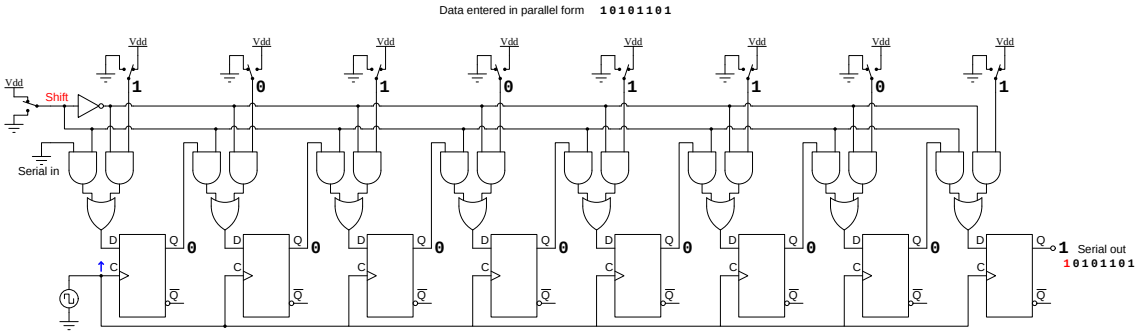

Data entered in parallel form 10101101
Vdd
Shift
Serial in
Serial out
01101
D
Q
C
Q̄

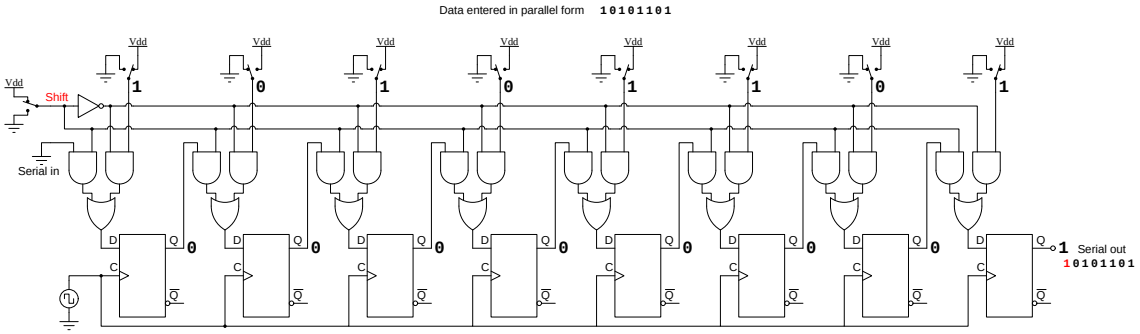

Data entered in parallel form 10101101
Vdd
Shift
Serial in
1
0
1
0
1
1
0
1
D
Q
C
Serial out
101101


Data entered in parallel form 10101101
Vdd
Shift
Serial in
1
0
1
0
1
1
0
1
D
Q
C
Q
0
1
Serial out
101101


Data entered in parallel form 10101101
Vdd
Shift
Serial in
1
0
1
0
1
1
0
1
D
Q
C
Serial out
0101101


Data entered in parallel form 10101101
Vdd
Shift
Serial in
1
0
1
0
1
1
0
1
D
Q
C
Q
0
1
Serial out
0101101


Data entered in parallel form 10101101
Vdd
Shift
Serial in
1
0
1
0
1
1
0
1
D
Q
C
Serial out
10101101


Data entered in parallel form 10101101
Vdd
Shift
Serial in
1
0
1
0
1
1
0
1
D
Q
C
Serial out
10101101

Chapter 7

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?

7.1 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.

7.1.1 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.

7.1.2 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.

Digital signal

Discrete signal

Analog signal

Bit

Noise immunity

Encoding

NRZ encoding

Manchester encoding

FSK encoding

Bit rate

Baud rate

Data frame

Synchronous versus asynchronous communication

Parity

UART

Frame check sequence

Hash

Flow control

Channel arbitration

Master-slave arbitration

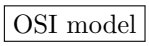
Token-passing arbitration

TDMA arbitration

CSMA arbitration

Determinism

Jabber


OSI model

Simplex

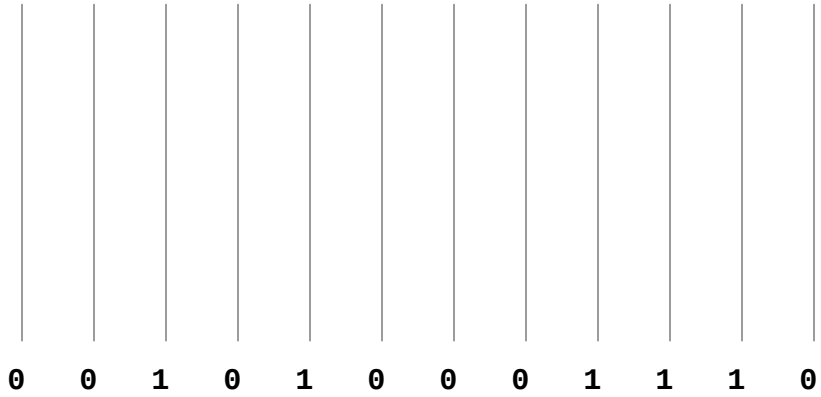
Duplex

7.1.3 Manchester encoding of a digital word

Decoding a Manchester-encoded waveform is challenging to many students, because it's not clear at first how to differentiate "real" signal transitions (i.e. those pulse edges representing actual binary data bits) from transitions that are merely "reversals" (i.e. those pulse edges that are simply set-up for the next "real" data pulse). Here, I will show you a practical problem-solving method to gain a deeper understanding.

We will apply the problem-solving technique of *working backwards* to understand the concept better. If the part we're struggling with is how to convert a waveform into a series of bits, then we'll turn the problem backwards by starting with a known series of bits and working to convert that series of bits into a waveform:

Here we see a series of bits aligned with a set of grey lines which we know will be pulse edges:



Begin by tracing the rising- and falling-edge pulses for each bit, following the standard of a rising edge representing a "1" bit and a falling edge representing a "0" bit. Feel free to draw small arrows distinguishing the rising versus falling transitions. Then, figure out how to connect these rising and falling edges together to form an actual pulse waveform.

It should quickly become apparent to you where "reversals" are necessary in order to "set up" properly for the next data pulse.

After you have done this, cover up the "1" and "0" bits so you can only see the waveform you've sketched. Erase any arrow-heads you might have sketched, so there is nothing visible to you except a clean pulse waveform. Now, explain to yourself how you would interpret this waveform to know which pulse edges represented real data as opposed to reversals.

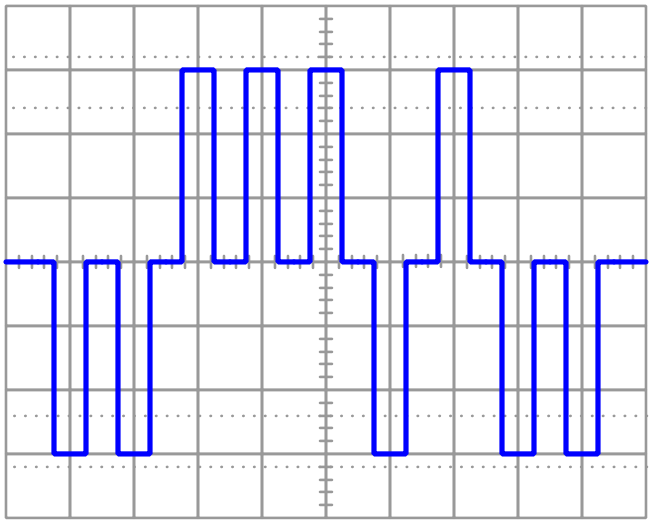
Challenges

- Manchester encoding is used to encode digital data on magnetic tape, where the playback speed of that tape may not be precisely constant. Explain why Manchester encoding makes sense for a medium where the bit rate will never be perfectly uniform.

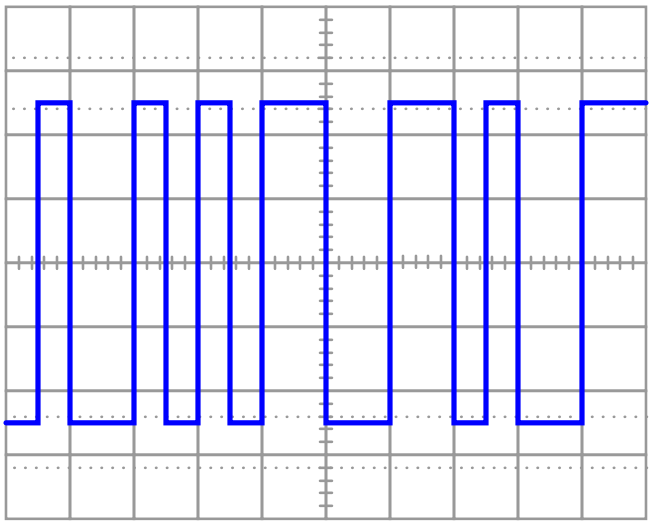
7.1.4 Serial data stream decoding

Decode the following serial data streams, each one encoded using a different method:

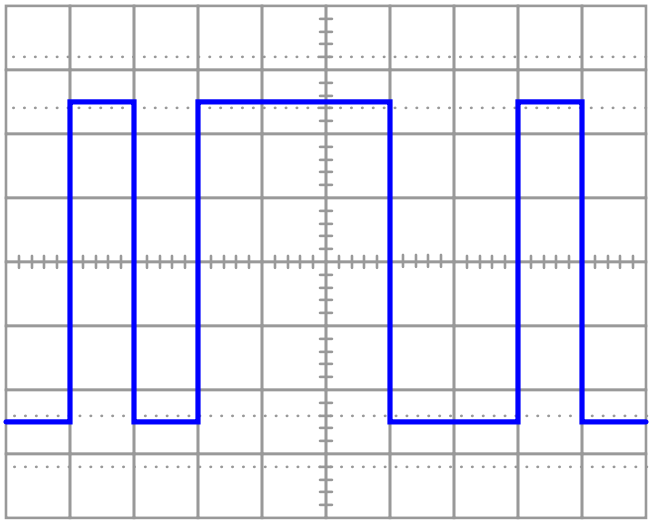
(RZ encoding)



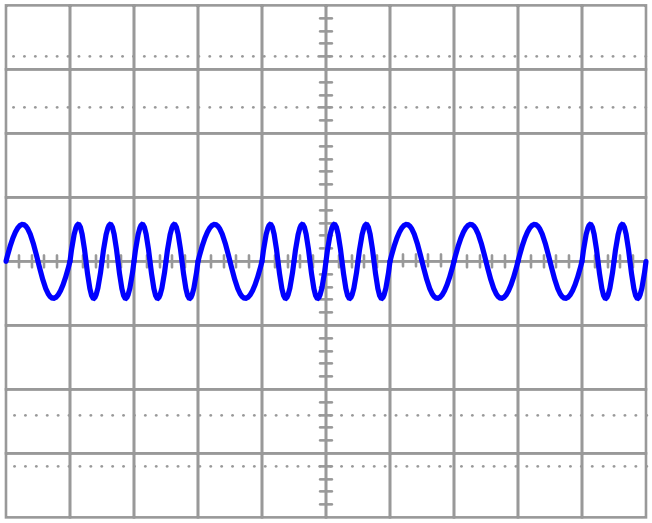
(Manchester encoding)



(NRZ encoding)



(FSK encoding)



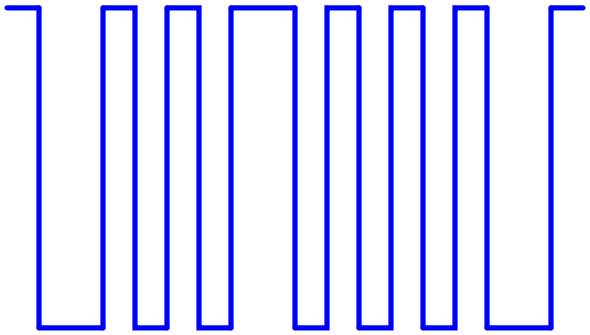
Challenges

- Which of these encoding schemes might be mistaken for another based on appearance alone?

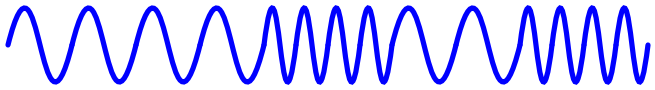
7.1.5 More Manchester and FSK decoding

Decode the following serial data streams, each one encoded using a different method:

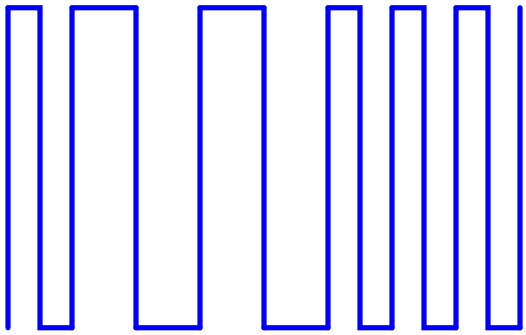
(Manchester encoding)



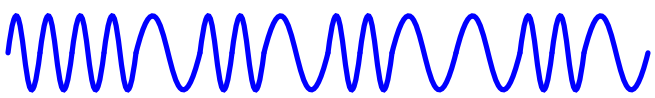
(FSK encoding)



(Manchester encoding)



(FSK encoding)

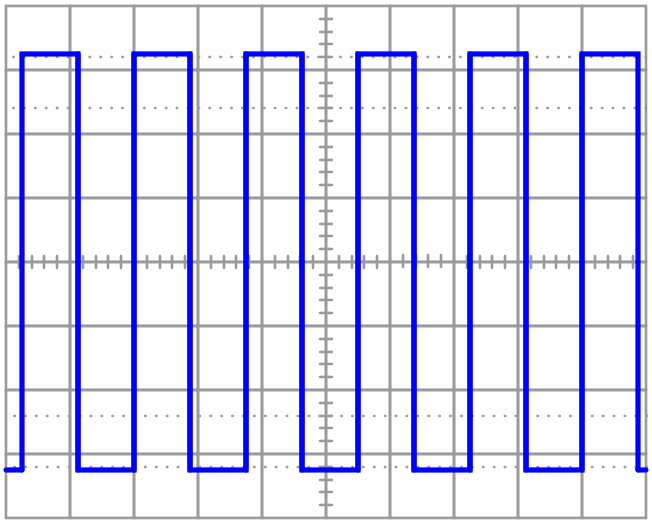


Challenges

- Assuming a common time scale for all data streams shown, which of the two has the highest *bit rate*?
- Assuming a common time scale for all data streams shown, which of the two has the highest *baud rate*?

7.1.6 Ambiguous Manchester data stream

Try to interpret this Manchester-encoded digital signal as viewed on an oscilloscope display, and explain why the task is difficult (or impossible) without further information:

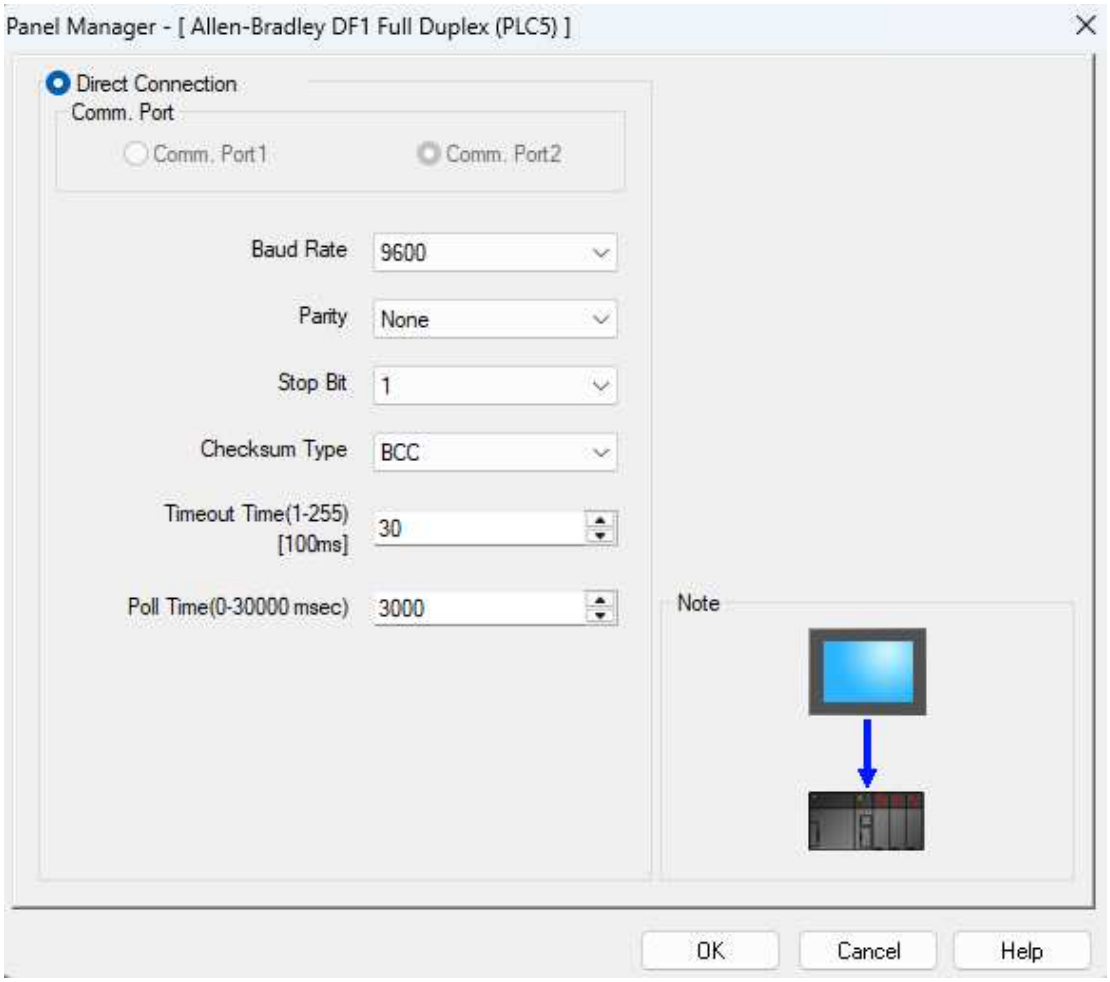


Challenges

- Identify multiple interpretations of this data stream.

7.1.7 Serial configuration for an HMI

The following screenshot is of an HMI (Human-Machine Interface) configuration software screen, viewing the "Panel Manager" dialogue window where serial communication parameters are set. In this example, the HMI is being configured to communicate with an Allen-Bradley model PLC-5 programmable logic controller:



Explain the significance of *baud rate* and of *parity* in this configuration. Why do these parameters matter, and what might happen if they were set incorrectly?

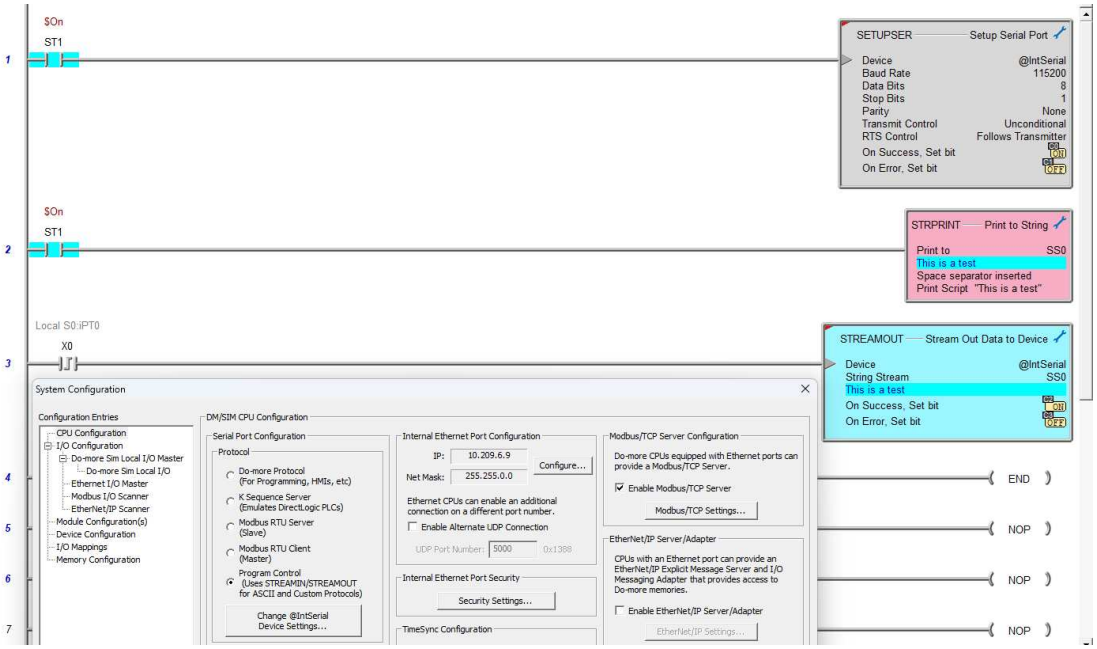
What do you suppose the *Poll Time* parameter signifies for the HMI?

Challenges

- What does "Full Duplex" mean in this context?

7.1.8 Serial data programming in a simulated PLC

The following screenshot shows the ladder-diagram editor for a simulated PLC (Programmable Logic Controller), using Automation Direct's Do-More programming software, along with the System Configuration dialogue window:



What parameter(s) in the System Configuration dialogue window relate to the instructions seen in the ladder-diagram PLC program?

Where in the ladder-diagram program are the serial port's basic communication parameters set?

What kind of data or message is being transmitted by this PLC, based on what you may discern from the program?

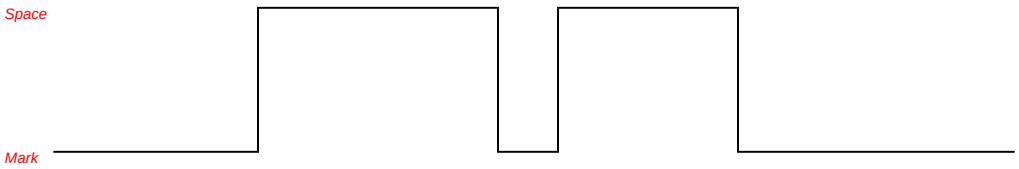
Challenges

- Why do you suppose these instructions provide bits to indicate success or error?

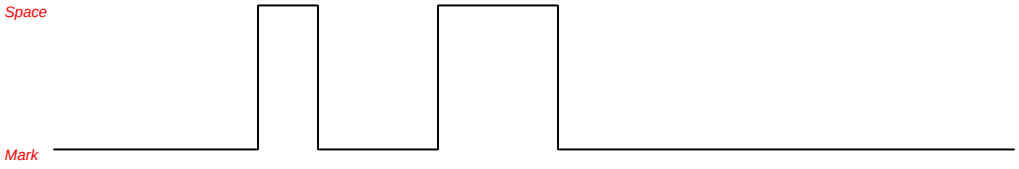
7.1.9 Interpreting NRZ data frames

The following oscillographs show NRZ data streams with different data bit widths and parities.

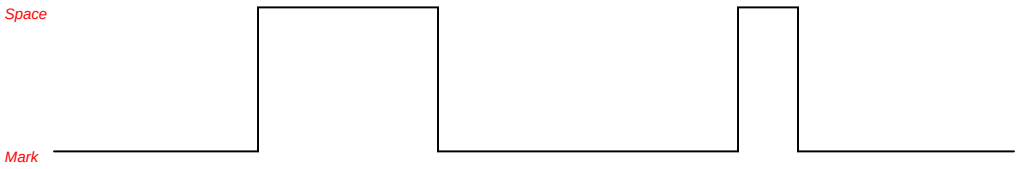
Seven-bit data, with no parity bit:



Seven-bit data, with no parity bit:



Eight-bit data, with no parity bit:

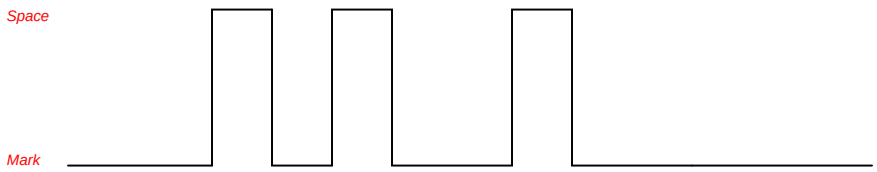


Challenges

- Explain how it is possible to identify the time period of a single bit, from any of these oscillographs.

7.1.10 Interpreting ASCII character from EIA/TIA-232 data frame

Here is an oscilloscope's view of an ASCII character (7 bits) sent asynchronously with even parity and 2 stop bits ("7-E-2"), using *NRZ* (Non-Return to Zero) encoding used by the EIA/TIA-232 serial communication standard:



Identify the specific ASCII character represented by the data in this frame, and determine whether or not the parity checks out okay.

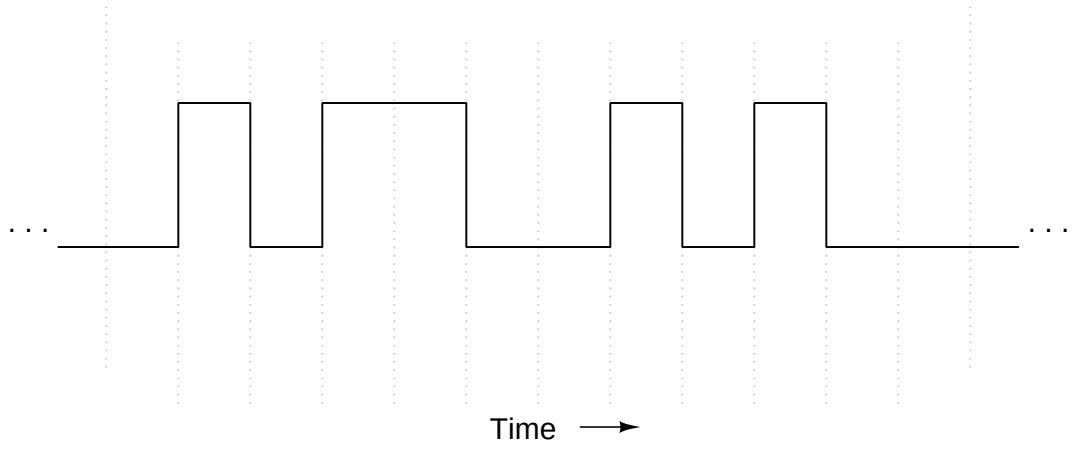
Challenges

- Label each of the bits on the waveform: start bit, data bits, parity bit, and stop bits.

7.1.11 Interpreting more NRZ data frames

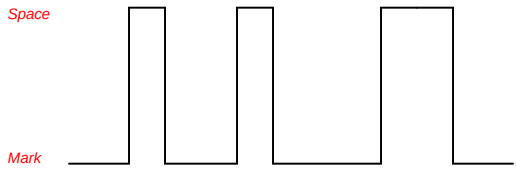
The following oscillographs show NRZ-encoded data. Interpret each and identify the features listed for each.

Eight data bits, no parity, two stop bits:



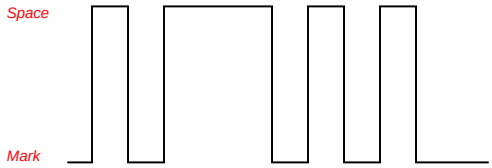
- All "Mark" and "Space" states
- The location and state of the *start bit*
- The eight-bit binary data represented by this waveform
- The location and state of the *stop bits*
- The default "idle" state between serial data packets
- Identify the binary data represented by this waveform.

Seven data bits, odd parity, one stop bit:



- All "Mark" and "Space" states
- The location and state of the *start bit*
- The eight-bit binary data represented by this waveform
- The location and state of the *stop bits*
- The default "idle" state between serial data packets
- Identify the ASCII character represented by this waveform, and determine whether or not the parity checks out okay.

Eight data bits, even parity, one stop bit:



- All "Mark" and "Space" states
- The location and state of the *start bit*
- The eight-bit binary data represented by this waveform
- The location and state of the *stop bits*
- The default "idle" state between serial data packets
- Identify the ASCII character represented by this waveform, and determine whether or not the parity checks out okay.

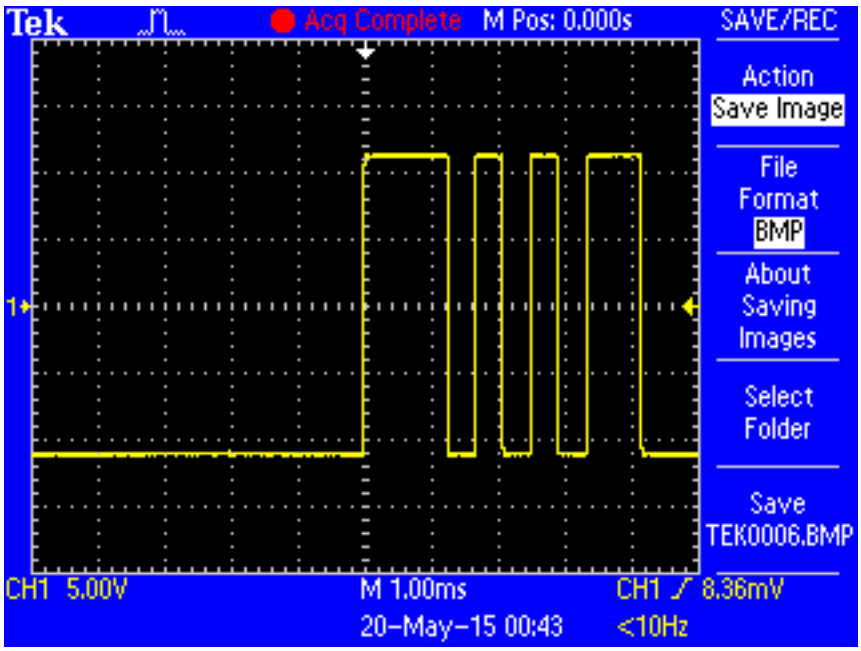
Challenges

- Explain why it is critically important that we know in advance the number of data bits contained in the frame.

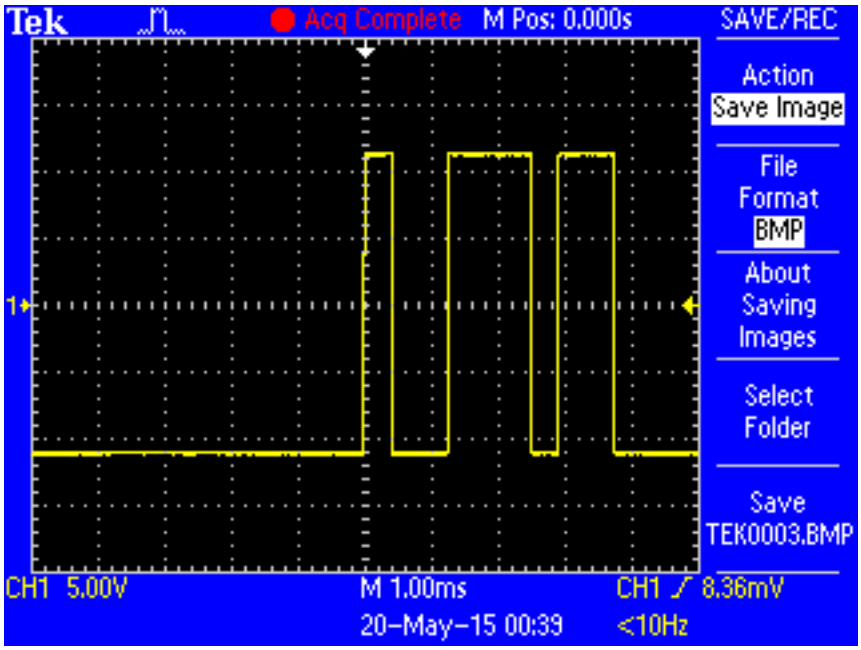
7.1.12 EIA/TIA-232 data frames of ASCII characters

Decode the following EIA/TIA-232 data "frames" captured on a digital oscilloscope, each image showing a single ASCII character. Note that the EIA/TIA-232 serial data communication protocol uses *NRZ encoding* where a "low" voltage is a "Mark" state (1) and a "high" voltage is a "Space" state (0), and where the data bits are transmitted in reverse order (LSB first, MSB last). Please note that each of these examples are of serial data communicated at the exact same bit rate, with a successful parity check (i.e. no corrupted bits):

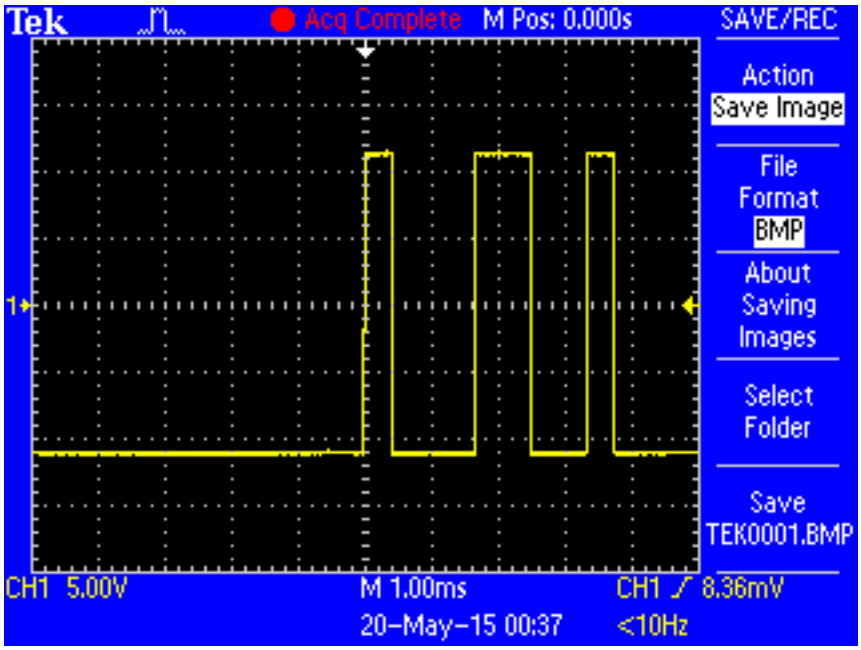
Example #1 – 8 data bits, odd parity, 1 stop bit:



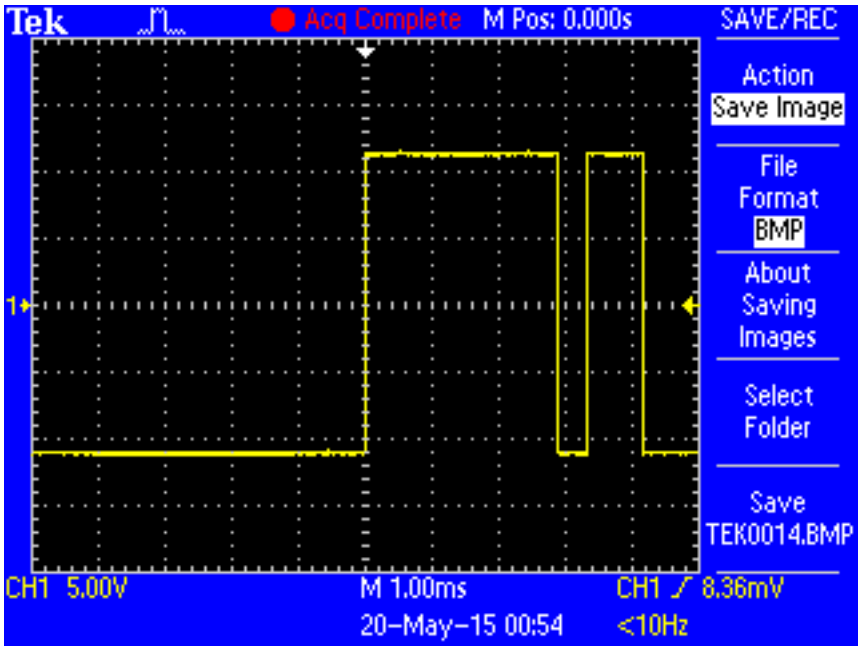
Example #2 – 8 data bits, even parity, 1 stop bit:



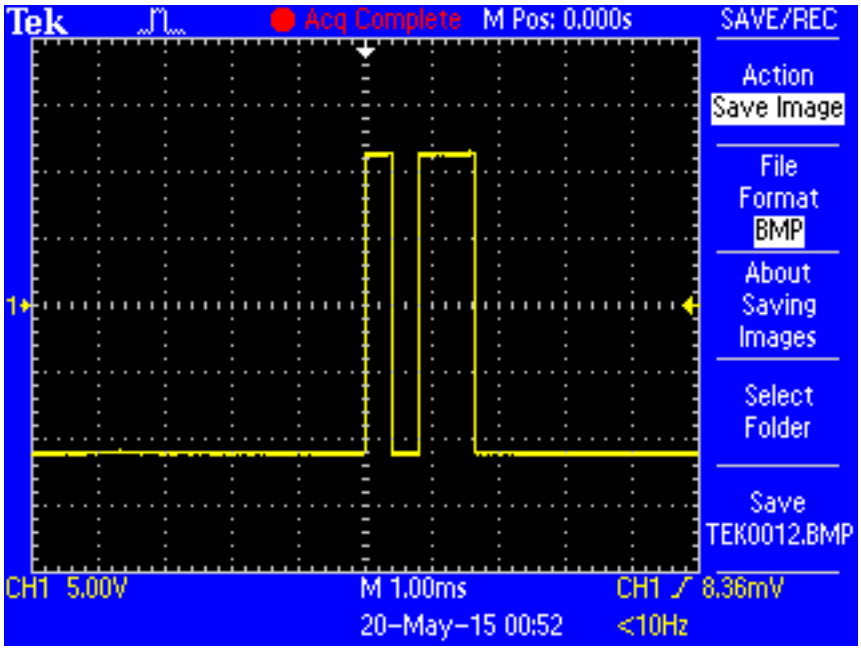
Example #3 – 8 data bits, even parity, 1 stop bit:



Example #4 – 8 data bits, odd parity, 1 stop bit:



Example #5 – 7 data bits, even parity, 2 stop bits:

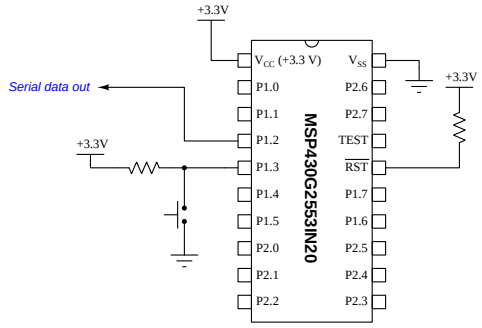


Challenges

- Identify the bit rate at which all these ASCII characters were transmitted.

7.1.13 Microcontroller UART communication program

The following schematic diagram and C-language code show how to configure a Texas Instruments MSP430G2553IN20 20-pin microcontroller to transmit simple text-based messages using its built-in UART, which TI refers to as its *Universal Serial Communications Interface A0*, abbreviated as `USCI_A0` or more simply as `UCA0`:



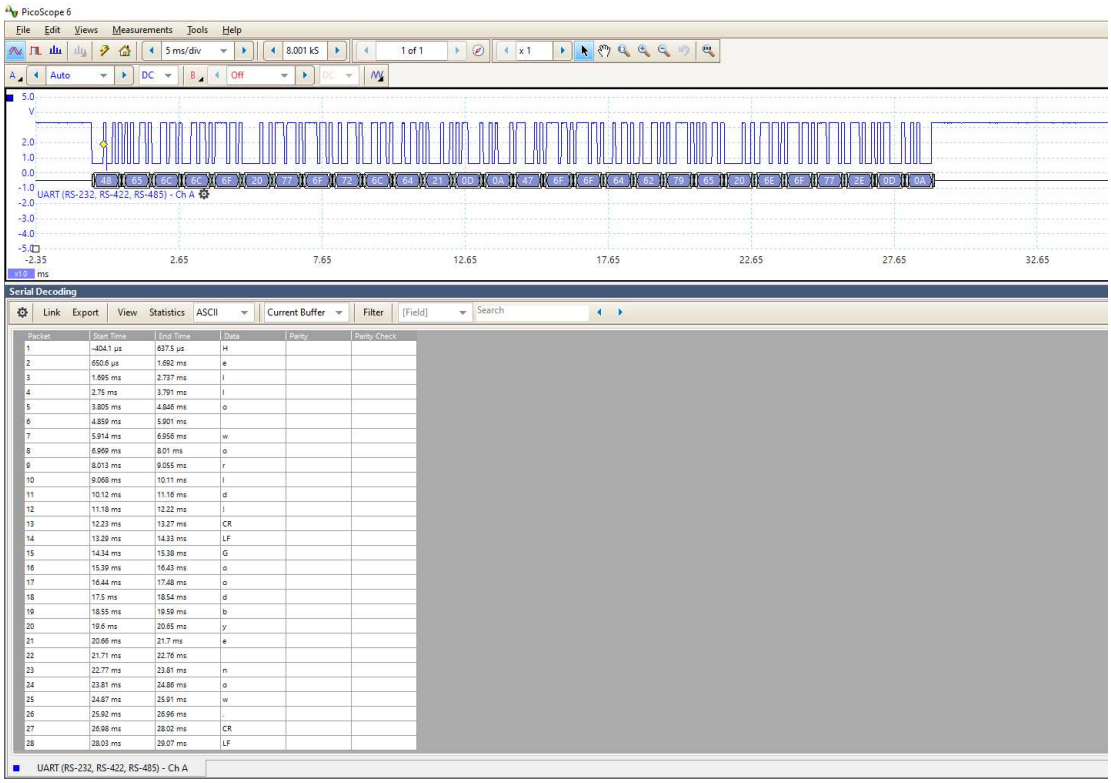

```
#include <msp430.h>

void transmit_ASCII(char * tx_data);   // Prototype for transmit function

void main(void)
{
  unsigned int button, lastbutton;

  WDTCTL = WDTPW + WDTHOLD;          // Stop watchdog timer

  BCSCTL1 = CALBC1_1MHZ;             // Set DCO to 1 MHz
  DCOCTL = CALDCO_1MHZ;

  P1DIR = 0xF7;                      // P1.3 input, all other P1 pins output
  P1REN = 0x08;                      // Enables pullup/pulldown resistor on P1.3
  P1OUT = 0x08;                      // Selects pullup resistor for P1.3
  P1SEL = BIT1 + BIT2;               // P1.1 is UCA0RXD and P1.2 is UCA0TXD
  P1SEL2 = BIT1 + BIT2;              //  when P1SEL and P1SEL2 bits set

  UCA0CTL1 |= UCSSEL_2;              // Have USCI use System Master Clock
  UCA0BR0 = 104;                     // These two USCI_A0 8-bit clock prescalers
  UCA0BR1 = 0;                       //  comprise a 16-bit value to set bit rate.
                                     //  Value of 104 = 9600 bps with 1MHz clock
                                     //  (as per table 15.4 in MSP430 user guide)
  UCA0MCTL = UCBRS0;                 // Bit rate may be further tweaked using the
                                     //  "modulation" register
```

```
  UCA0CTL1 &= ~UCSWRST;             // Start the USCI's state machine

  while(1)                          // Endless loop
  {
      if ((P1IN & BIT3) == BIT3)    // "button" represents state of P1.3
          button = 1;               // == 1 when P1.3 is high
      else
          button = 0;               // == 0 when P1.3 is low

      if((button == 0) && (lastbutton == 1))    // Detects P1.3 high-to-low
      {                                         //  without requiring interrupt
          transmit_ASCII("Hello world!\r\n");   // Transmit text
          transmit_ASCII("Goodbye now.\r\n");   // Transmit more text
      }
      lastbutton = button;
  }
}

void transmit_ASCII(char * char_array) // Function accepts char. array pointer
{
    unsigned int n = 0;
    while(char_array[n])             // Increment until null pointer (0) reached
    {
        while ((UCA0STAT & UCBUSY)); // Wait if line TX/RX module busy with data
        UCA0TXBUF = char_array[n];   // Load each character into transmit buffer
        ++n;                         // Increment variable for array address
    }
}
```

Why does this program use the variables `button` and `lastbutton` to trigger the UART to begin transmitting?

What sort of data gets passed to the `transmit_ASCII` function as an argument?

When the output signal (`P1.2`) is measured on a digital oscilloscope capable of EIA/TIA-232 serial data decoding, we see the following result:



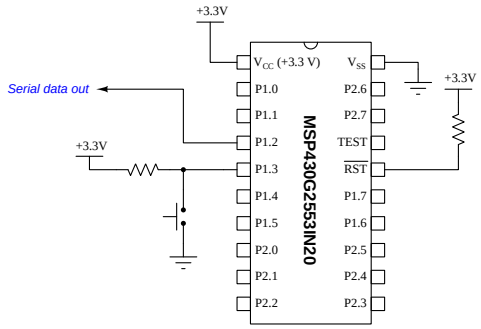
Explain how to decode these voltage pulses to interpret them as alphabetical characters.

Challenges

- Of what practical importance is bit rate to serial data communication?

7.1.14 Microcontroller UART text and number program

The following schematic diagram and C-language code show how to configure a Texas Instruments MSP430G2553IN20 20-pin microcontroller to transmit simple text-based messages including alphabetical characters and four-digit integer numbers using its built-in UART, which TI refers to as its *Universal Serial Communications Interface A0*, abbreviated as `USCI_A0` or more simply as `UCA0`:




```
#include <msp430.h>

void transmit_ASCII(char * tx_data); // Prototype for alpha transmit function
void transmit_Num(int num);   // Prototype for number transmit function

int x = 1486;  // Flame temperature, degrees C

void main(void)
{
  unsigned int button, lastbutton;

  WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
  BCSCTL1 = CALBC1_1MHZ; // Set DCO to 1 MHz
  DCOCTL = CALDCO_1MHZ;

  P1DIR = 0xF7; // P1.3 input, all other P1 pins output
  P1REN = 0x08; // Enables pullup/pulldown resistor on P1.3
  P1OUT = 0x08; // Selects pullup resistor for P1.3
  P1SEL = BIT1 + BIT2; // P1.1 is UCA0RXD and P1.2 is UCA0TXD
  P1SEL2 = BIT1 + BIT2; // when P1SEL and P1SEL2 bits set

  UCA0CTL1 |= UCSSEL_2; // Have USCI use System Master Clock
  UCA0BR0 = 104; // These two USCI_A0 8-bit clock prescalers
  UCA0BR1 = 0; // comprise a 16-bit value to set bit rate.
               // Value of 104 = 9600 bps with 1MHz clock
```

```
              // (as per table 15.4 in MSP430 user guide)

  UCA0MCTL = UCBRS0; // Bit rate may be further tweaked using the

  // "modulation" register
  UCA0CTL1 &= ~UCSWRST; // Start the USCI's state machine

  while(1) // Endless loop
  {
    if ((P1IN & BIT3) == BIT3) // "button" represents state of P1.3
      button = 1; // == 1 when P1.3 is high

    else
      button = 0; // == 0 when P1.3 is low

    if((button == 0) && (lastbutton == 1)) // Detects P1.3 high-to-low
    {                                      // without requiring interrupt
      transmit_ASCII("Flame temp = "); // Transmit text
      transmit_Num(x);                 // Transmit number
      transmit_ASCII(" deg C\r\n");    // Transmit more text
    }

    lastbutton = button;
  }
}

void transmit_ASCII(char * char_array) // Function accepts char. array pointer
{
  unsigned int n = 0;
  while(char_array[n]) // Increment until null pointer (0) reached
  {
    while ((UCA0STAT & UCBUSY)); // Wait if line TX/RX module busy with data
    UCA0TXBUF = char_array[n];   // Load each character into transmit buffer
    ++n;                         // Increment variable for array address
  }
}

void transmit_Num(int num) // Function accepts 4-digit integer number
{
  unsigned int n = 0;
  char char_array[5];

  // NOTE: (int) "casts" the calculated result into integer form:
  char_array[0] = (int)(num/1000) + 0x30;
```

```
  char_array[1] = (int)(num/100) - 10*(int)(num/1000) + 0x30;
  char_array[2] = (int)(num/10) - 10*(int)(num/100) + 0x30;
  char_array[3] = (int)(num) - 10*(int)(num/10) + 0x30;
  char_array[4] = 0;

  while(char_array[n]) // Increment until null pointer (0) reached
    {
      while ((UCA0STAT & UCBUSY)); // Wait if line TX/RX module busy with data
      UCA0TXBUF = char_array[n];   // Load each character into transmit buffer
      ++n;                         // Increment variable for array address
    }
}
```

Explain how the `transmit_Num` function takes a four-digit decimal value and produces four ASCII characters representing the digits of that number.

What sort of data gets passed to the `transmit_Num` function as an argument?

Challenges

- Of what practical importance is bit rate to serial data communication?
- Devise a "thought experiment" whereby you could test the operation of the `transmit_Num()` function, showing step-by-step how it converts an integer value into an ASCII character string.

7.2 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.

7.2.1 Miscellaneous physical constants

Note: constants shown in **bold** type are *exact*, not approximations. Values inside of parentheses show one standard deviation (σ) 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 (μ_0) = $1.25663706212(19) \times 10^{-6}$ Henrys per meter (H/m)

Electric permittivity of free space (ϵ_0) = $8.8541878128(13) \times 10^{-12}$ Farads per meter (F/m)

Characteristic impedance of free space (Z_0) = 376.730313668(57) Ohms (Ω)

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 (σ) = **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.

7.2.2 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	

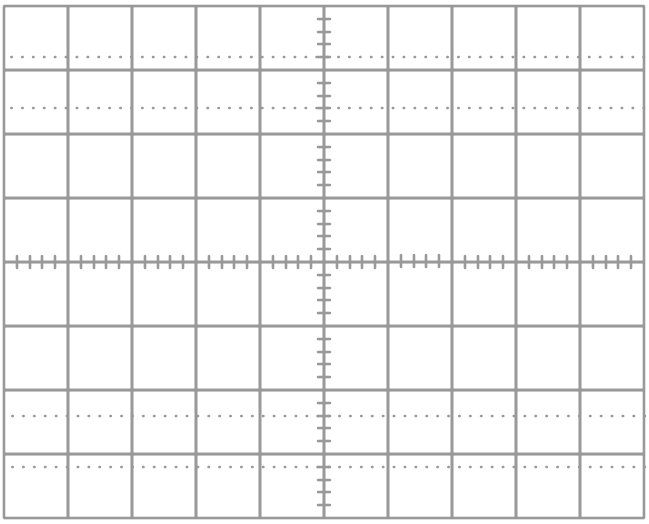
Note how the square-root term (y) is calculated in cell `C1`, and the denominator term (z) in cell `C2`. This makes the two final formulae (in cells `B1` and `B2`) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary[10] – all that matters is that they properly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

7.2.3 Manchester data frame with a specified bit rate

Sketch an oscillograph for a Manchester-encoded data stream `1011101` at a data rate of 14400 bits per second:



Be sure to specify an appropriate timebase setting for the oscilloscope to show the entire data frame.

Challenges

- Of what practical importance is bit rate to serial data communication?

7.3 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.

7.3.1 Testing determinism

Devise a diagnostic test you could apply to assess the determinism of any given serial data network.

Challenges

- Explain the significance of determinism in a serial communication network.

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

B.1 First principles of learning

- **Anyone can learn anything** given appropriate time, effort, resources, challenges, encouragement, and expectations. Dedicating time and investing effort are the student's responsibility; providing resources, challenges, and encouragement are the teacher's responsibility; high expectations are a responsibility shared by both student and teacher.

- **Transfer is not automatic.** The human mind has a natural tendency to compartmentalize information, which means the process of taking knowledge learned in one context and applying it to another usually does not come easy and therefore should never be taken for granted.

- **Learning is iterative.** The human mind rarely learns anything perfectly on the first attempt. Anticipate mistakes and plan for multiple tries to achieve full understanding, using the lessons of those mistakes as feedback to guide future attempts.

- **Information is absorbed, but understanding is created.** Facts and procedures may be memorized easily enough by repeated exposure, but the ability to reliably apply principles to novel scenarios only comes through intense personal effort. This effort is fundamentally creative in nature: casting new concepts in one's own words, running mental experiments to perceive patterns and trends, etc. These acts of making knowledge "one's own" need not be perfect in order to be effective, as the value lies in the activity and not necessarily the finished product. Depth of understanding and permanence of memory are both directly related to mental effort during the learning process – i.e. *easy learning is shallow and fleeting.*

- **Education trumps training.** There is no such thing as an entirely isolated subject, as all fields of knowledge are connected. Training is narrowly-focused and task-oriented. Education is broad-based and principle-oriented. When preparing for a life-long technical career, education beats training every time.

- **Character matters.** Poor habits are more destructive than deficits of knowledge or skill. This is especially true in collective endeavors, where a team's ability to function depends on trust between its members. Simply put, no one wants an untrustworthy person on their team. An essential component of education then, is character development.

- **People learn to be responsible by bearing responsibility.** An irresponsible person is someone who has never *had* to be responsible for anything that mattered enough to them. Just as anyone can learn anything, anyone can become responsible if the personal cost of irresponsibility becomes high enough.

- **What gets measured, gets done.** Accurate and relevant assessment of learning is key to ensuring all students learn. Therefore, it is imperative to measure what matters.

- **Failure is nothing to fear.** Every human being fails, and fails in multiple ways at multiple times. Eventual success only happens when we don't stop trying.

B.2 Proven strategies for instructors

- Assume every student is capable of learning anything they desire given the proper conditions. Treat them as capable adults by granting real responsibility and avoiding artificial incentives such as merit or demerit points.
- Create a consistent culture of high expectations across the entire program of study. Demonstrate and encourage patience, persistence, and a healthy sense of self-skepticism. Anticipate and de-stigmatize error. Teach respect for the capabilities of others as well as respect for one's own fallibility.
- Replace lecture with "inverted" instruction, where students first encounter new concepts through reading and then spend class time in Socratic dialogue with the instructor exploring those concepts and solving problems individually. There is a world of difference between observing someone solve a problem versus actually solving a problem yourself, and so the point of this form of instruction is to place students in a position where they *cannot* passively observe.
- Require students to read extensively, write about what they learn, and dialogue with you and their peers to sharpen their understanding. Apply Francis Bacon's advice that "reading maketh a full man; conference a ready man; and writing an exact man". These are complementary activities helping students expand their confidence and abilities.
- Use artificial intelligence (AI) to challenge student understanding rather than merely provide information. Find productive ways for AI to critique students' clarity of thought and of expression, for example by employing AI as a Socratic-style interlocutor or as a reviewer of students' journals. Properly applied, AI has the ability to expand student access to critical review well outside the bounds of their instructor's reach.
- Build frequent and rapid feedback into the learning process so that students know at all times how well they are learning, to identify problems early and fix them before they grow. Model the intellectual habit of self-assessing and self-correcting your own understanding (i.e. a cognitive *feedback loop*), encouraging students to do the same.
- Use "mastery" as the standard for every assessment, which means the exam or experiment or project must be done with 100% competence in order to pass. Provide students with multiple opportunity for re-tries (different versions of the assessment every time).
- Require students to devise their own hypotheses and procedures on all experiments, so that the process is truly a scientific one. Have students assess their proposed experimental procedures for risk and devise mitigations for those risks. Let nothing be pre-designed about students' experiments other than a stated task (i.e. what principle the experiment shall test) at the start and a set of demonstrable knowledge and skill objectives at the end.
- Have students build as much of their lab equipment as possible: building power sources, building test assemblies[1], and building complete working systems (no kits!). In order to provide

[1] In the program I teach, every student builds their own "Development Board" consisting of a metal chassis with DIN rail, terminal blocks, and an AC-DC power supply of their own making which functions as a portable lab environment they can use at school as well as take home.

this same "ground-up" experience for every new student, this means either previous students take their creations with them, or the systems get disassembled in preparation for the new students, or the systems grow and evolve with each new student group.

- Incorporate external accountability for you and for your students, continuously improving the curriculum and your instructional methods based on proven results. Have students regularly network with active professionals through participation in advisory committee meetings, service projects, tours, jobshadows, internships, etc. Practical suggestions include requiring students to design and build projects for external clients (e.g. community groups, businesses, different departments within the institution), and also requiring students attend all technical advisory committee meetings and to dialogue with the industry representatives at those meetings.
- Repeatedly explore difficult-to-learn concepts across multiple courses, so that students have multiple opportunities to build their understanding.
- Relate all new concepts, whenever possible, to previous concepts and to relevant physical laws. Challenge each and every student, every day, to *reason* from concept to concept and to explain the logical connections between. Challenge students to verify their conclusions by multiple approaches (e.g. double-checking their work using different methods). Ask *"Why?"* often.
- Maintain detailed records on each student's performance and share these records privately with them. These records should include academic performance as well as professionally relevant behavioral tendencies.
- Hold mandatory "check-in" meetings between all program faculty and each new student during their first term. Offer these to all other students as an option, except for any students continuing to manifest unprofessional behaviors, poor academic performance, or who have some other need for a face-to-face meeting with faculty.
- Address problems while they are small, before they grow larger. This is equally true for tutoring technical concepts as it is for helping students build professional habits.
- Build rigorous quality control into the curriculum to ensure every student masters every important concept, and that the mastery is retained over time. This includes (1) review questions added to every exam to re-assess knowledge taught in previous terms, (2) cumulative exams at the end of every term to re-assess all important concepts back to the very beginning of the program, and (3) review assessments in practical (hands-on) coursework to ensure critically-important skills were indeed taught and are still retained. What you will find by doing this is that it actually boosts retention of students by ensuring that important knowledge gets taught and is retained over long spans of time. In the absence of such quality control, student learning and retention tends to be spotty and this contributes to drop-out and failure rates later in their education.
- Finally, *never rush learning.* Education is not a race. Give your students ample time to digest complex ideas, as you continually remind yourself of just how long it took you to achieve mastery! Long-term retention and the consistently correct application of concepts are always the result of *focused effort over long periods of time* which means there are no shortcuts to learning.

B.3 Proven strategies for students

The single most important piece of advice I have for any student of any subject is to take responsibility for your own development in all areas of life including mental development. Expecting others in your life to entirely guide your own development is a recipe for disappointment. This is just as true for students enrolled in formal learning institutions as it is for auto-didacts pursuing learning entirely on their own. Learning to think in new ways is key to being able to gainfully use information, to make informed decisions about your life, and to best serve those you care about. With this in mind, I offer the following advice to students:

- **Approach all learning as valuable**. No matter what course you take, no matter who you learn from, no matter the subject, there is something useful in every learning experience. If you don't see the value of every new experience, you are not looking closely enough!
- **Continually challenge yourself**. Let other people take shortcuts and find easy answers to easy problems. The purpose of education is to stretch your mind, in order to shape it into a more powerful tool. This doesn't come by taking the path of least resistance. An excellent analogy for an empowering education is productive physical exercise: becoming stronger, more flexible, and more persistent only comes through intense personal effort.
- **Master the use of language**. This includes reading extensively, writing every day, listening closely, and speaking articulately. To a great extent language channels and empowers thought, so the better you are at wielding language the better you will be at grasping abstract concepts and articulating them not only for your benefit but for others as well.
- **Do not limit yourself to the resources given to you**. Read books that are not on the reading list. Run experiments that aren't assigned to you. Form study groups outside of class. Take an entrepreneurial approach to your own education, as though it were a business you were building for your future benefit.
- **Express and share what you learn**. Take every opportunity to teach what you have learned to others, as this will not only help them but will also strengthen your own understanding[2].
- Realize that **no one can give you understanding**, just as no one can give you physical fitness. These both must be *built*.
- **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[3] effort, and never give up! That concepts don't immediately come to you is not a sign of something wrong, but rather of something right: that you have found a worthy challenge!

[2] On a personal note, I was surprised to learn just how much my own understanding of electronics and related subjects was strengthened by becoming a teacher. When you are tasked every day with helping other people grasp complex topics, it catalyzes your own learning by giving you powerful incentives to study, to articulate your thoughts, and to reflect deeply on the process of learning.

[3] 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.

B.4 Design of these learning modules

"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. Every effort has been made to embed the following instructional and assessment philosophies within:

- 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[4] 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[5] 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[6] 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

[4] 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.

[5] 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.

[6] 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 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.

To high standards of education,

Tony R. Kuphaldt

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

"422 and 485 Standards Overview and System Configurations" Application Report SLLA070C, Texas Instruments Incorporated, Dallas, TX, 2002.

Baudot, J.M.E., *US Patent 388,244*, "Printing Telegraph", application 4 April 1883, patent granted 21 August 1888.

Floyd, Thomas L., *Digital Fundamentals*, 6th edition, Prentice-Hall, Inc., Upper Saddle River, NJ, 1997.

"FOUNDATION Fieldbus System Engineering Guidelines" (AG 181) Revision 2.0, The Fieldbus Foundation, 2004.

"FOUNDATION Specification System Architecture" (FF 581) Revision FS 1.1, The Fieldbus Foundation, 2000.

"Fundamentals of RS-232 Serial Communications" Application Note 83 (AN83), Maxim Integrated Products, 2001.

Graham, Frank D., *Audels New Electric Library, Volume IX*, Theo. Audel & Co., New York, NY, 1942.

HART Communications, Technical Information L452 EN; SAMSON AG, 1999.

Horak, Ray, *Telecommunications and Data Communications Handbook*, John Wiley & Sons, Inc., New York, NY, 2007.

Horak, Ray, *Webster's New World Telecom Dictionary*, Wiley Publishing, Inc., Indianapolis, IN, 2008.

"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.

Newton, Harry, *Newton's Telecom Dictionary*, CMP Books, San Francisco, CA, 2005.

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.

Polybius, *Histories*, cited from pages 49-50 of *Lapham's Quarterly* Volume IX, Number 1, Winter 2016, American Agora Foundation, New York, NY, 2016.

"Recommendation ITU-R M.1677 International Morse Code", ITU Radiocommunication Assembly, 2004.

Rector, B.E. et al.,*Industrial Electronics Reference Book*, Westinghouse Electric Corporation, John Wiley & Sons Inc., New York, NY, 1948.

"Selecting and Using RS-232, RS-422, and RS-485 Serial Data Standards" Application Note 723 (AN723), Maxim Integrated Products, 2000.

Spurgeon, Charles E., *Ethernet: The Definitive Guide*, O'Reilly Media, Inc., Sebastopol, CA, 2000.

Svacina, Bob, *Understanding Device Level Buses: A Tutorial*, *Interlink*BT, LLC, Minneapolis, MN, 1998.

Thoreau, Henry David, *Walden*, or *Life In The Woods*, Ticknor and Fields, Boston, MA, 1854.

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.

7 March 2026 – added a Morse Code Technical Reference section.

9-10 February 2026 – minor edits to the Tutorial. Also, added a comment to the C program listing in the "Microcontroller UART text and number program" Conceptual Reasoning question.

8 January 2026 – minor edit to one of the Conceptual Reasoning questions, changing PLC5 to PLC-5.

30 December 2025 – added a new Conceptual Reasoning question related to serial string programming in a PLC.

9 September 2025 – minor edits to the "Manchester data frame with a specified bit rate" Quantitative Reasoning question.

19 May 2025 – added a comment to the Tutorial chapter about how the Baudot code is still used in the present day, in the form of RTTY digital radio communication.

3-11 February 2025 – added a footnote on some common Linux tools for generating hash codes from files, and made other minor edits to the Tutorial as well. Also added a note in the "Flow control" section of the Tutorial defining RTS and CTS signals.

9 October 2024 – added content on return-to-zero (RZ) bit encoding to the "Physical encoding of bits" section of the Tutorial chapter.

9-15 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. Also added a new Case Tutorial section showing UART data frame examples with

and without parity.

7 February 2024 – added a Challenge Question to the "Microcontroller UART text and number program" Conceptual Reasoning question. Also added some introductory information to the "EIA/TIA-232 data frames of ASCII characters" Conceptual Reasoning question.

1 January 2024 – added a question based on the TI MSP430G2553 microcontroller's UART (serial RS-232) communications.

12 January 2023 – minor grammatical edit made in the Tutorial chapter.

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.

9 August 2022 – clarified distinctions between Morse and Continental Codes.

30 May 2022 – added reference for Morse Code pause lengths.

16 September 2021 – minor additions to some instructor notes.

9 July 2021 – replaced some TeX-style italicizing markup with LaTeX-style.

10 May 2021 – commented out or deleted empty chapters.

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

9-10 February 2021 – minor additions to the Introduction chapter, and added a Case Tutorial section showing an ASCII character communicated via EIA/TIA-232. Also, elaborated on the concept of determinism in the Tutorial chapter.

15-16 September 2020 – minor additions to the Introduction chapter, as well as minor typographical error correction thanks to Ty Weich. Also made minor additions to the Tutorial on how OSI Reference Model layers apply (or don't apply) to certain communication standards.

30 August 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.

9 August 2020 – uncommented graphic images which had been left commented out during development (to speed the LaTeX compilation process).

21 June 2020 – moved example of hash algorithms out of the Tutorial and into the Technical References chapter.

17 June 2020 – added discussion of UARTs to the Tutorial, as well as some Foundational Concepts.

19 May 2020 – completed Introduction section, added questions.

17 May 2020 – document first created.

Index

2G cellular network, 36
802.11, 38
802.3, 23, 30
802.4, 35
802.5, 35

Adding quantities to a qualitative problem, 134
Amateur radio, 16
American Morse code, 14
Annotating diagrams, 133
Arbitration, channel, 33
ARINC 429, 18
Asynchronous data transfer, 16, 23
Avionics, 18

Baud rate, 22
Baudot code, 16, 17
Bell 202 standard, 20
Bit rate, 21
bps, 21

CAN Bus network, 37
Carrier Sense Multiple Access (CSMA), 37
Channel arbitration, 33
Checking for exceptions, 134
Checking your work, 134
Code, computer, 143
Collision, 37
Continental code, 14
CRC, 30
CRC32 hash algorithm, 52
CSMA, 37
CSMA/BA channel arbitration, 37
CSMA/CA, 38
CSMA/CD channel arbitration, 37
Cyclic redundancy check, 30

Determinism, network communication, 38
Digest, 30
Dimensional analysis, 133
Duplex, 33

Edwards, Tim, 144
Error detection, 9
Ethernet, 19, 30
Even parity, 9

Farnsworth timing, 49
Fire signals, 44
Flow control (serial data communication), 31
FOUNDATION Fieldbus H1, 19
Frame check sequence, 30
Frequency Shift Keying, 20
FSK, 20
Full-duplex, 33

Graph values to solve a problem, 134
Greenleaf, Cynthia, 95
GSM cellular network, 36

Half-duplex, 33
Handshaking (serial data communication), 31
Hash algorithm, 30
Hash collision, 30
How to teach with these modules, 141
Hwang, Andrew D., 145
Hyperterminal, 24

Identify given data, 133
Identify relevant principles, 133
Intermediate results, 133
International Morse Code, 48
International Morse code, 14
Inverted instruction, 141

Jabber, 38

Jabber latch, 38

Kermit, 24
Knuth, Donald, 144

Lamport, Leslie, 144
Limiting cases, 134

Manchester encoding, 19
Mark, 17, 20
Master-slave channel arbitration, 34
Maxwell, James Clerk, 43
Metacognition, 100
Minicom, 24
Modbus, 45
Moolenaar, Bram, 143
Morse code, 14
Morse, Samuel, 14
Murphy, Lynn, 95

Non-Return-to-Zero, 17
NRZ, 17

Odd parity, 9
Open-source, 143
OSI Reference Model, 41

Parity, 11, 24, 25
Parity bit, 9, 26
PLC, 34
Polling, 34
Polybius, Greek historian, 44
Positive logic, 8
Problem-solving: annotate diagrams, 133
Problem-solving: check for exceptions, 134
Problem-solving: checking work, 134
Problem-solving: dimensional analysis, 133
Problem-solving: graph values, 134
Problem-solving: identify given data, 133
Problem-solving: identify relevant principles, 133
Problem-solving: interpret intermediate results, 133
Problem-solving: limiting cases, 134
Problem-solving: qualitative to quantitative, 134
Problem-solving: quantitative to qualitative, 134
Problem-solving: reductio ad absurdum, 134
Problem-solving: simplify the system, 133
Problem-solving: thought experiment, 5, 133
Problem-solving: track units of measurement, 133
Problem-solving: visually represent the system, 133
Problem-solving: work in reverse, 134
Profibus PA, 19
Programmable logic controller, 34

Qualitatively approaching a quantitative problem, 134

Radio teletype, 16
Radio, amateur, 16
Radiotelegraph, 14
Reading Apprenticeship, 95
Reductio ad absurdum, 134, 140, 141
Return-to-Zero, 18
RTTY, 16
RZ, 18

Schoenbach, Ruth, 95
Scientific method, 100
Self-clocking, 18, 19
SHA1 hash algorithm, 53
SHA256, 30
Simplex, 33
Simplifying a system, 133
Socrates, 140
Socratic dialogue, 141
Space, 17, 20
SPICE, 95
Stallman, Richard, 143
Store and forward, 38
Synchronous data transfer, 16, 23

TDMA, 36
Telegraph, 14
Thought experiment, 5, 133
Time Division Multiple Access (TDMA), 36
Token Bus, 35
Token Ring, 35
Token-passing channel arbitration, 35
Torvalds, Linus, 143

UART, 24
Units of measurement, 133

Variable-frequency drive, 34
VFD, 34
Visualizing a system, 133

WLAN, 38
Work in reverse to solve a problem, 134
WYSIWYG, 143, 144