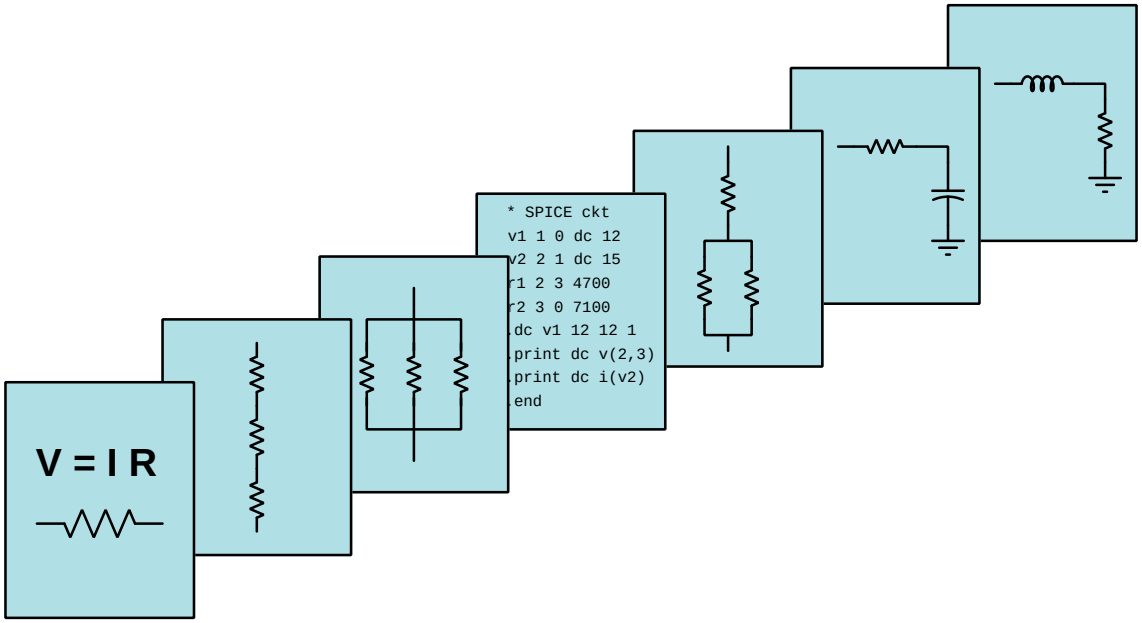
Modular Electronics Learning (ModEL) project



Digital Numeration

Last update = 21 November 2024

Contents

Chapter 1

Introduction

1.1 Recommendations for students

Digital words may represent a wide variety of things. We may use digital words to represent whole (natural) numbers, integers, alphabetical characters, machine positions, and analog quantities such as voltage, among others. In this learning module we will focus on the use of digital words to represent numerical quantities.

Important concepts related to digital numeration include **natural** numbers versus **integers** versus **rational** versus **irrational** numbers, **signed** versus **unsigned** integers, **radix**, **two's complement**, **encoders**, **fixed-point notation**, **binary-coded decimal** (BCD), **hexadecimal** notation, **octal** notation, **remainder** (division), **scientific notation**, **floating-point** notation, **mantissa**, **big-endian** versus **little-endian** formats, **registers**, **swapping** bytes and words, and number system **incompatibilities**.

An important problem-solving technique applied in the Tutorial is the *thought experiment*. Another technique is the *cut-and-try* method.

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

- What are some alternatives to our common decimal-based numeration system?
- Which numeration systems result in the most compact expressions?
- Which numeration systems result in the lengthiest expressions?
- What types of numerical quantities cannot be expressed in certain numeration systems?
- Why is binary the preferred numeration system for digital electronic circuits?
- How does the concept of a *place-weight* apply to multiple numeration systems?
- How may we calculate the total possible amount of numbers given a limited range of characters?
- How does BCD differ from binary?

- What practical purpose does octal or hexadecimal serve for human beings working with binary quantities?
- What are some different ways to perform the same type of numeration system conversion?
- How do we express negative quantities using binary?
- What purpose does floating-point notation serve?
- What does a shift register do?
- What is the concept of "serial" data communication?
- What are some different ways in which multiple numeration schemes may be incompatible with each other?

1.2 Challenging concepts related to numeration systems

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

- **Non-decimal numeration** – decimal numeration is so commonplace that many people assume this is the *only* possible way to represent numbers. So, when faced with non-decimal numeration schemes such as binary, octal, or hexadecimal the first impression may be unnerving. Fortunately, though, these are all *place-weighted* systems which means a close examination of how decimal actually works will shed light on how the others work too.

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 show how each of the counting sequences results were obtained by the author in the Tutorial chapter's examples.

- **Outcome** – Manually convert between different systems of integer numeration

  Assessment – Determine equivalent representations in binary, octal, and hexadecimal for given integer values; e.g. pose problems in the form of the "Integer conversion table" Quantitative Reasoning question.

- **Outcome** – Manually convert between decimal and floating-point binary numeration

  Assessment – Determine the decimal value of a given floating-point binary number.

  Assessment – Determine the floating-point binary equivalent for a given decimal value.

- **Outcome** – Independent research

  Assessment – Read and summarize in your own words reliable source documents on the subject of numeration systems utilized by cultures around the world.

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: signed integer examples

Here are all possible bit-combinations for a three-bit signed integer binary number, along with their decimal equivalent values. All place-weight values are the same as for unsigned binary, except for the most-significant bit (MSB) which has a place-weight of *negative* four:

- `000` = 0
- `001` = 1
- `010` = 2
- `011` = 3
- `100` = −4
- `101` = −3
- `110` = −2
- `111` = −1

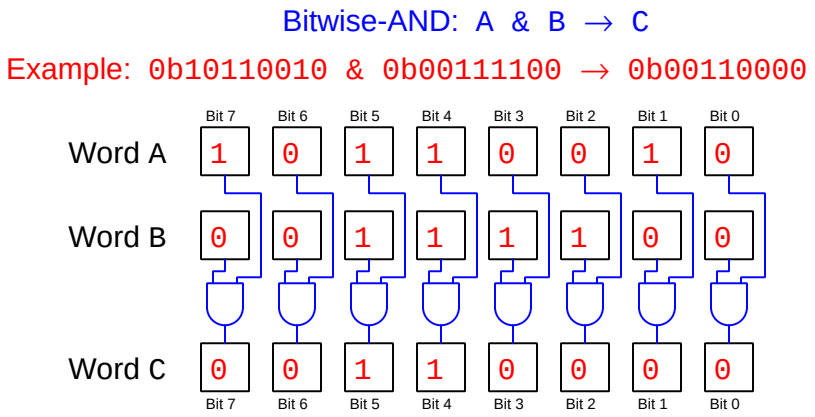
Here are all possible bit-combinations for a four-bit signed integer binary number, along with their decimal equivalent values. All place-weight values are the same as for unsigned binary, except for the most-significant bit (MSB) which has a place-weight of *negative* eight:

- `0000` = 0
- `0001` = 1
- `0010` = 2
- `0011` = 3
- `0100` = 4
- `0101` = 5
- `0110` = 6
- `0111` = 7
- `1000` = −8
- `1001` = −7
- `1010` = −6
- `1011` = −5
- `1100` = −4
- `1101` = −3
- `1110` = −2
- `1111` = −1

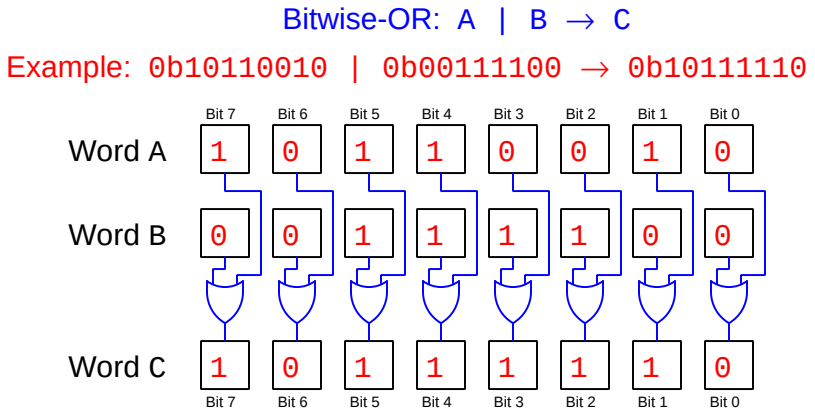
2.2 Example: bitwise logical operations

The following examples show bitwise operations being performed on 8-bit words (bytes).

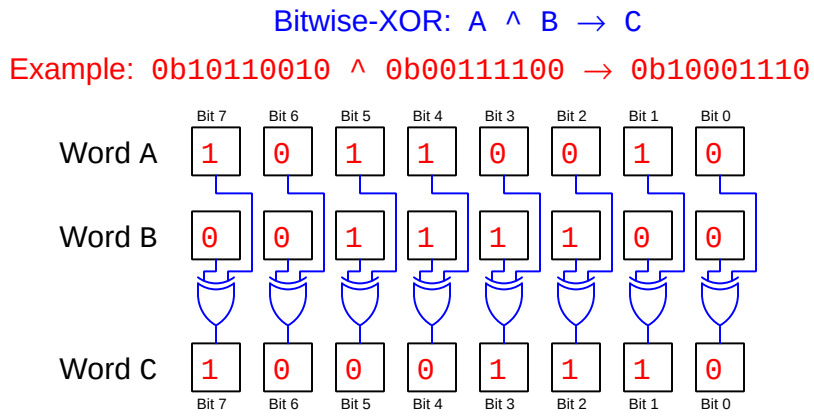
2.2.1 Bitwise-AND



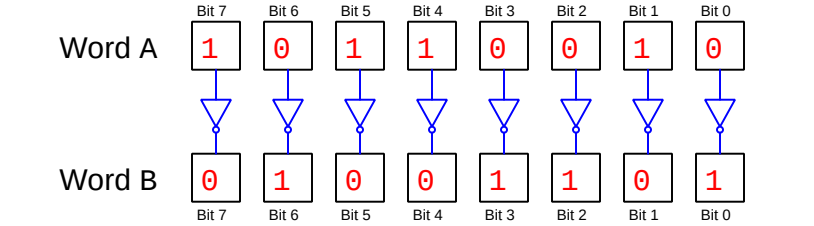
2.2.2 Bitwise-OR



2.2.3 Bitwise-XOR



2.2.4 Bitwise-complement



Chapter 3

Tutorial

For any technically literate person, numbers are as familiar as letters in the alphabet: we tend to take them for granted after learning how to express numerical quantities with these symbols. However, when studying the representation of numbers using electronic circuit states rather than written symbols, we must re-examine our assumptions.

3.1 Numbers versus numeration

First we must clearly distinguish between *numbers* versus the *symbols* (i.e. *numeration*) we commonly use to represent numbers. For example, if I say there are *one dozen* apples in a box, we clearly understand that to mean *twelve* apples. Furthermore, we would most likely write that on paper using two symbols, a "1" followed by a "2" as shown here:

`12` apples in the box

This, however, is not the only way to symbolically represent the number twelve. Alternatively, we could elect to use hash-marks to represent the number of apples, or even Roman numerals:

`|||||` `|||||` `||` apples in the box `XII` apples in the box

You may notice that our "decimal" system of numeration uses fewer written characters (called *ciphers*) to express the number twelve than either hash-marks or Roman numerals. This efficiency improves as the numbers in question become much larger, and it is entirely due to the *place-weighted* nature of the decimal numeration system. Each numeral, or cipher, occupies a *place* with a particular *weight value* associated. We would say that in the number twelve the numeral `1` occupies the *ten's* place while the numeral `2` occupies the *one's* place, and therefore the decimal number `12` means one portion of ten added to two portions of one (i.e. $12 = 10 + 2$). Larger decimal numbers use more places, with each successive place-weight being ten times the one before it: after the ten's place comes the *hundred's* place followed by the *thousand's* place, etc.

Please note that the decimal expression of twelve is not the same thing as the quantity *twelve*, but rather merely a convention for expressing that quantity. Decimal notation happens to be very efficient and works well for human beings to use, but it is not the only way to express quantities. As we will soon see, other methods exist for representing numerical quantities which lend themselves better to electronic switching circuits than the decimal numeration system.

Before we begin exploring alternative numeration systems, it is worth reviewing some of the different types of numerical quantities we may need to represent with any numeration system. What follows are some examples of common number types:

- Who1e numbers (or "Natural" numbers): 1, 2, 3, 4, 5, 6, 7
- Integer numbers: -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5
- Fractional (rational) numbers: $\frac{1}{2}$, $\frac{3}{4}$, $-\frac{5}{17}$
- Irrational[1] numbers: π, e
- Real numbers: the combined set of whole, integer, fractional, and irrational numbers
- Imaginary numbers: $\sqrt{-1}$ or any multiple thereof
- Complex numbers: the combined set of real and imaginary numbers

Standards exist to represent every one of these number types in digital form, but since the simplest case is whole numbers (commonly called *unsigned integers* in computer programming) we will start our exploration there.

3.2 Place-weighted numeration

Modern numeration schemes use the concept of *place-weighting* to express large ranges of numerical quantities with an economy of symbols. All readers of this text are no doubt very familiar with *decimal* notation, where we have a one's place, a ten's place, a hundred's place, and so on in successively weightier digit-places. For example, the number 378 really means "3" hundreds plus "7" tens plus "8" ones (i.e. $(3)(100) + (7)(10) + (8)(1)$) for a total of three hundred seventy eight. The decimal place-weighted system of numeration is far more efficient in terms of symbols used than hash-marks, Roman numerals (`CCCLXXVIII`), or some other schemes we might use.

The use of place-weights assigns different multiplier values to each of the digits of a multi-digit decimal number. The fact that each successive place-weight differs in value from the adjacent place ten-fold is a consequence of there being exactly ten unique symbols in the decimal system (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9). In other words, decimal is a *base-ten* numeration system with ten unique symbols and place-weights following powers of ten.

[1] Here, the term "irrational" does not mean illogical or nonsense as the word "irrational" is used in common parlance. In mathematics, a *rational* number is one which may be expressed as a quotient (i.e. a fraction) of integers, literally as a *ratio* (hence, "rational") of integers. By contrast, an *irrational* number is one that cannot be expressed by any quotient of integers. For example, many students of mathematics learn to approximate pi as $\frac{22}{7}$, but this is not actually equal to π as any quick check on a hand calculator will prove: $\frac{22}{7} = 3.\overline{142857} \cdots$; $\pi = 3.1415926535 \cdots$. One of the hallmarks of an irrational number is that its digits continue indefinitely without ever repeating, while a rational number's digits either always repeat at some point or do not repeat at all as in the case of an integer number.

If we count in decimal from zero upwards using two-digit representation, we see a pattern of how the digit values and the place-weights work together to make successively larger totals:

Decimal characters	**Word representation**	**Sum**
00	Zero	0 tens + 0 ones
01	One	0 tens + 1 one
02	Two	0 tens + 2 ones
03	Three	0 tens + 3 ones
04	Four	0 tens + 4 ones
05	Five	0 tens + 5 ones
06	Six	0 tens + 6 ones
07	Seven	0 tens + 7 ones
08	Eight	0 tens + 8 ones
09	Nine	0 tens + 9 ones
10	Ten	1 ten + 0 ones
11	Eleven	1 ten + 1 one
12	Twelve	1 ten + 2 ones
13	Thirteen	1 ten + 3 ones
14	Fourteen	1 ten + 4 ones
15	Fifteen	1 ten + 5 ones
16	Sixteen	1 ten + 6 ones
17	Seventeen	1 ten + 7 ones
18	Eighteen	1 ten + 8 ones
19	Nineteen	1 ten + 9 ones
20	Twenty	2 tens + 0 ones
21	Twenty-one	2 tens + 1 one

As soon as we run out of unique symbols when counting upwards, we must increment the digit in the next-higher place and start the pattern over again in the previous place: so, the next count after nine (09) is ten (10) where the one's place returns to 0 and the ten's place increments from 0 to 1. This same pattern occurs in the transition from 19 to 20, and so on.

Such patterns are easy to take for granted in a numeration system like decimal we've seen our entire lives, but their explicit understanding becomes essential when counting in numeration systems other than our familiar base-ten. For example, consider the *octal* numeration system which is *base-eight*, using eight unique symbols (0, 1, 2, 3, 4, 5, 6, and 7) with places having weights defined by powers of eight:

Octal characters	**Word representation**	**Sum**
00	Zero	0 eights + 0 ones
01	One	0 eights + 1 one
02	Two	0 eights + 2 ones
03	Three	0 eights + 3 ones
04	Four	0 eights + 4 ones
05	Five	0 eights + 5 ones
06	Six	0 eights + 6 ones
07	Seven	0 eights + 7 ones
10	Eight	1 eight + 0 ones
11	Nine	1 eight + 1 one
12	Ten	1 eight + 2 ones
13	Eleven	1 eight + 3 ones
14	Twelve	1 eight + 4 ones
15	Thirteen	1 eight + 5 ones
16	Fourteen	1 eight + 6 ones
17	Fifteen	1 eight + 7 ones
20	Sixteen	2 eights + 0 ones
21	Seventeen	2 eights + 1 one
22	Eighteen	2 eights + 2 ones
23	Nineteen	2 eights + 3 ones
24	Twenty	2 eights + 4 ones
25	Twenty-one	2 eights + 5 ones

As strange as it may look to see "10" representing the number eight, this makes perfect sense if the numeration system is base-eight rather than base-ten. Our association of "10" with the value *ten* is nothing more than a cultural assumption of all numbers belonging to a *decimal* system.

A *base-sixteen* system called *hexadecimal* uses sixteen different symbols (0 through 9 followed by A through F), with place-weights scaling by powers of sixteen:

Hexadecimal characters	**Word representation**	**Sum**
00	Zero	0 sixteens + 0 ones
01	One	0 sixteens + 1 one
02	Two	0 sixteens + 2 ones
03	Three	0 sixteens + 3 ones
04	Four	0 sixteens + 4 ones
05	Five	0 sixteens + 5 ones
06	Six	0 sixteens + 6 ones
07	Seven	0 sixteens + 7 ones
08	Eight	0 sixteens + 8 ones
09	Nine	0 sixteens + 9 ones
0A	Ten	0 sixteens + A (ten) ones
0B	Eleven	0 sixteens + B (eleven) ones
0C	Twelve	0 sixteens + C (twelve) ones
0D	Thirteen	0 sixteens + D (thirteen) ones
0E	Fourteen	0 sixteens + E (fourteen) ones
0F	Fifteen	0 sixteens + F (fifteen) ones
10	Sixteen	1 sixteen + 0 ones
11	Seventeen	1 sixteen + 1 one
12	Eighteen	1 sixteen + 2 ones
13	Nineteen	1 sixteen + 3 ones
14	Twenty	1 sixteen + 4 ones
15	Twenty-one	1 sixteen + 5 ones
16	Twenty-two	1 sixteen + 6 ones
17	Twenty-three	1 sixteen + 7 ones
18	Twenty-four	1 sixteen + 8 ones
19	Twenty-five	1 sixteen + 9 ones
1A	Twenty-six	1 sixteen + A (ten) ones

Perhaps the most useful numeration system in electronics is *binary* where only two symbols exist (0 and 1), and where the place-weights scale by powers of two. This results in a one's place followed by a two's place followed by a four's place, etc. The benefit of binary is that each character (called a *bit*) is either 1 or 0 which maps nicely to a transistor being either on or off, a switch being closed or open, etc. and so the binary numeration system lends itself well to numerical representation by digital electronic circuitry. The following table shows a count sequence of zero through fifteen using four binary bits:

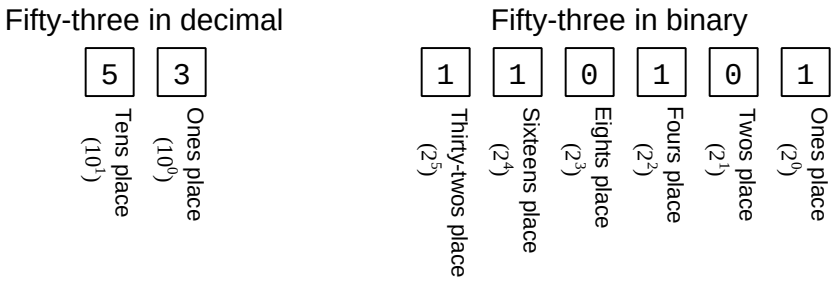
Binary characters	**Word representation**	**Sum**
0000	Zero	0 eights + 0 fours + 0 twos + 0 ones
0001	One	0 eights + 0 fours + 0 twos + 1 one
0010	Two	0 eights + 0 fours + 1 two + 0 ones
0011	Three	0 eights + 0 fours + 1 two + 1 one
0100	Four	0 eights + 1 four + 0 twos + 0 ones
0101	Five	0 eights + 1 four + 0 twos + 1 one
0110	Six	0 eights + 1 four + 1 two + 0 ones
0111	Seven	0 eights + 1 four + 1 two + 1 one
1000	Eight	1 eight + 0 fours + 0 twos + 0 ones
1001	Nine	1 eight + 0 fours + 0 twos + 1 one
1010	Ten	1 eight + 0 fours + 1 two + 0 ones
1011	Eleven	1 eight + 0 fours + 1 two + 1 one
1100	Twelve	1 eight + 1 four + 0 twos + 0 ones
1101	Thirteen	1 eight + 1 four + 0 twos + 1 one
1110	Fourteen	1 eight + 1 four + 1 two + 0 ones
1111	Fifteen	1 eight + 1 four + 1 two + 1 one

As you can see, there are multiple ways to apply the place-weighting concept. What we consider "normal" (base-ten) is nothing more than a cultural convention, most likely based on the fact that humans have ten fingers. Binary lends itself exceptionally well to digital electronic circuitry where logic signal states are limited to only two possibilities (1 or 0, true or false, high or low). Hexadecimal actually serves a useful purpose too for digital systems, as a condensed representation of binary numbers: each hexadecimal character with its range of zero to fifteen representing four binary bits with the same zero-to-fifteen range (0000 to 1111). For example, the binary number 10011101 is equal to the hexadecimal number 9D (1001 being 9 and 1101 being D). Likewise, the hexadecimal number B6 is equal to the binary number 10110110 (B being 1011 and 6 being 0110).

3.3 Unsigned integers

An electric or electronic *switching* circuit[2] has only two valid states, *on* and *off*, and we may use these discrete states to represent numerical values of 1 and 0. If we construct a numeration system using only these two ciphers (i.e. a *base-two* system, as opposed to our decimal system which is *base-ten* because it uses ten unique ciphers 0 through 9 and has place-weight values that are powers of ten) we may represent whole numbers in a manner similar to whole numbers in the decimal system. Instead of referring to these 1 and 0 cipher as *digits*, we will refer to them as *bits*.

Let's explore this idea by example. Consider the number *fifty three* represented in both decimal form (base-10) and binary form (base-2). In each numeration system we have a series of *places* for ciphers to occupy, and each of those places has an associated *weight*:



Fifty-three = $(5 \times 10^1) + (3 \times 10^0) = 50 + 3$

Fifty-three = $(1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$
Fifty-three = $(1 \times 32) + (1 \times 16) + (0 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1)$

In either numeration system – decimal or binary – the total value is equal to the sum of all the ciphers multiplied by their respective place-weight values. Fifty three written as a decimal number is five times ten, plus 3 times one. Fifty three written as a binary number is one times thirty two, plus one times sixteen, plus one times four, plus one times one[3].

It should not be difficult to see that the weight values for each place in either numeration system are mathematical powers of the numeration system's base value. For the decimal system (base-ten) the place-weights are powers of ten: a one's place (10^0) and a ten's place (10^1), and if our example number were much larger we would need a hundred's place (10^2) and perhaps a thousand's place (10^3). Similarly, in the binary system (base-two) the place-weights are powers of two: one's (2^0), two's (2^1), four's (2^2), eight's (2^3), sixteen's (2^4), and thirty-two's (2^5). Note how the least-significant place (i.e. the far-right digit or bit place in the number) weight is always one. Base value, also known as *radix*, is often shown as a trailing subscript to denote the numeration system when

[2] Actually, a great many physical systems may be reduced to just two states: magnetic polarization (north versus south), light waves (present or absent), particle spin (up or down), etc. which extends the discrete representation of data to systems other than electric/electronic switching circuits. In fact, many digital data storage technologies exploit this fact by using non-electrical means to encode 1 and 0 states.

[3] We may omit the place-weights carrying "0" bits, in this case the eight's and two's places, because they do not contribute to the sum.

written so as to avoid confusion[4]. Our example value of fifty three could be written in decimal as 53_{10} or in binary as 110101_2. In computer programming where the code exists as plain text lacking the capability of subscripts, decimal is assumed unless the number is prefaced by a special character sequence identifying it as non-decimal: in the case of binary that character sequence is often `0b`, so we would write the binary value fifty-three as `0b110101`.

The largest whole number representable in any place-weighted numeration system is equal to the base value raised to the power of the number of places (i.e. number of characters in the number) minus one, and is realized when all places are filled with the largest cipher. For example, a two-digit decimal number can represent ninety nine (99_{10}) as its maximum value ($10^2 - 1$) when all digits are set to "9", and a six-bit binary number can represent sixty three (111111_2) as its maximum value ($2^6 - 1$) when all bits are set to "1".

Below are listed some binary-decimal equivalent number pairs. You, the reader, are advised to try converting each of the binary numbers into decimal form by adding up all the place-weight values containing "1" bits, for practice. This is an example of *active reading* where you engage with the text in a deeper way than merely reading what's presented, in this case testing your understanding of binary numeration by trying to convert binary to decimal on your own:

- `0b101` (binary) = 5 (decimal)
- `0b111` (binary) = 7 (decimal)
- `0b11000` (binary) = 24 (decimal)
- `0b11101` (binary) = 29 (decimal)
- `0b110010` (binary) = 50 (decimal)
- `0b111111` (binary) = 63 (decimal)
- `0b1000000` (binary) = 64 (decimal)
- `0b1001101` (binary) = 77 (decimal)
- `0b1111111` (binary) = 127 (decimal)
- `0b10000100` (binary) = 132 (decimal)
- `0b10110111` (binary) = 183 (decimal)
- `0b11111111` (binary) = 255 (decimal)

Before we proceed to more advanced forms of binary numeration, it is useful to reflect on the significance of what we have accomplished so far: we now have a way to represent simple numerical quantities using nothing more than discrete *bits* which may take the form of electrical switch states, voltage or current pulses, magnetization states, optical transparency/reflectivity, etc.

[4] Imagine someone seeing the binary number `110101` written on a page and thinking it was supposed to be decimal, which would make it one hundred and ten *thousand* one hundred and one! This mistake can be avoided by writing the binary number as 110101_2.

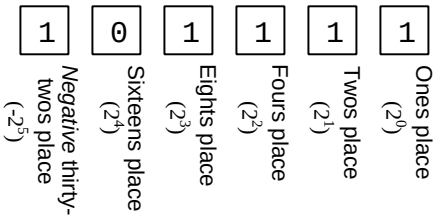
This fundamental principle allows us to encode not only numbers, but potentially *any* other form of data amenable to discrete states (e.g. alphabetical characters, color grades, standardized-pitch musical tones), into very simple physical forms. This is the essence of the *digital revolution*: encoding important data as large collections of bits which may be processed, archived, and communicated with relative ease.

3.4 Signed integers

As useful as whole numbers are, they do not form a number system complete enough for all practical purposes. It is important, for example, to be able to express *negative* quantities as well. The mathematical term for the set of numbers including whole numbers and their negative counterparts is *integer*. While "whole number" and "integer" are proper mathematical terms, it is more commonplace in the computer programming and digital electronics domains to refer to these number types as *unsigned* and *signed* integers, respectively.

When writing decimal numbers on paper, our solution for full integer representation is to prepend a "minus" symbol (−) to the left-hand side of the number, which is in essence adding one more cipher to our decimal numeration system. This is not an option for us in binary, the point of which is to represent numbers using *only* two states (1 and 0) for the purpose of exploiting the simplicity of electronic switching circuits, and therefore there is no third state available to serve as a negative symbol.

One possible solution is to simply create a new *place* for an additional bit, and interpret a 1 in that place as a negative symbol[5]. However, a more elegant solution has been invented called *two's complement* notation which achieves the same end. In two's complement notation, we make the most-significant bit (MSB) of the binary number bear a *negative* weight rather than a positive weight as all the other places. For example, the six-bit binary number scheme shown previously would have a *negative thirty two's place* followed by a sixteen's place followed by an eight's place, etc. In this scheme, a six-bit two's-complement representation of negative seventeen (−17) would be written as 101111_2:



A good way to understand two's complement notation is to test different patterns of 1 and 0 bits to see what these patterns will equate to when their place-weights are summed. This is an example of a kind of *thought experiment*, whereby we explore the consequences of some idea by imagining different conditions acting upon that idea. We will use the same six-bit digital word for consistency with prior examples, testing what happens when we set all bits to zero and when we set all bits to one:

$$000000_2 = (0)(-32) + (0)(16) + (0)(8) + (0)(4) + (0)(2) + (0)(1) = 0_{10}$$

$$111111_2 = (1)(-32) + (1)(16) + (1)(8) + (1)(4) + (1)(2) + (1)(1) = -1_{10}$$

Setting all bits to zero is unsurprising: the total sum is simply zero. Setting all bits to one, however, generates a novel result: *negative one*. In whole-number binary representation, setting all

[5] This scheme is referred to in computer science literature as *sign-magnitude notation.*

bits to one yields the largest possible value, but here (with the MSB bearing a negative weight) it does not.

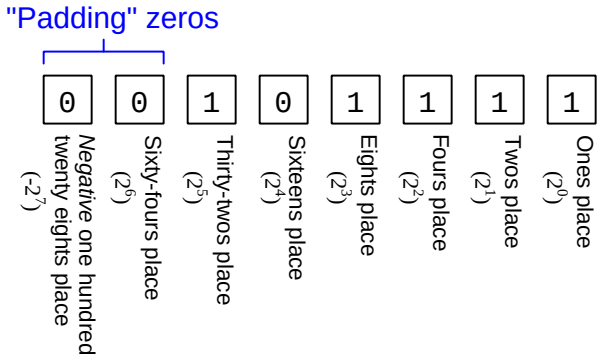
Having seen how the MSB of a two's complement binary number interacts with the other bits, we might try to find the binary patterns resulting in the largest positive and largest negative numbers possible with six bits. Since the MSB is the *only* negative place-weight, we ought to be able to achieve the largest negative number by setting only that bit to one (and the rest to zero). Likewise, we ought to be able to achieve the largest positive number possible by setting the MSB to zero and letting all other bits be one:

$$\texttt{100000}_2 = (1)(-32) + (0)(16) + (0)(8) + (0)(4) + (0)(2) + (0)(1) = -32_{10}$$

$$\texttt{011111}_2 = (0)(-32) + (1)(16) + (1)(8) + (1)(4) + (1)(2) + (1)(1) = 31_{10}$$

From these "thought experiments" we see that a six-bit two's complement binary number has a range extending from -32 to $+31$. This stands in contrast to the six-bit *whole* binary number with could range from 0 to $+63$. Note that in either case the six-bit number still has sixty four possible values (including zero as one of those), but with two's complement notation that range is offset with zero approximately in the middle instead of zero being the low end.

The fact that two's complement notation works by granting a negative weight to *only* the most-significant place means it is important for us to define the *width* of our digital word prior to using it to represent numerical values. Consider how our earlier example of a six-bit representation of -17 would appear if cast into a binary word *eight* bits in width:



$$\texttt{00101111}_2 = (0)(128) + (0)(64) + (1)(32) + (0)(16) + (1)(8) + (1)(4) + (1)(2) + (1)(1) = 47_{10}$$

Casting smaller binary numbers into larger word fields is not a problem when dealing with whole numbers only, where every place-weight has a positive value. Here, however, we must be careful – clearly, -17 and 47 are not the same quantities, but `101111` could mean either one depending on how many total bits are in the binary word!

Any signed negative binary number will differ in value from an unsigned binary number made up of the exact same bits by twice the value of its most-significant place-weight. Consider the previous example of `101111` which represents -17 as a signed integer but 47 as an unsigned integer: $47 - (-17) = 64$ is twice the most significant place-weight of a six-bit number ($2^5 = 32$). The reason for this should be clear with just a little thought: if that most-significant place-weight of a six-bit binary number represents -32 as signed but $+32$ as unsigned, then we would logically expect the signed-versus-unsigned interpretations of that six-bit number to disagree by the difference between $+32$ and -32, or 64.

This fact may be exploited as a means of identifying the bits needed to represent any given negative integer quantity. Suppose we wished[6] to express -50 in 8-bit signed binary format. The most-significant place-weight of an 8-bit number is either -128 (signed) or $+128$ (unsigned), and so whatever the combination of bits needed in signed binary to represent -50 will be the same combination of bits that represent $-50 + (2)(128)$ in unsigned form. This would be $+206$ which is `11001110` as an unsigned binary number. Therefore, `11001110` is the proper representation of -50 in 8-bit signed binary form.

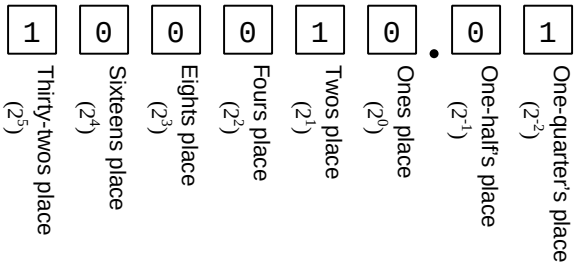
The *Introduction to computer programming* subsection of the *Quantitative reasoning* section of this learning module (beginning on page 77) contains example programs contrasting the use of signed versus unsigned binary numbers.

[6] Most modern electronic hand calculators will easily perform decimal-binary conversion for unsigned integers, but not necessarily for signed integers. This technique of adding double the most-significant place-weight works well to quickly determine a signed binary number's bit-states if we have an easy means of converting decimal to (unsigned) binary.

3.5 Fixed-point notation

In many applications we must digitally represent numbers with fractional values, lying between integer increments. If we assume that a "point" symbol lies between two places within a number, this will force characters to the left of that point to be whole and characters to the right of that point to be fractional. This is commonly called *fixed-point binary notation*.

Consider a case where an eight-bit word has a "binary point" located between the second and third bits (from the least-significant end) as shown in the following illustration:



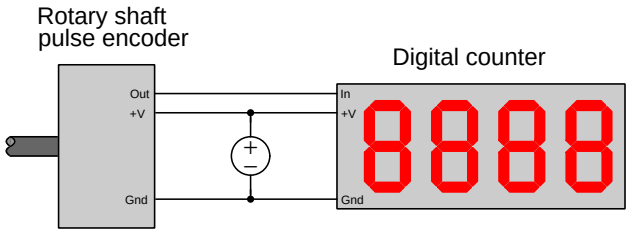
$$\texttt{100010.01}_2 = (1)(32) + (0)(16) + (0)(8) + (0)(4) + (1)(2) + (0)(1) + (0)(\tfrac{1}{2}) + (1)(\tfrac{1}{4}) = 34.25_{10}$$

In this scheme, the digital word `10001001` would be interpreted to have the binary value $\texttt{100010.01}_2$, which in turn would be equal to a decimal value of 34.25_{10}. If desired, we could also apply signed (two's complement) notation with a fixed binary point by making the MSB have a weight of -32 instead of $+32$, and thusly be able to represent both positive and negative quantities with fractional portions. The word *interpreted* is used quite intentionally here, for the existence of this "binary point" as well as the decision to use two's complement (or not) are our own arbitrary choices, neither one of which appears literally as a discrete electrical state or other physical entity. To the digital circuitry, this is nothing but an *eight bit word* (`10001001`). It is only our human *interpretation* of these bits as being an unsigned binary number having a fixed point located between six whole-weighted bits and two "fractional" bits (`wwwwww.ff`) that gives it definite numerical meaning.

With this example – an eight-bit word having six whole-numbered bits and two "fractional" bits – the smallest increment we may represent is one-quarter (0.25_{10}). Finer precision would require more bits to the right of the binary point, which may be achieved either by moving the binary point to the left (which will give us finer resolution but also limit the maximum representable values) and/or by adding more bits to the digital word which requires an expansion of hardware. The fundamental limit of eight bits is that it only has 256 possible combinations (2^8) of ones and zeroes, and so to optimize one form of numerical expression means we must compromise elsewhere.

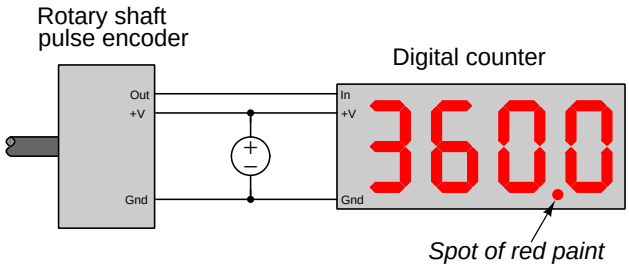
Now that we have seen how integers may be "forced" to express fractional values by interpretation, we will explore another way of applying the same concept: forcing the placement of a "point" after the binary integer value has been converted into decimal form.

Many years ago I encountered a very literal application of this concept while working on a digital motion indicator for a large industrial machine. This machine used a *rotary shaft pulse encoder* to track the angular position of one of its parts, to be displayed as a decimal number using 7-segment LED indicators. The encoder happened to output exactly 3600 pulses per revolution, or ten pulses per degree of shaft rotation, and the digital counter display had four decimal digits. With every low-to-high pulse transition output by the encoder, the counter would increment by one:



Obviously, if the encoder shaft turned one complete revolution, the counter would increment from `0000` to `3600`. What we desired, however, was for the counter to register actual *degrees* of rotation. In other words, what we needed was for the counter to register `360.0` after one full revolution of the shaft rather than `3600`.

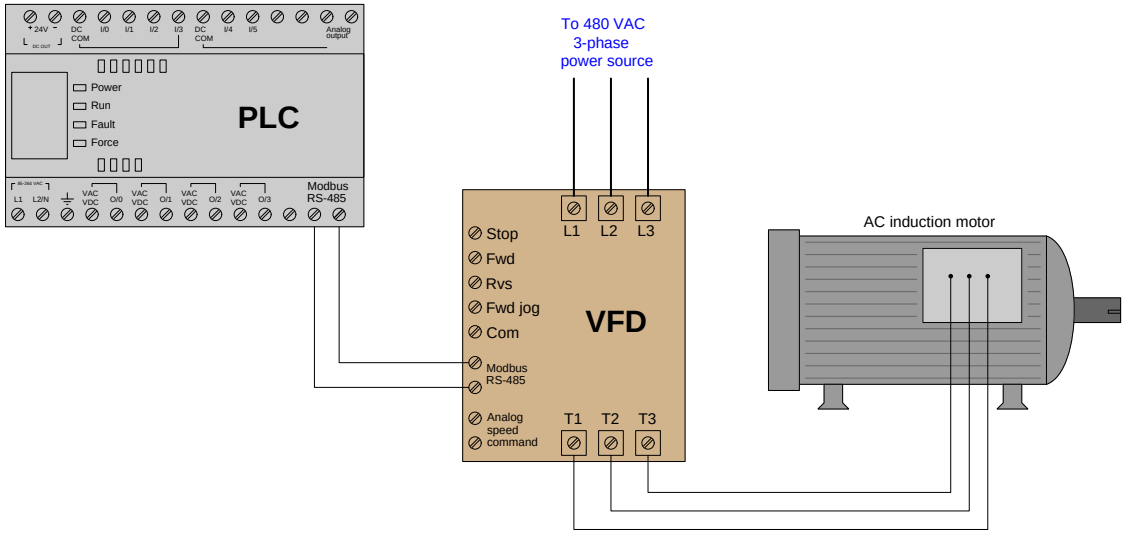
Internally, the digital counter used a 12-bit (unsigned) binary number to represent the number of pulses it received from the encoder, giving it the ability to accumulate up to 4095 pulses. It had no provision for a decimal point on its display, and so our solution was to place a dot of bright red paint between the least-significant digit and the next-most-significant digit on the counter's face:



This "paintbrushed" solution was a literal realization of *fixed-point decimal notation* where we forced a decimal point to appear between digits of the decimal number for the purpose of representing fractional values (in this case, tenths of a degree of rotation), even though the underlying binary value driving that display was just a simple integer.

Modern *Human-Machine Interface* (*HMI*) graphical displays provide this same solution in a more sophisticated form: the HMI device may be programmed to receive a binary integer from some other digital device via a *data network*, then insert a decimal point between specified digits while converting and displaying that binary word as a decimal number on a graphical screen.

A common industrial application of fixed-point decimal notation is in the speed command for a motor-control device called a *Variable Frequency Drive* or *VFD*. A VFD serves the purpose of controlling the shaft speed of an AC induction motor by varying the frequency of the AC electricity powering that motor. VFDs are digital devices, and as such they usually come equipped with digital network connections for receiving command codes from a computer. These command codes include such commands as Start, Stop, Forward, Reverse, and Speed (frequency). In the following illustration, a VFD is shown connected to the terminals of a three-phase AC induction motor, and also connected to the network[7] port of an industrial control computer called a *PLC* (*Programmable Logic Controller*).



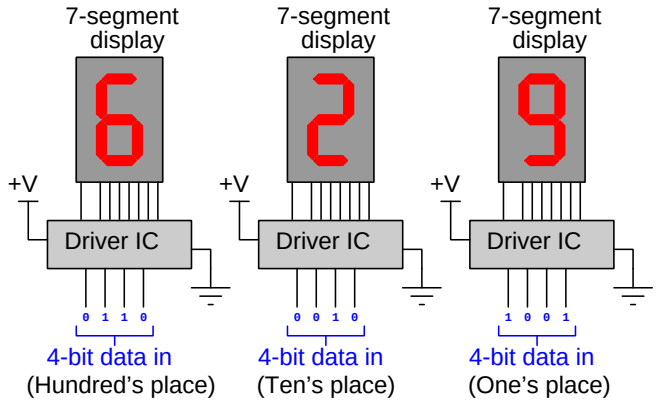
With the standard AC line frequency being either 60 Hz or 50 Hz in most parts of the world, an integer value for VFD output frequency would result in fairly coarse speed control (i.e. 0 Hz, 1 Hz, 2 Hz, . . . 59 Hz, 60 Hz). In order achieve speed adjustment finer than 1 Hz intervals, the integer number received by the VFD as the speed command is usually assumed to have an implied decimal point at the "tenth's" place, commonly notated as `XXX.X`. For example, to command such a VFD to output a frequency of 30 Hz (half-speed for a 60 Hz motor), the control computer would need to transmit the binary equivalent[8] of 300_{10} to the VFD. The VFD is programmed to interpret the received value of `300` as `30.0` which it then uses as a target value for output frequency. This fixed-point scheme allows for a speed resolution of 0.1 Hz which is much finer than the 1 Hz increments we would be forced to use without any decimal point fixing.

[7] Many different types of digital networks are popular for industrial use, including the Modbus/RS-485 network type shown here which uses a single pair of wires between the PLC and the VFD.

[8] For an unsigned integer, a decimal value of 300 would be represented as `1 0010 1100`, with as many padding zeros as necessary prepended to the left-hand side to round out the digital word. Since the Modbus network shown in the illustration happens to assume 16-bit words, the transmitted speed data would be `0000 0001 0010 1100`.

3.6 Binary-Coded Decimal

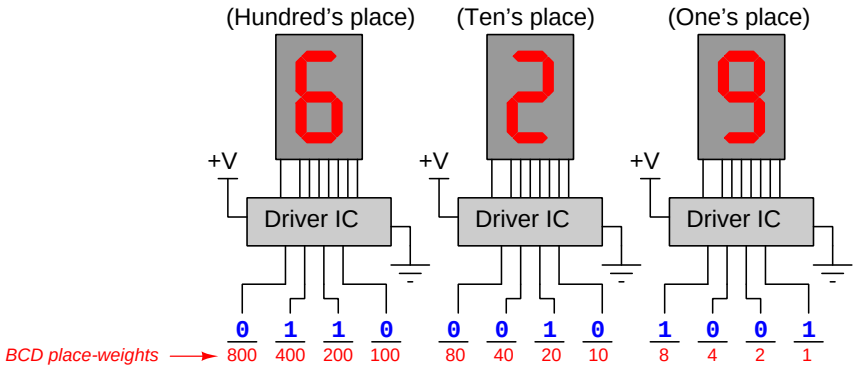
Legacy numerical display technologies such as 7-segment LEDs, 7-segment LCDs, and Nixie tubes[9] driven by special-purpose *display driver* integrated circuits (ICs) were once very common in digital systems. The seven illuminated (or shaded) segments of a 7-segment display are activated by the driver IC based on the status of the four data input lines. The illustration below shows a three-digital decimal display built using this technology, with examples of 4-bit data and the corresponding 7-segment patterns for the decimal number 629:



If we combine the three sets of 4-bit data into a single 12-bit digital word, we find that it does *not* represent six hundred twenty nine if interpreted as a regular binary integer:

$$\texttt{0110 0010 1001}_2 \neq 629_{10}$$

The reason for this mismatch is the place-weights of the respective bits. For normal binary integers, the place-weights are all powers of two. Here, however, since each group of four bits is supposed to represent a distinct *decimal* digit, the place-weights between 7-segment displays must differ by factors of *ten*:



[9] For the uninitiated, a *Nixie tube* is a clear glass tube containing metal elements shaped like numerals and filled with a gas such as neon which visibly glows when ionized. By electrically energizing different metal elements within a Nixie tube, any one of several numerals would be made to glow and thereby display one digit of a decimal number.

This digital numeration format is called *Binary Coded Decimal*, or *BCD*, because each set of four binary bits represents (i.e. encodes) a single decimal digit. Its use, although much less common now than it was in the days of discrete-digit numerical displays, was not limited just to display but also for input of number values into digital systems. One such legacy example is shown in the following photograph, where a set of four *thumb-wheel* switches appears on the face of an industrial weigh-feeding scale:

The digits 1120 are clearly visible in this photograph, as are the plastic "thumb-drive" tabs on each wheel allowing a human operator to increment or decrement each digit of the four-digit decimal value. Pushing up or down on these tabs would rotate a wheel, each wheel having the numerals 0 through 9 printed in white text along their circumference, with one of those numerals visible from the front of the switch at any given time. These thumb-wheel switches contained four sets of electrical contacts which generated a BCD code corresponding to the displayed value on the thumb-wheel's edge.

Occasionally you may even encounter a computer using BCD internally for its numeration. Certain models of industrial PLC (Programmable Logic Controller) were like this, which was often a source of confusion to any programmer accustomed to plain binary numeration.

3.7 Shorthand representations of digital words

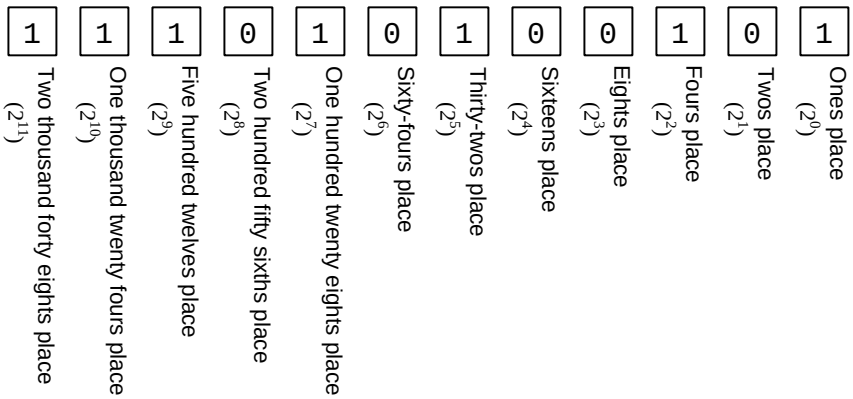
Modern digital systems can have very long word lengths, typically multiples of 8 bits (i.e. one *byte*[10]), with 32 bits being quite common for many applications. It is not difficult to understand the human difficulties of accurately reading and writing long binary words. Consider the following 32-bit binary number, written with spaces between every four bits to aid in readability:

```
1001 0010 0111 0100 1101 1000 0110 1010
```

Even with the spaces, it would be very easy to mis-read this long string of 32 bits.

For this reason "shorthand" notations for binary exist called *octal* and *hexadecimal* numeration. Both of these are place-weighted numeration systems like decimal and binary, differing only by base value: octal is base-eight while hexadecimal is base-sixteen. This means octal numbers are written only with the characters 0 through 7 as valid ciphers, and place-weights are powers of eight; hexadecimal numbers use all ten Arabic numerals (0 through 9) plus the first six letters of the English alphabet, usually capitalized (A through F), and its place-weights are powers of sixteen.

The base values for octal and hexadecimal are not arbitrary, but were chosen such that each character is directly equivalent to three or four binary bits, respectively[11]. Let's see how each of these works by example, first taking the number three thousand seven hundred forty nine and expressing in 12-bit binary form:

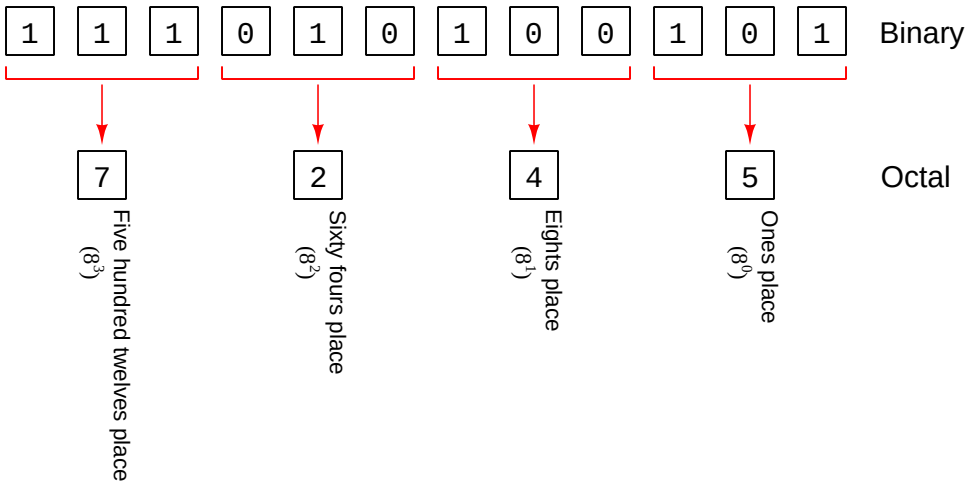


Our goal will be to take this binary representation of 3749_{10} and express it in both octal and hexadecimal notations.

[10] The term "byte" is actually a formal label for a group of eight bits. Not wanting anyone to think computer scientists were lacking in humor, a lexicon of terms grew from byte including *nybble* (4 bits), *playte* (16 bits), and even *dynner* (32 bits).

[11] Note that each octal character is equivalent to three bits because $2^3 = 8$, and each hexadecimal character is equivalent to four bits because $2^4 = 16$.

If we separate these bits into groups of three (starting from the least-significant bit, or LSB) and tally the place-weights of the bits within each group (i.e. treating each three-bit cluster as its own binary number), we get the following result:



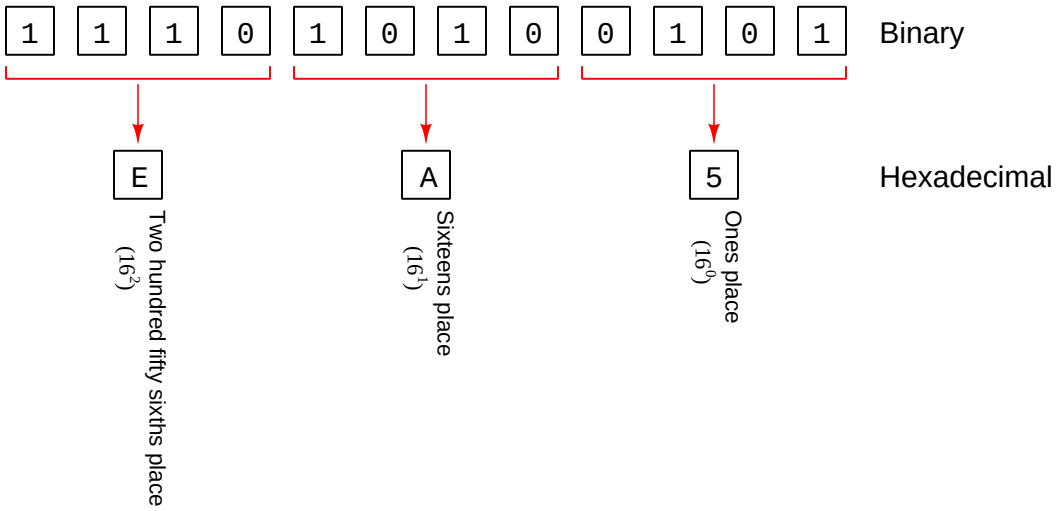
$$111\ 010\ 100\ 101_2 = 7\ 2\ 4\ 5_8$$

Here is a listing of equivalent values between three-bit binary clusters and octal characters to aid in the conversion from binary to octal:

$000_2 = 0_8$
$001_2 = 1_8$
$010_2 = 2_8$
$011_2 = 3_8$
$100_2 = 4_8$
$101_2 = 5_8$
$110_2 = 6_8$
$111_2 = 7_8$

Note the convention of appending the subscript "8" to the character string 7245 so that any reader will know this is an octal number rather than a decimal number.

Conversion from binary to hexadecimal is not much different. Instead of grouping bits by three we will group them by four, and then tally place-weights for the bits within each of those groups to find the equivalent hexadecimal character for each. Starting with the same binary number as before (111010100101), beginning our four-bit groupings at the LSB as before, we apply this technique and arrive at the following result:



$$1110\ 1010\ 0101_2 = \mathtt{E\ A\ 5}_{16}$$

Here is a listing of equivalent values between four-bit binary clusters and hexadecimal characters to aid in the conversion from binary to hexadecimal:

$0000_2 = 0_{16}$ $\quad$ $1000_2 = 8_{16}$
$0001_2 = 1_{16}$ $\quad$ $1001_2 = 9_{16}$
$0010_2 = 2_{16}$ $\quad$ $1010_2 = \mathtt{A}_{16}$
$0011_2 = 3_{16}$ $\quad$ $1011_2 = \mathtt{B}_{16}$
$0100_2 = 4_{16}$ $\quad$ $1100_2 = \mathtt{C}_{16}$
$0101_2 = 5_{16}$ $\quad$ $1101_2 = \mathtt{D}_{16}$
$0110_2 = 6_{16}$ $\quad$ $1110_2 = \mathtt{E}_{16}$
$0111_2 = 7_{16}$ $\quad$ $1111_2 = \mathtt{F}_{16}$

Note once again the convention of using a subscript to denote the radix, or "base", of the numeration system. Although the presence of *lettered characters* in the number $\mathtt{EA5}_{16}$ should be enough to indicate it is neither decimal or octal, the "16" subscript shows its merit in cases where the hexadecimal number in question happens to lack letter-characters.

Subscripts work well for this purpose, but in documents such as computer program source code where alternative fonts, subscripts, and other typesetting features are notably absent we must find other means of denoting when a number is hexadecimal. Shown here are some common notations using `EA5` as the example number, the last form used in this series of learning modules:

- A trailing[12] "H" character (e.g. `EA5H`)
- A leading dollar-sign ($) character (e.g. `$EA5`)
- A leading pound-sign (#) character (e.g. `#EA5`)
- Leading "0h" characters (e.g. `0hEA5`)
- Leading[13] "0x" characters (e.g. `0xEA5`)

It should be clear to see how either 7245_8 or $EA5_{16}$ are easier for a person to manage than the binary form 111010100101_2, given their dramatically smaller character counts. This is the sole purpose of octal and hexadecimal: to make it easier for human beings to manage large binary words by "compressing" groups of bits into single characters easier to read, write, and remember. Octal representation "compresses" a binary word by a ratio of 3:1 (i.e. three binary bits are represented in each octal character), while hexadecimal "compresses" at a 4:1 ratio.

[12] This same convention of appending or prepending a letter to the end of a non-decimal number is sometimes used for binary numbers as well as hexadecimal. For example 10110_2 could be written as `10110B`, or as `0b10110` instead. However, this introduces another problem: what if someone sees the letter "B" and thinks this is a *hexadecimal* number because "B" is a legitimate hexadecimal cipher?

[13] This same convention of prepending characters to the front end of a non-decimal number is sometimes used for octal numbers as well as hexadecimal. For example, in the C and C++ programming languages, prepending a zero (0) or 0o to a constant typically denotes it as being octal, while prepending 0x to a constant value denotes it as hexadecimal.

3.8 Decimal conversions

In the course of working with binary numbers, one must frequently convert between our conventional *decimal* numeration and one of the other numeration systems (binary, BCD, octal, hexadecimal). We saw examples of this in Unsigned integers section (3.3) and Signed integers section (3.4) of this tutorial, where we assigned place-weight values to each of the bits within a binary integer and tallied their sum to arrive at the equivalent decimal value. We saw a very similar example in the Binary-Coded Decimal section (3.6) where we assigned place-weight values for each of the bits in every four-bit grouping, and again tallied the sum of the weighted places bearing "1" bits.

This technique is not limited to finding the decimal equivalent of binary or BCD numbers, but in fact is applicable to *any* non-decimal place-weighted numeration system: simply label the place-weight values for each digit or character, compute the product of each numeral and its respective weight, then sum all of those values. To illustrate, we will convert three different numbers into decimal format using this technique.

3.8.1 Binary to decimal

First, let us consider the unsigned binary number `101100`. The six bits of this number all have place-weights, from 2^5 at the most-significant-bit (MSB weight = thirty two) to 2^0 at the least-significant bit (LSB weight = one). To convert to decimal, we simply find the product of each bit and its corresponding weight, then sum all those products:

Bit 5 = (1)(2^5) = 32 (MSB)
Bit 4 = (0)(2^4) = 0
Bit 3 = (1)(2^3) = 8
Bit 2 = (1)(2^2) = 4
Bit 1 = (0)(2^1) = 0
Bit 0 = (0)(2^0) = 0 (LSB)

This sum of all the multiplied place-weights is $32+0+8+4+0+0$ which is equal to 44_{10} for this unsigned binary number. If we happened to know this number was *signed* binary, our conversion procedure would only differ in making the MSB's weight negative rather than positive, in which case the value would be -20.

3.8.2 Octal to decimal

Next, consider the octal number `4207`. Each of the four octal characters' places have weight values ordered as powers of eight. Arranging all the characters in order with their respective place-weights as we did with binary, performing the multiplication, and then summing the total will result in the decimal equivalent value:

Octal character 3 = (4)(8^3) = 2048 (Most-significant character)
Octal character 2 = (2)(8^2) = 128
Octal character 1 = (0)(8^1) = 0
Octal character 0 = (7)(8^0) = 7 (Least-significant character)

The final tally is 2048 + 128 + 0 + 7 = 2183_{10}, and so we can say that $4207_8 = 2183_{10}$.

3.8.3 Hexadecimal to decimal

Finally, consider the hexadecimal number `3D9`. Each of the three hexadecimal characters' places have weight values ordered as powers of sixteen. Arranging all the characters in order with their respective place-weights as we did with binary and octal, performing the multiplication, and then summing the total will result in the decimal equivalent value:

Hexadecimal character 2 = (3)(16^2) = 768 (Most-significant character)
Hexadecimal character 1 = (D = 13)(16^1) = 208
Hexadecimal character 0 = (9)(16^0) = 9 (Least-significant character)

The final tally is 768 + 208 + 9 = 985_{10}, and so we know $\texttt{3D9}_{16} = 985_{10}$.

As you can see, converting from any non-decimal, place-weighted numeration to decimal is a fairly clear procedure based on what it means for any numeration system to be place-weighted. Each cipher in such a numeration system occupies a place, and each place has its own unique weight value based on the place number and the "base" of that numeration system.

3.8.4 Decimal to binary by cut-and-try

Converting *from* decimal into some non-decimal, place-weighted format requires a different approach. One way to do it is by *cut-and-try*, where we repeatedly plug different cipher values into the places of the target numeration system until we obtain the desired sum. This is easiest to demonstrate with binary, and so for our example we will convert the decimal number 93 into binary.

Our first task is to determine how many binary bits we will need to represent 93_{10}. This may be done by examining place-weight values for a series of binary bits, as we have done here with eight bits:

Bit 7 place-weight = $2^7 = 128$
Bit 6 place-weight = $2^6 = 64$
Bit 5 place-weight = $2^5 = 32$
Bit 4 place-weight = $2^4 = 16$
Bit 3 place-weight = $2^3 = 8$
Bit 2 place-weight = $2^2 = 4$
Bit 1 place-weight = $2^1 = 2$
Bit 0 place-weight = $2^0 = 1$ (LSB)

Clearly, we will not require the bit 7 or beyond to represent 93_{10}, because the place-weight value for that bit is already larger than our desired value. Therefore, we begin by setting the bit value of bit 6 to a "1" and proceed down toward the LSB setting bit values at either "1" or "0" as needed to meet our decimal goal. After setting the 2^6 bit to `1` we subtract that value from our target to see how much is left ($93 - 64 = 29$). If the next-lower bit's place-weight is more than that we make it `0`, if not we make it `1`. Here we see that the next bit (2^5) is larger than our remainder, so we set that bit to `0` and move on to the 2^4 bit, which we set to `1`. At this point our remainder is $29 - 16 = 13$ and so all we need now is thirteen from the remaining bits to make our goal. This will require setting the 2^3, 2^2, and 2^0 bits all to `1` states.

Reviewing our work:

Bit 6 = (1)(2^6) = 64 (MSB)
Bit 5 = (0)(2^5) = 0
Bit 4 = (1)(2^4) = 16
Bit 3 = (1)(2^3) = 8
Bit 2 = (1)(2^2) = 4
Bit 1 = (0)(2^1) = 0
Bit 0 = (1)(2^0) = 1 (LSB)

Arranging all the bits in order to form a binary number, we have $\texttt{1011101}_2$, which is the binary (unsigned) integer equivalent of 93_{10}. If we were asked to convert a positive decimal integer into a *signed* binary integer (i.e. two's complement notation), we would follow all the same steps, but include an additional bit (set to 0) as the new MSB which holds an (unused) negative place-weight. To use the decimal example of 93 once again, $93_{10} = \texttt{01011101}_2$ (signed).

Converting a *negative* decimal integer into signed binary format is not much different, except now to determine the requisite number of binary bits to use in our word we must find the smallest negative place-weight that is *not more negative* than our target number. For example, if our given decimal value was −50, we would need to use a seven-bit signed binary number because −32 is not negative enough but −64 will suffice (with some positive-weighted bits set to 1, of course, to bring the total to negative fifty).

More often than not, though, the binary word size will already be prescribed for us, since most digital systems are fixed at bit widths some multiple of eight (e.g. 8-bit, 16-bit, 32-bit, 64-bit). In this case we set the MSB at 1 to make sure the result will be negative, then set as many positive-weighted bits at 1 as necessary to reach our desired value.

3.8.5 Decimal to octal or hexadecimal by cut-and-try

We may apply this same cut-and-try method to the conversion of decimal into any place-weighted numeration system including octal and hexadecimal, but this requires substantially more work. Conversion into binary by cut-and-try is relatively easy because the decision to mark each place with either a "1" or a "0" as we consider each bit from MSB to LSB is a simple yes-or-no decision based on whether the sum would be less than or greater than or equal to the target value. When converting to octal or hexadecimal, we must not only consider the value of each place-weight, but also the effect of marking each place with one out of eight (or sixteen!) different ciphers. This typically requires multiple "trials" per place, making the process laborious for all but small quantities.

A hybrid alternative is to use the cut-and-try method to convert the given decimal quantity into binary (as previously shown), then group those binary bits into clusters of three or four, then convert each bit-cluster into one octal or hexadecimal character, respectively. Using the previous example decimal value of 93_{10} first converted into an unsigned binary number (1011101_2), we may show how to group bits into threes and convert to octal as a multi-step process:

Step	Description	Numerical value
1	The given decimal value	93_{10}
2	After conversion to binary using cut-and-try	1011101_2
3	Grouping bits into threes starting from the LSB	1 011 101
4	Translating each group into an octal cipher	135_8

Shown in this next table is the value 93_{10} similarly converted into hexadecimal form:

Step	Description	Numerical value
1	The given decimal value	93_{10}
2	After conversion to binary using cut-and-try	1011101_2
3	Grouping bits into fours starting from the LSB	101 1101
4	Translating each group into a hexadecimal cipher	$5D_{16}$

As with any lengthy mathematical procedure, it is always a good idea to *check your work* at the end. A good work-checking strategy for this application is to take the octal or decimal quantity and convert back into decimal[14] to prove it matches the original value:

$$135_8 = (1)(64) + (3)(8) + (5)(1) = 93_{10}$$

$$5D_{16} = (5)(16) + (13)(1) = 93_{10}$$

[14] This is easily done by summing the products of ciphers and place-weights as shown previously.

3.8.6 Conversion from decimal by repeated division

A more efficient method for converting a decimal quantity into any non-decimal, place-weighted numeration system involves repeated integer division by the radix (i.e. the base-value). The phrase *integer division* means dividing one integer value by another, expressing any "remainder" quantity as another integer.

Anyone familiar with the arithmetic procedure of *long division* knows what a "remainder" is. For example, if we divide 5 by 3 using an electronic calculator, the decimal result is displayed as 1.666666667. If we express this quotient more accurately as a *mixed number* (i.e. an integer plus a fraction), we obtain $1\frac{2}{3}$. The value "2" which appears as the numerator in the fractional portion of the mixed number is the *remainder*. When I learned long division many years ago, prior to my introduction to fractions, I was taught to express this quotient as 1 R 2 (i.e. one with a remainder of 2). When we divide integers to convert from decimal to some non-decimal numeration system, we will pay close attention to these remainders. If the concept of a remainder is too abstract, know that you can calculate the remainder of any fractional answer by multiplying the fractional portion of the answer by whatever integer you were dividing by. For this example, we would take the 0.666666667 (i.e. $\frac{2}{3}$) and multiply by 3 to compute a remainder of 2.

The general procedure of repeated division is as follows, iterated until the integer quantity is reduced to zero (i.e. there is nothing left to divide):

1. Divide the integer quantity by the radix
2. Note the remainder left over after the division – each remainder constitutes one character in the converted number
3. Discard the remainder and return to step one using just the integer portion

Consider an example where we intend to convert the decimal quantity 105_{10} into binary. Since binary has a radix (base) value of two, we must use repeated division by *two*. What follows is the complete procedure for this particular conversion:

$$105 \div 2 = 52.5 = 52\frac{1}{2} = 52 \text{ R } \texttt{1}$$

$$52 \div 2 = 26.0 = 26\frac{0}{2} = 26 \text{ R } \texttt{0}$$

$$26 \div 2 = 13.0 = 13\frac{0}{2} = 13 \text{ R } \texttt{0}$$

$$13 \div 2 = 6.5 = 6\frac{1}{2} = 6 \text{ R } \texttt{1}$$

$$6 \div 2 = 3.0 = 3\frac{0}{2} = 3 \text{ R } \texttt{0}$$

$$3 \div 2 = 1.5 = 1\frac{1}{2} = 1 \text{ R } \texttt{1}$$

$$1 \div 2 = 0.5 = \frac{1}{2} = 0 \text{ R } \texttt{1}$$

Listing all the remainders together as a binary word constitutes our binary integer: $\texttt{1101001}_2$. Checking our work to make sure it is correct, we will sum all the "1" place-weights to ensure we obtain 105_{10}:

$$(1)(64) + (1)(32) + (0)(16) + (1)(8) + (0)(4) + (0)(2) + (1)(1) = 105_{10}$$

This procedure is generally trouble-free, but there is often a point of uncertainty among those first learning it: *do the remainder values begin at the LSB, or the MSB?* In this example, someone new to the procedure might be left wondering, *is the binary number `1001011` or is it `1101001`?* For multiple reasons[15], the first remainder calculated must be the least-significant cipher.

[15] At least two lines of reasoning conclusively address this question. The first is to consider what it *means* when we begin the procedure by dividing an integer by two for the first time. What does it mean if our first remainder is either one or zero? This, simply put, is a test for whether that integer is *odd* or *even*. You will note that all place-weighted integer numeration systems have only one place-weight that is odd, and it is always the least-significant – all other place-weights are even-numbered. This means the even-ness or odd-ness of any number is defined by its least-significant cipher. If our first step of division-by-two tells us whether the given number is odd or even, and we know it is the LSB of a binary number that defines its odd-ness or even-ness, then *the first remainder we calculate must be the LSB*. The second line of reasoning is based on the conclusion of the repeated-division procedure: what would happen if we continued dividing by two? The answer to this question is that we would obtain an endless string of 0 remainder values. If those additional zeroes were placed on the LSB-end of the number (i.e. on the right-hand end), it would affect the number's value and thereby make that value depend entirely on when we happened to stop calculating more zeroes. This makes no sense, as it would render the result arbitrary instead of definite. Therefore, by *reductio ad absurdum* we demonstrate that last remainder cannot be least-significant.

Repeated division-by-radix works just as well for numeration systems other than binary. Let's take another decimal number, say 589_{10}, and repeatedly divide by eight to convert into octal:

$$589 \div 8 = 73.625 = 73\frac{5}{8} = 73 \text{ R } \texttt{5}$$

$$73 \div 8 = 9.125 = 9\frac{1}{8} = 9 \text{ R } \texttt{1}$$

$$9 \div 8 = 1.125 = 1\frac{1}{8} = 1 \text{ R } \texttt{1}$$

$$1 \div 8 = 0.125 = 0\frac{1}{8} = 0 \text{ R } \texttt{1}$$

Listing all the remainders together constitutes our octal integer: $\texttt{1115}_8$. Checking our work to make sure it is correct:

$$(1)(512) + (1)(64) + (1)(8) + (5)(1) = 589_{10}$$

Of course, this also works for hexadecimal. Let's begin with an example decimal value of 491_{10} and begin dividing by sixteen:

$$491 \div 16 = 30.6875 = 30\frac{11}{16} = 30 \text{ R } \texttt{11}$$

$$30 \div 16 = 1.875 = 1\frac{7}{8} = 1\frac{14}{16} = 1 \text{ R } \texttt{14}$$

$$1 \div 16 = 0.0625 = 0\frac{1}{16} = 0 \text{ R } \texttt{1}$$

Assembling the remainders[16] into a hexadecimal integer results in an answer of $\texttt{1EB}_{16}$. Checking our work to make sure it is correct:

$$(1)(256) + (14)(16) + (11)(1) = 491_{10}$$

[16] Recall that hexadecimal uses sixteen unique ciphers, `0` through `9` plus `A` through `F`. This means ten is represented by `A`, eleven by `B`, twelve by `C`, thirteen by `D`, fourteen by `E`, and fifteen by `F`.

As your familiarity with the repeated-division technique grows, you will likely notice two efficiencies to your benefit:

1. The larger the radix, the faster the conversion (i.e. fewer steps necessary)
2. The last step can actually be skipped if you pay close attention to the quotients

The first efficiency is of benefit to you if your goal is to convert a large decimal number into binary. Rather than divide by two over and over again (once for every bit of the binary number), you can save time and effort by converting into hexadecimal (which, as you recall, is merely a shorthand notation for binary), then quickly translating each hexadecimal character into four binary bits.

The second efficiency is realized in any case by noting whether the whole-numbered portion of the quotient is less than the radix. If so, that will be the remainder of the next (last) step! For example, note the second-to-last step of the decimal-to-hexadecimal conversion example: there we divided 30 by 16 to obtain 1 R 14. Since the whole-numbered portion of this quotient (1) is less than 16, we know it is going to be the remainder of the last quotient ($1 \div 16 = 0$ R 1) and therefore we can skip this last act of division.

3.9 Floating-point

All binary number representation is limited in range, but integer representation is especially limited. For example, a 16-bit unsigned integer has a range from 0 to 65535 (decimal), while a signed 16-bit integer ranges from −32768 to +32767. Increasing the number of bits will of course extend the range, but even with 32 bits an unsigned integer is limited to a range of 0 to 4,294,967,296, and a signed integer limited to a range of $-2,147,483,648$ to $+2,147,483,647$. If we require fractional capability this means assuming a binary point somewhere within the binary word, or similarly fixing a decimal point in the displayed value, with a corresponding decrease in range.

This fundamental problem has an analogue in the world of pencil-and-paper: how to write very large as well as very small numbers using a minimum of printed characters. A solution to this problem commonly used in scientific endeavors is *scientific notation*, where some fractional quantity between 1 and 10 is immediately followed by a power-of-ten multiplier which effectively sets the placement of the decimal point. For example, the decimal number 3600 would be written in scientific notation as 3.6×10^3. If we needed to express a number such as 3,600,000 or 0.000000000036 it would be as easy as writing 3.6×10^6 or 3.6×10^{-11}, respectively. Instead of a *fixed* decimal point, the power-of-ten gives us a *floating* decimal point whose position is defined by the exponent. No longer need we waste paper or pencil by writing place-holding zero digits for very large or very small numbers.

This same concept has been applied to binary numbers, and it is called *floating-point* notation. A floating-point number is defined by a digital word of some length (typically 32 bits) with groups of those bits serving different purposes. One of them is used as a *sign* bit, a small group represents the exponent, and the remaining bits represent the *significand* (the "significant" bits of the number, analogous to the 3.6 in the previous scientific notation example).

ANSI/IEEE standard 754 specifies a form of binary floating-point notation that is nearly universal in digital systems today, with multiple word-lengths for differing degrees of precision and range capabilities. The basic format follows this mathematical pattern, with m and E representing binary numbers comprised of different groups of bits in the floating-point word:

$$\pm 1.m \times 2^E$$

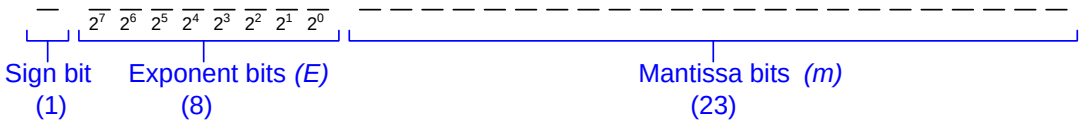
Alert readers will note that the format shown here ($\pm 1.m \times 2^E$) provides no way to represent the number zero, since 1.0×2^0 ($m = 0$ and $E = 0$) is actually equal to one! Here is our first case where floating-point notation must provide *special representation* of quantities. In the IEEE 754 standard, the significand's format shifts to $\pm 0.m \times 2^E$ if ever all exponent bits are 0 ($E = 0$). In order to still be able to represent 1 (1.0×2^0), the IEEE standard assumes the exponent value is *biased* with a negative number[17], so that an exponent bit field of 0 does not mean 2^0, but rather 2^{-bias}. This makes it possible to have an exponent value of 0 using non-zero E bits. Similarly, an exponent field consisting entirely of 1 bits is used as special representation for *infinity* or for an error code called *Not a Number* (NaN), depending on the values of the m bits. These special representations are important for handling the results of calculations such as division-by-zero and the square-roots or logarithms of negative numbers.

[17] Having an implied negative bias added to an integer number is an alternative scheme to two's complement for representing negative numbers in binary numeration, and is commonly referred to as *excess* in computer science literature. The IEEE 754 standard's exponent bias could be said as being *excess-127*.

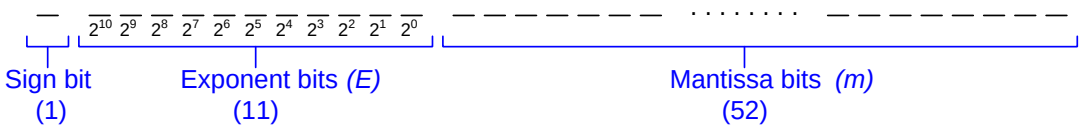
Given these special cases of representation required in floating-point notation, the task of doing calculations with floating-point numbers requires special processor circuitry designed to implement these rules. Inside a digital computer, this task is managed by a *floating-point processor* unit, usually a special section of the microprocessor. Some simpler microprocessors cannot support floating-point arithmetic, and thus some control system hardware (e.g. low-cost PLCs) must do all tasks using integer numbers (or fixed-point notation, if fractional quantities must be represented).

The ANSI/IEEE standard 754 specifies multiple floating-point number formats, including one that is 32 bits in length ("single-precision") and another that is 64 bits in length ("double-precision"). In the IEEE standard, one bit is reserved for the sign of the number (0 for positive and 1 for negative), a certain number of bits for the power-of-two exponent[18], and the rest of the bits for the *mantissa*[19] (the fractional portion of the normalized value). Both formats[20] are shown here:

Single-precision IEEE floating-point number format



Double-precision IEEE floating-point number format



A third floating-point IEEE standard called *extended* uses 80 total bits: 1 for the sign, 15 for the exponent, and 64 for the mantissa.

Floating-point number representation greatly simplifies the task of digitally calculating real-world values. Integer numbers, by contrast, are rather clumsy when used to represent most real-world measurements or statistics. For this reason, floating-point numbers are sometimes referred to as *real* values in digital computer systems.

[18] Note how the place-weights shown for the exponent field do not seem to allow for negative values. There is no negative place-weight in the most significant position as one might expect, to allow negative exponents to be represented. Instead the IEEE standard implies a *bias value* of −127 (i.e. the exponent integer is in "excess-127" signed format). For example, in a single-precision IEEE floating-point number, an exponent value of 11001101 represents a power of 78 (since 11001101 = 205, the exponent's actual value is 205 − 127 = 78).

[19] The term *mantissa* is somewhat ambiguous because it has alternative definitions in mathematics. Some references prefer to use the phrase *trailing significand* instead, but the term "mantissa" has been used for so long that it is still common to see.

[20] Both formats shown here, with the sign bit on the left and mantissa bits on the right, assume *big endian* formatting, which is another topic entirely!

The ANSI/IEEE 754 floating-point format can be rather confusing to learn, and so some illustrative examples are helpful. I will be using a simple computer program written in C++[21] to take any given floating-point number and display its constituent bits.

For the first example we will enter the floating-point decimal number -2.75 and see what the analysis of bits looks like:

```
Enter a floating-point value: -2.75

Sign    Exponent                Mantissa
1       1000 0000      0110 0000 0000 0000 0000 000
```

Since -2.75_{10} is a negative number, the sign bit is set to `1`. The 8-bit exponent value of `1000 0000` is equal to 128_{10}, and added to the ANSI/IEEE standard's bias value of -127_{10} for single-precision floating-point numbers gives a power-of-two value of 1. The mantissa bits will be appended to a leading `1` bit followed by a "binary point", so that the whole number in binary format will read as follows:

$$\texttt{1.0110 0000 0000 0000 0000 0000} \times 2^1$$

This, of course, is equivalent to $\texttt{10.11}_2$ in fixed-point binary notation because the 2^1 factor shifts the "binary" point one place right, just as 10^1 shifts the decimal point one place right in scientific notation. Remember that floating-point numeration is just scientific notation *in binary*. The binary number `10.11` is then converted into decimal form by summing all the place-weights bearing "1" bit values. Place-weights to the left of the binary point ascend by multiples of two (as we have seen before), while place-weights to the right of the binary point descend by factors of two:

$$(1)(2) + (0)(1) + (1)\left(\frac{1}{2}\right) + (1)\left(\frac{1}{4}\right) = 2.75$$

[21] To see the source code for this program, refer to section 5.3 beginning on page 79. The program written to "dissect" a floating-point number is one of several programs shown and demonstrated in this section of the module.

For the second example we will analyze the floating-point decimal number 3×10^{-8} which happens to be the approximate value of the speed of light in a vacuum expressed in meters per second:

```
Enter a floating-point value: 3e8

Sign    Exponent                  Mantissa
0       1001 1011      0001 1110 0001 1010 0011 000
```

The sign bit is zero, which indicates this is a positive quantity. The exponent value of `1001 1011` is equal to 155_{10}, which yields a power-of-two value of twenty eight after the -127 bias value has been added to it. Prepending a `1` to the mantissa bits and including the power-of-two factor, we arrive at the following:

$$\texttt{1.0001 1110 0001 1010 0011 000} \times 2^{28}$$

Our multiplication factor of 2^{28} shifts the binary point twenty eight places to the right, moving it to the far right and requiring five additional "0" bits for *padding*. The result is the following binary integer:

$$\texttt{1 0001 1110 0001 1010 0011 0000 0000}$$

Converting this binary integer into decimal form by summing all the place-weights occupied by "1" bits reveals what we expected, a value of 300 million:

$$268,435,456 + 16,777,216 + 8,388,608 + 4,194,304 + 2,097,152 +$$

$$65,536 + 32,768 + 8,192 + 512 + 256 = 300,000,000$$

As useful as floating-point numbers are, they are not without liabilities. One of these is the computational expense (i.e. the number of steps necessary for a computing circuit to complete the calculation) of floating-point addition and subtraction. Multiplication and division of floating-point values is relatively simple, for the same reason[22] that multiplication and division of decimal quantities in scientific notation is relatively simple; however, addition and subtraction is not nearly as straightforward a process which means an inordinate amount of time may be required for a computer to add or subtract two floating-point values. For this reason it is common to find smaller digital computers (especially so-called *embedded* computers used in a wide range of consumer, commercial, and industrial applications, and even some industrial PLCs[23] at the time of this writing) entirely lacking floating-point capability, which means these devices must perform all their arithmetic functions using only integer quantities[24].

[22] Multiplication of scientific-notation numbers consists of multiplying the significands and summing the exponents. For example 3.2×10^5 multiplied by 1.8×10^{11} is equal to 5.76×10^{16} because 3.2 times 1.8 is 5.76, and 5 plus 11 is 16. Division is similar: divide the significands and subtract the exponents.

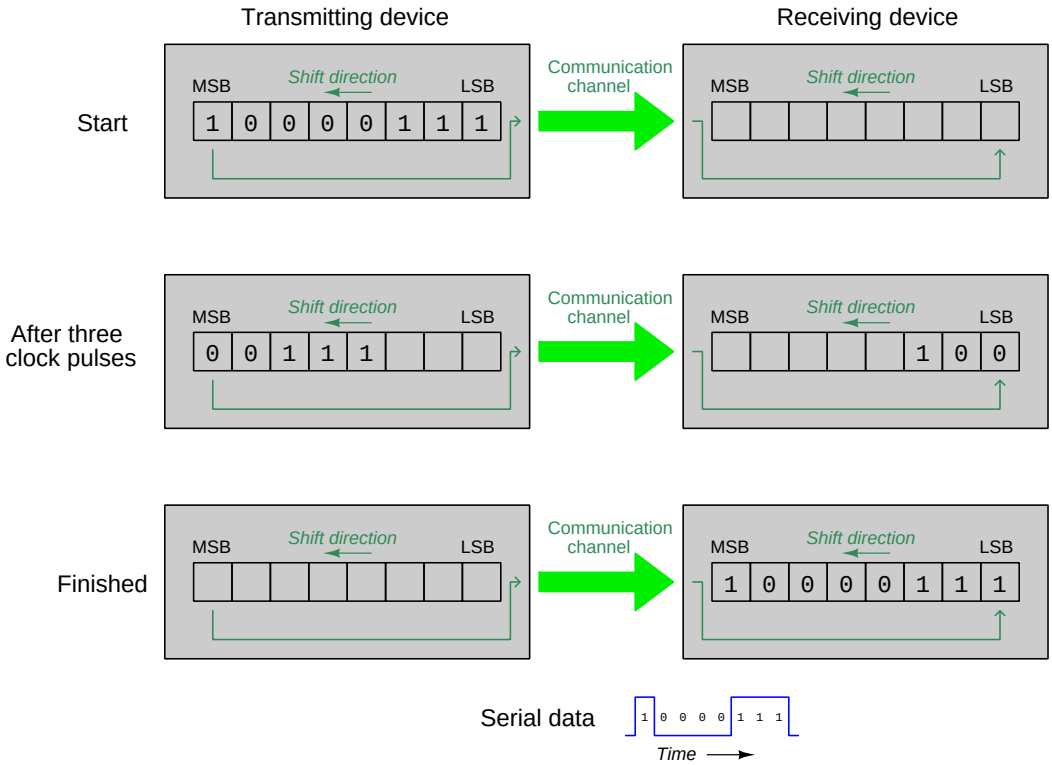
[23] The Rockwell (Allen-Bradley) MicroLogix 1000 programmable logic controller (Bulletin 1761), for example, is only capable of integer arithmetic, and cannot even represent a quantity using floating-point notation.

[24] Early personal computers suffered this disadvantage as well. A common upgrade for these computers was to install a *math co-processor* IC which served the purpose of performing floating-point calculations using definite-purpose logic circuitry rather than have the main processor IC emulate floating-point arithmetic using integer values.

3.10 Big endian and little endian formats

A very popular form of digital data communication is called *serial*, where the individual bits of a multi-bit word travel along some communications channel (e.g. copper cable, optical fiber, radio waves) one bit at a time. An important design decision must be made for both the transmitting and receiving devices in order for this to work, and that is *which bit will be sent first?*.

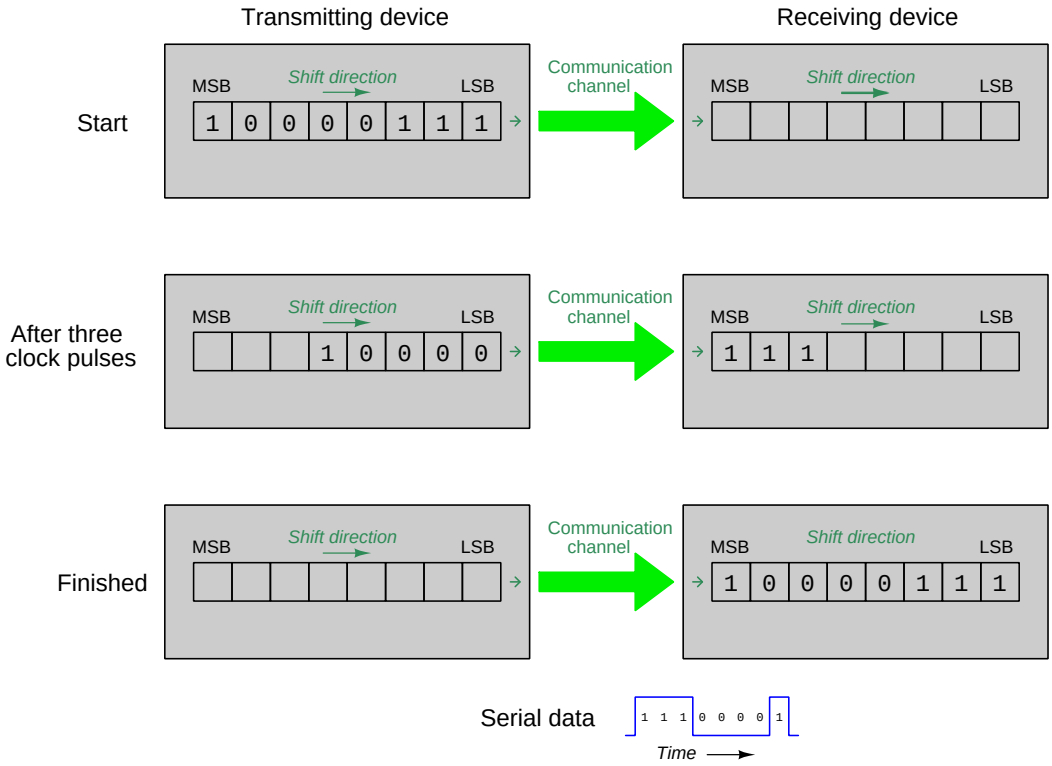
Imagine two digital devices, each one with a *register*[25] storing an eight-bit word. Both registers are designed to be able to *shift* their bit-contents from one place to the next following the timing of a "clock" pulse signal, and the way serial communication occurs is to tie the one communications line to one of the far-end bits of each register, so that when these registers are both shifted, the data gets "pumped out" of one and "pumped in" to the other. The following illustration shows one way of doing this, with the transmitting register's most-significant bit (MSB) linked to the communication line, which in turn connects with the least-significant bit (LSB) of the receiving register:



As this illustration shows, the eight-bit digital word held in the transmitting device gets shifted out of its register beginning with the MSB (the "big end") while the receiving register fills up from its LSB end. This is called a *big-endian* sequence, where the MSB (the "big end") of the word gets sent first.

[25] A "register" is simply a collection of electronic circuits having the ability to store a digital word as individual "1" and "0" states.

This next illustration shows the same eight-bit data communicated in reverse order, or *little-endian*:



As you can see from this illustration, the eight-bit word gets shifted out of the transmitting register starting with the LSB (the "little end") while the receiving register fills up from its MSB end. This is called *little-endian* sequence, where the LSB (the "little end") of the word gets sent first.

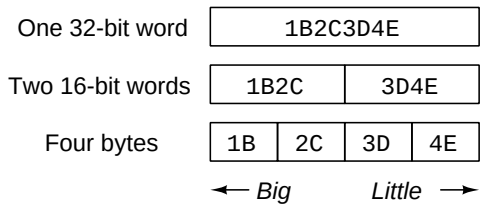
There is no functional advantage of one scheme over the other – both big-endian and little-endian sequences get the job done. However, it should be clear that both transmitting and receiving devices must agree on one of these formats, or else the digital word is going to be have all of its bits rearranged in reverse order at the conclusion of the serial communication! Big-endian versus little-endian serial sequence is just one of many parameters which must be agreed upon when the transmitting and receiving devices are of different manufacture, which is why *common standards* are important to codify and also to heed.

Although the question of big-endian versus little-endian arises from the challenge of serial data communication, it does not end there. Even if two digital systems perfectly agree on the communication order of bits, they may still disagree on the meanings of those bits once they arrive.

It is important to note that most digital systems manage data in *words*, which are quantities comprised of multiple bits. Digital computers have a standard word-length dependent on the hardware used in the processor (or *micro*processor) unit. For example, legacy microprocessor ICs such as the Motorola 6502 operated using eight-bit words. Later microprocessors such as the Motorola 68000 used sixteen-bit words. Even later processors such as the Intel Pentium used 32-bit words, and at the time of this writing the state-of-the-art for personal computer microprocessors is a word length of 64 bits. At first one might be tempted to think that a computer using small words would be incapable of handling large numbers, but that is not the case. For example, an 8-bit (i.e. one-*byte*) digital computer may still be capable of performing certain operations on 16- or 32-bit binary quantities, but this is done by regarding multiple bytes as comprising larger words: using two consecutive bytes to represent a 16-bit word, or four consecutive bytes to represent a 32-bit word.

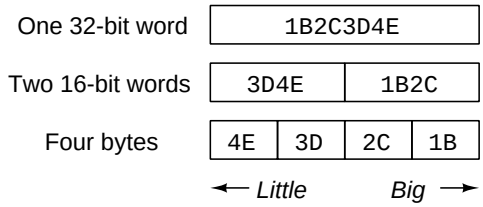
Here is where *endianness* rears its ugly head again: the way in which a digital system subdivides large words into smaller words, bytes, and bits is arbitrary. In other words, no natural law makes one type of ordering more sensible than another. In the absence of some formal standard declaring a "conventional" order, the inevitable result is that different engineers designing digital systems for different manufacturers wind up choosing different ordering schemes.

Consider the example of a 32-bit digital word represented by the following hexadecimal value: $\texttt{1B2C3D4E}_{16}$. Since we know each hexadecimal character is "shorthand" for four binary bits, every two hexadecimal characters must represent one byte, and every four characters a 16-bit word. As readers of English, we might assume the breakdown would maintain consistent left-to-right reading order, so that any string of smaller words would read in the same order as the original 32-bit word (i.e. `1B2C3D4E` left-to-right):



However, this is only true in a *big-endian* system where the first sub-word in a set begins with the highest-weight (i.e. the biggest) portion of the original word. In a "big endian" structure, the most significant bit/byte/word (i.e. the "bigger end") always comes before the others.

If a digital system is *little-endian*, the structure is reversed so that the "little end" always comes before the others. In such a case, our original 32-bit word gets its portions "swapped" about as shown in this next illustration:



Alternative descriptions for these scenarios include the terms *byte-swapped* and *word-swapped*[26]. In the "big-endian" scheme previously shown, there is no swapping at all because everything reads in the same order no matter how it is subdivided. In the "little-endian" scheme previously shown, it is both word-swapped and byte-swapped because with each level of subdivision the front and back halves get reversed.

If we consider (16-bit) word-swapping and (8-bit) byte-swapping to be separate configurations, we find exactly four combinations are possible. These are shown in the following table, using the 32-bit data from the previous example ($\texttt{1B2C3D4E}_{16}$) along with written descriptions of swapping and a *byte order* written symbolically using the four letters A, B, C, and D:

Data as four bytes	Swapping	Byte order
1B 2C 3D 4E	No swapping (original 32-bit data word)	ABCD
3D 4E 1B 2C	Word-swapped, but no byte-swapping	CDAB
2C 1B 4E 3D	No word-swapping, but byte-swapped	BADC
4E 3D 2C 1B	Word-swapped and byte-swapped	DCBA

If all of this seems confusing and perhaps even pointless, you are in good company. Within the confines of any given digital system, whichever ordering scheme is chosen should be consistent across all portions of the system and therefore transparent to most (if not all) people using that system. In other words, if the system is well-designed you probably won't have to worry about any of this. However – and this is a *big* "however" – when different digital systems must exchange data across some form of communications network, problems are likely to arise if the byte-ordering of one system differs from the other. This problem is often compounded by a lack of comprehensive documentation on one or both systems in question, which means it might be necessary to experimentally determine the byte-ordering of data by trial-and-error. In such cases it is very good to know what byte-swapping is and the various forms it takes.

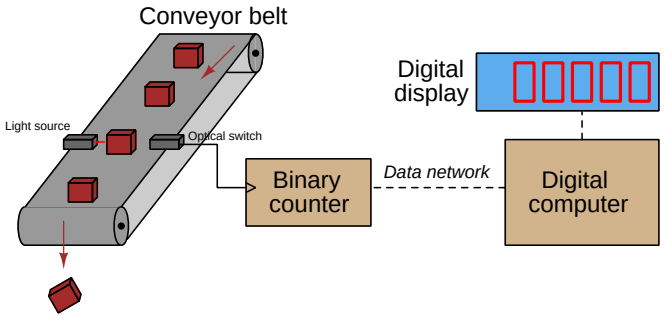
[26] The use of "word" here can be misleading, since technically the bit-width of a word is system-dependent. However, for a long time "word" was taken to mean *16 bits*, and in cases such "word-swapping" it is assumed to mean the swapping of two *16-bit* words.

3.11 Incompatible format errors

At a very fundamental level all digital systems are identical: information is represented by collections of discrete (on or off) states. This means it ought to be possible, at least on a theoretical basis, to translate any digital data to make sense within any digital system. As you might imagine, though, the fact that this is theoretically possible does not mean it will be easy[27].

One of the many obstacles complicating the inter-operation of digital systems is differing numeration formats. This section will not attempt to be comprehensive in describing every type of incompatibility, but rather will give a few examples to showcase the nature of the problem.

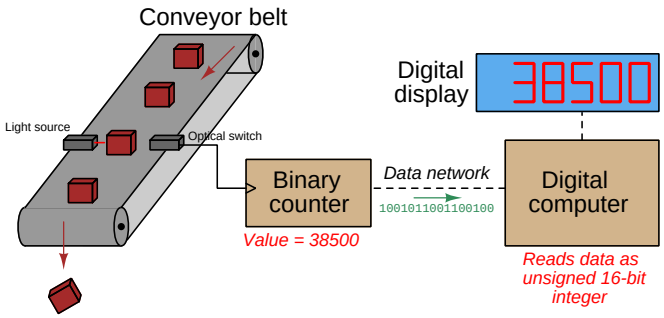
Let's begin with a numerical example to show how different numeration formats complicate the exchange of digital data. Suppose we have a sensor counting the number of items passing by on a conveyor belt. Perhaps these items are assembled machines, or perhaps containers of liquid, or something else – their identity is unimportant to this discussion. What matters to us is that this sensor generates an electrical pulse with the passing of each object, and that a digital counter unit tallies these pulses as a running total:



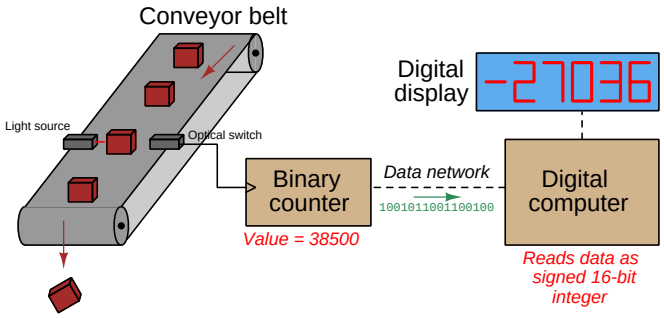
The digital counter simply increments by one with each pulse from the sensor, and so its value is best interpreted as an unsigned integer. If we assume a 16-bit counter capacity, and furthermore assume exactly thirty-eight thousand five hundred items have passed by on the conveyor belt, the counter should register the binary equivalent of $38{,}500_{10}$. Converting to binary numeration, this quantity appears as $\texttt{1001 0110 0110 0100}_2$.

[27] A phrase often used in computer science to describe this theoretical versus practical dichotomy is to say we have fallen into a *Turing tar pit*. Alan Turing was an English mathematician who greatly assisted the Allied forces during World War Two with his pioneering work using computers to decrypt German military messages which were encrypted by a sophisticated algorithm. He also described a universal computing machine (called a *Turing machine*) theoretically capable of executing any mathematical algorithm, which used a long tape upon which characters could be written and read, those characters directing the machine to increment and decrement the tape's position to other character places, continuing the process indefinitely. Turing's idea was not far removed from that of a digital processor, which reads data from and writes data to locations in a digital memory array, the data stored in that array giving commands to the processor what to do next. His concept for a computer, though, was so rudimentary as to require an inordinate amount of time and processing time to complete even the simplest of practical tasks. That is to say, even though a Turing machine is *theoretically* capable of performing any kind of mathematical operation, trying to program a Turing machine to *actually perform some practical operation* is akin to crawling out of a tar pit: possible in principle but nearly impossible in practice.

If the receiving computer accepts this 16-bit binary quantity as the unsigned integer it is, and all other aspects of the system are compatible, the digital display will faithfully register 38500 after the sensor counts 38,500 items as shown in the following illustration:



Let's suppose, though, that someone mistakenly programmed the computer to interpret this exact same 16-bit data as a *signed* 16-bit integer rather than the *unsigned* 16-bit integer it actually is. The data received via the communication link is nothing but a stream of sixteen sequential bits, and so the receiving computer cannot naturally "know" what these bits are supposed to represent. It is the programmer's responsibility to instruct the computer to properly interpret this data, and in this scenario the interpretation is improper. What would happen in this case? We can tell from the binary expression of this quantity that the leading bit is 1, which means in two's complement notation it will have a negative value (−27036 to be precise).

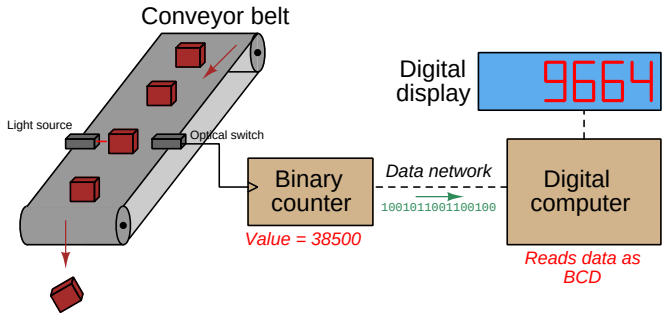


Not only will this improperly-configured system register an entirely non-sensical count value (i.e. a *negative* number of items!), but the problem won't appear for quite some time. Up to a count value of 32767 the MSB of this 16-bit binary number will be zero, and for that reason will be interpreted as a positive quantity whether the computer regards it as signed or unsigned. Only after the count value exceeds 32767_{10} will the discrepancy between unsigned and signed integer formats become apparent (i.e. when the display begins to register negative values).

This example not only demonstrates one way in which disparate number formats can cause problems, but it also proves why it is necessary in the design of complex systems to *fully test* all functions of that system before relying on them to perform as designed. Here we have an incompatibility that does not reveal itself until a large number of items have been detected by the sensor, which may equate to weeks or months or even years of operation in this conveyor belt

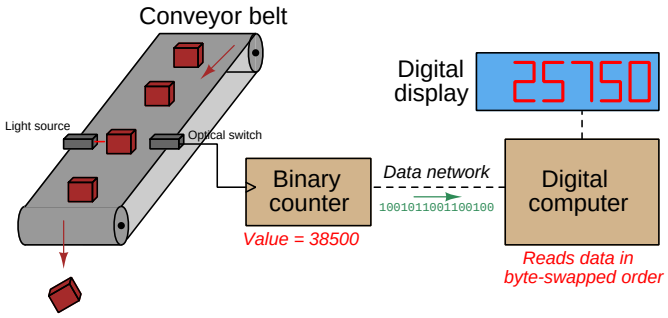
application. A full test of the system would have included at least a simulation of item count up to the maximum possible with an unsigned 16 bit binary number (65535_{10}) in order to verify no unusual problems lurked in the system.

Let us now assume a different programming flaw, one where the computer is incorrectly programmed to interpret the received 16-bit data as *BCD* rather than unsigned binary. In this case, the computer will translate each grouping of four bits in the 16-bit word as a separate decimal digit, and so the data $\texttt{1001 0110 0110 0100}_2$ will be interpreted as the decimal value 9664_{10} as the following illustration shows:



What is more, every time the binary count value happens to generate a grouping of four bits whose value exceeds nine, the computer will display some sort of nonsensical character for that digit, since BCD expects only the codes `0000` through `1001` (i.e. 0 through 9) for each four-bit group. What exactly the computer does with any non-conforming four-bit code (e.g. `1011`, `1100`, etc.) depends on details of its operating code which may not even be accessible to the engineers and technicians tasked with installing and configuring the computer for this conveyor belt system. The programming manual for this computer might not even document the response for non-compliant BCD data.

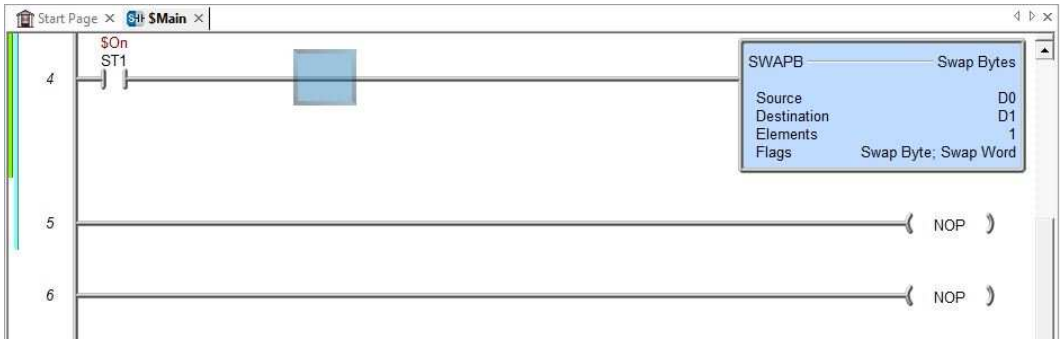
Yet another faulty scenario is one where the computer assumes a different byte order than what is transmitted by the counter. In this case a binary count value of $\texttt{1001 0110 0110 0100}_2$ would be interpreted as $\texttt{0110 0100 1001 0110}_2$, which is interpreted as 25750_{10}:



An interesting characteristic of this error is that it is not user-generated. In other words, it was not the fault of the person programming this computer to receive and display the data, but rather

an inherent incompatibility between the counter's 16-bit register and the computer's 16-bit registers resulting from disparate data-organizing philosophies on the part of each manufacturer. In order to correct this problem, the people installing and programming the computer for this conveyor belt application must include *additional* programming code to intentionally swap bytes in this 16-bit word so that it will read correctly.

This problem of byte-order and word-order is so common between manufacturers of industrial computing hardware that many programmable computers (e.g. PLCs) provide ready-to-use functions just for this purpose. A screenshot showing an image of such a data-swapping function in the programming editor of a Koyo brand "Do-More" PLC appears here:



This particular function provides user-selectable "flag" options to swap bytes, swap words, or both.

And, of course, it is possible for *multiple* data-format incompatibilities to simultaneously coexist: consider a case with byte-swapped data *on top of* a signed/unsigned mismatch. These problems can be frustrating to diagnose because they require just the right combination of fixes to remedy.

A helpful tool for diagnosing such incompatibilities is to view the system's data in hexadecimal form. Examining the "raw" data in hexadecimal form rather than formatted decimal values eliminates one step of translation, allowing you to more easily discern patterns indicating data format. Many industrial computing systems such as Programmable Logic Controllers (PLCs) provide means to view data in raw form with little or no display formatting. In text-programmed systems you may need to insert your own code to force the computer to display hexadecimal values if no "raw data view" feature exists. This approach is analogous to taking electrical measurements of signals within an electronic circuit rather than merely relying on the ordinary indicators of that circuit's function, giving yourself a view of the circuit's "raw" status.

Another way to attack a problem such as this is to force certain "test" values such as zero, one, or any other value providing an easily-predicted result. Forcing certain input values and examining their results provides a way to actively explore how the system interprets data. This approach is an example of the general problem-solving strategy of *simplifying the system*: inserting a numerical value into the digital system for which the results are simpler to predict than the random value that happened to be there when you first encountered the problem.

So far we have discussed errors resulting from the exchange of data between incompatible number formats. A different type of incompatibility error is seen with floating-point digital numbers: an *internal* incompatibility with specific numerical values whereby it is impossible to precisely represent a seemingly simple decimal value using binary floating-point. The following example shows a set of simple arithmetic calculations performed on a computer[28], each of them intended to generate the same result of 1.8:

```
>>> 1.3 + 0.5
1.8
>>> 1.2 + 0.6
1.7999999999999998
>>> 1.1 + 0.7
1.8
>>> 1.0 + 0.8
1.8
>>> 0.9 + 0.9
1.8
>>> 0.8 + 1.0
1.8
>>> 0.7 + 1.1
1.8
>>> 0.6 + 1.2
1.7999999999999998
>>> 0.5 + 1.3
1.8
```

As you can see, all the sums are correct except for those adding 1.2 and 0.6. Clearly, the problem is not an inability to represent 1.8 because most of these sums are exactly that. The problem must have something to do either with representing 1.2, or representing 0.6, or both. A second experiment helps us diagnose the problem a little further:

```
>>> 0.6 + 0.6
1.2
>>> 1.2 + 1.2
2.3999999999999999
```

[28]The software used to perform these computations is an *interpreter* for a text-based programming language called *Python*. Rather than type commands into a text file and then use *compiler* software to translate those commands into a format the computer can execute, an *interpreter* does the translation as soon as you type the commands. This immediacy makes interpreted programming languages especially easy for beginners to learn. Python in particular is a very well-designed language with relatively simple syntax, yet possesses many advanced functions, and is highly recommended for beginning programmers to learn.

Clearly, 0.6 is representable in floating-point format, but it has a problem representing the value 1.2. Continuing our experiments with the value 1.2 represented as a floating-point number:

```
>>> 1.2 * 1
1.2
>>> 1.2 * 2
2.3999999999999999
>>> 1.2 * 3
3.5999999999999996
>>> 1.2 * 4
4.7999999999999998
>>> 1.2 * 5
6.0
>>> 1.2 * 6
7.1999999999999993
>>> 1.2 * 7
8.4000000000000004
>>> 1.2 * 8
9.5999999999999996
>>> 1.2 * 9
10.799999999999999
>>> 1.2 * 10
12.0
```

Some additional experimentation with different numbers reveals other anomalous values:

```
>>> 0.5 - 0.6
-0.09999999999999998
>>> -0.4 - (-0.3)
-0.10000000000000003
```

It is important to note that none of these errors are the fault of poorly-written software. Rather, they reveal inherent limitations of the binary floating-point numbers themselves, and/or limitations of the algorithms responsible for floating-point arithmetic. Imprecise floating-point representations of numbers such as these may result in malfunctioning computer programs if the computer is programmed to check a floating-point number for some expected result. If the expected result happens to be one of these anomalous cases the program may fail to properly recognize it (e.g. the product of 1.2 and 8 ought to be 9.6, but if the result comes out as 9.5999999999999996 it won't be deemed "equal" to 9.6 and the computer will not recognize it for what it ought to be).

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 A binary resistance box

The Scottish physicist James Clerk Maxwell (1831-1879) wrote a book entitled *A Treatise on Electricity and Magnetism*, within which he describes the construction and use of a "resistance box" which could be used to simulate any number of different resistance values by shorting past certain resistive wire coils in particular patterns.

> There are various arrangements by which resistance coils may be easily introduced into a circuit.
>
> For instance, a series of coils of which the resistances are 1, 2, 4, 8, 16, &c, arranged according to the powers of 2, may be placed in a box in series.
>
> The electrodes consist of stout brass plates, so arranged on the outside of the box that by inserting a brass plug or wedge between two of them as a shunt, the resistance of the corresponding coil may be put out of the circuit. This arrangement was introduced by Siemens.
>
> Each interval between the electrodes is marked with the resistance of the corresponding coil, so that if we wish to make

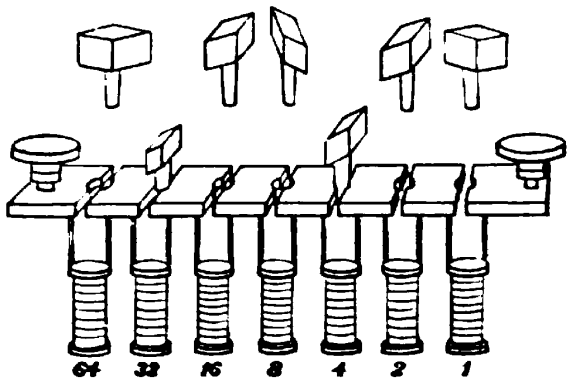


Fig. 29.

> the resistance in the box equal to 107 we express 107 in the binary scale as 64 + 32 + 8 + 2 + 1 or 1101011. We then take the plugs out of the holes corresponding to 64, 32, 8, 2 and 1, and leave the plugs in 16 and 4.
>
> This method, founded on the binary scale, is that in which the smallest number of separate coils is needed, and it is also that which can be most readily tested. For if we have another coil equal to 1 we can test the quality of 1 and 1', then that of 1 + 1 and 2, then that of 1 + 1' + 2 and 4, and so on.
>
> The only disadvantage of the arrangement is that it requires a familiarity with the binary scale of notation, which is not generally possessed by those accustomed to express every number in the decimal scale. [page 470]

Maxwell continues by giving an example of a different kind of resistance box also employing binary patterns. In this design, each resistance coil is labeled with a resistance value (in Ohms) as before, but with this design the insertion of brass plugs connects these resistances in parallel with each other to produce greater values of total *conductance* (measured in units of Siemens, or *Mhos*). Recall that conductance (G) is the reciprocal of resistance ($G = \frac{1}{R}$), so that a resistance of 2 Ohms would be a conductance of 0.5 Siemens, a resistance of 8 Ohms would be a conductance of 0.125 Siemens, etc. Just as resistances add together when connected in series, conductances add together when connected in parallel:

> A box of resistance coils may be arranged in a different way to the purpose of measuring conductivities instead of resistances. [page 470]
>
> The coils are placed so that one end of each is connected with a long thick piece of metal which forms one electrode of the box, and the other end is connected with a stout piece of brass plate as in the former case.
>
> The other electrode of the box is a long brass plate, such that by inserting brass plugs between it and the electrodes of the coils it may be connected to the first electrode through any given set of coils. The conductivity of the box is then the sum of the conductivities of the coils.

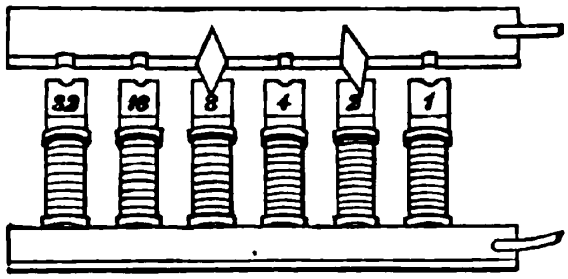


Fig. 30.

> In the figure, in which the resistances of the coils are 1, 2, 4, &c, and the plugs are inserted at 2 and 8, the conductivity of the box is $\frac{1}{2} + \frac{1}{8} = \frac{5}{8}$, and the resistance of the box is therefore $\frac{8}{5}$ or 1.6.
>
> This method of combining resistance coils for the measurement of fractional resistances was introduced by Sir W. Thomson under the name of the method of multiple arcs. See Art. 276. [page 471]

4.2 Big-endians and Little-endians

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

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

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

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

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

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

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

At the time of this writing (2018), nearly four decades after Cohen's missive, the state of anarchy described by Cohen remains alive and well, with little-endian and big-endian formats commonly found coexisting across digital data networks. While the problem of which bit to send first in a serial (i.e. one bit at a time) communication channel seems to be settled[1] within each of the various network standards (e.g. Ethernet, EIA/TIA-232, etc.), the problem of byte and word order for large data segments remains. It is not uncommon, for example, to find manufacturers of industrial data equipment arbitrarily using different 16-bit word orders for storing 32-bit binary numbers, so that when a 32-bit binary number is received by a digital device of different manufacture, swapping of word or byte orders may be necessary in order to preserve the meaning of that 32-bit number.

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

Chapter 5

Programming References

A powerful tool for mathematical modeling is text-based *computer programming*. This is where you type coded commands in text form which the computer is able to interpret. Many different text-based languages exist for this purpose, but we will focus here on just two of them, *C++* and *Python*.

5.1 Programming in C++

One of the more popular text-based computer programming languages is called *C++*. This is a *compiled* language, which means you must create a plain-text file containing C++ code using a program called a *text editor*, then execute a software application called a *compiler* to translate your "source code" into instructions directly understandable to the computer. Here is an example of "source code" for a very simple C++ program intended to perform some basic arithmetic operations and print the results to the computer's console:

```
#include <iostream>
using namespace std;

int main (void)
{
  float x, y;

  x = 200;
  y = -560.5;

  cout << "This simple program performs basic arithmetic on" << endl;
  cout << "the two numbers " << x << " and " << y << " and then" << endl;
  cout << "displays the results on the computer's console." << endl;

  cout << endl;

  cout << "Sum = " << x + y << endl;
  cout << "Difference = " << x - y << endl;
  cout << "Product = " << x * y << endl;
  cout << "Quotient of " << x / y << endl;

  return 0;
}
```

Computer languages such as C++ are designed to make sense when read by human programmers. The general order of execution is left-to-right, top-to-bottom just the same as reading any text document written in English. Blank lines, indentation, and other "whitespace" is largely irrelevant in C++ code, and is included only to make the code more pleasing[1] to view.

[1] Although not included in this example, *comments* preceded by double-forward slash characters (`//`) may be added to source code as well to provide explanations of what the code is supposed to do, for the benefit of anyone reading it. The compiler application will ignore all comments.

Let's examine the C++ source code to explain what it means:

- `#include <iostream>` and `using namespace std;` are set-up instructions to the compiler giving it some context in which to interpret your code. The code specific to your task is located between the brace symbols ({ and }, often referred to as "curly-braces").

- `int main (void)` labels the "Main" function for the computer: the instructions within this function (lying between the { and } symbols) it will be commanded to execute. Every complete C++ program contains a `main` function at minimum, and often additional functions as well, but the `main` function is where execution always begins. The `int` declares this function will return an *integer* number value when complete, which helps to explain the purpose of the `return 0;` statement at the end of the `main` function: providing a numerical value of zero at the program's completion as promised by `int`. This returned value is rather incidental to our purpose here, but it is fairly standard practice in C++ programming.

- Grouping symbols such as (parentheses) and {braces} abound in C, C++, and other languages (e.g. Java). Parentheses typically group data to be processed by a function, called *arguments* to that function. Braces surround lines of executable code belonging to a particular function.

- The `float` declaration reserves places in the computer's memory for two *floating-point* variables, in this case the variables' names being `x` and `y`. In most text-based programming languages, variables may be named by single letters or by combinations of letters (e.g. `xyz` would be a single variable).

- The next two lines assign numerical values to the two variables. Note how each line terminates with a semicolon character (`;`) and how this pattern holds true for most of the lines in this program. In C++ semicolons are analogous to periods at the ends of English sentences. This demarcation of each line's end is necessary because C++ ignores whitespace on the page and doesn't "know" otherwise where one line ends and another begins.

- All the other instructions take the form of a `cout` command which prints characters to the "standard output" stream of the computer, which in this case will be text displayed on the console. The double-less-than symbols (`<<`) show data being sent *toward* the `cout` command. Note how verbatim text is enclosed in quotation marks, while variables such as `x` or mathematical expressions such as `x - y` are not enclosed in quotations because we want the computer to display the numerical values represented, not the literal text.

- Standard arithmetic operations (add, subtract, multiply, divide) are represented as `+`, `-`, `*`, and `/`, respectively.

- The `endl` found at the end of every `cout` statement marks the end of a line of text printed to the computer's console display. If not for these `endl` inclusions, the displayed text would resemble a run-on sentence rather than a paragraph. Note the `cout << endl;` line, which does nothing but create a blank line on the screen, for no reason other than esthetics.

After saving this *source code* text to a file with its own name (e.g. `myprogram.cpp`), you would then *compile* the source code into an *executable* file which the computer may then run. If you are using a console-based compiler such as *GCC* (very popular within variants of the Unix operating system[2], such as Linux and Apple's OS X), you would type the following command and press the Enter key:

```
g++ -o myprogram.exe myprogram.cpp
```

This command instructs the `GCC` compiler to take your source code (`myprogram.cpp`) and create with it an executable file named `myprogram.exe`. Simply typing `./myprogram.exe` at the command-line will then execute your program:

```
./myprogram.exe
```

If you are using a graphic-based C++ development system such as Microsoft Visual Studio[3], you may simply create a new console application "project" using this software, then paste or type your code into the example template appearing in the editor window, and finally run your application to test its output.

As this program runs, it displays the following text to the console:

```
This simple program performs basic arithmetic on
the two numbers 200 and -560.5 and then
displays the results on the computer's console.

Sum = -360.5
Difference = 760.5
Product = -112100
Quotient of -0.356824
```

As crude as this example program is, it serves the purpose of showing how easy it is to write and execute simple programs in a computer using the C++ language. As you encounter C++ example programs (shown as source code) in any of these modules, feel free to directly copy-and-paste the source code text into a text editor's screen, then follow the rest of the instructions given here (i.e. save to a file, compile, and finally run your program). You will find that it is generally easier to

[2] A very functional option for users of Microsoft Windows is called *Cygwin*, which provides a Unix-like console environment complete with all the customary utility applications such as GCC!

[3] Using Microsoft Visual Studio community version 2017 at the time of this writing to test this example, here are the steps I needed to follow in order to successfully compile and run a simple program such as this: (1) Start up Visual Studio and select the option to create a New Project; (2) Select the Windows Console Application template, as this will perform necessary set-up steps to generate a console-based program which will save you time and effort as well as avoid simple errors of omission; (3) When the editing screen appears, type or paste the C++ code within the `main()` function provided in the template, deleting the "Hello World" `cout` line that came with the template; (4) Type or paste any preprocessor directives (e.g. `#include` statements, `namespace` statements) necessary for your code that did not come with the template; (5) Lastly, under the Debug drop-down menu choose either Start Debugging (F5 hot-key) or Start Without Debugging (Ctrl-F5 hotkeys) to compile ("Build") and run your new program. Upon execution a console window will appear showing the output of your program.

learn computer programming by closely examining others' example programs and modifying them than it is to write your own programs starting from a blank screen.

5.2 Programming in Python

Another text-based computer programming language called *Python* allows you to type instructions at a terminal prompt and receive immediate results without having to compile that code. This is because Python is an *interpreted* language: a software application called an *interpreter* reads your source code, translates it into computer-understandable instructions, and then executes those instructions in one step.

The following shows what happens on my personal computer when I start up the Python interpreter on my personal computer, by typing `python3`[4] and pressing the Enter key:

```
Python 3.7.2 (default, Feb 19 2019, 18:15:18)
[GCC 4.1.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The `>>>` symbols represent the prompt within the Python interpreter "shell", signifying readiness to accept Python commands entered by the user.

Shown here is an example of the same arithmetic operations performed on the same quantities, using a Python interpreter. All lines shown preceded by the `>>>` prompt are entries typed by the human programmer, and all lines shown without the `>>>` prompt are responses from the Python interpreter software:

```
>>> x = 200
>>> y = -560.5
>>> x + y
-360.5
>>> x - y
760.5
>>> x * y
-112100.0
>>> x / y
-0.35682426404995538
>>> quit()
```

[4] Using version 3 of Python, which is the latest at the time of this writing.

More advanced mathematical functions are accessible in Python by first entering the line `from math import *` which "imports" these functions from Python's math *library* (with functions identical to those available for the C programming language, and included on any computer with Python installed). Some examples show some of these functions in use, demonstrating how the Python interpreter may be used as a scientific calculator:

```
>>> from math import *
>>> sin(30.0)
-0.98803162409286183
>>> sin(radians(30.0))
0.49999999999999994
>>> pow(2.0, 5.0)
32.0
>>> log10(10000.0)
4.0
>>> e
2.7182818284590451
>>> pi
3.1415926535897931
>>> log(pow(e,6.0))
6.0
>>> asin(0.7071068)
0.78539819000368838
>>> degrees(asin(0.7071068))
45.000001524425265
>>> quit()
```

Note how trigonometric functions assume angles expressed in *radians* rather than *degrees*, and how Python provides convenient functions for translating between the two. Logarithms assume a base of e unless otherwise stated (e.g. the `log10` function for common logarithms).

The interpreted (versus compiled) nature of Python, as well as its relatively simple syntax, makes it a good choice as a person's first programming language. For complex applications, interpreted languages such as Python execute slower than compiled languages such as C++, but for the very simple examples used in these learning modules speed is not a concern.

Another Python math library is `cmath`, giving Python the ability to perform arithmetic on complex numbers. This is very useful for AC circuit analysis using *phasors*[5] as shown in the following example. Here we see Python's interpreter used as a scientific calculator to show series and parallel impedances of a resistor, capacitor, and inductor in a 60 Hz AC circuit:

```
>>> from math import *
>>> from cmath import *
>>> r = complex(400,0)
>>> f = 60.0
>>> xc = 1/(2 * pi * f * 4.7e-6)
>>> zc = complex(0,-xc)
>>> xl = 2 * pi * f * 1.0
>>> zl = complex(0,xl)
>>> r + zc + zl
(400-187.38811239154882j)
>>> 1/(1/r + 1/zc + 1/zl)
(355.837695813625+125.35793777619385j)
>>> polar(r + zc + zl)
(441.717448903332, -0.4381072059213295)
>>> abs(r + zc + zl)
441.717448903332
>>> phase(r + zc + zl)
-0.4381072059213295
>>> degrees(phase(r + zc + zl))
-25.10169387356105
```

When entering a value in rectangular form, we use the `complex()` function where the arguments are the real and imaginary quantities, respectively. If we had opted to enter the impedance values in polar form, we would have used the `rect()` function where the first argument is the magnitude and the second argument is the angle in radians. For example, we could have set the capacitor's impedance (`zc`) as $X_C \angle -90^o$ with the command `zc = rect(xc,radians(-90))` rather than with the command `zc = complex(0,-xc)` and it would have worked the same.

Note how Python defaults to rectangular form for complex quantities. Here we defined a 400 Ohm resistance as a complex value in rectangular form (400 +j0 Ω), then computed capacitive and inductive reactances at 60 Hz and defined each of those as complex (phasor) values ($0 - jX_c$ Ω and $0 + jX_l$ Ω, respectively). After that we computed total impedance in series, then total impedance in parallel. Polar-form representation was then shown for the series impedance (441.717 Ω $\angle -25.102^o$). Note the use of different functions to show the polar-form series impedance value: `polar()` takes the complex quantity and returns its polar magnitude and phase angle in *radians*; `abs()` returns just the polar magnitude; `phase()` returns just the polar angle, once again in radians. To find the polar phase angle in degrees, we nest the `degrees()` and `phase()` functions together.

The utility of Python's interpreter environment as a scientific calculator should be clear from these examples. Not only does it offer a powerful array of mathematical functions, but also unlimited

[5] A "phasor" is a voltage, current, or impedance represented as a complex number, either in rectangular or polar form.

assignment of variables as well as a convenient text record[6] of all calculations performed which may be easily copied and pasted into a text document for archival.

It is also possible to save a set of Python commands to a text file using a text editor application, and then instruct the Python interpreter to execute it at once rather than having to type it line-by-line in the interpreter's shell. For example, consider the following Python program, saved under the filename `myprogram.py`:

```
x = 200
y = -560.5

print("Sum")
print(x + y)

print("Difference")
print(x - y)

print("Product")
print(x * y)

print("Quotient")
print(x / y)
```

As with C++, the interpreter will read this source code from left-to-right, top-to-bottom, just the same as you or I would read a document written in English. Interestingly, whitespace *is* significant in the Python language (unlike C++), but this simple example program makes no use of that.

To execute this Python program, I would need to type `python myprogram.py` and then press the Enter key at my computer console's prompt, at which point it would display the following result:

```
Sum
-360.5
Difference
760.5
Product
-112100.0
Quotient
-0.35682426405
```

As you can see, syntax within the Python programming language is simpler than C++, which is one reason why it is often a preferred language for beginning programmers.

[6] Like many command-line computing environments, Python's interpreter supports "up-arrow" recall of previous entries. This allows quick recall of previously typed commands for editing and re-evaluation.

If you are interested in learning more about computer programming in *any* language, you will find a wide variety of books and free tutorials available on those subjects. Otherwise, feel free to learn by the examples presented in these modules.

5.3 Numeration formats in Python and C++

In this section we will explore the use of two text-based computer programming languages, Python and C++, to perform number-format conversions. As we do so, we will discuss the following programming principles:

- Interpreted versus compiled languages
- Number base prefixes
- Conversion functions (`bin`, `hex`, `oct`)
- Preprocessor directives, namespaces
- Variable types (`int_16t`, `uint_16t`), names, and declarations
- Printing text output (`cout`, `<<`, `endl`, `dec`, `oct`, `hex`, `uppercase`)
- Accepting user input (`cin`, `>>`)
- Loops (`while`)
- Basic arithmetic (`+`, `%`)
- Numerical overflow
- Arrays
- Unions
- Bitwise operators (`&`), masking
- Comments (`//`)

The Python programming language is *interpreted*, which means a computer is able to immediately execute single Python commands as they are typed. This makes Python very useful as a scientific calculator, and also for easily demonstrating elementary programming concepts.

Here, we see a Python console running Python version 3.6.1[7], being used to convert between binary, octal, hexadecimal, and decimal expressions of unsigned integers (i.e. whole numbers):

```
Python 3.6.1 (default, Mar 21 2017, 21:49:16)
[GCC 5.4.0] on cygwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 0b1011011
91
>>> 0x4a32
18994
>>> 0o3721
2001
>>> bin(494)
'0b111101110'
>>> hex(494)
'0x1ee'
>>> oct(494)
'0o756'
>>> quit()
```

Any number prefaced by `0b` is interpreted by Python as a binary number, and its decimal equivalent immediately appears when that quantity is entered. Similarly, `0x` denotes a hexadecimal quantity, and `0o` denotes octal. If we wish to enter a decimal number and see its binary, hexadecimal, or octal expression, we must use the functions `bin()`, `hex()`, and `oct()`, respectively, with the decimal value to be converted placed between the parentheses as an "argument" to that function.

To exit the Python interpreter console, simply enter the function `quit()` as shown.

[7] In order to initiate Python from a command-line environment, simply type `python3` at the command line and press Enter. The Python version running will be shown as it appears here, followed by a `>>>` prompt where you may type and enter lines of Python code.

C++ is a *compiled* programming language, which means all lines of C++ code must be written to a file which is then processed by a piece of software called a *compiler* before the computer can execute that code. Unlike Python, which may be interpreted live (line by line, as it is typed), C++ executes code in batches saved as plain-text files.

Here is an example C++ program useful to displaying a decimal value entered by the user as a binary, octal, and hexadecimal 16-bit signed number.

```
#include <iostream>
#include <cstdint>
#include <bitset>
using namespace std;

int main (void)
{
  int16_t x;

  while (1)
    {
      cout << "Enter the decimal number: ";
      cin >> x;

      cout << dec << "Decimal " << x << " is equal to ";

      cout << std::bitset<16>(x) << " (binary)  ";
      cout << oct << x << " (octal)  ";
      cout << hex << uppercase << x << " (hex)" << endl;

      cout << endl;
    }

  return 0;
}
```

This program uses the `cin` command of the C++ language to receive typed input from whomever runs the program, and it uses a `while` loop to repeatedly accept input and display results, so that the user does not have to re-start the program for each new conversion. Execution of this program may be interrupted at any time using the <*Ctrl-C*> key sequence.

The `cstdint` inclusion near the top of the code listing tells the C++ compiler to include all necessary definitions for fixed-size integer variables, which is necessary[8] in this case because we wish to use 16-bit signed integers only. If we simply declared the variable `x` to be a regular integer instead of a 16-bit integer, the compiler software would default to whatever the standard bit-width of the operating system happened to be. For this example I wanted to limit the program to using only

[8] Depending on the version of C++ compiler installed on your computer.

16-bit integers, hence the special declaration for `x` as well as the `#include <cstdint>` statement. Similarly, the `#include <bitset>` statement was necessary in order to have the `cout` command express `x` as a binary number.

Shown here is one test run of this program, with the numerical values 355 and −10722 entered when prompted by the program:

```
Enter the decimal number: 355
Decimal 355 is equal to 0000000101100011 (binary) 543 (octal) 163 (hex)

Enter the decimal number: -10722
Decimal -10722 is equal to 1101011000011110 (binary) 153036 (octal) D61E (hex)
```

Next is an example C++ program demonstrating simple arithmetic using 16-bit binary numbers:

```
#include <iostream>
#include <cstdint>
using namespace std;

int main()
{
  int16_t x, y, sum;

  cout << "Enter the 1st number: ";
  cin >> x;

  cout << "Enter the 2nd number: ";
  cin >> y;

  sum = x + y;

  cout << endl << "The sum of " << x << " and " << y << " is: ";

  cout << dec << sum << " (decimal)  ";
  cout << oct << sum << " (octal)  ";
  cout << hex << uppercase << sum << " (hex)" << endl;

  return 0;
}
```

As with the previous example, this program uses the `cin` command of the C++ language to receive typed input from whomever runs the program. This way, different numerical values may be tested without having to re-compile the source code.

Shown here is one test run of this program, with the numerical values 4553 and 8882 entered when prompted by the program:

```
Enter the 1st number: 4553
Enter the 2nd number: 8882

The sum of 4553 and 8882 is: 13435 (decimal)  32173 (octal)  347B (hex)
```

As you can see, everything seems to work well in this example. However, the program generates strange results when we try a larger sum:

```
Enter the 1st number: 32341
Enter the 2nd number: 2943

The sum of 32341 and 2943 is: -30252 (decimal)  104724 (octal)  89D4 (hex)
```

There is no way that the sum of 32341 and 2943 should be -30252, but yet this is what the program concludes. The problem here is that the actual sum is too large to be represented by a 16-bit signed integer, and so the result "overflows" with the most-significant-bit (MSB) being 1 rather than 0 which results in a negative value according to two's-complement (signed integer) notation.

A solution for this problem is to declare the integer variables in this program to be *unsigned* rather than *signed* 16-bit integers, which allows for higher count values. Note how the first line within the `main()` function now declares `x`, `y`, and `sum` to be unsigned (`uint16_t`) rather than signed (`int16_t`) variables:

```
#include <iostream>
#include <cstdint>
using namespace std;

int main()
{
  uint16_t x, y, sum;

  cout << "Enter the 1st number: ";
  cin >> x;

  cout << "Enter the 2nd number: ";
  cin >> y;

  sum = x + y;

  cout << endl << "The sum of " << x << " and " << y << " is: ";

  cout << dec << sum << " (decimal)  ";
  cout << oct << sum << " (octal)  ";
  cout << hex << uppercase << sum << " (hex)" << endl;

  return 0;
}
```

Shown here is a test run of this program, with the same numerical values of 32341 and 2943 as entered in the previous program:

```
Enter the 1st number: 32341
Enter the 2nd number: 2943

The sum of 32341 and 2943 is: 35284 (decimal)  104724 (octal)  89D4 (hex)
```

We can still cause a "collision" with this new program, though, if we make the sum large enough:

```
Enter the 1st number: 36401
Enter the 2nd number: 51849

The sum of 36401 and 51849 is: 22714 (decimal)  54272 (octal)  58BA (hex)
```

Even unsigned numbers have their limits (the limit for a 16-bit unsigned number being $2^{16} - 1$, or 65535). The true sum of 36401 and 51849 is 88250, which is why the result "overflows"[9] to yield only 22714.

[9] To understand exactly how this overflow results in the incorrect sum of 22714, it is instructive to imagine a simpler case (i.e. applying the problem-solving technique of *simplifying the problem* in order to obtain a clearer view of the principles at work). We know that a 16-bit unsigned number can only go as high as 65535, so what would happen if our true sum happened to be 65536? The correct binary expression for 65536, of course, would be a *17-bit* number with a 1 in as the MSB and all other bits 0. However, the computer "throws away" this 17th bit because it must fit the result into the 16-bit space allocated, which means the computed result reads zero rather than 65536. Taking another example, if our true sum happened to be 65537 (just one more), the result would "overflow" to read *one*. Therefore, the way to predict the results of an overflow is to take the true sum and subtract 65536. Returning to our last C++ program example, 36401 + 51849 - 65536 = 22714.

This example program displays all 32 bits of a floating point number:

```
#include <iostream>
#include <cstdint>
using namespace std;

int main (void)
{
  int n;
  bool bits[32];               // This is an array of bits 32 long

  union {
    uint32_t integer;
    float fltpnt;  } x;

  cout << "Enter a floating-point value: ";
  cin >> x.fltpnt;

  cout << endl << "Sign    Exponent                 Mantissa" << endl;

  for (n = 0 ; n < 32 ; ++n)
  {
    bits[n] = x.integer & 1;   // Stores LSB in bits[n]
    x.integer = x.integer / 2; // Shifts all bits to the right
  }

  for (n = 0 ; n < 32 ; ++n)
  {
    cout << bits[31-n];        // This displays each bit in the proper order

    if (n == 0)
      cout << "      ";        // Adds whitespace between Sign and Exponent

    if ((n+1) % 4 == 1)
      cout << " ";             // Adds one space between every four bits

    if (n == 8)                // Adds whitespace between Exponent and Mantissa
      cout << "     ";
  }

  cout << endl;

  return 0;
}
```

This program makes use of a construct known as a *data union*, where two different digital words having the exact same number of bits are "joined" so that any data written to one may be read from the other. In this particular case, a union named `x` is made between a 32-bit unsigned integer (`x.integer`) and a 32-bit floating-point value (`x.fltpnt`). The `cin` statement reads in a floating-point value typed by the user, and then this data becomes available for display as an integer number.

Next, this program uses a `for` loop to read the contents of that integer value bit by bit, exactly 32 times in a row. This is done using the *bitwise* operator `&` to "AND" the integer's 32 bits against an integer value of 1 (`0000 0000 0000 0000 0000 0000 0000 0001`). This technique, called *masking*, "masks off" all the bits of that integer value except the LSB, the result being placed into one of the elements of the Boolean array `bits[]`. After masking the integer and copying its LSB to the array, the integer is divided by two which has the effect of shifting all its bits one place to the right, with the old LSB disappearing. Lastly, a second `for` loop prints the bits stored in the `bits[]` array, one by one, formatting them on the console for a pleasant presentation.

Something included in this program to enhance its readability is *comments*. These are words written to explain to any human being reading the code how it works. However, we must delineate comments as such, so the compiler doesn't try to interpret them as C++ code. For this reason, we use double-forward-slash characters (`//`) which C/C++ compilers recognize as comment symbols. Anything to the right of a comment symbol is ignored by the compiler, but still exists in the source code as a cue to any human readers.

Here is an example of this program being run, "dissecting" the floating-point number -8.443×10^5 into its constituent bit-fields:

```
Enter a floating-point value: -8.443e5

Sign    Exponent                Mantissa
1       1001 0010      1001 1100 0100 0001 1000 000
```

This program is a particularly useful tool when learning the ANSI/IEEE 754 floating-point number format (for single-precision, 32-bit floating point numbers) because it breaks down the 32-bit digital word into separate bit-fields (Sign bit, Exponent field, Mantissa field) for easier analysis.

Chapter 6

Questions

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

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

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

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

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

General challenges following tutorial reading

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

General follow-up challenges for assigned problems

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

General follow-up challenges for experiments or projects

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

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

6.1 Conceptual reasoning

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

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

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

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

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

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

6.1.1 Reading outline and reflections

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

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

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

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

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

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

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

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

6.1.2 Foundational concepts

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

Thought experiments as a problem-solving strategy

Working "backwards" to validate calculated results

Simplification as a problem-solving strategy

Reductio ad absurdum

Cut-and-try problem-solving strategy

Numbers versus Numeration

Binary

Place-weight

Whole numbers

Integer numbers

Rational numbers

Irrational numbers

Real numbers

Imaginary numbers

Complex numbers

Radix

MSB versus LSB

Word width

Two's complement

Fixed-point notation

Pulse encoding

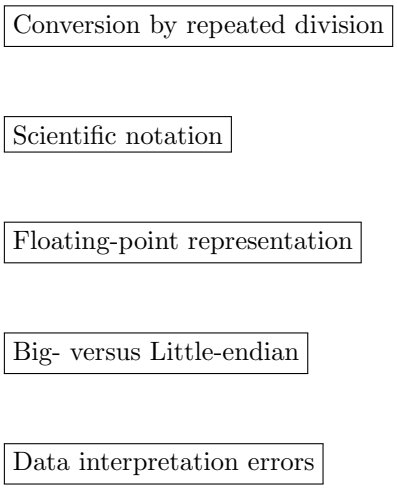
AC motor speed control

Binary-Coded Decimal (BCD)

Programmable Logic Controller (PLC)

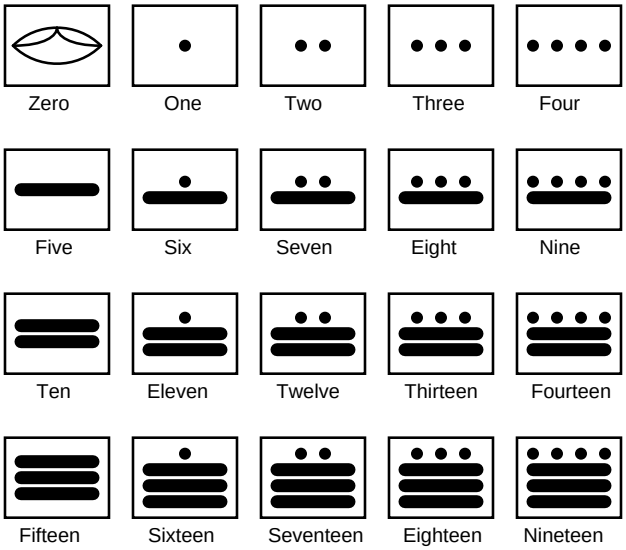
Octal

Hexadecimal

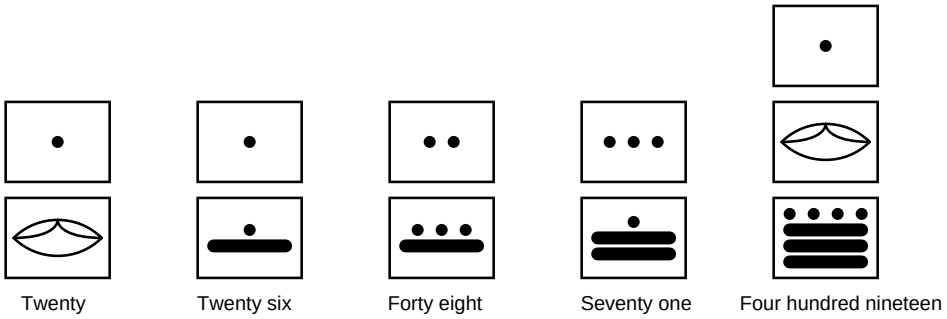

Conversion by repeated division
Scientific notation
Floating-point representation
Big- versus Little-endian
Data interpretation errors

6.1.3 Mayan numeration

The ancient Mayans used a *vigesimal*, or base-twenty, numeration system in their mathematics. Each "digit" was a actually a composite of dots and/or lines, as such:

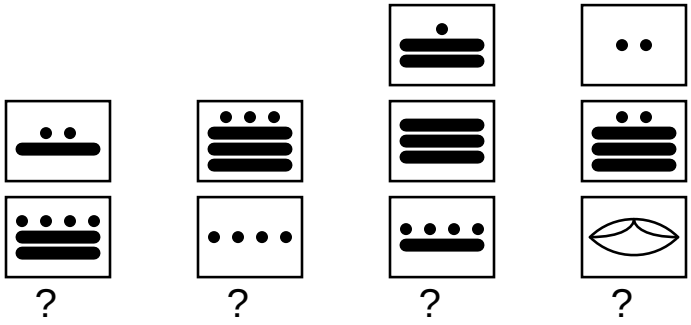


To represent numbers larger than twenty, the Mayans combined multiple "digits" the same way we do to represent numbers larger than ten. For example:



Based on the examples shown above, determine the place-weighting of each "digit" in the vigesimal numeration system. For example, in our denary, or base-ten, system, we have a one's place, a ten's place, a hundred's place, and so on, each successive "place" having ten times the "weight" of the place before it. What are the values of the respective "places" in the Mayan system?

Also, determine the values of these Mayan numbers:

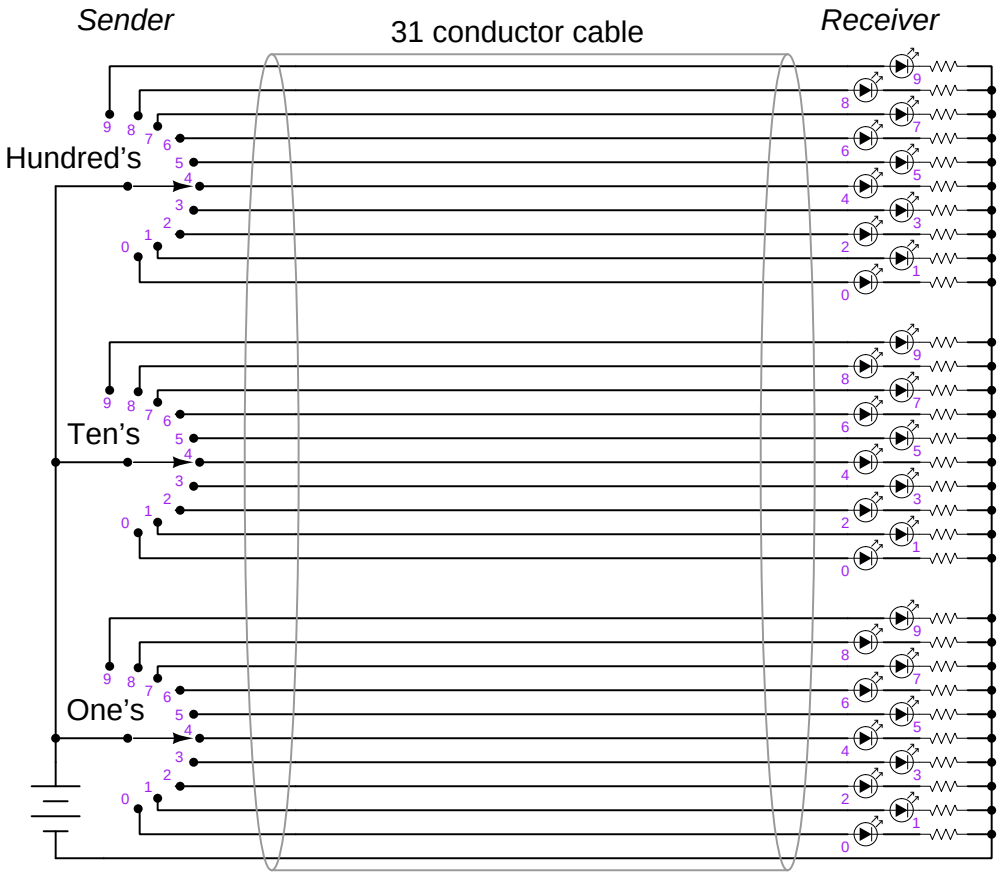


Challenges

- Compare the advantages and disadvantages of the Mayan versus the decimal systems of numeration.

6.1.4 Number transmission via cable

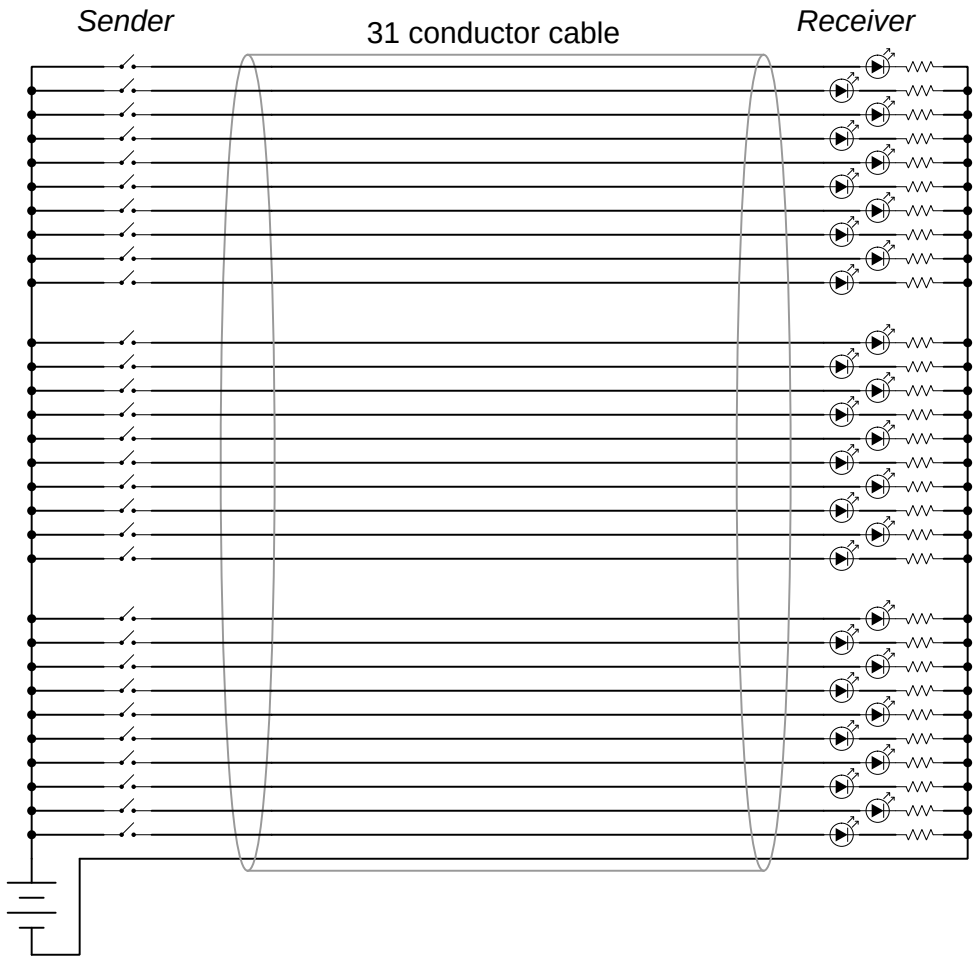
The circuit shown in this diagram is used to transmit a numerical value from one location to another, by means of multi-position electrical switches and LEDs:



Given the switches and lights shown, any whole number between 0 and 999 may be transmitted from the switch location to the LED location.

In fact, the arrangement shown here is not too different from an obsolete design of electronic base-ten indicators known as *Nixie tube* displays, where each digit was represented by a neon-filled glass tube in which one of ten distinct electrodes (each in the shape of a digit, 0-9) could be energized, providing glowing numerals for a person to view.

However, this base-ten circuit is somewhat wasteful of wiring. If we were to use the same thirty-one conductor cable, we could represent a much broader range of numbers if each conductor represented a distinct binary bit, and we used binary rather than base-ten for the numeration system:



How many unique numbers are representable in this simple communications system? Also, what is the greatest individual number which may be sent from the "Sender" location to the "Receiver" location?

Challenges

- Identify a practical application for either of these systems.

6.2 Quantitative reasoning

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

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

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

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

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

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

6.2.1 Miscellaneous physical constants

Note: constants shown in **bold** type are *exact*, not approximations. Values inside of parentheses show one standard deviation (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.

6.2.2 Introduction to spreadsheets

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2.3 Counting in binary, octal, and hexadecimal

Count from one to thirty-one in binary, octal, and hexadecimal, noting patterns you see within each counting sequence:

	Binary	Octal	Hex
Zero			
One			
Two			
Three			
Four			
Five			
Six			
Seven			
Eight			
Nine			
Ten			
Eleven			
Twelve			
Thirteen			
Fourteen			
Fifteen			

	Binary	Octal	Hex
Sixteen			
Seventeen			
Eighteen			
Nineteen			
Twenty			
Twenty one			
Twenty two			
Twenty three			
Twenty four			
Twenty five			
Twenty six			
Twenty seven			
Twenty eight			
Twenty nine			
Thirty			
Thirty one			

Challenges

- Which numeration system uses the least number of places to represent any given quantity? Which numeration system uses the most?
- If binary is how most digital computers represent numbers, why do we ever use octal or hexadecimal?

6.2.4 Binary to decimal and hex conversions

Convert the following unsigned integer numbers from binary (base-two) to decimal (base-ten) as well as to hexadecimal (base-sixteen):

- $10_2 =$
- $1010_2 =$
- $10011_2 =$
- $11100_2 =$
- $10111_2 =$
- $101011_2 =$
- $11100110_2 =$
- $10001101011_2 =$

Challenges

- How would the results differ if you knew these were *signed* binary integers instead?

6.2.5 Decimal to binary conversions

Convert the following unsigned integer numbers from decimal (base-ten) to binary (base-two):

- 7_{10} =
- 10_{10} =
- 19_{10} =
- 250_{10} =
- 511_{10} =
- 824_{10} =
- 1044_{10} =
- 9241_{10} =

Challenges

- How would the results differ if you knew these were *signed* binary integers instead?

6.2.6 Half-life of an anesthetic

Suppose an anesthetic substance administered to a patient has a *half-life* of 9 minutes within the patient's body. This means the effectiveness of the chemical decreases by half its previous value every 9 minutes (e.g. 50% after 9 minutes, 25% after 18 minutes, etc.).

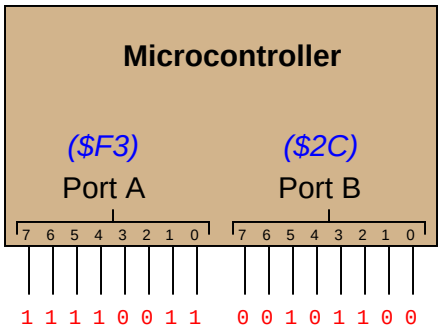
Determine how long after ceasing the flow of this anesthetic will its effectiveness be less than 1 percent, and explain how binary numeration is a helpful model to solve this problem.

Challenges

- Suppose your doctor told you the half-life of the anesthetic she was going to give you prior to a procedure. Explain how this information might be useful to you.

6.2.7 Stepper motor sequence

Digital computers communicate with external devices through *ports*: sets of terminals usually arranged in groups of 4, 8, 16, or more (4 bits = 1 *nybble*, 8 bits = 1 *byte*, 16 bits = 2 bytes). These terminals may be set to high or low logic states by writing a program for the computer that sends a numerical value to the port. For example, here is an illustration of a microcontroller being instructed to send the hexadecimal number `F3` to port A and `2C` to port B:



Suppose we wished to use the upper four bits of port A (pins 7, 6, 5, and 4) to drive power MOSFET gates, which in turn drive coils of a stepper motor in this eight-step sequence:

1. `0001`
2. `0011`
3. `0010`
4. `0110`
5. `0100`
6. `1100`
7. `1000`
8. `1001`

As each pin goes high, it drives a power MOSFET on, which sends current through that respective coil of the stepper motor. By following a "shift" sequence as shown, the motor will rotate a small amount for each cycle.

Write the necessary sequence of numbers to be sent to port A to generate this specific order of bit shifts, in hexadecimal. Leave the lower four bits of port A all in the low logic state.

Also, sketch a diagram showing how one of the microcontroller's terminals could connect to a MOSFET, and that MOSFET to just one coil of a stepper motor.

Challenges

- Sketch a circuit whereby each of the four pins on the microcontroller port connect to MOSFET gates, the rest of each MOSFET connected to drive coils on the stepper motor.

6.2.8 Integer conversion table

Complete this table, performing all necessary conversions between numeration systems of these unsigned integer quantities:

Binary	Octal	Decimal	Hexadecimal
10010			12
	134	92	
	32		1A
110111	67		
1100101		101	
		290	122
1111101000		1000	
	336		DE
1011010110			2D6

Challenges

- Which type of conversion do you find most difficult to perform, and why is that?

6.2.9 Fixed-point integer conversion table

Complete this table, performing all necessary conversions between numeration systems of these fixed-point unsigned quantities:

Binary	Octal	Decimal	Hexadecimal
101.011	5.3		
	31.146	25.2	
	4.54		4.B
111010.101	72.52		
1011.101			B.A
		172.066	AC.11
1110100110.110		934.79	
	641.7		1A1.E
101100.1		44.5	

Challenges

- Which type of conversion do you find most difficult to perform, and why is that?

6.2.10 Signed integer conversion table

Complete this table, performing all necessary conversions between numeration systems of these 16-bit signed integer quantities:

Decimal	**Binary**	**Hexadecimal**
-1		
	0001 0001 1001 0100	1194
-32768		
		AFAF
	0001 1100 0010 1110	
		0041

Challenges

- Which type of conversion do you find most difficult to perform, and why is that?

6.2.11 Using Python to convert between bases

Python is a computer programming language that is able to be run in an *interpreted* environment. This means you can start up a software application called a *Python interpreter*, and within that application type Python commands which will be immediately executed. One of the many features of this programming language is the ability to convert between different numeration systems (i.e. bases).

The following example shows the decimal value 57 being converted into binary, and the binary value 1100101 being converted into decimal. Commands typed at the prompt (`>>>`) of a Python interpreter[11] are executed immediately upon pressing the "Enter" key. The final command, `quit()`, exits the Python interpreter:

```
Python 3.8.4 (default, Jul 13 2020, 17:36:52)
[GCC 4.7.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> bin(57)
'0b111001'
>>> 0b1100101
101
>>> quit()
```

Once you have Python installed and working on your computer[12], demonstrate the following:

- Converting `1A4F` (hexadecimal) to decimal
- Converting 2088 (decimal) to binary
- Converting `371` (octal) to decimal
- Converting 9301 (decimal) to hexadecimal
- Converting `101101` (binary) to hexadecimal

Challenges

- Demonstrate a conversion that Python does *not* perform correctly.

[11] To start the Python interpreter, simply type `python3` (for version 3 of Python, the newest at the time of this writing) at the command-line prompt of any computer with Python installed.

[12] Python may be easily downloaded and installed to your computer from `https://python.org`. However, an alternative to installing Python on your computer is to use the online interpreter available at `https://python.org/shell`.

6.2.12 C++ program converting decimal to other formats

The following computer program written in the C++ language inputs a decimal value from the user and displays its binary, octal, and hexadecimal equivalents, all assuming a 16-bit signed data word:

```
#include <iostream>
#include <cstdint>
#include <bitset>
using namespace std;

int main (void)
{
  int16_t x;

  while (1)
    {
      cout << "Enter the decimal number: ";
      cin >> x;

      cout << dec << "Decimal " << x << " is equal to ";

      cout << std::bitset<16>(x) << " (binary)  ";
      cout << oct << x << " (octal)  ";
      cout << hex << uppercase << x << " (hex)" << endl;

      cout << endl;
    }

  return 0;
}
```

Compile and run this program[13], testing it with a few different number values to verify that it indeed works as it should. Then, modify this program to use an *unsigned* 16-bit word instead (using the integer variable type `uint16_t`) and test-run it again. Also try signed and unsigned 32-bit words (`int32_t` and `uint32_t`, respectively).

[13] Using Microsoft Visual Studio community version 2017, here are the steps I needed to follow in order to successfully compile and run a simple program such as this: (1) Start up Visual Studio and select the option to create a New Project; (2) Select the Windows Console Application template, as this will perform necessary set-up steps to generate a console-based program which will save you time and effort as well as avoid simple errors of omission; (3) When the editing screen appears, type or paste the C++ code within the `main()` function provided in the template, deleting the "Hello World" `cout` line that came with the template; (4) Type or paste any preprocessor directives (e.g. `#include` statements, `namespace` statements) necessary for your code that did not come with the template; (5) Lastly, under the Debug drop-down menu choose either Start Debugging (F5 hot-key) or Start Without Debugging (Ctrl-F5 hotkeys) to compile ("Build") and run your new program. Upon execution a console window will appear showing the output of your program.

Use this program to generate practice problems for yourself: entering random decimal numbers, and then using the resulting binary, octal, and hexadecimal equivalents to either check your own conversion work (decimal into binary, octal, and hex) or to take any of those non-decimal values and check your own conversion work into the other formats.

This program may also serve as a tool to explore the upper and lower range limits of binary words. How large or small of a number can be represented using a 16-bit signed integer? How about a 16-bit unsigned integer? 32 bit?

Challenges

- When switching from 16-bit to 32-bit word sizes, you must change more than just the declaration of the `x` variable. What other portion of the code must be altered for proper 32-bit operation?

6.2.13 Dissecting floating-point numbers

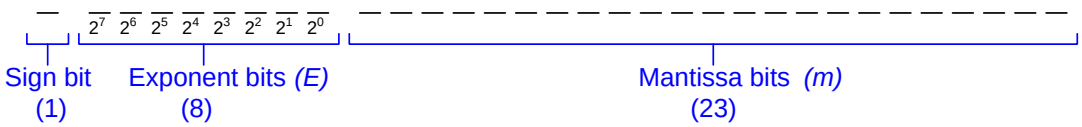
The following computer program written in the C++ programming language prompts the user to enter floating-point numbers, and then displays the 32-bit contents of each number as an eight-character hexadecimal number:

```
#include <iostream>
#include <cstdint>
using namespace std;

int main (void)
{
  union
  {
    uint32_t integer;
    float fltpnt;
  } x;

  while(1)
  {
    cout << "Enter a floating-point value: ";
    cin >> x.fltpnt;

    cout << "Floating point " << x.fltpnt;
    cout << " equivalent to hexadecimal " << hex << uppercase;
    cout << x.integer << endl;
    cout << endl;
  }

  return 0;
}
```

When compiled and run, with the user entering the values 1, −1, 0, −0, 3600, and 3.1415, the following results are obtained:

```
Enter a floating-point value: 1
Floating point 1 equivalent to hexadecimal 3F800000

Enter a floating-point value: -1
Floating point -1 equivalent to hexadecimal BF800000

Enter a floating-point value: 0
Floating point 0 equivalent to hexadecimal 0

Enter a floating-point value: -0
Floating point -0 equivalent to hexadecimal 80000000

Enter a floating-point value: 3600
Floating point 3600 equivalent to hexadecimal 45610000

Enter a floating-point value: 3.1415
Floating point 3.1415 equivalent to hexadecimal 40490E56
```

Analyze each of the hexadecimal values displayed by this program by converting each one to binary form and then mapping the bits to the ANSI/IEEE 754 standard for 32-bit floating-point numbers:

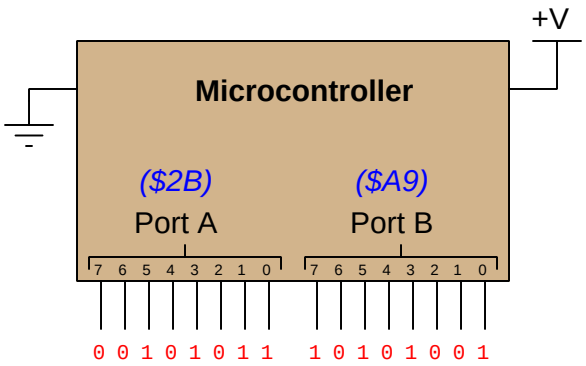


Challenges

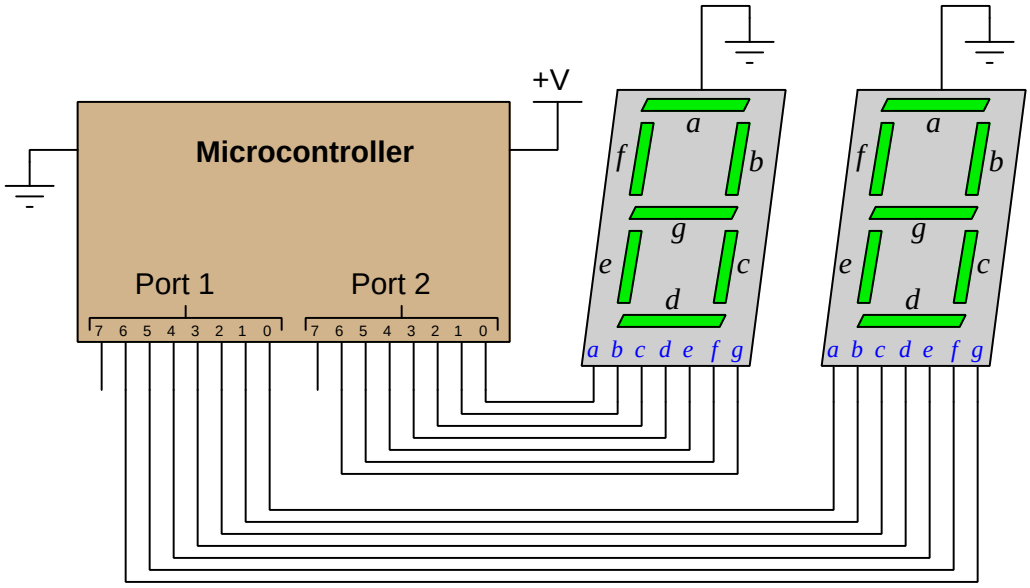
- What might be a practical application for floating-point number representation?

6.2.14 Microcontroller driving seven-segment displays

Digital computers communicate with external devices through *ports*: sets of terminals usually arranged in groups of 4, 8, 16, or more. These terminals may be set to high or low logic states by writing a program for the computer that sends a numerical value to the port. For example, here is an illustration of a single-IC computer called a *microcontroller* being instructed to send the hexadecimal number `2B` to port A and `A9` to port B:



Suppose we wished to use the first seven bits of each port (pins 0 through 6) to drive two 7-segment, common-cathode displays, rather than use BCD-to-7-segment decoder ICs:



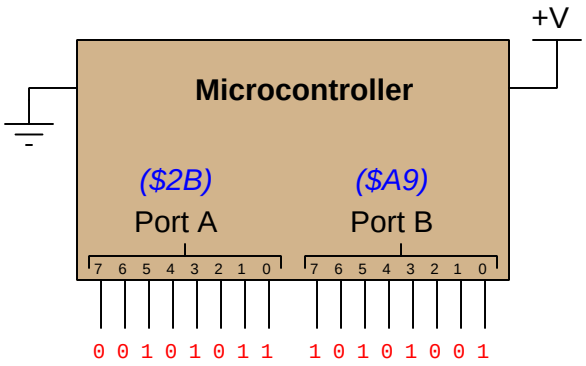
Write the necessary hexadecimal values to be output at ports 1 and 2 to generate the display "42" at the two 7-segment display units.

Challenges

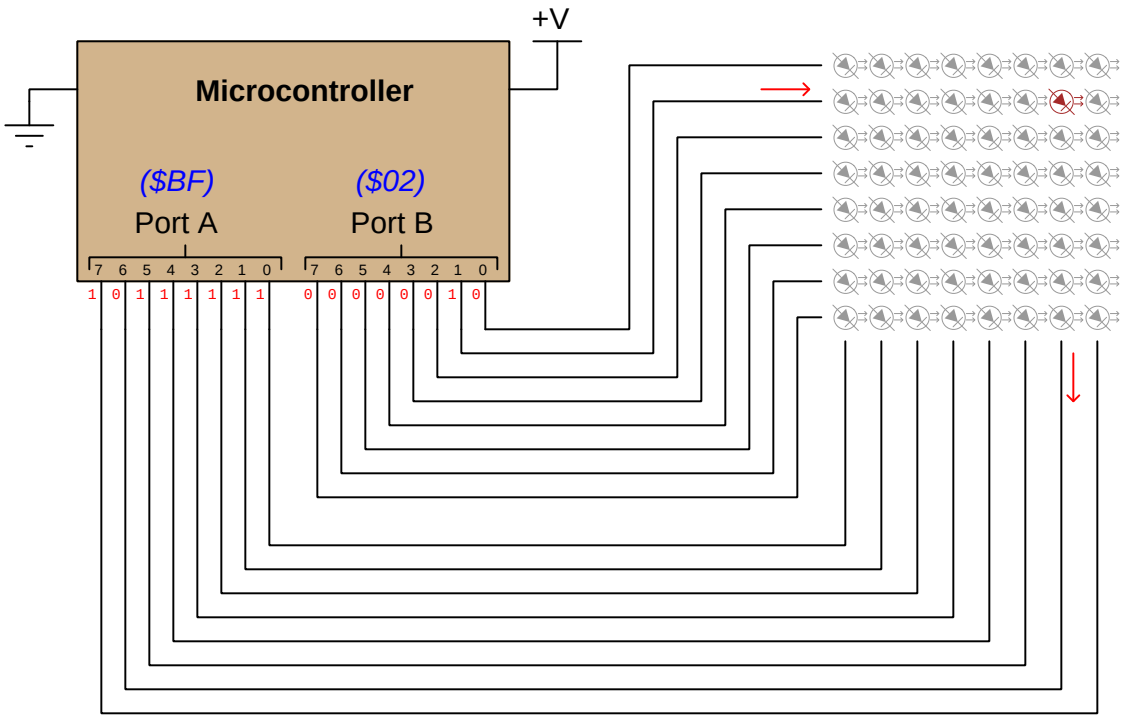
- What advantage is there in programming port values in hexadecimal rather than use plain binary?

6.2.15 Microcontroller driving an LED array

Digital computers communicate with external devices through *ports*: sets of terminals usually arranged in groups of 4, 8, 16, or more. These terminals may be set to high or low logic states by writing a program for the computer that sends a numerical value to the port. For example, here is an illustration of a single-IC computer called a *microcontroller* being instructed to send the hexadecimal number `2B` to port A and `A9` to port B:



One method of driving pixels in a grid-based display is to organize the pixels into rows and columns, then select individual pixels for illumination by the intersection of a specific row line and a specific column line. In this example, we are controlling an 8 × 8 grid of LEDs with two 8-bit (1-byte) ports of a microcontroller:



Note that a *high* state is required on one of port B's pins to activate a row, and a *low* state is required on one of port A's pins to activate a column, because the LED anodes connect to port B and the LED cathodes connect to port A.

Determine the hexadecimal codes we would need to output at ports A and B to energize the LED in the far lower-left corner of the 8 × 8 grid.

Port A =

Port B =

Challenges

- Is it possible to energize more than one LED at a time in this array?

6.3 Diagnostic reasoning

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

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

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

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

6.3.1 Strange floating-point addition

The following computer program written in the C++ language takes a floating-point value of 99997_{10} and displays the results of adding small values to it:

```
#include <iostream>
using namespace std;

int main()
{
  float x = 999997.0;

  cout << x + 0 << endl;
  cout << x + 1 << endl;
  cout << x + 2 << endl;
  cout << x + 3 << endl;
  cout << x + 4 << endl;
  cout << x + 5 << endl;

  return 0;
}
```

When compiled and executed, the result is as follows:

```
999997
999998
999999
1e+06
1e+06
1e+06
```

Explain what is wrong with these results.

Describe a similar scenario with decimal scientific notation which could serve as an analogy to explain what is happening here in this program.

Challenges

- ???.
- ???.
- ???.

6.3.2 Testing endianness

The following computer program written in the C++ language is designed to test how an integer value represented by a 32-bit unsigned binary number is subdivided into 16-bit words and 8-bit bytes within the computer's memory:

```
#include <iostream>
#include <cstdint>
using namespace std;

int main (void)
{
  int n;

  union {
    uint32_t word32;
    uint16_t word16[2];
    uint8_t byte[4];
  } x;

  x.word32 = 999998877;

  cout << "32-bit word = " << hex << uppercase << x.word32 << endl << endl;

  for (n = 0 ; n < 2 ; ++n) {
    cout << "16-bit word " << n << " = " << hex << uppercase << x.word16[n];
    cout << endl; }

  cout << endl;

  for (n = 0 ; n < 4 ; ++n) {
    cout << "Byte " << n << " = " << hex << uppercase << (int)(x.byte[n]);
    cout << endl; }

  return 0;
}
```

When compiled and executed on a particular model of personal computer, the result is as follows:

```
32-bit word = 3B9AC59D

16-bit word 0 = C59D
16-bit word 1 = 3B9A

Byte 0 = 9D
Byte 1 = C5
Byte 2 = 9A
Byte 3 = 3B
```

Describe the word-swapping and byte-swapping displayed in this result.

Challenges

- ???.
- ???.
- ???.

Appendix A

Problem-Solving Strategies

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

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

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

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

Appendix B

Instructional philosophy

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

To high standards of education,

Tony R. Kuphaldt

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

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

Appendix C

Tools used

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

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

The `GNU/Linux` computer operating system

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

Bram Moolenaar's `Vim` text editor

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

Donald Knuth's TEX typesetting system

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

Leslie Lamport's LATEX extensions to TEX

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

Tim Edwards' `Xcircuit` drafting program

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

`Gimp` graphic image manipulation program

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

`SPICE` circuit simulation program

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

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

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

`gnuplot` mathematical visualization software

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

`Python` programming language

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

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

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

Section 1 – Definitions.

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

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

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

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

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

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

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

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

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

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

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

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

Section 2 – Scope.

a. License grant.

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

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

B. produce, reproduce, and Share Adapted Material.

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

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

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

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

5. Downstream recipients.

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

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

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

b. Other rights.

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

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

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

Section 3 – License Conditions.

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

a. Attribution.

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

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

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

ii. a copyright notice;

iii. a notice that refers to this Public License;

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

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

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

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

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

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

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

Section 4 – Sui Generis Database Rights.

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

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

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

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

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

Section 5 – Disclaimer of Warranties and Limitation of Liability.

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

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

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

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

Section 6 – Term and Termination.

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

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

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

2. upon express reinstatement by the Licensor.

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

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

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

Section 7 – Other Terms and Conditions.

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

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

Section 8 – Interpretation.

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

be made without permission under this Public License.

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

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

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

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

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

Appendix E

References

Blanc, Bertrand and Maaraoui, Bob, "Endianness or Where is Byte 0?", 3B Consultancy, December, 2005.

Cohen, Danny, "On Holy Wars and a Plea For Peace", IEN 137, USC/ISI, April 1, 1980.

"DRAFT Standard for Floating-Point Arithmetic P754", Draft 1.2.5, IEEE, New York, October 4, 2006.

Giancoli, Douglas C., *Physics for Scientists & Engineers*, Third Edition, Prentice Hall, Upper Saddle River, NJ, 2000.

Goldberg, David, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", ACM Computing Surveys, Volume 23, Number 1, March 1991.

Hecker, Chris, "Let's Get to the (Floating) Point", Game Developer magazine, pages 19-24, February-March, 1996.

"ieee754.h" header file, GNU C Library, 1999.

"Intel 64 and IA-32 Architectures Software Developer's Manual", Volume 2 Instruction Set Reference, order number 325383-060US, 2016.

Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice Hall, New Jersey, 1978.

Maxwell, James Clerk, *A Treatise on Electricity and Magnetism*, Volume I, Third Edition, Clarendon Press, Oxford, 1904.

Overton, Michael L., "Floating Point Representation", 1996.

PDP11/45 Processor Handbook, Digital Equipment Corporation, 1973.

Swift, Jonathan, *Gulliver's Travels*, 1726.

Waser, Schlomo and Flynn, Michael J., *Introduction to Arithmetic for Digital Systems Designers*, Holt, Rinehard & Winston, New York, 1982.

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.

21 November 2024 – corrected a copy-and-paste error in the decimal number counting table where it said "zero" instead of "ten".

9 November 2024 – added an Introduction section on challenging concepts.

19 August 2024 – added a Case Tutorial chapter with a section on bitwise operations with graphic images borrowed from mod_c.

13 August 2024 – additions and edits made to the "Place-weighted numeration" section of the Tutorial chapter, most notably to include a hexadecimal count sequence.

26 July 2024 – minor formatting to C++ code example in the Programming References chapter to get the program to fit onto a single page.

15 July 2024 – divided the Introduction chapter into two sections, one for students and one for instructors, and added content to the instructor section recommending learning outcomes and measures.

26-29 April 2024 – added some instructor notes, and added a new Tutorial section on place-weighting. Also elaborated on signed integer numbers.

4 December 2023 – added image and text from "Microcontroller driving seven-segment displays" Quantitative Reasoning question to the "Microcontroller driving an LED array" question. Otherwise it felt as though some context were missing, to anyone just reading the latter question without first reading the former. Also added +V and Ground terminals to the microcontrollers shown in images 2402, 2403, and 2404.

17 September 2023 – edited image_1167 to show more explicitly how binary place-weighting works.

16 August 2023 – minor edits to the Tutorial chapter, especially comments about number types. Also added some binary-decimal equivalent examples to the Tutorial, and fixed a typo where I said "binary" but meant to say "hexadecimal".

27 April 2023 – corrected an error in the "Microcontroller driving an LED array" Quantitative Reasoning question where I mistakenly said port A's lines connected to the LED anodes and port B's to the LED cathodes.

6 December 2022 – added a new Case Tutorial section showing examples of counting in signed integer format.

5 December 2022 – minor edits to the Tutorial.

15 August 2022 – deleted a confusing comment in the Historical References section on James Clerk Maxwell's presentation of binary resistance boxes regarding his life versus the publication date of the copy of the book I referenced (which was published after Maxwell died).

2 May 2022 – corrected an omission in the "Binary to decimal and hex conversions" Quantitative Reasoning question where I did not actually ask the reader to convert to hex as well as to decimal.

28 April 2022 – improved and added graphical examples of different-base numbers broken down into individual ciphers and place-weight products. This included editing image_1167 and image_1168 and image_1169 and image_1181, as well as adding some new images.

6-7 December 2021 – corrected typo in a Tutorial section title (!). Instead of "Conversion to decimal by repeated division" it should read "Conversion from decimal by repeated division". Also expanded one of the Quantitative Reasoning questions to include conversion from binary to hexadecimal (rather than just to decimal).

30 June 2021 – edited footnote in the Quantitative Reasoning question "Using Python to convert between bases" to eliminate suggestion of installing `Cygwin`, and also edited the sample Python console to show binary-to-decimal and decimal-to-binary conversions.

8 May 2021 – commented out or deleted empty chapters.

5 December 2020 – expanded the "Stepper motor sequence" question to include sketching a diagram showing how the MCU could connect to a MOSFET and to one coil of the stepper motor.

3 December 2020 – minor edits to the Tutorial.

13 October 2020 – added some instructor notes to questions.

2 October 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.

10 September 2020 – minor edit to "Microcontroller driving seven-segment displays" question,

altering Port A/B to be Port 1/2.

20 July 2020 – added pagebreaks on some questions that contained "framed" verbatim text.

1 July 2020 – added some Challenge questions, and also a Quantitative question on anesthetic. Personal note: this was added after the author recovered from a routine medical procedure in which the administered sedative had a 9-minute half-life. This is called "homework imitating life".

10 May 2020 – renamed "Trial-and-fit" method to "Cut-and-try".

28 January 2020 – added more Foundational Concepts to the list in the Conceptual Reasoning section.

27 January 2020 – added Foundational Concepts to the list in the Conceptual Reasoning section.

5 January 2020 – added bullet-list of relevant programming principles to the Programming References section.

2 January 2020 – removed from from C++ code execution output, to clearly distinguish it from the source code listing which is still framed.

1 January 2020 – changed `main ()` to `main (void)` in C++ programming examples.

23 December 2019 – added a Python programming question, challenging students to use the Python interpreter environment as a number-base converter.

13 November 2019 – swapped a couple of questions from the Quantitative section to the Diagnostic section.

18 May 2019 – typographical error correction in the Tutorial, having to do with interpreting one of the hexadecimal characters in a sample conversion – `1DB` should have been `1EB`. Also, added some clarifying text to the Tutorial based on student feedback.

16 May 2019 – typographical error correction in the Historical References chapter, and another one in a different section. Clarified the process of converting negative decimal values into two's complement signed binary. Added Python programming example to the Quantitative Reasoning section. Added 0o as prepended characters denoting octal in the Tutorial.

12 May 2019 – added an Experiment to demonstrate an integer arithmetic error by programming a computer to specifically for this purpose.

9 May 2019 – added output from a test-run of the numeration conversion C++ program, as well as a quantitative problem based on use of this program.

8 May 2019 – added an "include" statement for `cstdint` in all the C++ code examples using definite-width integer variables such as `int16_t`, which is necessary for some C++ compilers to avoid a scope error.

7 May 2019 – added more questions.

6 May 2019 – removed the word "module" from the title.

9 January 2019 – added examples of floating-point addition errors using Python.

3 January 2019 – corrected missing numerical examples from a sentence describing one of the C++ programming examples.

23 December 2018 – elaborated on methods to denote the radix (base) of written numbers, commented on alternatives to the term "mantissa", finished the sections on "endianness" and incompatible format errors, and made other small edits.

20 December 2018 – document first created.

Index