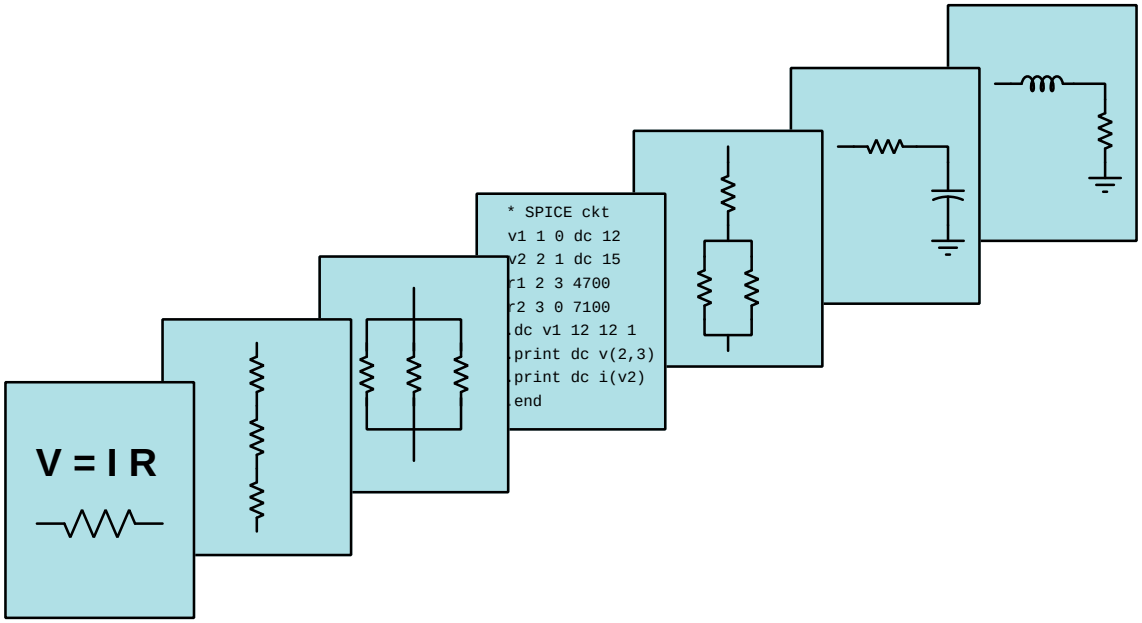
Modular Electronics Learning (ModEL) project



Human-Machine Interfaces

© 2025-2026 by Tony R. Kuphaldt – under the terms and conditions of the Creative Commons Attribution 4.0 International Public License

Last update = 24 March 2026

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

Contents

1 Introduction **3**

2 Case Tutorial **5**
2.1 Example: NAND function in a PLC . . . 6
2.2 Example: simple PLC comparisons . . . 8

3 Tutorial **11**
3.1 Review of basic PLC functionality . . . 11
3.2 Human-Machine Interface basics . . . 20
3.3 Tag name databases . . . 23
3.4 Advanced HMI functionality . . . 27
3.5 Discrete (Boolean) tag programming . . . 27
3.6 Integer tag programming . . . 30
3.7 Integer incompatibility . . . 34
3.8 Floating-point (real) tag programming . . . 39

4 Derivations and Technical References **43**
4.1 Feature comparisons between PLC models . . . 44
4.1.1 Viewing live values . . . 44
4.1.2 Forcing live values . . . 45
4.1.3 Single-bit addressing . . . 45
4.1.4 Special “system” values . . . 46
4.1.5 Free-running clock pulses . . . 46
4.1.6 Standard counter instructions . . . 46
4.1.7 High-speed counter instructions . . . 47
4.1.8 Timer instructions . . . 47
4.1.9 ASCII text message instructions . . . 47
4.1.10 Analog signal scaling . . . 48
4.2 Legacy Allen-Bradley memory maps and I/O addressing . . . 48

5 Programming References **55**
5.1 Programming in C++ . . . 56
5.2 Programming in Python . . . 60
5.3 Displaying different integer types using C++ . . . 65

5.4 Numeration formats in Python and C++ . . . 66

6 Questions **77**
6.1 Conceptual reasoning . . . 81
6.1.1 Reading outline and reflections . . . 82
6.1.2 Foundational concepts . . . 83
6.1.3 Valve motion timer . . . 85
6.2 Quantitative reasoning . . . 87
6.2.1 Miscellaneous physical constants . . . 88
6.2.2 Introduction to spreadsheets . . . 89
6.2.3 Decoding 32-bit real numbers . . . 92
6.3 Diagnostic reasoning . . . 93
6.3.1 Possible faults in a PLC/HMI pump control system . . . 94
6.3.2 Diagnostic tests on a failed PLC/HMI pump control system . . . 96
6.3.3 Troubleshooting a new HMI/PLC system . . . 97
6.3.4 Possible faults in a PLC/HMI package-counting system . . . 99
6.3.5 Failed parking garage counter . . . 100

A Problem-Solving Strategies **103**

B Instructional philosophy **105**
B.1 First principles of learning . . . 106
B.2 Proven strategies for instructors . . . 107
B.3 Proven strategies for students . . . 109
B.4 Design of these learning modules . . . 110

C Tools used **113**

D Creative Commons License **117**

E References **125**

F Version history **127**

Index **127**

Chapter 1

Introduction

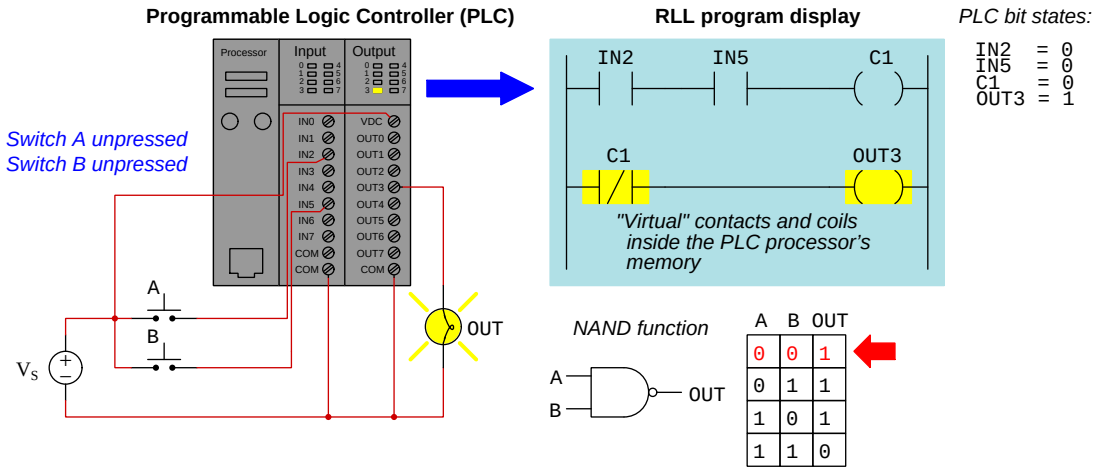
Chapter 2

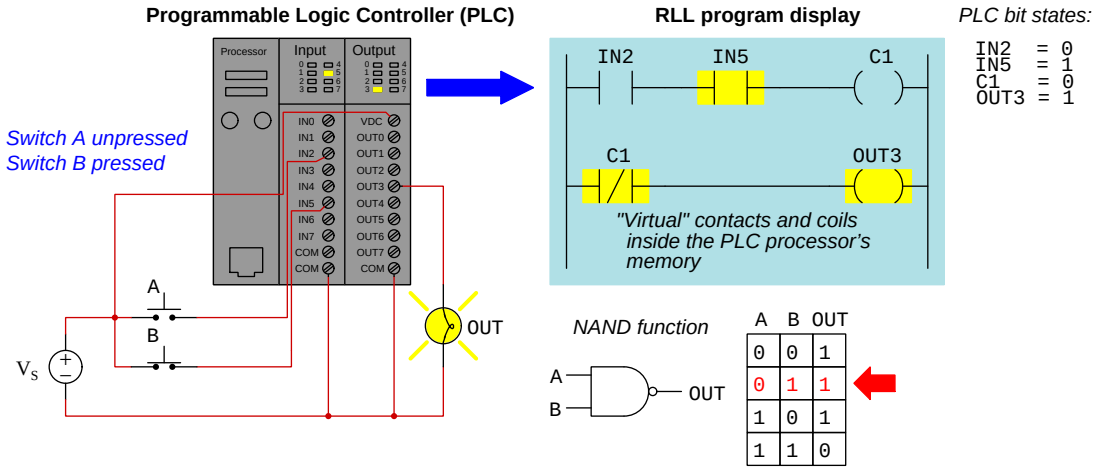
Case Tutorial

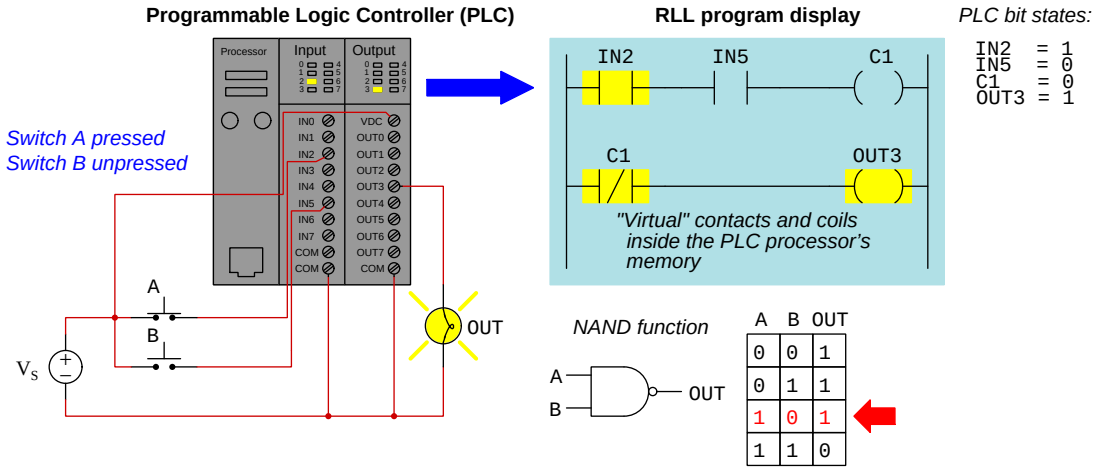
The idea behind a *Case Tutorial* is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as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

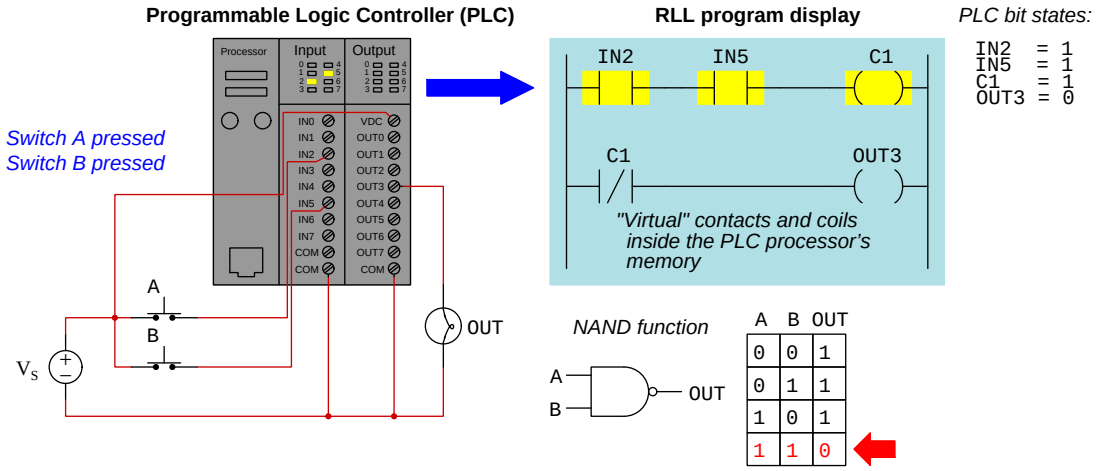
These examples also serve well as challenges following your reading of the other Tutorial(s) in this module – can you explain *why* the circuits behave as they do?

2.1 Example: NAND function in a PLC



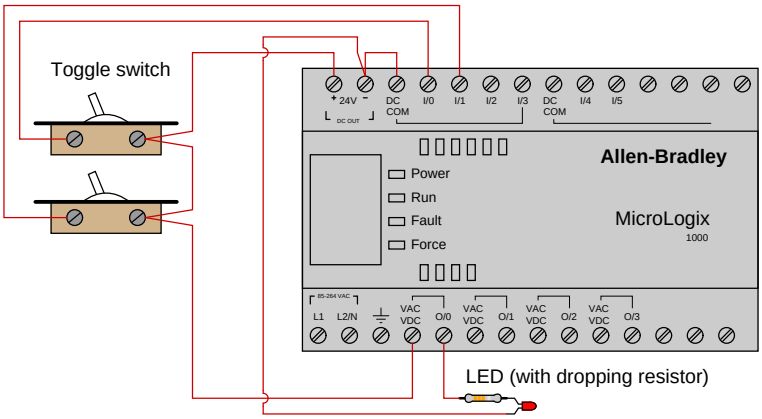



Programmable Logic Controller (PLC)
RLL program display
PLC bit states:
Processor
Input
Output
IN0
IN1
IN2
IN3
IN4
IN5
IN6
IN7
COM
COM
VDC
OUT0
OUT1
OUT2
OUT3
OUT4
OUT5
OUT6
OUT7
COM
Switch A pressed
Switch B unpressed
A
B
OUT
V_S
IN2
IN5
C1
C1
OUT3
"Virtual" contacts and coils inside the PLC processor's memory
IN2 = 1
IN5 = 0
C1 = 0
OUT3 = 1
NAND function
A
B
OUT
A B OUT
0 0 1
0 1 1
1 0 1
1 1 0

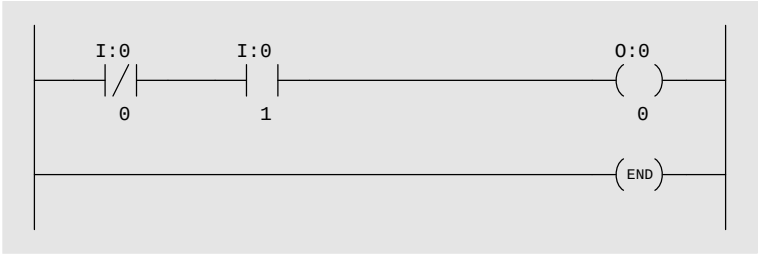

Programmable Logic Controller (PLC)
RLL program display
PLC bit states:
Processor
Input
Output
IN0
IN1
IN2
IN3
IN4
IN5
IN6
IN7
COM
COM
VDC
OUT0
OUT1
OUT2
OUT3
OUT4
OUT5
OUT6
OUT7
COM
Switch A pressed
Switch B pressed
A
B
OUT
V_S
IN2
IN5
C1
C1
OUT3
"Virtual" contacts and coils inside the PLC processor's memory
IN2 = 1
IN5 = 1
C1 = 1
OUT3 = 0
NAND function
A
B
OUT
A B OUT
0 0 1
0 1 1
1 0 1
1 1 0

2.2 Example: simple PLC comparisons

The following illustration shows wiring and a sample relay ladder logic (RLL) program for an Allen-Bradley MicroLogix 1000 PLC:



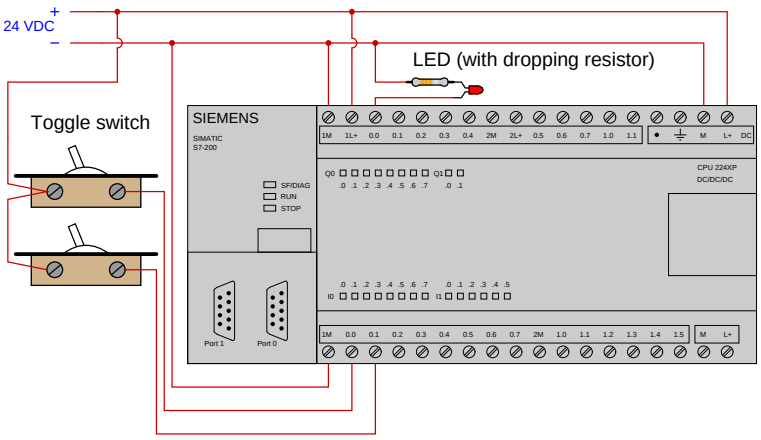
Ladder-Diagram program written to PLC:



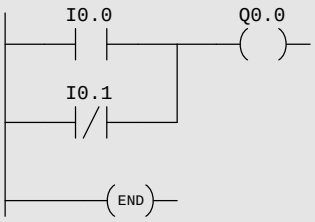
Note how Allen-Bradley I/O is labeled in the program: input bits designated by the letter `I` and output bits designated by the letter `O`.

In order to energize the LED, the switch connected to input terminal 0 must be off (open) and the switch connected to input terminal 1 must be on (closed).

The following illustration shows wiring and a sample relay ladder logic (RLL) program for a Siemens Simatic S7-200 PLC:



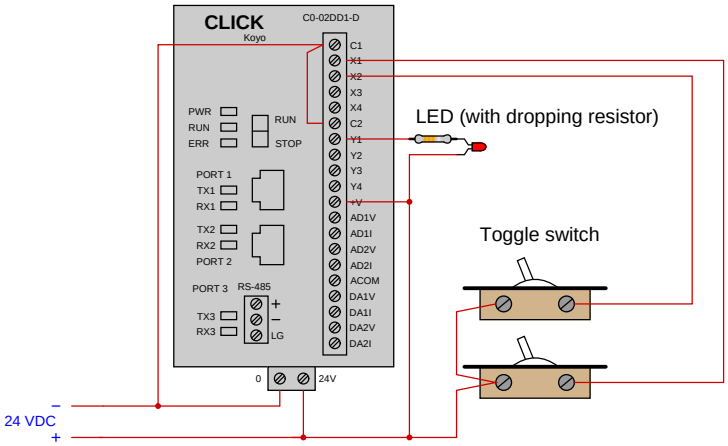
Ladder-Diagram program written to PLC:



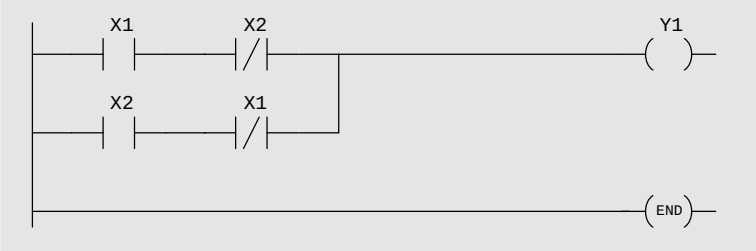
Note how Siemens I/O is labeled in the program: input bits designated by the letter `I` and output bits designated by the letter `Q`.

In order to energize the LED, either the switch connected to input terminal 0.0 must be on (closed) or the switch connected to input terminal 0.1 must be off (open).

The following illustration shows wiring and a sample relay ladder logic (RLL) program for a Koyo CLICK PLC:



Ladder-Diagram program written to PLC:



Note how Koyo I/O is labeled in the program: input bits designated by the letter `X` and output bits designated by the letter `Y`.

In order to energize the LED, at least one of the following conditions must be met:

- X1 switch turned on (closed) and X2 switch turned off (open)
- X2 switch turned on (closed) and X1 switch turned off (open)

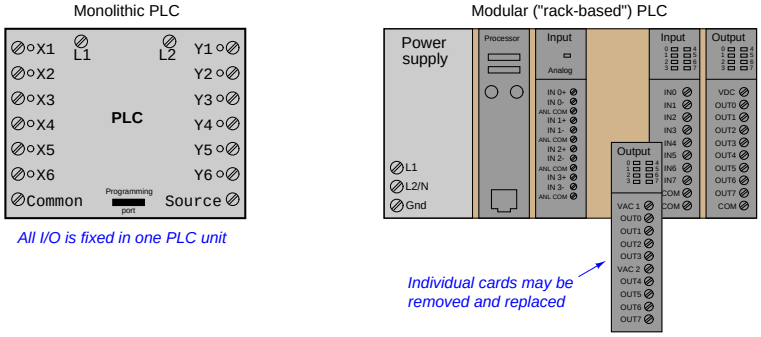
Either the switch connected to input terminal 0.0 must be on (closed) or the switch connected to input terminal 0.1 must be off (open).

Chapter 3

Tutorial

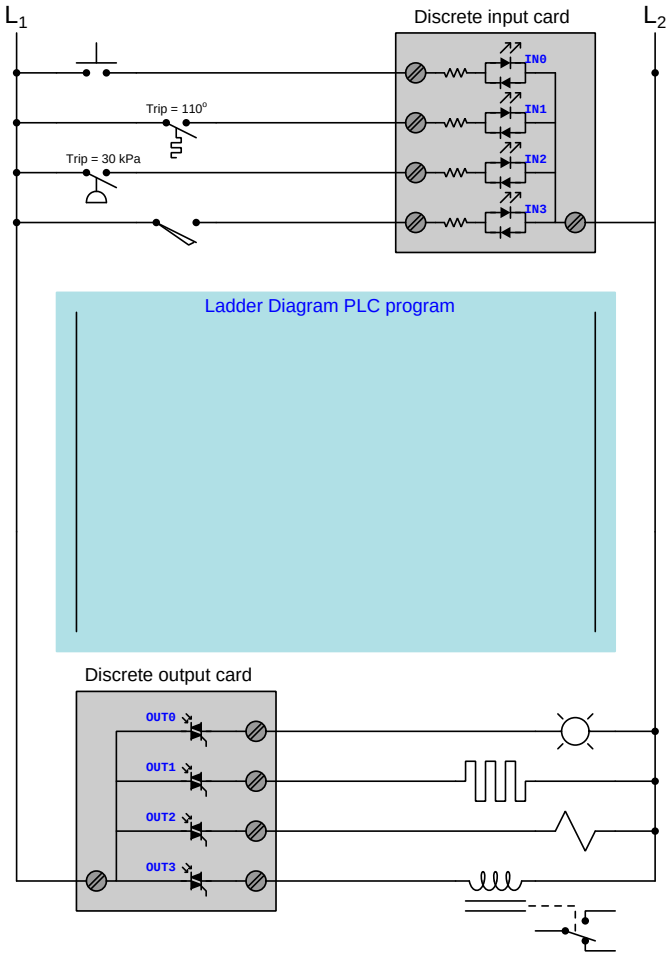
3.1 Review of basic PLC functionality

A *Programmable Logic Controller*, or *PLC*, is a general-purpose industrial computer designed to be easily programmed by end-user maintenance and engineering personnel for specific control functions. PLCs have input and output channels (often hosted on removable "I/O cards") intended to connect to field sensor and control devices such as proximity switches, pushbuttons, solenoids, lamps, sirens, etc. The user-written *program* instructs the PLC how to energize its outputs in accordance with input conditions.



PLCs were originally invented as a replacement for hard-wired relay control systems, and a popular PLC programming language called *Ladder Diagram* was invented to allow personnel familiar with relay ladder logic diagrams to write PLC programs performing the same discrete (on/off) functions as control relays. With a PLC, the discrete functionality for any system could be altered merely by editing the Ladder Diagram program rather than by re-wiring connections between physical relays.

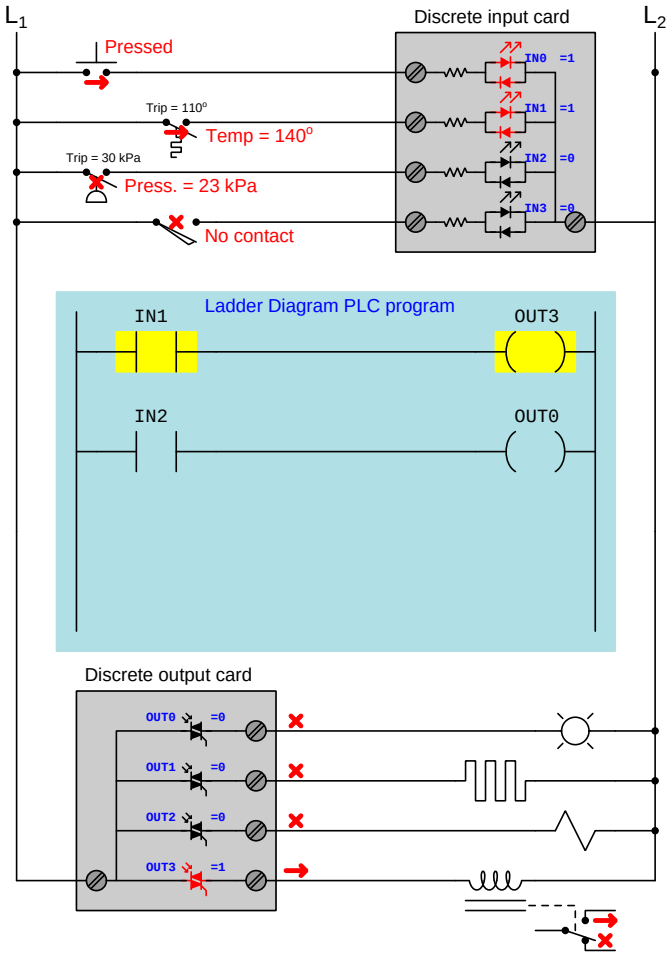
The following diagram shows a PLC separated into three sections: (1) a *discrete input card*, (2) the *program* space in the processor's memory, and (3) a *discrete output card*:



Inputs `IN0` through `IN3` are connected to a pushbutton switch, temperature switch, pressure switch, and limit switch, respectively. Outputs `OUT0` through `OUT3` connect to an indicator lamp, electric heater, solenoid coil, and electromechanical relay coil, respectively. The input card triggers[1] bits in the PLC's memory to switch from 0 to 1 when each respective input is electrically energized, and another set of bits in the PLC's memory control TRIACs inside the output card to turn on when 1 and off when 0. However, with no program installed in the processor, this PLC will not actually *do* anything. As the switch contacts open and close, the only thing the PLC will do is represent their discrete states by the bits `IN0` through `IN3` (0 = de-energized and 1 = energized).

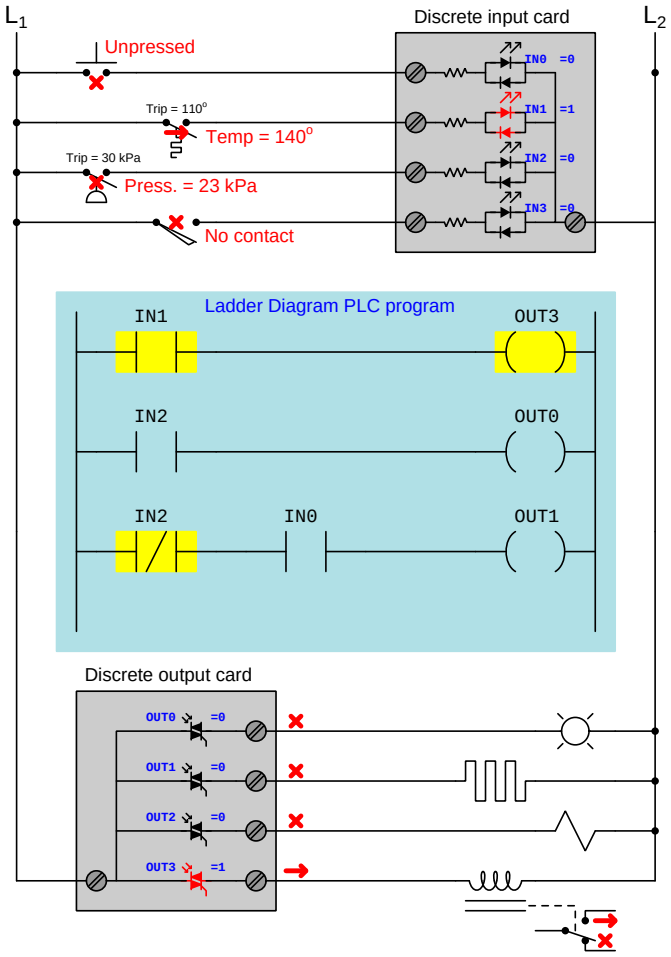
[1] Not shown in this simplified diagram are the optotransistors coupled to the LEDs inside the input card, translating each LED's state to a discrete logic level at the transistor to be interpreted by the PLC's digital processor. Likewise, another set of LEDs driven by the processor's outputs couple to the opto-TRIACs in the output card. Optical isolation of all I/O points is standard design practice for industrial PLCs.

This next diagram shows the same PLC, but this time with a very simple Ladder Diagram program running in the processor, and with stimuli applied to some of the switches:



Inputs `IN0` and `IN1` are energized by their closed switches (pushbutton and temperature), triggering those bits to "1" states in the PLC's memory. The Ladder Diagram program consists of two virtual "contact" instructions and two virtual "coil" instructions, the contact instructions controlled by input bits `IN1` and `IN2` and the coil instructions controlling output bits `OUT3` and `OUT0`. Contact instruction `IN1` "connects" (virtually) to coil instruction `OUT3`, contact `IN2` connecting to coil `OUT0` similarly. Colored highlighting shows the "virtual electricity" status of these instructions, as though they were relays being energized with real electricity. Contact instruction `IN1` is colored because it is a "normally-open" that is being stimulated into its closed state by its "1" bit status. Contact instruction `IN2` is also normally-open, but since its bit is "0" it remains uncolored, and so is the coil it's connected to. The end-result of this program is that the relay's state follows the temperature switch, and the lamp's state follows the pressure switch.

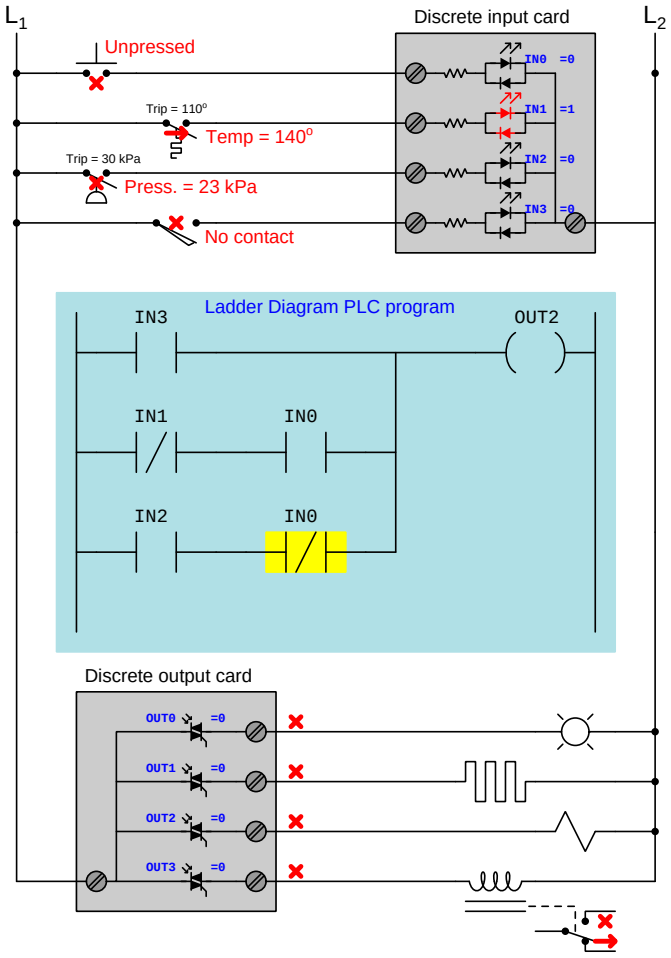
Things get more complex when we begin adding *normally-closed* contact instructions to the program. Consider this next diagram, with updated stimuli and an expanded Ladder Diagram program for the PLC to follow:



The first two rungs of the program are unchanged, as are the temperature and pressure switch statuses, and so outputs `OUT3` and `OUT0` do precisely what they did before. A new rung has been added to the program, with contact instructions linked to bits `IN2` and `IN0`, and the pushbutton switch is no longer being pressed. Both bits `IN0` and `IN2` are currently "0" and so their respective contact instructions are both in their "normal" (i.e. resting) states. The normally-closed contact instruction `IN2` is colored because it is "closed" but the `OUT1` coil in that rung is uncolored because the normally-open contact instruction `IN0` is uncolored and therefore blocks virtual electricity from reaching that coil.

Practically any logic function may be made simply by drawing virtual contact and coil instructions controlling the flow of "virtual electricity". We could describe the above program in Boolean terms: $OUT0 = IN2$; $OUT1 = (\overline{IN2})(IN0)$; $OUT2 = 0$; $OUT3 = IN1$.

This next diagram shows the same PLC with a completely re-written program. The program is now written so that the solenoid coil will energize if the limit switch makes contact, *or* if the temperature is below 110^o *and* the pushbutton is pressed, *or* if the pressure rises above 30 kPa *and* the pushbutton is unpressed:



All switch stimuli are the same as before, resulting in a "0" state for bit `OUT2` and a correspondingly de-energized solenoid. It should be clear to see how this program implements the intended AND and OR functionality by means of series-connected and parallel-connected contact instructions, respectively, with inversion (i.e. the NOT function) implemented by normally-closed rather than normally-open contact instructions.

The logical chain of causality from input to output on a PLC is very important to understand, and will be represented here by a sequence of numbered statements:

1. Energization of input channels controls input bit states (no current = 0 and current = 1)
2. Bit states control the resting/actuated status of contact instructions (0 = resting and 1 = actuated)
3. The resting/actuated status of a contact instruction, combined with its "normal" type determines virtual conductivity (open = uncolored and closed = colored)
4. Continuous color on a rung activates that rung's coil instruction
5. The coil's status controls output bits (uncolored = 0 and colored = 1)
6. Output bits control energization of output channels (0 = off and 1 = on)

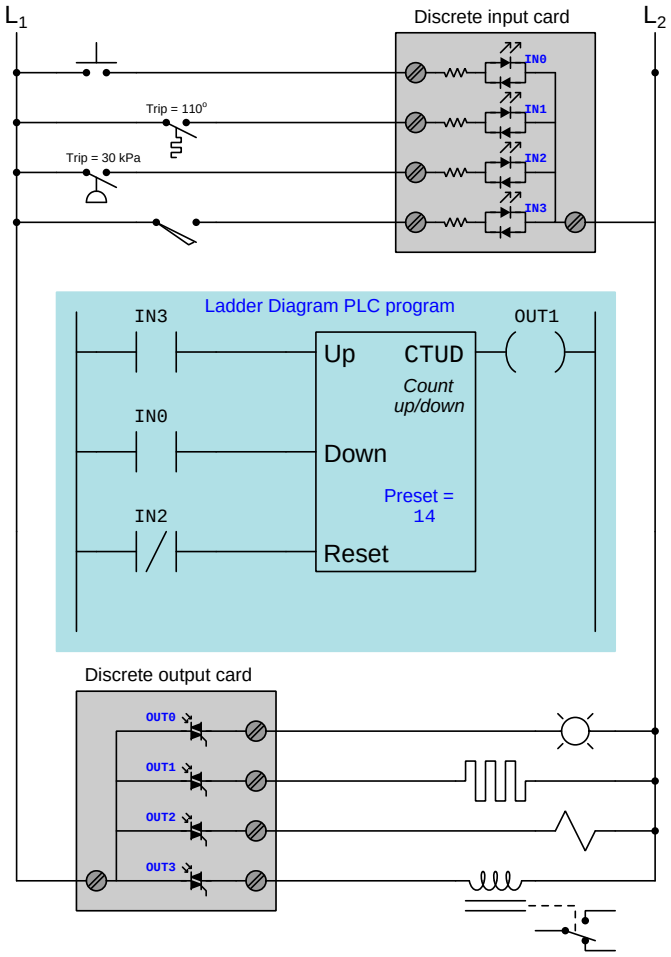
All PLCs follow this chain of logic precisely, and do so in a repeated pattern called the *scan cycle*. Each cycle begins with the PLC's processor reading the electrical states at the input terminals and updating input registers in its memory to reflect those electrical states. The processor then proceeds to evaluate the ladder-diagram program in much the same way we read sentences and paragraphs written in English: from left to right, top rung to bottom rung. Contact instructions are evaluated as either "conductive" or not based on their bit states and their type (e.g. NO or NC), then coil instructions write their states to the PLC's memory, rung by rung through the end of the program. After that the PLC turns on or off its outputs[2] to control real-world devices based on the output register states just updated by the program. Self-diagnostic and other auxiliary evaluations follow that before the entire cycle begins again. This cycle repeats as a continuous loop so long as the PLC is in the "run" mode, each cycle typically taking tens of milliseconds to repeat.

This same step-by-step causality *must* be respected in order to successfully analyze a Ladder Diagram program in a PLC. The most confusing part of this for new students seems to be the relationship of contact instructions to real-world switch inputs. Many students have an unfortunate tendency to want to directly[3] associate real-world switch status with Ladder Diagram color, and/or to believe that the "normal" status of a Ladder Diagram contact instruction must always match the "normal" status of the real-world switch. These and other such misconceptions stem from the same error, namely assuming that contact instructions somehow model or represent real-world contacts, rather than seeing contact instructions as entities *controlled* by the real-world inputs. Only by rigorously following the chain of causation from beginning to end (i.e. input energization → input bit state → contact instruction actuation → color based on normal type *and* bit state → coil color → output bit state → output energization) is one able to properly understand how PLCs function.

[2] At least this is how *most* coil instructions operate. Some PLCs offer a special type of coil instruction called an *immediate* coil that do not wait until the end of the program's scan to drive real-world outputs, but do so immediately upon evaluation mid-way through the program scan.

[3] For a normally-open contact instruction, this association is direct. However, for a normally-closed contact instruction it is inverted!

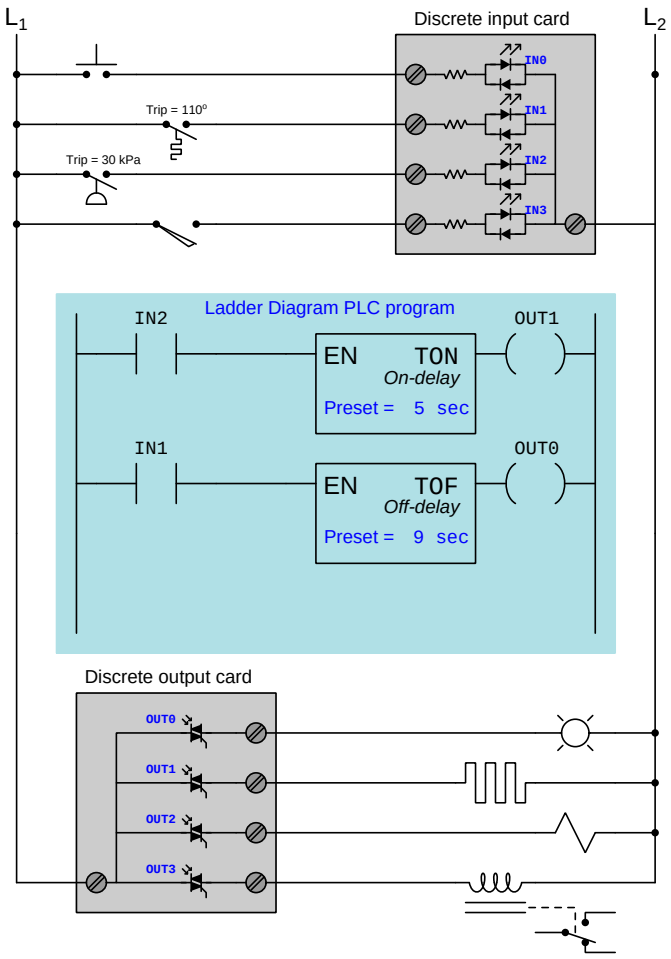
Being fully-fledged digital computers in their own right, PLCs are not limited to executing simple Boolean functions represented by "virtual relay" contacts and coils. Other digital functions include *counters* and *timers*. An example of a counter program is shown here:



The `CTUD` instruction is an *up/down counter* receiving three discrete inputs and generating one discrete output. The program is written so that this counter instruction's count value will increment (i.e. count up) once for every closure of the limit switch, decrement (i.e. count down) once for every closure of the pushbutton switch, and reset to zero if the pressure falls below 30 kPa. The output signal ("wired" to coil `OUT1`) energizes the heating element if this count value reaches or exceeds the "preset" value of 14.

The PLC's scan cycle is exactly the same for this and for all programs as it was for the last: the PLC first scans its inputs and updates those bits in its input memory registers, next scans the program rung-by-rung from left to right and top to bottom as it writes to internal and output registers, next updates all its actual outputs based on those updated output registers, and after some internal "housekeeping" tasks repeats the cycle again.

Next we see an example PLC program showcasing two *timing* instructions, an *on-delay* timer and an *off-delay* timer:



When the pressure exceeds 30 kPa and closes the pressure switch connected to input `IN2`, the `TON` timer instruction begins counting. After 5 seconds of continuous activation, output `OUT1` activates to energize the heating element. When the pressure falls below 30 kPa, the heating element immediately de-energizes.

When the temperature exceeds 110^o and closes the temperature switch connected to input `IN1`, the `TOF` timer instruction immediately activates its output (`OUT0`) to energize the indicator lamp. When the temperature cools down below 110^o, the off-delay timer begins timing and does not de-energize the indicator lamp until 9 seconds after the temperature switch has opened.

Both the utility and versatility of programmable logic controllers should be evident in this brief tutorial. These are digital computers, fully programmable by the end-user in a simple instructional language, designed to implement discrete logic functions, counting functions, timing functions, and a whole host of other useful operations for the purpose of controlling electrically-based systems. Originally designed to replace hard-wired electromechanical relay control circuits, PLCs are designed to mimic the functionality of relays while providing superior reliability and reconfigurability.

PLCs are not limited to contact, coil, counter, and timer instructions, either. A typical PLC literally offers dozens of instruction types in its set, which may be applied and combined in nearly limitless fashion. Other types of PLC programming instructions include *latch instructions* (offering bistable "set" and "reset" capability), *one-shot instructions* (outputting an active state for exactly one "scan" of the PLC's program every time the input transitions from inactive to active), *sequencers* (controlling a pre-determined sequence of discrete states based on a count value), *arithmetic instructions* (e.g. addition, subtraction, multiplication, division, etc.), *comparison instructions* (comparing two numerical values and generating a discrete signal indicating equality, inequality, etc.), *data communication instructions* (sending and receiving digital messages over a communications network), and *clock/calendar functions* (tracking time and date).

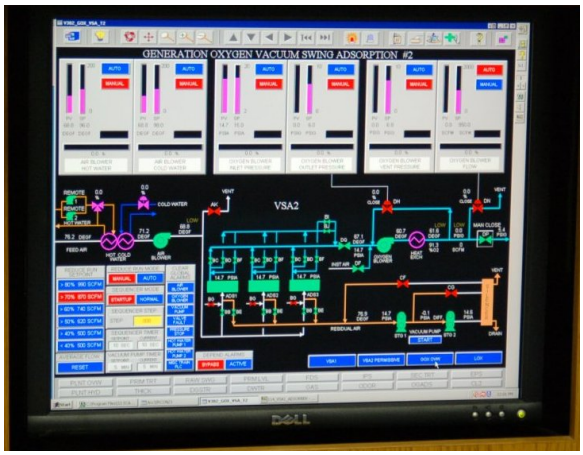
One advantage of PLCs over relay circuitry which may not be evident at first inspection is the fact that the number of virtual "contacts" and "coils" and other instructions is limited only by how much memory the PLC's processor has. The example programs shown on the previous pages were extremely short, but a real PLC program may be dozens of pages long! Electromechanical control and timing relays are, of course, limited in the number of physical switch contacts each one offers, which in turn limits how elaborate the control system may be. For the sake of illustration, a PLC with a single discrete input (say, `IN0` wired to a pushbutton switch) may contain a program with *hundreds* of virtual contacts labeled `IN0` triggering all kinds of logical, counting, and timing functions.

3.2 Human-Machine Interface basics

Programmable logic controllers are built to input various signal types (discrete, analog), execute control algorithms on those signals, and then output signals in response to control processes. By itself, a PLC generally lacks the capability of displaying those signal values and algorithm variables to human operators. A technician or engineer with access to a personal computer and the requisite software for editing the PLC's program may connect to the PLC and view the program's status "online" to monitor signal values and variable states, but this is not a practical way for operations personnel to monitor what the PLC is doing on a regular basis. In order for operators to monitor and adjust parameters inside the PLC's memory, we need a different sort of interface allowing certain variables to be read and written without compromising the integrity of the PLC by exposing too much information or allowing any unqualified person to alter the program itself.

One solution to this problem is a dedicated computer display programmed to provide selective access to certain variable's in the PLC's memory, generally referred to as *Human*[4]*-Machine Interface*, or *HMI*.

HMIs may take the form of general-purpose ("personal") computers running special graphic software to interface with a PLC, or as special-purpose computers designed to be mounted in sheet metal panel fronts to perform no task but the operator-PLC interface. This first photograph shows an example of an ordinary personal computer (PC) with HMI software running on it:

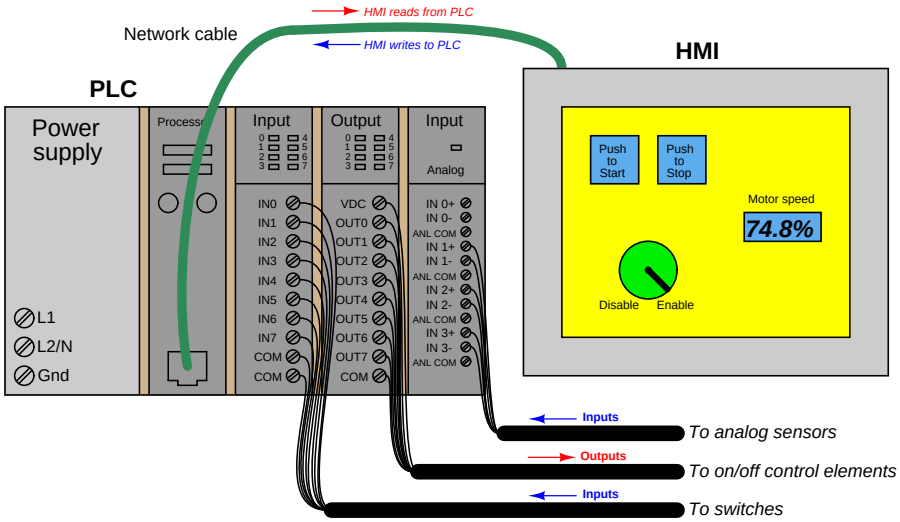


The display shown here happens to be for monitoring a vacuum swing adsorption (VSA) process for purifying oxygen extracted from ambient air. Somewhere, a PLC (or collection of PLCs) is monitoring and controlling this VSA process, with the HMI software acting as a "window" into the PLC's memory to display pertinent variables in an easy-to-interpret form for operations personnel. The personal computer running this HMI software connects to the PLC(s) via digital network cables such as Ethernet.

[4] An older term for an operator interface panel was the "Man-Machine Interface" or "MMI." However, this fell out of favor due to its sexist tone.

This next photograph shows an example of a special-purpose HMI panel designed and built expressly to be used in industrial operating environments:

These HMI panels are really nothing more than "hardened[5]" personal computers built ruggedly and in a compact format to facilitate their use in industrial environments. Most industrial HMI panels come equipped with touch-sensitive screens, allowing operators to press their fingertips on displayed objects to change screens, view details on portions of the process, etc.



Technicians and/or engineers program HMI displays to read and write data via a digital network to one or more PLCs. Graphical objects arrayed on the display screen of an HMI often mimic real-world indicators and switches, in order to provide a familiar interface for operations personnel.

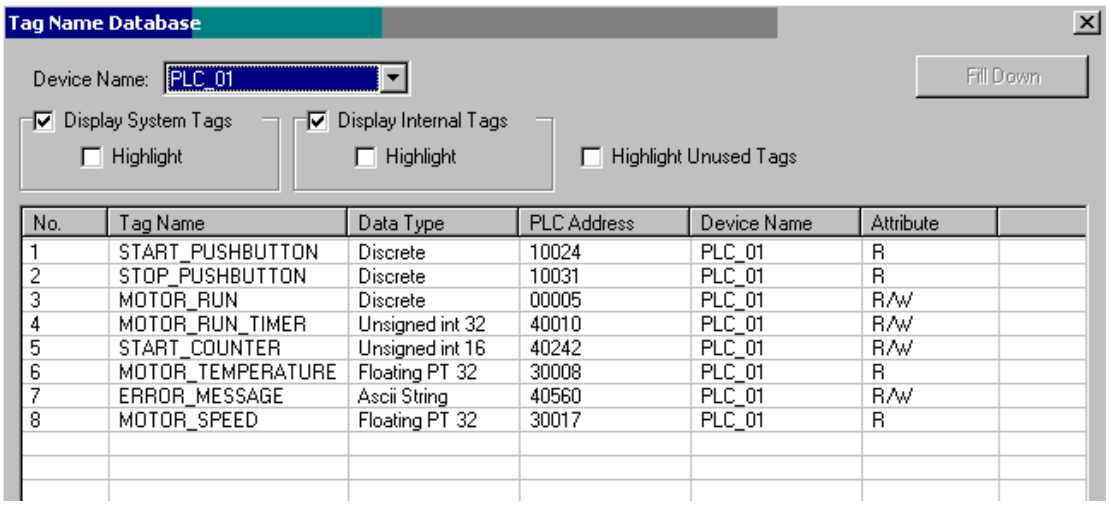
[5]Industrial computers typically lack moving parts such as cooling fans and hard drives with rotating disks in order to achieve significantly greater operating lifespans and reliability. The lack of cooling fans, especially for a computer which may very well be installed in an environment much hotter than most homes or offices, generally means slower-speed microprocessors (to generate less heat) and passive cooling systems with heat sinks large enough that forced-air cooling becomes unnecessary.

A “pushbutton” object on the face of an HMI panel, for example, would be configured to *write* one bit of data to the PLC, in a manner similar to a real-world switch writing one bit of data to the PLC’s input register.

3.3 Tag name databases

Modern HMI panels and software are almost exclusively tag-based, with each graphic object on the screen associated with at least one data tag name, which in turn is associated to data points (bits, or words) in the PLC by way of a tag name database file resident in the HMI. Graphic objects on the HMI screen either accept (read) data from the PLC to present useful information to the operator, send (write) data to the PLC from operator input, or both. The task of programming an HMI unit consists of building a tag name database and then drawing screens to illustrate the process to as good a level of detail as operators will need to run it.

An example screenshot of a tag name database table for a modern HMI is shown here:



No.	Tag Name	Data Type	PLC Address	Device Name	Attribute
1	START_PUSHBUTTON	Discrete	10024	PLC_01	R
2	STOP_PUSHBUTTON	Discrete	10031	PLC_01	R
3	MOTOR_RUN	Discrete	00005	PLC_01	R/W
4	MOTOR_RUN_TIMER	Unsigned int 32	40010	PLC_01	R/W
5	START_COUNTER	Unsigned int 16	40242	PLC_01	R/W
6	MOTOR_TEMPERATURE	Floating PT 32	30008	PLC_01	R
7	ERROR_MESSAGE	Ascii String	40560	PLC_01	R/W
8	MOTOR_SPEED	Floating PT 32	30017	PLC_01	R

The tag name database is accessed and edited using the same software to create graphic images in the HMI. In this particular example you can see several tag names (e.g. `START_PUSHBUTTON`, `MOTOR_RUN_TIMER`, `ERROR_MESSAGE`, `MOTOR_SPEED`) associated with data points within the PLC's memory (in this example, the PLC addresses are shown in Modbus register format). In many cases the tag name editor will be able to display corresponding PLC memory points in the same manner as they appear in the PLC programming editor software (e.g. `I:5/10`, `SM0.4`, `C11`, etc.).

An important detail to note in this tag name database display is the read/write attributes of each tag. Note in particular how four of the tags shown are *read-only*: this means the HMI only has permission to read the values of those four tags from the PLC's memory, and not to write (alter) those values. The reason for this in the case of these four tags is that those tags refer to PLC input data points. The `START_PUSHBUTTON` tag, for instance, refers to a discrete input in the PLC energized by a real pushbutton switch. As such, this data point gets its state from the energization of the discrete input terminal. If the HMI were to be given *write* permission for this data point, there would likely be a conflict. Suppose input terminal on the PLC were energized (setting the `START_PUSHBUTTON` bit to a "1" state) and the HMI simultaneously attempted to write a "0" state to the same tag. One of these two data sources would win, and other would lose, possibly resulting in unexpected behavior from the PLC program. For this reason, data points in the PLC linked to

real-world inputs should always be limited as "read-only" permission in the HMI's database, so the HMI cannot possibly generate a conflict.

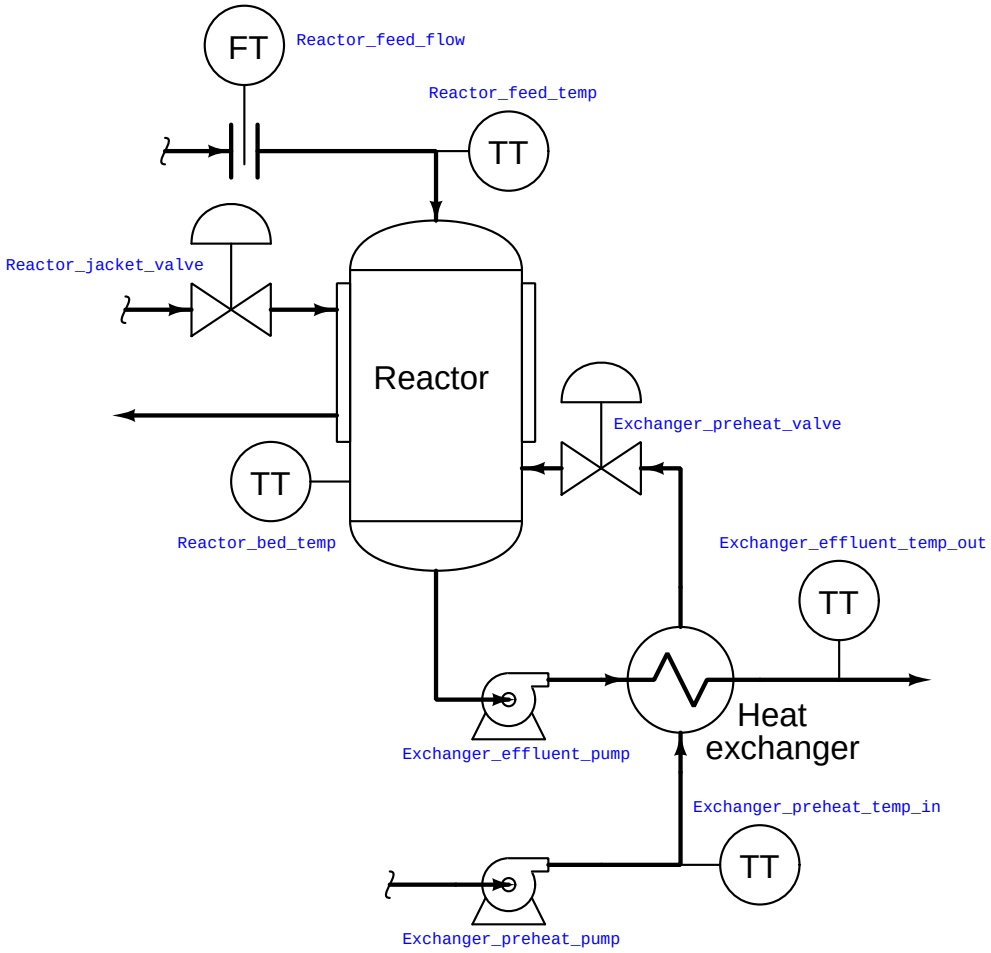
The potential for data conflict also exists for some of the other points in the database, however. A good example of this is the `MOTOR_RUN` bit, which is the bit within the PLC program telling the real-world motor to run. Presumably, this bit gets its data from a coil in the PLC's Ladder Diagram program. However, since it also appears in the HMI database with *read/write* permission, the potential exists for the HMI to over-write (i.e. conflict) that same bit in the PLC's memory. Suppose someone programmed a toggling "pushbutton" screen object in the HMI linked to this tag: pushing this virtual "button" on the HMI screen would attempt to set the bit (1), and pushing it again would attempt to reset the bit (0). If this same bit is being written to by a coil in the PLC's program, however, there exists the distinct possibility that the HMI's "pushbutton" object and the PLC's coil will conflict, one trying to tell the bit to be a "0" while the other tries to tell that bit to be a "1". This situation is quite similar to the problem experienced when multiple coils in a Ladder Diagram program are addressed to the same bit.

The general rule to follow here is *never allow more than one element to write to any data point.* In my experience teaching PLC and HMI programming, this is one of the more common errors students make when first learning to program HMIs: they will try to have both the HMI and the PLC writing to the same memory locations, with weird results.

One of the lessons you will learn when programming large, complex systems is that it is very beneficial to define all the necessary tag names *before* beginning to lay out graphics in an HMI. The same goes for PLC programming: it makes the whole project go faster with less confusion if you take the time to define all the necessary I/O points (and tag names, if the PLC programming software supports tag names in the programming environment) before you begin to create any code specifying how those inputs and outputs will relate to each other.

Maintaining a consistent convention for tag names is important, too. For example, you may wish to begin the tag name of every hard-wired I/O point as either `INPUT` or `OUTPUT` (e.g. `INPUT_PRESSURE_SWITCH_HIGH`, `OUTPUT_SHAKER_MOTOR_RUN`, etc.). The reason for maintaining a strict naming convention is not obvious at first, since the whole point of tag names is to give the programmer the freedom to assign *arbitrary names* to data points in the system. However, you will find that most tag name editors list the tags in alphabetical order, which means a naming convention organized in this way will present all the input tags contiguously (adjacent) in the list, all the output tags contiguously in the list, and so on.

Another way to leverage the alphabetical listing of tag names to your advantage is to begin each tag name with a word describing its association to a major piece of equipment. Take for instance this example of a chemical fluid processing system with several data points[6] defined in a PLC control system and displayed in an HMI:



[6] Each circle with "TT" written inside is a *temperature transmitter*, which is the industrial instrumentation term for a temperature sensor. The "FT" circle is a *flow transmitter*, reporting the rate of flow of fluid through that pipe.

If we list all these tags in alphabetical order, the association is immediately obvious:

- `Exchanger_effluent_pump`
- `Exchanger_effluent_temp_out`
- `Exchanger_preheat_pump`
- `Exchanger_preheat_temp_in`
- `Exchanger_preheat_valve`
- `Reactor_bed_temp`
- `Reactor_feed_flow`
- `Reactor_feed_temp`
- `Reactor_jacket_valve`

As you can see from this tag name list, all the tags directly associated with the heat exchanger are located in one contiguous group, and all the tags directly associated with the reactor are located in the next contiguous group. In this way, judicious naming of tags serves to group them in hierarchical fashion, making them easy for the programmer to locate at any future time in the tag name database.

You will note that all the tag names shown here lack space characters between words (e.g. instead of "`Reactor bed temp`", a tag name should use hyphens or underscore marks as spacing characters: "`Reactor_bed_temp`"), since spaces are generally assumed by computer programming languages to be delimiters (separators between different variable names).

3.4 Advanced HMI functionality

Like programmable logic controllers themselves, the capabilities of HMIs have been steadily increasing over time. Modern HMIs support graphic trending, data archival, event monitoring, audio file playback, and even web server ability allowing other computers to easily access certain data over wide-area networks. The ability of HMIs to log data over long periods of time relieves the PLC of having to do this task, which is very memory-intensive. This way, the PLC merely "serves" current data to the HMI, and the HMI is able to keep a record of current and past data using its vastly larger memory reserves[7].

Some modern HMI panels even have a PLC built inside the unit, providing control and monitoring in the same device. Such panels provide terminal connection points for discrete and even analog I/O wires, allowing all control and interface functions to be located in a single panel-mount unit.

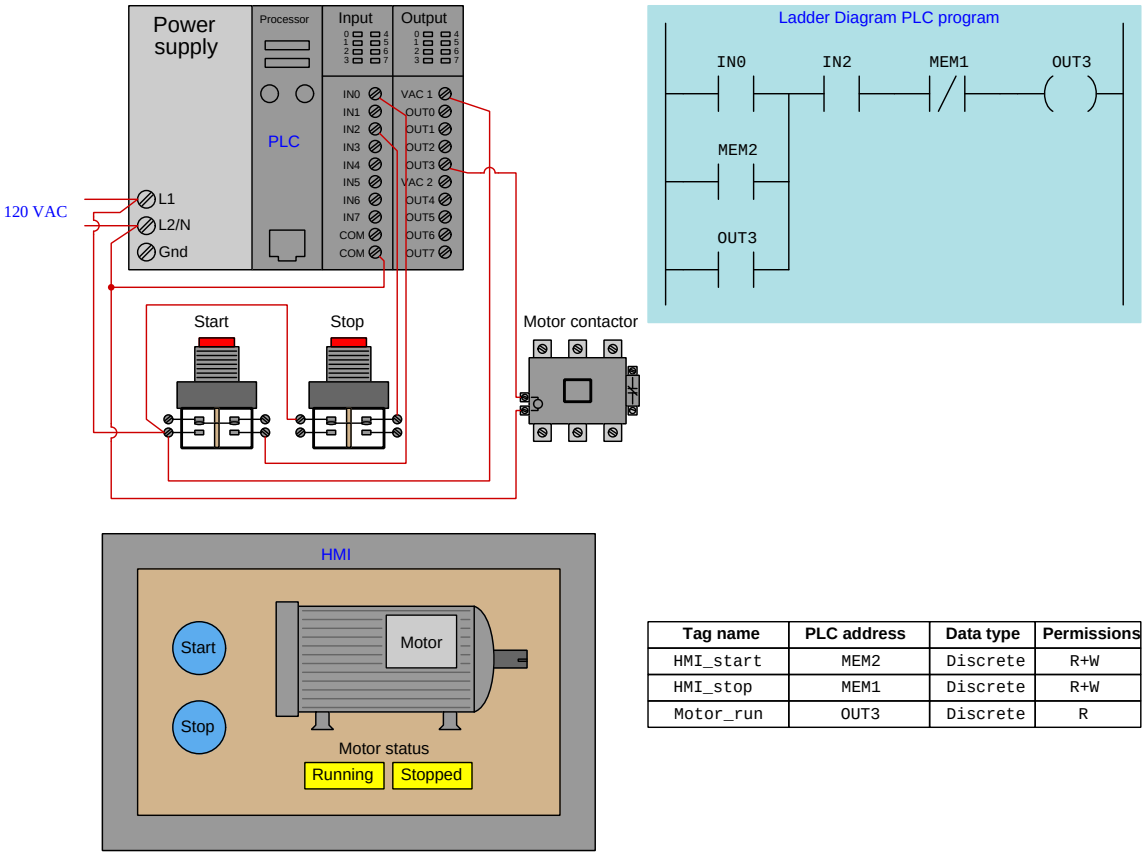
An example of an advanced HMI function is *recipe management*. The concept of a "recipe" is a parameter set corresponding to a particular desired outcome. For example, a PLC controlling a paint-blending machine would require a different recipe to mix red gloss-finish paint versus blue matte-finish paint versus orange flat-finish paint, each recipe specifying a different combination of ingredient doses, blending times, etc. If a PLC was controlling an automated baking machine, the term "recipe" would take on a very literal sense as the PLC would need to know how much of each ingredient to mix together, in what order, and at what temperatures and times to heat the mix. An HMI programmed with recipes will be able to send data to the PLC for all these parameters at the press of a single virtual button on the HMI's screen, one button per recipe. Having pre-programmed recipes residing within the HMI means the human operators pressing virtual buttons on the HMI need not know the details of each recipe. Recipes may be made read-only so that operations personnel cannot change a recipe arbitrarily and thereby compromise quality control.

3.5 Discrete (Boolean) tag programming

The most elementary data type within any computer, including all PLCs, is the *discrete* or *Boolean* type. This is a single bit of information. Such data types usually represent things like on/off status, pushbutton states, true/false conditions, etc. In an HMI, every virtual "indicator lamp" shown on the screen is controlled by the state of a discrete tag, which it reads from memory. Similarly, every virtual pushbutton or toggle switch on the screen of an HMI writes to a discrete tag in its memory.

[7] If the HMI is based on a personal computer platform (e.g. Rockwell RSView, Ignition, Wonderware, FIX/Intellution software), it may even be equipped with one or more hard disk drives for enormous amounts of historical data storage.

A good way to explore these concepts is by practical example. Consider the following PLC-based motor start/stop system using both physical pushbutton switches and virtual pushbuttons on the HMI to control the motor's state:



Tag name	PLC address	Data type	Permissions
HMI_start	MEM2	Discrete	R+W
HMI_stop	MEM1	Discrete	R+W
Motor_run	OUT3	Discrete	R

In this system two physical pushbuttons, one normally-open and the other normally-closed, connect to inputs `IN0` and `IN2` of the PLC's discrete input card, respectively. A single discrete output `OUT3` sends 120 VAC power to the coil of the motor contactor to start up the motor, the contactor's three-phase power conductors and motor omitted from the wiring diagram for simplicity. Inside the PLC is a ladder-diagram program controlling the discrete status of the `OUT3` bit based on not only the pushbutton states but also the states of the virtual pushbutton objects programmed into the HMI. Those two graphical "button" objects link to two tags in the HMI's database: `HMI_start` and `HMI_stop`, which in turn write to PLC bits `MEM2` and `MEM1`, respectively. The HMI also reads data from the PLC's `OUT3` bit so as to be aware of the motor's run state, this state given the name `Motor_run` in the HMI's tag name database. Two motor status indicators (virtual lamps) on the HMI's screen associate with this `Motor_run` tag, the "Running" indicator configured to illuminate when that tag is true and the "Stopped" indicator to illuminate when it is false. The graphical

depiction of the motor on the HMI's screen might also be configured to *animate* (e.g. display a rotating shaft) when the `Motor_run` bit is true.

A very important detail to note in this example is how the PLC program has distinctly separate contact instructions to read the physical pushbutton states versus the HMI's virtual pushbutton states. If the HMI's tag name database were configured to write the `HMI_start` and `HMI_stop` states to the PLC's `IN0` and `IN2` bits, a memory conflict would result. The PLC scans its input card channel states with every program cycle and writes those to the input register which includes bits `IN0` and `IN2`, and if some other device such as an HMI attempts to over-write those same bits in the PLC's memory one of two things would happen (depending on the design of the PLC): either the HMI's attempt to over-write the PLC input bits would fail in which case the HMI pushbutton controls would not work at all, or the over-write attempt would succeed and the physical pushbutton controls would not work at all. Neither situation is good. This is why the ladder-diagram PLC program needs to read the physical pushbuttons and the HMI's virtual pushbuttons as separate bits. Failing to respect this concept is a very common mistake made by new HMI/PLC programmers.

Another important detail here is the normally-open `IN2` stop contact instruction versus the normally-closed `MEM1` stop contact instruction in the PLC program, both intended to prevent energization of the PLC's `OUT3` coil instruction and thereby stop the motor. The `IN2` instruction is configured to be normally-open because the real physical "Stop" pushbutton switch is already wired as normally-closed, making the `IN2` instruction colored when no one presses the physical Stop pushbutton and un-colored when someone does. This is standard practice with motor stop controls in PLC programming, done so in order to stop the motor if ever the physical switch "stop" circuit suffers an open fault. In contrast, the `MEM1` instruction is normally-closed, necessitating the `HMI_stop` tag to be true inside the HMI to command the PLC to stop the motor. Unlike a real pushbutton switch where the probability of failing open is much greater than the probability of failing shorted, there is no such fault tendency within an HMI. If an HMI's network cable becomes severed from the PLC, it simply loses the ability to write *any* data to the PLC's memory at all, in which case the PLC memory simply retains the last states transmitted by the HMI. Therefore, when programming the HMI we might as well use whatever default configuration comes with each virtual pushbutton object – which is "true" when pressed – and this in turn necessitates a normally-closed `MEM1` contact instruction in the PLC program that will un-color whenever someone presses the "Stop" virtual pushbutton on the HMI screen.

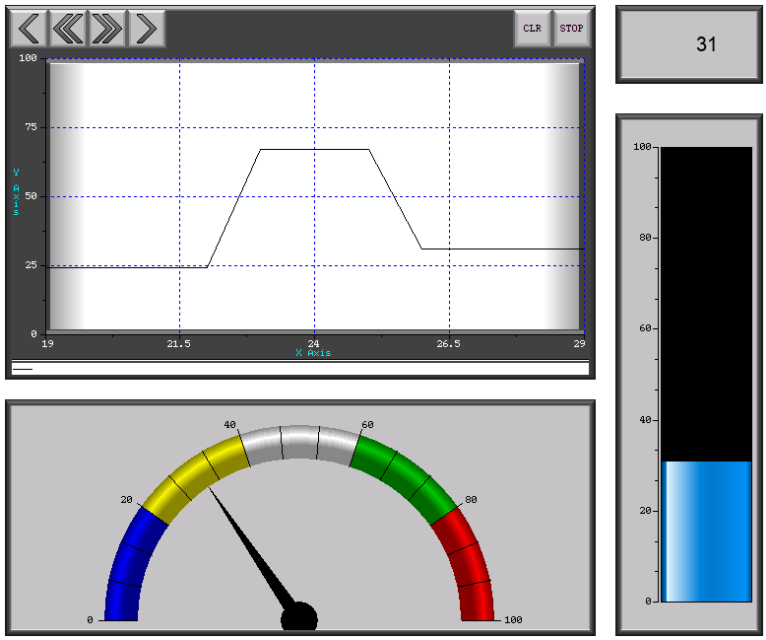
3.6 Integer tag programming

HMIs are also capable of displaying and setting the values of binary *integers*[8] in addition to discrete (Boolean) variables. The total count of filled bottles in a beverage processing facility is an example of an integer, because bottles only exist in whole-numbered quantities and never partial or fractional quantities. After a PLC counts the bottles, we may have an HMI networked to that PLC display the count value as a number on its screen, and perhaps have a numerical entry feature on the same HMI for a human operator to type in a certain target value for bottles filled which may be transmitted back to the PLC to have some influence over that conveyor belt or associated equipment.

HMI graphical objects capable of displaying integer number values include the following:

- Numerical displays showing actual digits
- Bargraph indicators
- "Gauge" style indicators
- Traces ("pens") on a trend graph

Examples of all four may be seen in the HMI screenshot below, each of the four indicator objects linked to the same integer tag:



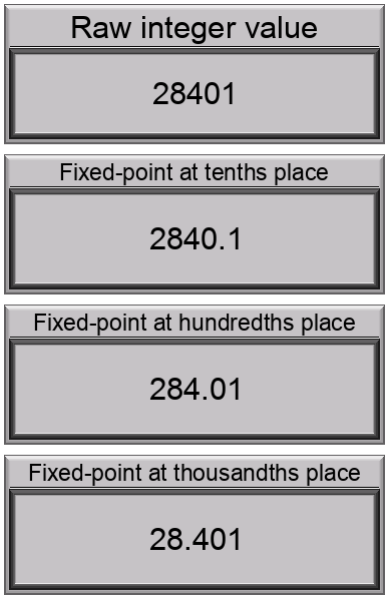
At the moment in time when this screenshot was captured the integer's value happened to be 31, but as we can see from the trend graph its value had previously been both greater than 31 and less than 31 at various times.

[8] Integers are whole numbers and their negative counterparts. For example, 9 would be an integer, and so would −15. However, 21.83 would not be an integer because it lies *between* the consecutive integer values of 21 and 22.

Integers are even useful for representing quantities that are not intrinsically whole-numbered. Take for example the air temperature of a room, which could very well be represented in integer degrees (e.g. 18, 19, 20, 21, 22 degrees Celsius) even though we know temperature can vary fractions of a degree, so long as we don't mind the imprecision of only resolving the temperature down to a limit of ± 1 degree C. Motor speed in revolutions per minute (RPM) is another common example of a parameter representable in integer form with the same caveat of ± 1 RPM resolution.

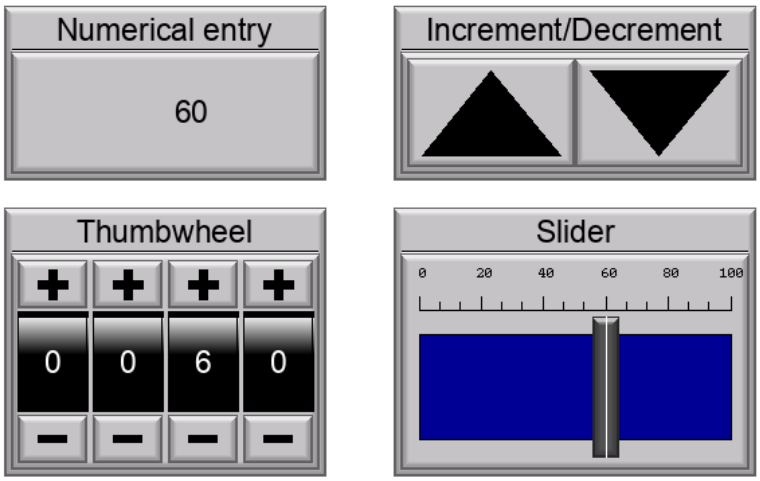
One reason for using integer numbers to represent non-integer quantities is that integers typically occupy less space within a computer's memory and require less processor cycles to compute than numbers with movable decimal points such as *floating-point* numbers (to be discussed in a later section). Another reason is that some digital devices can *only* represent numbers in integer form, lacking floating-point capability altogether. In applications where we either must or prefer to use integers to represent continuously-variable quantities, we may use a "trick" whereby we simply scale the integer value larger by some power of ten and then force a decimal point to appear on the HMI's screen where we wish it to be so that the quantity *looks* as though it resolves to tenths, hundredths, etc. of a unit. For example, we could program a PLC such that it scales a measured temperature of 22.73 degrees to an integer value of 2273, and then program the HMI to read that value as 2273 from the PLC but impose a decimal point between the middle of these four digits so it looks to the human viewer as 22.73 – this creative interpretation of integer values is known as *fixed-point* notation.

This next HMI screenshot shows different indicators linked to the same integer tag, but with different fixed-point decimal placements to make that integer appear as though it is able to represent non-integer values:



HMI graphical objects capable of manipulating integer number values include the following:

- Numerical entry controls
- "Slider" controls
- "Thumbwheel" controls
- Increment/Decrement pushbuttons



Pressing the numerical entry field produces a keypad where the desired integer number may be typed. Increment/Decrement buttons work by shifting the integer's value up or down by a pre-set amount with each press. Thumbwheel controls function as increment/decrement controls for each separate decimal digit. Slider controls have an analog feel to them, allowing the integer's value to be quickly set by finger position.

Integer quantities are more complicated than discrete (Boolean) quantities not only because they contain multiple bits, but also because there are multiple ways in which to regard those bits as representing integers. For example, a PLC may contain a 16-bit integer variable capable of representing any whole-numbered quantity between 0 and 65,535 if we assume *unsigned*[9] binary notation. However, if we interpret those same 16 bits as a *signed* binary number, the range becomes −32,768 to +32,767. A device might alternatively reserve 32 bits rather than 16 bits for integer values, again with the provision to be either signed or unsigned. Another possibility, though less popular in modern PLC/HMI systems, is something called *BCD* notation where every four binary bits within an integer directly represents one decimal digit.

A very important fact to bear in mind here is that the bits themselves do not suggest how they are to be interpreted. If you were to read one byte (eight bits) of memory inside a PLC and view `1001 0111` it would be impossible to tell from these bits alone whether this was an 8-bit unsigned binary number having a decimal value of 151, or an 8-bit signed binary number having a decimal value of −105, or a two-digit BCD number having a decimal value of 97, or just a portion of a 16-bit or 32-bit or even larger binary value!

This means you will be faced with options on how to interpret bits when populating an HMI's tag name database with integer tags, and in order to avoid serious interpretational errors you must configure each integer tag to be the same specific type of integer as assumed by the PLC. The PLC program editor software may or may not make these details clear to you, either, so it is advised to consult the manufacturer's documentation on that PLC to be sure.

To illustrate, the table below shows several different 16-bit binary numbers interpreted as unsigned, signed, and BCD for comparison:

Raw 16-bit binary word	Unsigned integer	Signed integer	BCD integer
`0000 0000 0000 0000`	0	0	0000
`0000 0000 0000 0001`	1	1	0001
`0000 0001 0011 0110`	310	310	0136
`0000 0011 1001 1010`	922	922	039?
`1111 1111 1111 1111`	65535	−1	????
`1111 1000 1001 0100`	63636	−1900	?894
`1000 1001 0100 0000`	35136	−30400	8940

Manually interpreting each of these 16-bit binary words is a good "active reading" exercise, and will help reinforce your understanding of how unsigned, signed, and BCD notation work. Note that the most significant bit (MSB) of a signed binary number has a negative place-weight (in this case with 16 bits, a place weight of −32768), and also note that only the binary combinations `0000` through `1001` have meaning as BCD-encoded decimal digits.

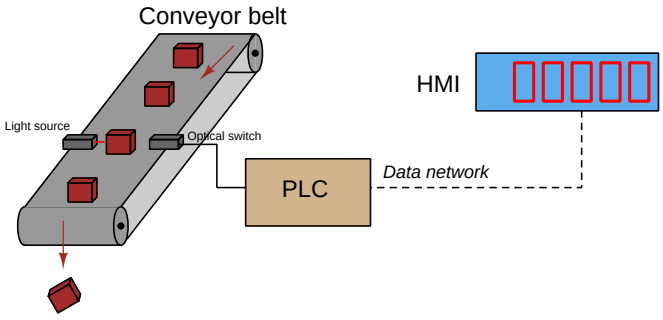
[9] Those familiar with the strict mathematical definition of "integer" may balk at the suggestion of an *unsigned integer*, as integers by definition include both positive and negative quantities, while whole numbers are positive only. However, in the world of industrial electronics "unsigned integer" and "signed integer" are the terms that seem to have become most popular.

3.7 Integer incompatibility

The mathematical definition of an *integer* is any number that either is a whole number (0, 1, 2, 3, 4 . . .) or a negative version of a whole number (−1, −2, −3, −4 . . .). In digital electronic systems where we use collections of binary bits to represent numerical quantities, multiple ways exist in which to express whole and integer numbers. First and foremost we have *unsigned integers* which are more properly known as whole numbers, expressed in binary form by the standard place-weighted format where each successive binary bit has a place-weight multiplier twice that of the previous bit. An alternative to the standard place-weighted binary scheme for unsigned integers is something called *Binary Coded Decimal* or *BCD* where groups of four bits each represent a single decimal digit (`0000` representing 0, `0001` representing 1, through `1001` representing 9). Next we have *signed integers* (or, as a mathematician might say, *true* integers) with the ability to express both positive and negative quantities. Most commonly we do this by assigning a negative place-weight value to the most-significant bit of a binary word, also known as *two's complement notation*. Finally, for each of these schemes we are faced with the limitation of the binary word width, where PLCs and HMIs default to 8 bits, 16 bits, 32 bits, or more depending on the "data type" reserved for a particular location in the device's memory.

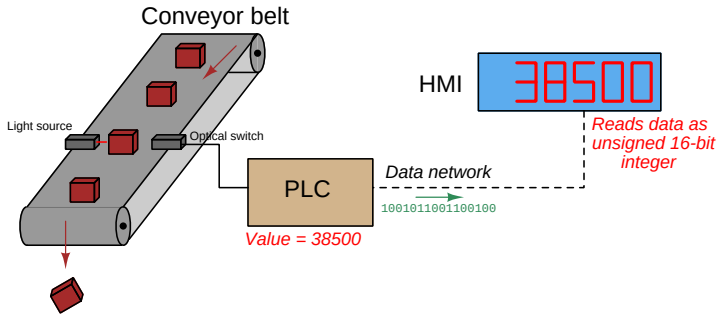
If a PLC is configured to store a numerical value using one data type, and an HMI is configured to read that same value but interpreting it as a different data type, interpretational errors will result. This section will not attempt to be comprehensive in describing every type of incompatibility, but 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, a PLC counts these pulses, and an HMI receives the total count value from the PLC via a digital network:

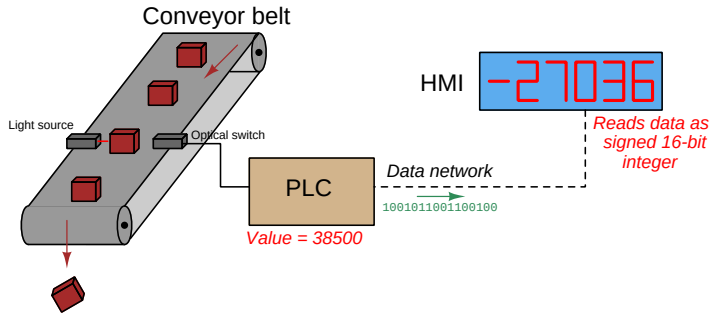


If we assume the counter instruction within the PLC keeps track of sensor pulses using a 16-bit unsigned integer count value, and exactly thirty-eight thousand five hundred items have passed by on the conveyor belt since the counter was reset, the PLC counter should register the binary equivalent of 38,500 in decimal. As a 16-bit unsigned binary integer this quantity will be `1001 0110 0110 0100` within the PLC's memory.

If the HMI receives this binary quantity as the 16-bit 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 configured the HMI to read this exact same 16-bit data as a *signed* 16-bit integer rather than the *unsigned* 16-bit integer it actually is coming from the PLC. The data received via the communication link is nothing but a stream of sixteen sequential bits, and so the receiving HMI cannot naturally "know" what sort of quantity these bits are supposed to represent. It is the programmer's responsibility to instruct the HMI 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).

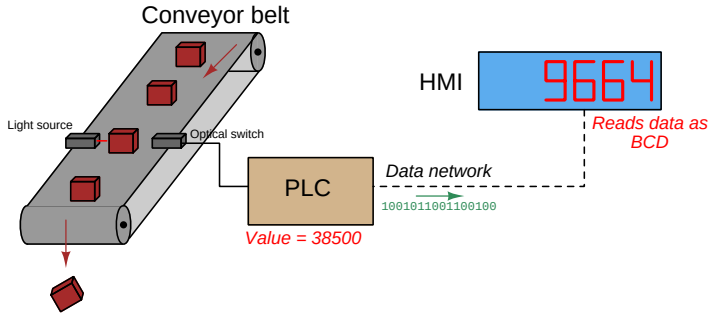


Not only will the improperly-configured HMI register an entirely non-sensical count value (i.e. a *negative* number of items!), but the problem won't appear for quite some time. Up to a count value of +32767 the MSB of this 16-bit binary number will be zero, and for that reason will be interpreted as a positive quantity whether the HMI regards it as signed or unsigned. Only *after* the count value exceeds 32767 (decimal) 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

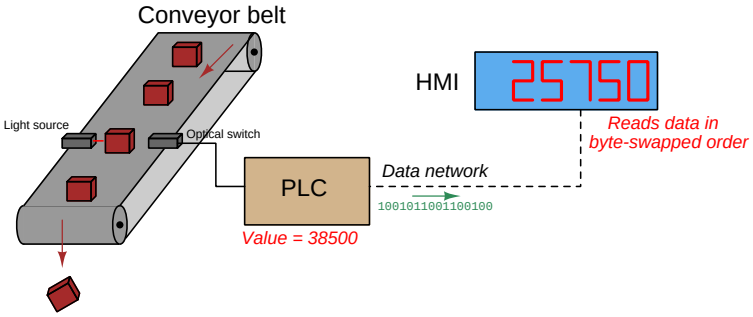
application. A full test of the system would have included at least a simulation of item count up to the maximum possible with an unsigned 16 bit binary number (65535 decimal) in order to verify no unusual problems lurked in the system.

Let us now assume a different configuration flaw, one where the HMI is incorrectly programmed to interpret the PLC counter's 16-bit unsigned integer count value as Binary Coded Decimal (*BCD*) rather than standard unsigned binary. Here the HMI will translate each grouping of four bits in the PLC's 16-bit word as a separate decimal digit, and so the data `1001 0110 0110 0100` will be interpreted as the decimal value 9664 as the following illustration shows:



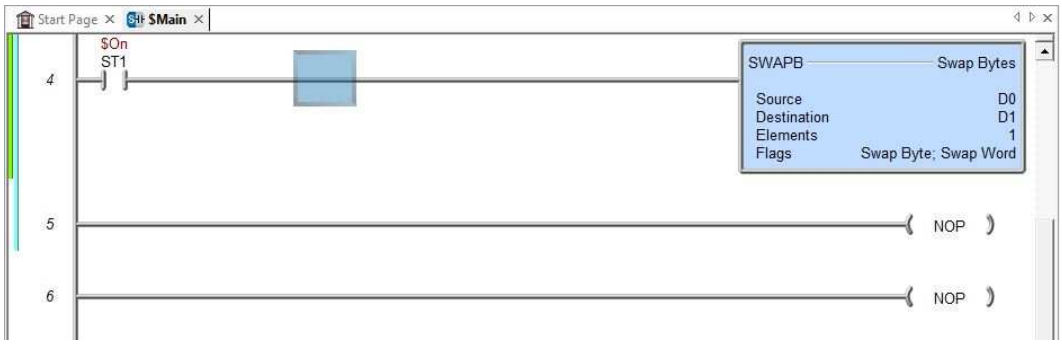
What is more, every time the binary count value happens to generate a grouping of four bits whose value exceeds nine, the HMI will display some sort of nonsensical character for that digit, since BCD expects only the codes `0000` through `1001` (i.e. 0 through 9) for each four-bit group. What exactly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

Yet another faulty scenario is one where the HMI assumes a different byte order than what is transmitted by the counter. In this case a binary count value of `1001 0110 0110 0100` would be interpreted as `0110 0100 1001 0110` and displayed as decimal 25750:



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 HMI to receive and display the data, but rather an inherent incompatibility between the PLC's 16-bit register and the HMI'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 PLC and HMI 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 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 binary or hexadecimal form. Examining the "raw" data rather than formatted decimal values eliminates one step of translation, allowing you to more easily discern patterns indicating data format. PLC program-editing software always provides tools[10] to view numerical data in raw form with little or no display formatting. 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.

[10] Rockwell Logix5000 PLC programming software provides a *Controller Tags* folder in which data values may be displayed. Legacy Rockwell (Allen-Bradley) PLC programming software for the PLC-5, SLC-500, and MicroLogix series of PLCs offer a *Data Files* view of stored data to the same effect. Siemens Step-7 (S7) programming software provides a *Status Chart* window for viewing data. Koyo DirectLogic and CLICK PLC programming software provides customizable *Data View* windows to do the same.

3.8 Floating-point (real) tag programming

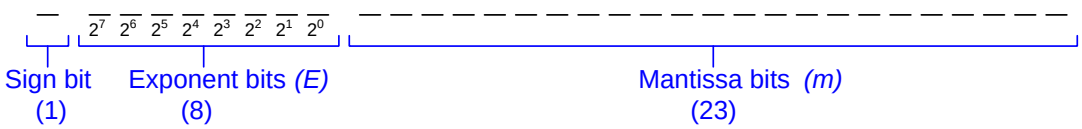
A *floating-point* or *real* number in a PLC or an HMI is a special form of binary representation where some of the bits specify the 1's and 0's making up the number as it would be written on a page, while other bits specify where a "binary point" (similar to a decimal point) will be located within those 1's and 0's. The following table shows a set of floating-point binary numbers and their decimal-equivalent values:

Power-of-two notation	Binary number with point	Decimal equivalent
1.0011×2^{5}	`100110.`	38
1.0011×2^{4}	`10011.`	19
1.0011×2^{3}	`1001.1`	9.5
1.0011×2^{2}	`100.11`	4.75
1.0011×2^{1}	`10.011`	2.375
1.0011×2^{0}	`1.0011`	1.1875
1.0011×2^{-1}	`.10011`	0.59375
1.0011×2^{-2}	`.010011`	0.296875

You, the reader, should take some time and manually translate these binary numbers into their equivalent decimal values as an active reading exercise. To do this, you must understand the place-weight values for each binary place, multiplied by two for each place to the left of the binary point (i.e. 1's place, 2's place, 4's place, etc.) and divided by two for each place to the right of the binary point (i.e. one-half's place, one-quarter's place, one-eighth's place, etc.). The binary values equivalent to 9.5 and 4.75 decimal are perhaps the easiest place to start in this table of values.

The ANSI/IEEE 754 standard specifies multiple methods for encoding floating-point numbers as a set of binary bits, the most popular by far within industrial control systems being the 32-bit "single-precision" version:

Single-precision (32-bit) IEEE floating-point number format



32-bit floating-point numbers are capable of representing numerical values over a tremendously wide range, from zero (0) to approximately 1.17549×10^{-38} as the smallest finite value all the way up to 3.40282×10^{38} as the largest finite value, and of course all their negative counterparts. Contrast this against a 32-bit unsigned integer which only ranges from zero (0) to 4,294,967,295 (i.e. 4.29497×10^{9}) and only in increments of $\pm$ 1.

This stark disparity in numerical range might prompt one to wonder why we don't represent all numerical values in floating-point notation, and dispense with integers altogether, given the superior range of floating-point and its ability to represent fractional values? The reason why integer binary notation is still valid is due to two major factors: the unique ability of an integer to *precisely* represent every whole number within its range (whereas floating-point may only *approximate* many whole numbers), and the relative simplicity of integer arithmetic compared with floating-point arithmetic. Of these two reasons for maintaining integer notation in digital systems, the first remains eternally valid while the second yields to technological advancement as modern microprocessors have become far more adept at floating-point arithmetic than microprocessors of old.

Simply put, **one should always use integer data types to represent things that are naturally whole-numbered, and use floating-point numbers for things that aren't**. Integers are perfect for any process or algorithm requiring discrete entities to be *counted*. This would include such things as counting discrete items passing by on a conveyor belt, tracking individual steps within a sequence of steps, etc. Floating-point numbers should be used for quantities known to *continuously vary* such as analog sensor values.

HMI graphical objects capable of displaying floating-point (real) number values are the same as for displaying integer values, including:

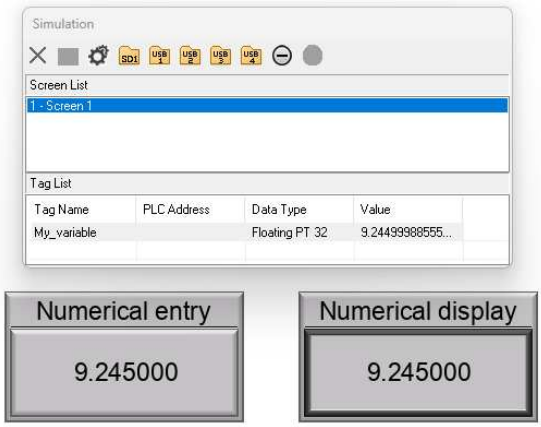
- Numerical displays showing actual digits
- Bargraph indicators
- "Gauge" style indicators
- Traces ("pens") on a trend graph

Similarly, HMI graphical objects capable of manipulating floating-point number are similar to those used to manipulate integers, including:

- Numerical entry controls
- "Slider" controls
- Increment/Decrement pushbuttons

A notable example of an integer-only HMI manipulator object is the *thumbwheel*, where a set of virtual wheels controls the value of each digit within a multi-digit decimal number. Since the ANSI/IEEE 754 binary format cannot precisely represent *every* possible decimal number, it would be disingenuous of an HMI to present the illusion of full control over each digit of a decimal number. Numerical entry controls where the user types in a decimal value digit-by-digit sometimes results in a form of rounding as that entered value gets converted to the closest floating-point binary equivalent.

An example of floating-point imprecision is seen below in a screenshot taken from an HMI simulation, where a single 32-bit floating-point tag named `My_variable` is linked to both a numerical entry object and a numerical display object, so that typing a number in the entry window results in the same number appearing on the display window:



In this particular demonstration, the number typed was 9.245. Not surprisingly, this is the same value appearing in the numerical display. However, what is a bit surprising is how the tag's value appears in the simulation's tag list: 2.4499988555 with ellipses suggesting even more trailing digits. The decimal number 9.245 is an example of just one value the ANSI/IEEE 754 32-bit standard cannot precisely represent. This imprecision is hidden from normal view on the HMI because the numerical display object in this case has been constrained to seven total decimal digits, but "behind the scenes" within the HMI's memory the true floating-point value happens to be slightly less.

This phenomenon is not unique to floating-point notation, and in fact we even experience this with familiar decimal numbers in the form of certain rational fractions being impossible to precisely represent by any decimal number having a finite digit count. Consider the fraction $\frac{1}{3}$ which is equivalent to 0.333333333 (with an infinite string of "3" digits), or the fraction $\frac{5}{7}$ which is approximately equal to 0.714285714285 (with the digit sequence *714285* repeating forever). Decimal numbers are based on power-of-ten place weights, just like binary being based on power-of-two place weights. This means any fractional quantity must be represented by some limited series of $\frac{1}{10^n}$ fractions for decimal and some limited series of $\frac{1}{2^n}$ fractions for binary, and for this simple reason some real-numbered quantities cannot ever be *exactly* represented by any fixed number of digits or bits. The more digits or bits we have at our disposal, the closer we can get to these unrepresentable quantities, but we can never *exactly* express them. In the HMI simulation example shown above, the fractional portion of the entered value (0.245) consists of $\frac{2}{10} + \frac{4}{100} + \frac{5}{1000}$, and this sum is simply unequal to *any* series of binary-weighted rational fractions (i.e. $\frac{n}{2} + \frac{m}{4} + \frac{p}{8}$, etc.) no matter how far we take that series with additional bits.

As a practical matter, floating-point number imprecision usually doesn't create any problem for us in terms of displayed numbers. As we saw in the HMI simulation, having a display of seven or fewer decimal digits rounds any 32-bit real number's value to a decimal equivalent we cannot distinguish from the original. Where this may lead to problems, however, is in HMI or PLC programming if we attempt to set up a conditional test for equality, expecting the floating-point number to exactly equal our test condition value at some point in time.

Consider the following calculation and test-for-equality performed in the Python programming language:

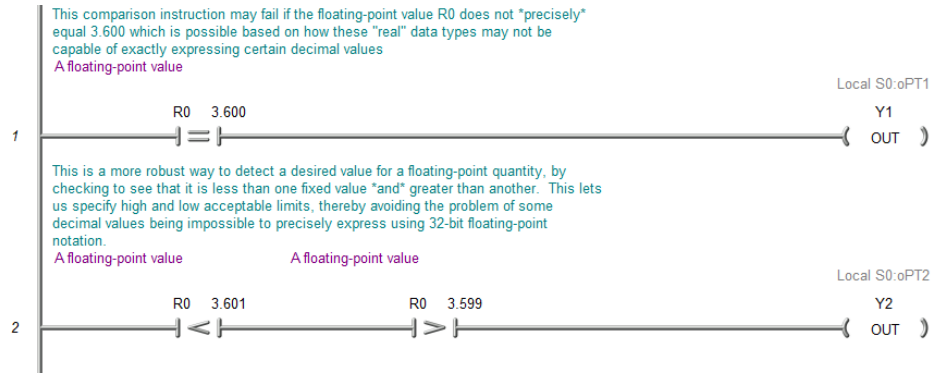
```
>>> 1.2 * 3
3.5999999999999996
>>> 1.2 * 3 == 3.6
False
```

We know as human beings accustomed to decimal notation that 1.2 times 3 should exactly equal 3.6, but the calculation `1.2 * 3` yields a "rounded" answer of `3.5999999999999996` due to the floating-point imprecision. Note the second Python command where we asked the computer to test `1.2 * 3` against `3.6` for equality, and it returned a result of *False* because it thought those two quantities were unequal! Whenever we need to reliably test a floating-point number for equality with some specified value, it is actually safer to perform a test to see if the number falls within a narrow range as shown below (once again using Python):

```
>>> (1.2 * 3 < 3.601) and (1.2 * 3 > 3.599))
True
```

Instead of having the computer test for an exact value match, we anticipate there possibly being "rounded" calculation results and so instead we have the computer perform a less-than test as well as a greater-than test for values just outside of what we want. In the above example the result of `1.2 * 3` is tested for being less than 3.601 and for being more than 3.599, the test passing when *both* conditions are met using the logical `and` function.

Performing such a bounded-value test within a ladder-diagram PLC program is fairly easy using two comparison instructions strung in series with each other such that *both* would have to evaluate as true in order to pass virtual power to the output coil. In the example shown below, the variable `R0` in the PLC's memory is a 32-bit floating-point number:



Chapter 4

Derivations and Technical References

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

4.1 Feature comparisons between PLC models

In most cases, similarities are far greater for different models of PLC than differences. However, differences do exist, and it is worth exploring the differences in basic features offered by an array of PLC models.

4.1.1 Viewing live values

- Allen-Bradley Logix 5000: the *Controller Tags* folder (typically on the left-hand pane of the programming window set) lists all the tag names defined for the PLC project, allowing you to view the real-time status of them all. Discrete inputs do not have specific input channel tag names, as tag names are user-defined in the Logix5000 PLC series.

- Allen-Bradley PLC-5, SLC 500, and MicroLogix: the *Data Files* listing (typically on the left-hand pane of the programming window set) lists all the data files within that PLC's memory. Opening a data file window allows you to view the real-time status of these data points. Discrete inputs are the `I` file addresses (e.g. `I:0/2`, `I:3/5`, etc.). The letter "`I`" represents "input," the first number represents the slot in which the input card is plugged, and the last number represents the bit within that data element (a 16-bit word) corresponding to the input card.

- Siemens S7-200: the *Status Chart* window allows the user to custom-configure a table showing the real-time values of multiple addresses within the PLC's memory. Discrete inputs are the `I` memory addresses (e.g. `I0.1`, `I1.5`, etc.).

- Koyo (Automation Direct) DirectLogic and CLICK: the *Data View* window allows the user to custom-configure a table showing the real-time values of multiple addresses within the PLC's memory. Discrete inputs are the `X` memory addresses (e.g. `X1`, `X2`, etc.).

4.1.2 Forcing live values

- Allen-Bradley Logix 5000: forces may be applied to specific tag names by right-clicking on the tag (in the program listing) and selecting the "Monitor" option. Discrete outputs do not have specific output channel tag names, as tag names are user-defined in the Logix5000 PLC series.
- Allen-Bradley PLC-5, SLC 500, and MicroLogix: the *Force Files* listing (typically on the left-hand pane of the programming window set) lists those data files within the PLC's memory liable to forcing by the user. Opening a force file window allows you to view and set the real-time status of these bits. Discrete outputs are the `O` file addresses (e.g. `O:0/7`, `O:6/2`, etc.). The letter "`O`" represents "output," the first number represents the slot in which the output card is plugged, and the last number represents the bit within that data element (a 16-bit word) corresponding to the output card.
- Siemens S7-200: the *Status Chart* window allows the user to custom-configure a table showing the real-time values of multiple addresses within the PLC's memory, and enabling the user to force the values of those addresses. Discrete outputs are the `Q` memory addresses (e.g. `Q0.4`, `Q1.2`, etc.).
- Koyo (Automation Direct) DirectLogic and CLICK: the *Override View* window allows the user to force variables within the PLC's memory. Discrete outputs are the `Y` memory addresses (e.g. `Y1`, `Y2`, etc.).

4.1.3 Single-bit addressing

- Allen-Bradley Logix 5000: a period serves as the bit delimiter. For example, `ThisRegister.0` is the least-significant bit (bit #0) of the multi-bit word `ThisRegister`.
- Allen-Bradley PLC-5, SLC 500, and MicroLogix: a forward-slash symbol serves as the bit delimiter. For example, `N7:1/15` is the most-significant bit (bit #15) the 16-bit word `N7:1`.
- Siemens S7-200: a period serves as the bit delimiter. For example, `SM0.5` is the #5 bit of the word `SM0`.

4.1.4 Special "system" values

Every PLC has special registers holding data relevant to its operation, such as error flags, processor scan time, etc.

- Allen-Bradley Logix 5000: various "system" values are accessed via `GSV` (Get System Value) and `SSV` (Save System Value) instructions.
- Allen-Bradley PLC-5, SLC 500, and MicroLogix: the `Data Files` listing (typically on the left-hand pane of the programming window set) shows file number 2 as the "Status" file, in which you will find various system-related bits and registers.
- Siemens S7-200: the *Special Memory* registers contain various system-related bits and registers. These are the `SM` memory addresses (e.g. `SM0.1`, `SMB8`, `SMW22`, etc.).
- Koyo (Automation Direct) DirectLogic and CLICK: the *Special* registers contain various system-related bits and registers. These are the `SP` memory addresses (e.g. `SP1`, `SP2`, `SP3`, etc.) in the DirectLogic PLC series, and the `SC` and `SD` memory addresses in the CLICK PLC series.

4.1.5 Free-running clock pulses

- Allen-Bradley SLC 500: status bit `S:4/0` is a free-running clock pulse with a period of 20 milliseconds, which may be used to clock a counter instruction up to 50 to make a 1-second pulse (because 50 times 20 ms = 1000 ms = 1 second).
- Siemens S7-200: Special Memory bit `SM0.5` is a free-running clock pulse with a period of 1 second.
- Koyo (Automation Direct) DirectLogic: Special relay `SP4` is a free-running clock pulse with a period of 1 second.

4.1.6 Standard counter instructions

- Allen-Bradley Logix 5000: `CTU` count-up, `CTD` count-down, and `CTUD` count-up/down instructions. A `RES` reset instruction exists to reset the accumulator value of the specified counter.
- Allen-Bradley SLC 500: `CTU` and `CTD` instructions. A `RES` reset instruction exists to reset the accumulator value of the specified counter.
- Siemens S7-200: `CTU` count-up, `CTD` count-down, and `CTUD` count-up/down instructions.
- Koyo (Automation Direct) DirectLogic: `UDC` counter instruction.

4.1.7 High-speed counter instructions

- Allen-Bradley SLC 500: `HSU` high-speed count-up instruction.
- Siemens S7-200: `HSC` high-speed counter instruction, used in conjunction with the `HDEF` high-speed counter definition instruction.

4.1.8 Timer instructions

- Allen-Bradley Logix 5000: `TOF` off-delay timer, `TON` on-delay timer, `RTO` retentive on-delay timer, `TOFR` off-delay timer with reset, `TONR` on-delay timer with reset, and `RTOR` retentive on-delay timer with reset instructions.
- Allen-Bradley SLC 500: `TOF` off-delay timer, `TON` on-delay timer, and `RTO` retentive on-delay timer instructions.
- Siemens S7-200: `TOF` off-delay timer, `TON` on-delay timer, and `TONR` retentive on-delay timer instructions.

4.1.9 ASCII text message instructions

- Allen-Bradley Logix 5000: the "ASCII Write" instructions `AWT` and `AWA` may be used to do this. The "ASCII Write Append" instruction (`AWA`) is convenient to use because it may be programmed to automatically insert linefeed and carriage-return commands at the end of a message string.
- Allen-Bradley SLC 500: the "ASCII Write" instructions `AWT` and `AWA` may be used to do this. The "ASCII Write Append" instruction (`AWA`) is convenient to use because it may be programmed to automatically insert linefeed and carriage-return commands at the end of a message string.
- Siemens S7-200: the "Transmit" instruction (`XMT`) is useful for this task when used in Freeport mode.
- Koyo (Automation Direct) DirectLogic: the "Print Message" instruction (`PRINT`) is useful for this task.

4.1.10 Analog signal scaling

- Allen-Bradley Logix 5000: the I/O configuration menu (specifically, the *Module Properties* window) allows you to directly and easily scale analog input signal ranges into any arbitrary numerical range desired. Floating-point ("REAL") format is standard, but integer format may be chosen for faster processing of the analog signal.
- Allen-Bradley PLC-5, SLC 500, and MicroLogix: raw analog input values are 16-bit signed integers. The `SCL` and `SCP` instructions are custom-made for scaling these raw integer ADC count values into ranges of your choosing.
- Siemens S7-200: raw analog input values are 16-bit signed integers. Interestingly, the S7-200 PLC provides built-in potentiometers assigned to special word registers (`SMB28` and `SMB29`) with an 8-bit (0-255 count) range. These values may be used for any suitable purpose, including combination with the raw analog input register values in order to provide mechanical calibration adjustments for the analog input(s).
- Koyo (Automation Direct) DirectLogic: you must use standard math instructions (e.g. `ADD`, `MUL`) to implement a $y = mx + b$ linear equation for scaling purposes.
- Koyo (Automation Direct) CLICK: the I/O configuration menu allows you to directly and easily scale analog input signal ranges into any arbitrary numerical range desired.

4.2 Legacy Allen-Bradley memory maps and I/O addressing

A wise PLC programmer once told me that the first thing any aspiring programmer should learn about the PLC they intend to program is how the digital memory of that PLC is organized. This is sage advice for any programmer, especially on systems where memory is limited, and/or where I/O has a fixed association with certain locations in the system's memory. Virtually every microprocessor-based control system comes with a published *memory map* showing the organization of its limited memory: how much is available for certain functions, which addresses are linked to which I/O points, how different locations in memory are to be referenced by the programmer.

Discrete input and output channels on a PLC correspond to individual *bits* in the PLC's memory array. Similarly, analog input and output channels on a PLC correspond to multi-bit *words* (contiguous blocks of bits) in the PLC's memory. The association between I/O points and memory locations is by no means standardized between different PLC manufacturers, or even between different PLC models designed by the same manufacturer. This makes it difficult to write a general tutorial on PLC addressing, and so my ultimate advice is to consult the engineering references for the PLC system you intend to program.

The most common brand of PLC in use in the United States at the time of this writing (2019) is Allen-Bradley (Rockwell), and a great many of these Allen-Bradley PLCs still in service happen to use a unique form of I/O addressing[1] students tend to find confusing.

[1] The most modern Allen-Bradley PLCs have all but done away with fixed-location I/O addressing, opting instead for *tag name* based I/O addressing. However, enough legacy Allen-Bradley PLC systems still exist in industry to warrant coverage of these addressing conventions.

The following table shows a partial memory map for an Allen-Bradley SLC 500 PLC[2]:

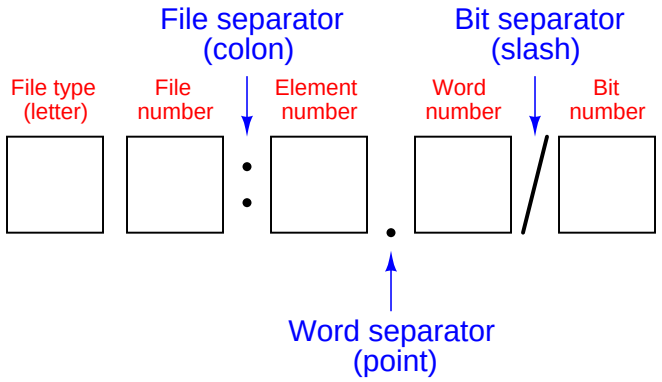
File number	File type	Logical address range
0	Output image	`O:0` to `O:30`
1	Input image	`I:0` to `I:30`
2	Status	`S:0` to `S:`n
3	Binary	`B3:0` to `B3:255`
4	Timers	`T4:0` to `T4:255`
5	Counters	`C5:0` to `C5:255`
6	Control	`R6:0` to `R6:255`
7	Integer	`N7:0` to `N7:255`
8	Floating-point	`F8:0` to `F8:255`
9	Network	`x9:0` to `x9:255`
10 through 255	User defined	`x10:0` to `x255:255`

Note that Allen-Bradley's use of the word "file" differs from personal computer parlance. In the SLC 500 controller, a "file" is a block of random-access memory used to store a particular type of data. By contrast, a "file" in a personal computer is a contiguous collection of data bits with collective meaning (e.g. a word processing file or a spreadsheet file), usually stored on the computer's hard disk drive. Within each of the Allen-Bradley PLC's "files" are multiple "elements," each element consisting of a set of bits (8, 16, 24, or 32) representing data. Elements are addressed by number following the colon after the file designator, and individual bits within each element addressed by a number following a slash mark. For example, the first bit (bit 0) of the second element in file 3 (Binary) would be addressed as `B3:2/0`.

In Allen-Bradley PLCs such as the SLC 500 and PLC-5 models, files 0, 1, and 2 are exclusively reserved for discrete outputs, discrete inputs, and status bits, respectively. Thus, the letter designators O, I, and S (file types) are redundant to the numbers 0, 1, and 2 (file numbers). Other file types such as B (binary), T (timers), C (counters), and others have their own default file numbers (3, 4, and 5, respectively), but may also be used in some of the user-defined file numbers (10 and above). For example, file 7 in an Allen-Bradley controller is reserved for data of the "integer" type (N), but integer data may also be stored in any file numbered 10 or greater at the user's discretion. Thus, file numbers and file type letters for data types other than output (O), input (I), and status (S) always appear together. You would not typically see an integer word addressed as `N:30` (integer word 30 in the PLC's memory) for example, but rather as `N7:30` (integer word 30 *in file 7* of the PLC's memory) to distinguish it from other integer word 30's that may exist in other files of the PLC's memory.

[2]Also called the *data table*, this map shows the addressing of memory areas reserved for programs entered by the user. Other areas of memory exist within the SLC 500 processor, but these other areas are inaccessible to the technician writing PLC programs.

This file-based addressing notation bears further explanation. When an address appears in a PLC program, special characters are used to separate (or "delimit") different fields from each other. The general scheme for Allen-Bradley SLC 500 PLCs is shown here:



Not all file types need to distinguish individual words and bits. Integer files (N), for example, consist of one 16-bit word for each element. For instance, `N7:5` would be the 16-bit integer word number five held in file seven. A discrete input file type (I), though, needs to be addressed as individual bits because each separate I/O point refers to a single bit. Thus, `I:3/7` would be bit number seven residing in input element three. The "slash" symbol is necessary when addressing discrete I/O bits because we do not wish to refer to all sixteen bits in a word when we just mean a single input or output point on the PLC. Integer numbers, by contrast, are collections of 16 bits each in the SLC 500 memory map, and so are usually addressed as entire words rather than bit-by-bit[3].

Certain file types such as timers are more complex. Each timer "element[4]" consists of *two* different 16-bit words (one for the timer's accumulated value, the other for the timer's target value) in addition to no less than *three* bits declaring the status of the timer (an "Enabled" bit, a "Timing" bit, and a "Done" bit). Thus, we must make use of both the decimal-point and slash separator symbols when referring to data within a timer. Suppose we declared a timer in our PLC program with the address `T4:2`, which would be timer number two contained in timer file four. If we wished to address that timer's current value, we would do so as `T4:2.ACC` (the "Accumulator" word of timer number two in file four). The "Done" bit of that same timer would be addressed as `T4:2/DN` (the "Done" bit of timer number two in file four)[5].

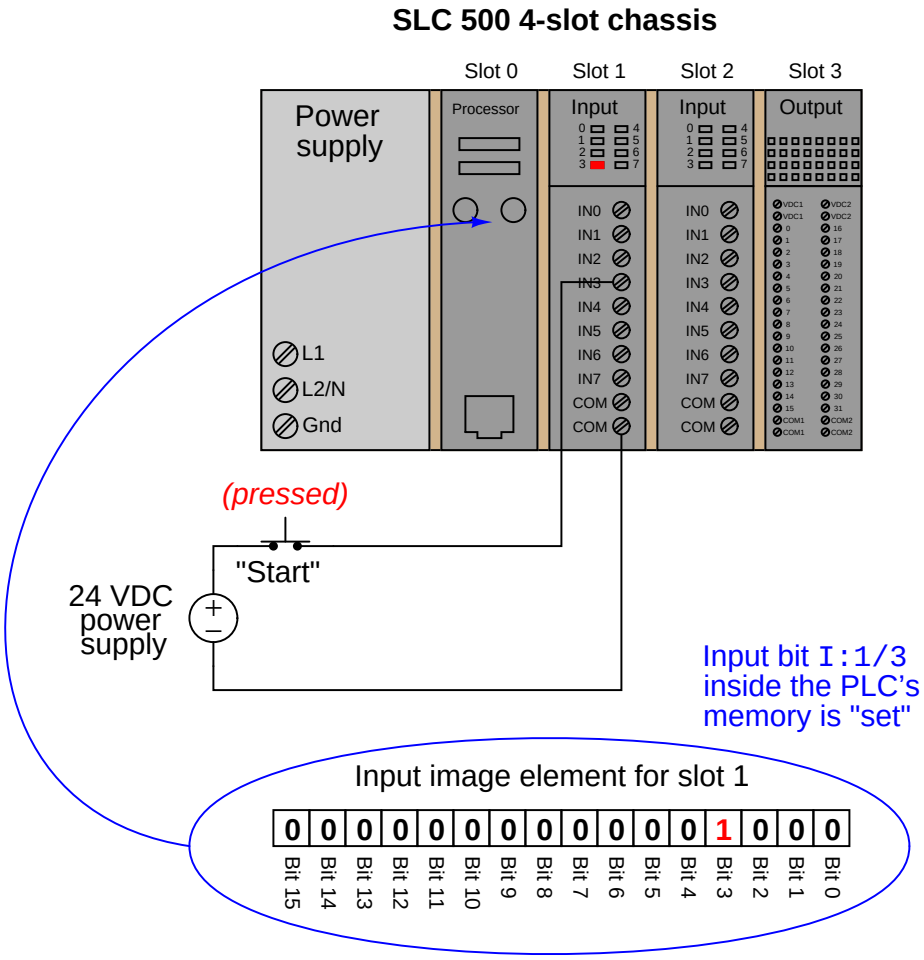
[3] This is not to say one *cannot* specify a particular bit in an otherwise whole word. In fact, this is one of the powerful advantages of Allen-Bradley's addressing scheme: it gives you the ability to precisely specify portions of data, even if that data is not generally intended to be portioned into smaller pieces!

[4] Programmers familiar with languages such as C and C++ might refer to an Allen-Bradley "element" as a *data structure*, each type with a set configuration of words and/or bits.

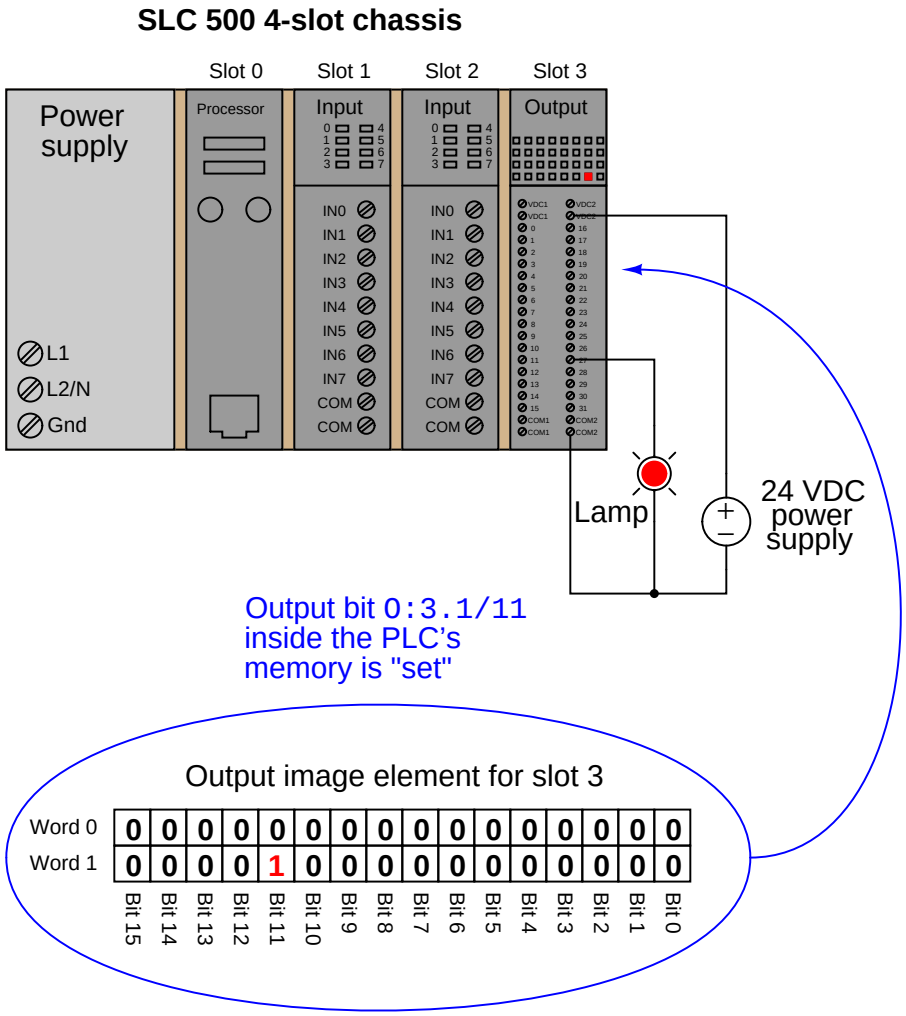
[5] Referencing the Allen-Bradley engineering literature, we see that the accumulator word may alternatively be addressed by number rather than by mnemonic, `T4:2.2` (word 2 being the accumulator word in the timer data structure), and that the "done" bit may be alternatively addressed as `T4:2.0/13` (bit number 13 in word 0 of the timer's data structure). The mnemonics provided by Allen-Bradley are certainly less confusing than referencing word and bit numbers for particular aspects of a timer's function!

A hallmark of the SLC 500's addressing scheme common to many legacy PLC systems is that the address labels for input and output bits explicitly reference the physical locations of the I/O channels. For instance, if an 8-channel discrete input card were plugged into slot 4 of an Allen-Bradley SLC 500 PLC, and you wished to specify the second bit (bit 1 out of a 0 to 7 range), you would address it with the following label: `I:4/1`. Addressing the seventh bit (bit number 6) on a discrete output card plugged into slot 3 would require the label `O:3/6`. In either case, the numerical structure of that label tells you exactly where the real-world input signal connects to the PLC.

To illustrate the relationship between physical I/O and bits in the PLC's memory, consider this example of an Allen-Bradley SLC 500 PLC, showing one of its discrete input channels energized (the switch being used as a "Start" switch for an electric motor):



If an input or output card possesses more than 16 bits – as in the case of the 32-bit discrete output card shown in slot 3 of the example SLC 500 rack – the addressing scheme further subdivides each element into *words* and bits (each "word" being 16 bits in length). Thus, the address for bit number 27 of a 32-bit input module plugged into slot 3 would be `I:3.1/11` (since bit 27 is equivalent to bit 11 of word 1 – word 0 addressing bits 0 through 15 and word 1 addressing bits 16 through 31):



A close-up photograph of a 32-bit DC input card for an Allen-Bradley SLC 500 PLC system shows this multi-word addressing:

The first sixteen input points on this card (the left-hand LED group numbered 0 through 15) are addressed `I:X.0/0` through `I:X.0/15`, with "`X`" referring to the slot number the card is plugged into. The next sixteen input points (the right-hand LED group numbered 16 through 31) are addressed `I:X.1/0` through `I:X.1/15`.

Legacy PLC systems typically reference each one of the I/O channels by labels such as "I:1/3" (or equivalent[6]) indicating the actual location of the input channel terminal on the PLC unit. The IEC 61131-3 programming standard refers to this channel-based addressing of I/O data points as *direct addressing*. A synonym for direct addressing is *absolute addressing*.

Addressing I/O bits directly by their card, slot, and/or terminal labels may seem simple and elegant, but it becomes very cumbersome for large PLC systems and complex programs. Every time a technician or programmer views the program, they must "translate" each of these I/O labels to some real-world device (e.g. "Input `I:1/3` is actually the *Start* pushbutton for the middle tank mixer motor") in order to understand the function of that bit. A later effort to enhance the clarity of PLC programming was the concept of addressing variables in a PLC's memory by arbitrary names rather than fixed codes. The IEC 61131-3 programming standard refers to this as *symbolic addressing* in contrast to "direct" (channel-based) addressing, allowing programmers arbitrarily

[6] Some systems such as the Texas Instruments 505 series used "X" labels to indicate discrete input channels and "Y" labels to indicate discrete output channels (e.g. input `X9` and output `Y14`). This same labeling convention is still used by Koyo in its DirectLogic and "CLICK" PLC models. Siemens continues a similar tradition of I/O addressing by using the letter "I" to indicate discrete inputs and the letter "Q" to indicate discrete outputs (e.g. input channel `I0.5` and output `Q4.1`).

name I/O channels in ways that are meaningful to the system as a whole. To use our simple motor "Start" switch example, it is now possible for the programmer to designate input `I:1/3` (an example of a *direct address*) as "`Motor_start_switch`" (an example of a *symbolic address*) within the program, thus greatly enhancing the readability of the PLC program. Initial implementations of this concept maintained direct addresses for I/O data points, with symbolic names appearing as supplements to the absolute addresses.

The modern trend in PLC addressing is to avoid the use of direct addresses such as `I:1/3` altogether, so they do not appear anywhere in the programming code. The Allen-Bradley "Logix" series of programmable logic controllers is the most prominent example of this new convention at the time of this writing. Each I/O point, regardless of type or physical location, is assigned a *tag name* which is meaningful in a real-world sense, and these tag names (or *symbols* as they are alternatively called) are referenced to absolute I/O channel locations by a database file. An important requirement of tag names is that they contain no space characters between words (e.g. instead of "`Motor start switch`", a tag name should use hyphens or underscore marks as spacing characters: "`Motor_start_switch`"), since spaces are generally assumed by computer programming languages to be delimiters (separators between different variables).

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 Displaying different integer types using C++

This program prompts the user to enter an integer value compatible with a signed 16-bit format, then displays that number in raw binary form, in unsigned 16-bit integer form, and in signed 16-bit integer form:

```
#include <iostream>
#include <bitset>

using namespace std;

int main (void)
{
  union
  {
    int16_t s16bit;
    uint16_t u16bit;
  } n;

  while(1)
  {
    cout << "Enter an integer value between -32768 and +32767: ";
    cin >> n.s16bit;
    cout << endl;

    cout << "Represented as raw binary = 0b" << bitset<16>(n.s16bit) << endl;
    cout << "Represented as 16-bit unsigned integer = " << n.u16bit << endl;
    cout << "Represented as 16-bit signed integer = " << n.s16bit << endl;
    cout << endl;
  }

  return 0;
}
```

5.4 Numeration formats in Python and C++

Here 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.

Energy

Conservation of Energy

Simplification as a problem-solving strategy

Thought experiments as a problem-solving strategy

Limiting cases as a problem-solving strategy

Annotating diagrams as a problem-solving strategy

Interpreting intermediate results as a problem-solving strategy

Graphing as a problem-solving strategy

Converting a qualitative problem into a quantitative problem

Converting a quantitative problem into a qualitative problem

Working "backwards" to validate calculated results

Reductio ad absurdum

Re-drawing schematics as a problem-solving strategy

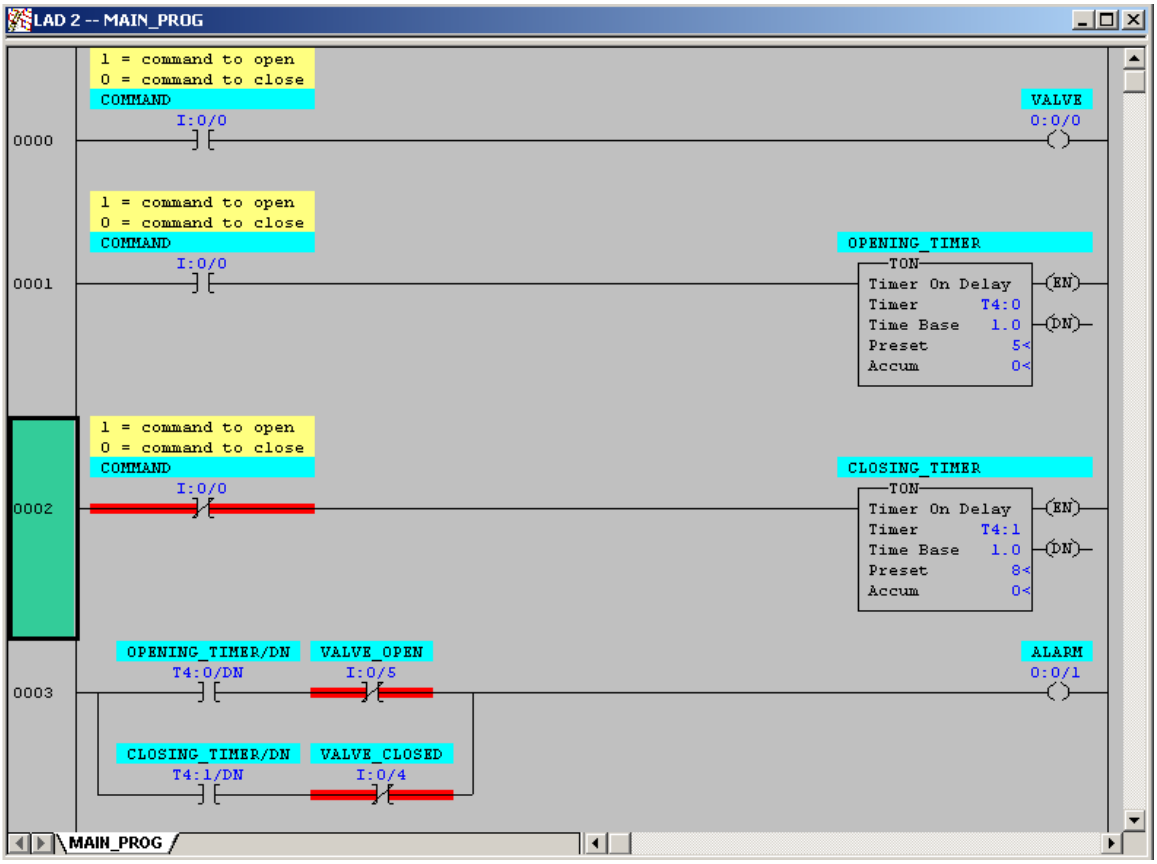
Cut-and-try problem-solving strategy

Algebraic substitution

???

6.1.3 Valve motion timer

An Allen-Bradley (Rockwell) MicroLogix connects to an electrically-actuated valve controlling the flow of oil through a pipeline. The purpose of this PLC program is to check to see that the large valve moves with sufficient speed for proper operation of the pipeline. If the valve takes too long to open, or too long to close, the PLC program will activate an alarm:



Based on your analysis of this program, determine the necessary connections for each limit switch monitoring the door's position:

- `VALVE_OPEN` limit switch: *normally-open* (NO) or *normally-closed* (NC)?
- `VALVE_CLOSED` limit switch: *normally-open* (NO) or *normally-closed* (NC)?

Furthermore, determine the energization status of the PLC output controlling the valve actuator:

- `VALVE` discrete output channel: *energize* to open the valve or *de-energize* to open the valve?

Suppose we wanted an HMI to display the last time-to-open and the last time-to-close for this valve, as maintenance personnel have learned these measurements are a good indicator of friction build-up within the valve mechanism deserving of attention. Explain how you could program an HMI and (possibly) add to this PLC's program to provide such recorded measurements.

Furthermore, suppose we wished the same HMI to express the last opening and closing times in *percentages* of manufacturer ratings, the valve manufacturer specifying a normal opening time of 3 seconds and a normal closing time of 5 seconds. Explain how the PLC and HMI be programmed to provide these percentage figures.

Challenges

- Are floating-point values necessary for implementing any of the new features?

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	

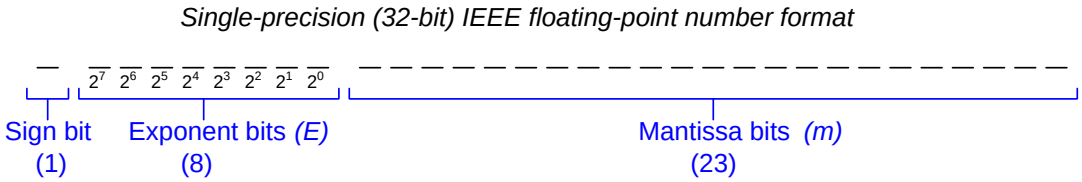
Note how the square-root term (y) is calculated in cell C1, and the denominator term (z) in cell C2. This makes the two final formulae (in cells B1 and B2) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary[10] – all that matters is that they properly reference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

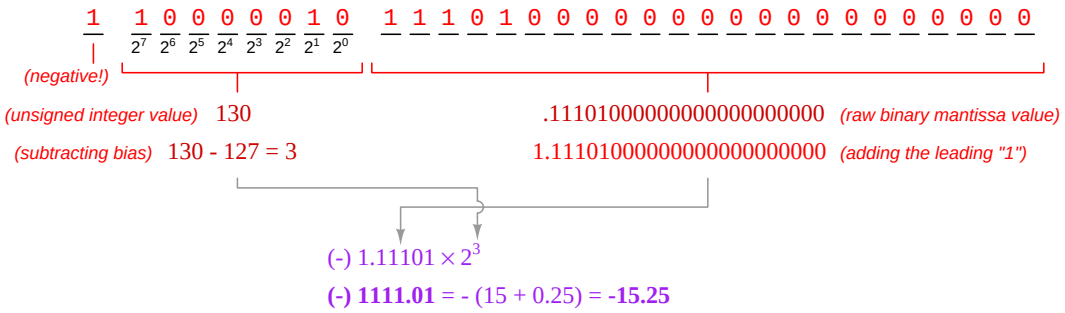
6.2.3 Decoding 32-bit real numbers

The ANSI/IEEE 754 standard defines a very popular method of representing floating-point ("real") numbers as 32-bit words, following this format where m is the *mantissa* and E is the *exponent*:



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

The exponent in this standard has a bias of -127 which means the eight exponent bits are to be interpreted as a standard unsigned-binary integer, from which 127 is subtracted to obtain the actual power-of-two exponent for the floating-point number. For example, the number -15.25 would be represented as `1100 0001 0111 0100 0000 0000 0000 0000` in 32-bit ANSI/IEEE 754 form:



Next, you will be challenged to similarly decode multiple 32-bit words representing ANSI/IEEE 754 floating-point numbers.

- `0100 0010 0110 0000 0000 0000 0000 0000`
- `1100 0011 0011 0111 1000 0000 0000 0000`
- `0100 1111 1000 1000 1001 0101 1000 0100`
- `1011 1011 0001 1011 0101 0010 0000 0000`

Challenges

- Identify an application for floating-point numbers that an integer could not fulfill.

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.

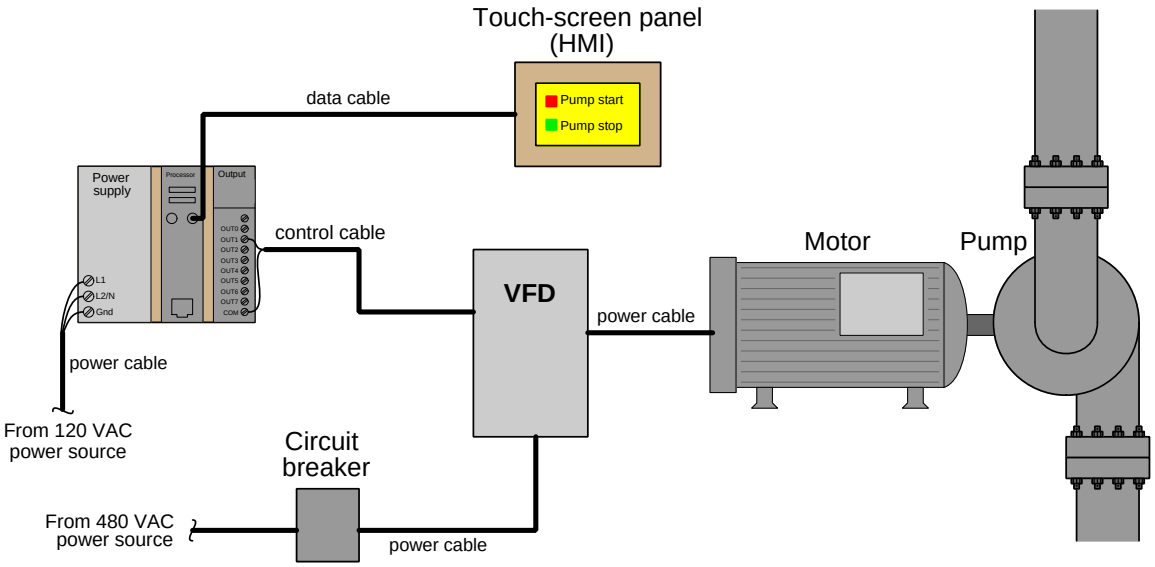
As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of 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 Possible faults in a PLC/HMI pump control system

A large water pump at a wastewater treatment facility is speed-controlled by a VFD, receiving commands from a PLC/HMI control system:



This system is newly constructed, and when the operators try starting up the pump by pressing the "Pump start" icon on the touch-screen, nothing happens. A technician temporarily connects a jumper wire across the two terminals at the VFD where the control cable lands. At this, the motor starts up and runs.

Identify the likelihood of each specified fault for this circuit. Consider each fault one at a time (i.e. no coincidental faults), determining whether or not each fault is compatible with *all* measurements and symptoms in this circuit.

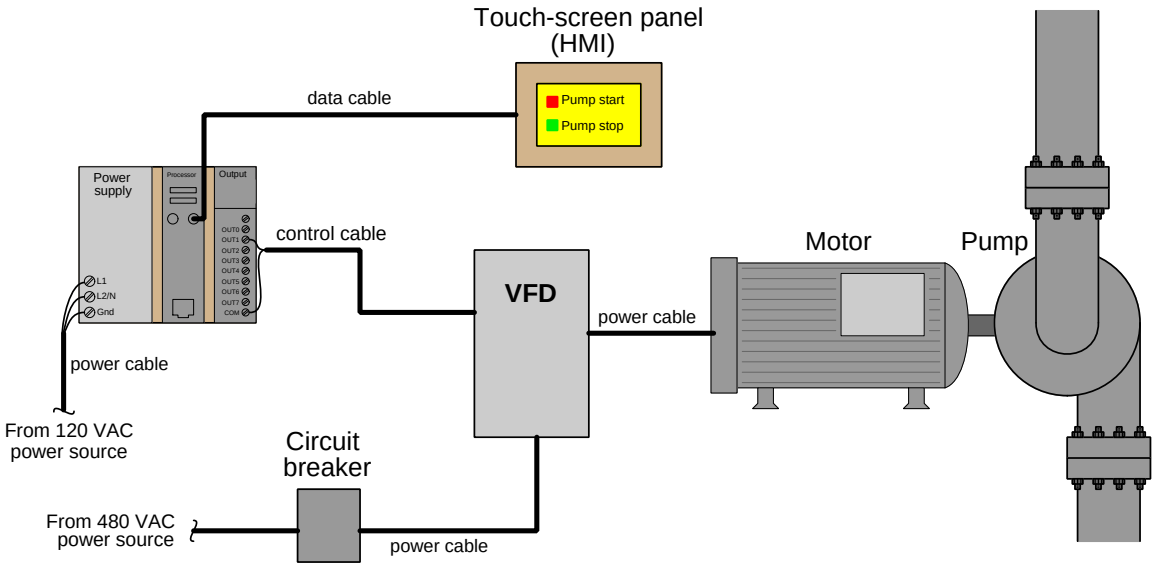
- Circuit breaker off
- Touch-screen panel malfunctioning
- Programming error in PLC
- Faulted power cable between VFD and motor
- Faulted power cable between breaker and VFD
- PLC output card malfunctioning
- Open control cable
- Shorted control cable
- Open data cable

Challenges

- Identify the *next* diagnostic test or measurement you would make on this system. Explain how the result(s) of this next test or measurement help further identify the location and/or nature of the fault.

6.3.2 Diagnostic tests on a failed PLC/HMI pump control system

A large water pump at a wastewater treatment facility is speed-controlled by a VFD, receiving commands from a PLC/HMI control system:



This system is newly constructed, and when the operators try starting up the pump by pressing the "Pump start" icon on the touch-screen, nothing happens. You happen to notice a glowing "Power" indicator LED on the VFD, but none of the warning LEDs on the VFD are lit.

Assess whether each of the following diagnostic tests would be useful on this failed system:

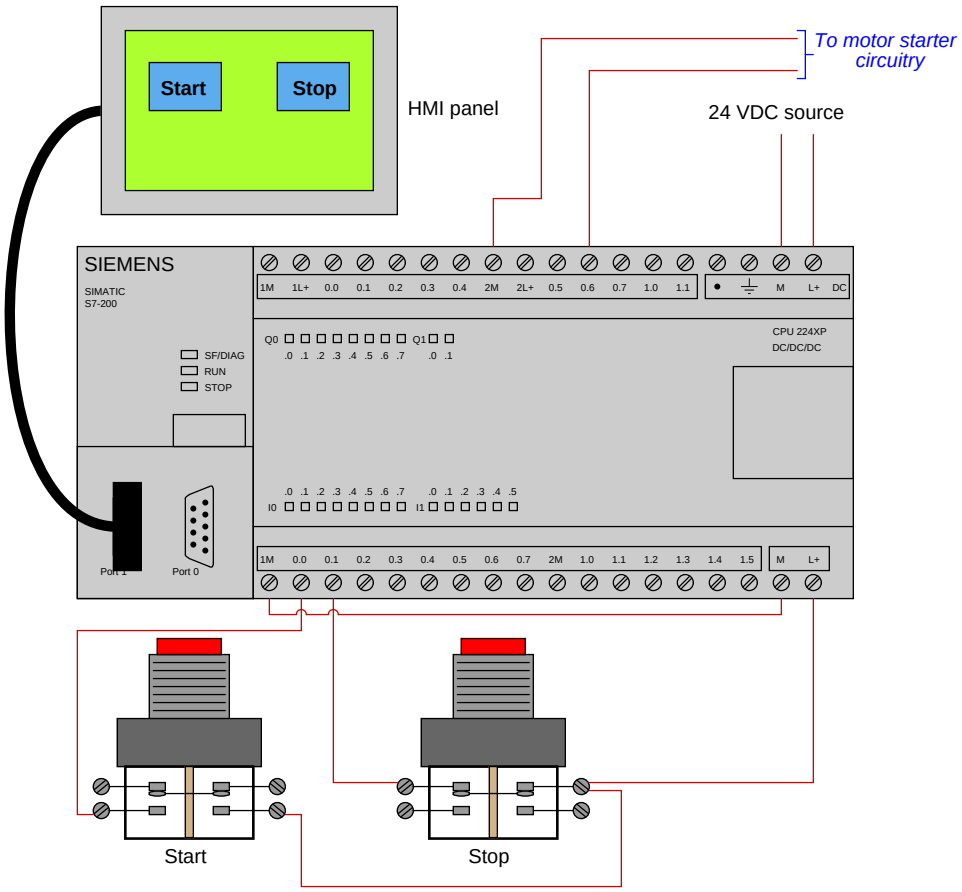
- Measure AC Volts at VFD input
- Measure AC Volts at VFD output
- Jumper control cable terminals at VFD
- Jumper control cable terminals at PLC output card
- Force PLC output bit on
- Measure DC Volts between PLC output terminals
- Check PLC mode (Run/Terminal/Stop)

Challenges

- What, exactly, makes a diagnostic test useful?

6.3.3 Troubleshooting a new HMI/PLC system

A technician is troubleshooting a problem with a PLC program she just wrote. The program is a simple motor start/stop control, with an HMI panel interfacing with the Siemens S7-200 PLC for another set of start/stop "buttons" to complement the hard-wired momentary-contact pushbuttons:



The problem is, the motor starts and stops just fine by pushing the real pushbuttons, but not by pressing the "button" objects on the HMI's touch-screen display. For some reason, the HMI is unable to either start or stop the motor. The tagname database programmed into the HMI only has two tags: `Motor_Start` and `Motor_Stop`, assigned (with "write" permission) to PLC addresses `I0.0` and `I0.1` respectively.

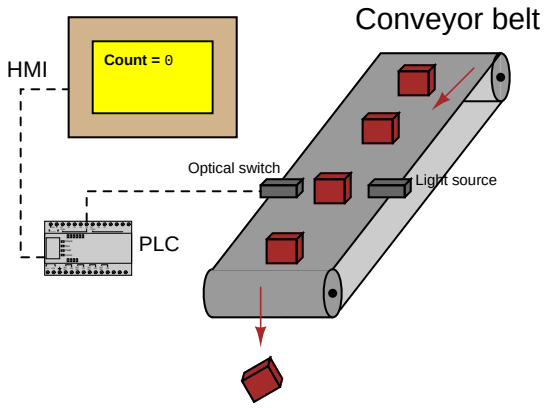
Identify what the problem is in this newly-programmed system, and also how to fix it. Additionally, determine whether the pushbutton switch inputs are wired to *source* current to the PLC or *sink* current from it.

Challenges

- If the "Start" pushbutton switch's wire came disconnected from input `I0.0`, would the HMI panel then be able to start up the motor?
- If the "Stop" pushbutton switch's wire came disconnected from input `I0.1`, would the HMI panel then be able to stop the motor?
- If the "Start" and "Stop" pushbutton switches were re-wired to connect to inputs `I0.5` and `I0.6` respectively, would this solve the problem?

6.3.4 Possible faults in a PLC/HMI package-counting system

A PLC counts packages coming by on a conveyor belt in a manufacturing facility. An optical sensor detects these packages as they travel by on the conveyor belt:



Unfortunately, something is not working correctly in this system. The HMI display continues to read a count value of zero no matter how many packages pass by the sensor switch. This very same system worked just fine three days ago, and had been working fine for one whole year before that.

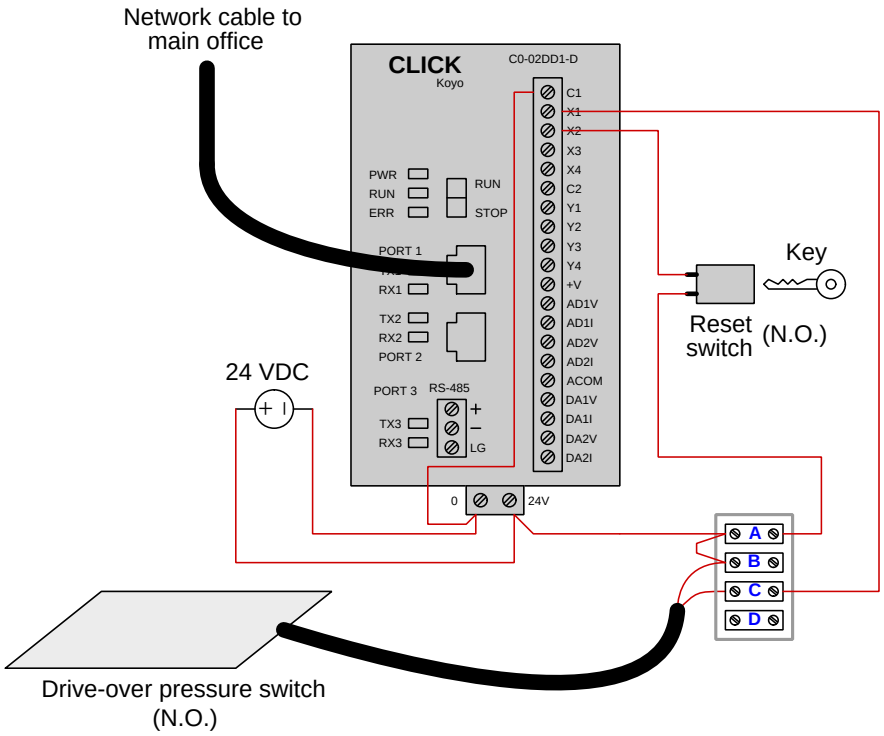
Brainstorm at least five different faults that could account for this problem, and then devise a "next test" you would conduct to narrow the field of potential faults. The simpler this test (i.e. the least amount of time to conduct and the less complicated test equipment required), the better!

Challenges

- Explain how your proposed diagnostic test would either confirm or eliminate certain fault possibilities.

6.3.5 Failed parking garage counter

This PLC senses cars entering a parking garage, and is supposed to count every one as it drives over a pressure-sensitive mat. The system is intended for the parking attendant to be able to reset the count to zero at the end of their shift using a key-switch:



For each of the following scenarios, identify possible faults to explain all symptoms, as well as whether or not the diagnostic test described yields any new and useful information. Regard each scenario on its own merits, without reference to any others.

Scenario #1

An HMI connected to this PLC over the network cable registers a fixed value of 4381 despite several more cars entering the garage. A voltmeter connected between terminals `X2` and `C1` registers 0.0 Volts DC all the time.

- Possible fault =
- Possible fault =
- Possible fault =
- Was the voltmeter test useful?

Scenario #2

An HMI connected to this PLC over the network cable registers a fixed value of 0 despite several more cars entering the garage. A voltmeter connected between terminals `X2` and `C1` registers 24.1 Volts DC all the time.

- Possible fault =
- Possible fault =
- Was the voltmeter test useful?

Scenario #3

An HMI connected to this PLC over the network cable registers a fixed value of 921 despite several more cars entering the garage. A voltmeter connected between terminals `A` and `C` registers 23.7 Volts DC all the time.

- Possible fault =
- Possible fault =
- Was the voltmeter test useful?

Scenario #4

An HMI connected to this PLC over the network cable incorrectly displays the count of cars after ten cars have entered the garage. From zero to nine cars the count displays correctly, but for ten through fifteen it just says "0" and then at the sixteenth car the HMI displays a count of "10", the seventeenth car "11", and the eighteenth car "12". A voltmeter connected between terminals `A` and `C` registers 23.9 Volts DC falling to 0.2 Volts every time a car enters the garage.

- Possible fault =
- Was the voltmeter test useful?

Challenges

- What wiring fault(s) could account for the reset switch not functioning anymore?

Appendix A

Problem-Solving Strategies

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

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

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

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

Appendix B

Instructional philosophy

B.1 First principles of learning

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

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

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

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

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

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

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

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

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

B.2 Proven strategies for instructors

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

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

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

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

B.3 Proven strategies for students

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

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

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

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

B.4 Design of these learning modules

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

These learning modules, although useful for self-study, were designed to be used in a formal learning environment where a subject-matter expert challenges students to digest the content and exercise their critical thinking abilities in the answering of questions and in the construction and testing of working circuits. Every effort has been made to embed the following instructional and assessment philosophies within:

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

These learning modules were expressly designed to be used in an "inverted" teaching environment[4] where students first read the introductory and tutorial chapters on their own, then individually attempt to answer the questions and construct working circuits according to the experiment and project guidelines. The instructor never lectures, but instead meets regularly with each individual student to review their progress, answer questions, identify misconceptions, and challenge the student to new depths of understanding through further questioning. Regular meetings between instructor and student should resemble a Socratic[5] dialogue, where questions serve as scalpels to dissect topics and expose assumptions. The student passes each module only after consistently demonstrating their ability to logically analyze and correctly apply all major concepts in each question or project/experiment. The instructor must be vigilant in probing each student's understanding to ensure they are truly *reasoning* and not just *memorizing*. This is why "Challenge" points appear throughout, as prompts for students to think deeper about topics and as starting points for instructor queries. Sometimes these challenge points require additional knowledge that hasn't been covered in the series to answer in full. This is okay, as the major purpose of the Challenges is to stimulate analysis and synthesis on the part of each student.

The instructor must possess enough mastery of the subject matter and awareness of students' reasoning to generate their own follow-up questions to practically any student response. Even completely correct answers given by the student should be challenged by the instructor for the purpose of having students practice articulating their thoughts and defending their reasoning. Conceptual errors committed by the student should be exposed and corrected not by direct instruction, but rather by reducing the errors to an absurdity[6] through well-chosen questions and thought experiments posed by the instructor. Becoming proficient at this style of instruction requires time and dedication, but the positive effects on critical thinking for both student and instructor are spectacular.

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

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

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

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

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

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

To high standards of education,

Tony R. Kuphaldt

Appendix C

Tools used

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

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

The `GNU/Linux` computer operating system

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

Bram Moolenaar's `Vim` text editor

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

Donald Knuth's TEX typesetting system

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

Leslie Lamport's LATEX extensions to TEX

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

Tim Edwards' `Xcircuit` drafting program

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

`Gimp` graphic image manipulation program

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

`SPICE` circuit simulation program

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

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

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

`gnuplot` mathematical visualization software

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

`Python` programming language

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

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

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

Section 1 – Definitions.

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

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

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

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

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

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

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

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

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

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

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

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

Section 2 – Scope.

a. License grant.

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

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

B. produce, reproduce, and Share Adapted Material.

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

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

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

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

5. Downstream recipients.

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

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

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

b. Other rights.

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

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

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

Section 3 – License Conditions.

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

a. Attribution.

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

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

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

ii. a copyright notice;

iii. a notice that refers to this Public License;

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

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

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

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

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

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

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

Section 4 – Sui Generis Database Rights.

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

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

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

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

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

Section 5 – Disclaimer of Warranties and Limitation of Liability.

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

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

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

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

Section 6 – Term and Termination.

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

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

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

2. upon express reinstatement by the Licensor.

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

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

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

Section 7 – Other Terms and Conditions.

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

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

Section 8 – Interpretation.

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

be made without permission under this Public License.

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

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

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

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

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

Appendix E

References

"HMI Handbook", document ADC-001-HMI, Automation Direct.

"Human Factors Design Standard", document HF-STD-001B, U.S. Department of Transportation, Federal Aviation Administration, 30 December 2016.

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.

24 March 2026 – minor edits to the Tutorial chapter.

10 March 2026 – minor edits to the Tutorial chapter, and also added some Challenge questions.

11 February 2026 – added more wires to image_1708 to show power to the PLC.

1 February 2026 – added a new section to the Tutorial chapter discussing integer formatting errors in more detail.

14-23 December 2025 – added new content to the Tutorial and added new questions.

26 November 2025 – added some references.

17 September 2025 – added some Diagnostic Reasoning questions.

28 July 2025 – document first created.

Index

Absolute addressing, 53
Active reading, 33, 39
Adding quantities to a qualitative problem, 104
Annotating diagrams, 103
ANSI/IEEE standard 754 for floating-point numbers, 39, 75

BCD, 34
BCD notation, 33
Binary Coded Decimal, 34
Bitwise operator, 75

C++, 56
Card, I/O, 11
Checking for exceptions, 104
Checking your work, 104
Code, computer, 113
Coil, immediate, 16
Compiler, C++, 56
Computer programming, 55

Data type, 34
Data union, 75
Dimensional analysis, 103
Direct addressing, 53
Do-More, 37

Edwards, Tim, 114

FIX/Intellution HMI software, 27
Fixed-point notation, 31
Floating-point number, 31

Graph values to solve a problem, 104
Greenleaf, Cynthia, 77

HMI panel, 20
How to teach with these modules, 111
Human-Machine Interface panel, 20
Hwang, Andrew D., 115

I/O, 11
Identify given data, 103
Identify relevant principles, 103
Ignition HMI software, 27
Immediate coil, 16
Integer number, 30, 34
Integer, signed, 72
Integer, unsigned, 72
Intermediate results, 103
Interpreter, Python, 60
Inverted instruction, 111

Java, 57

Knuth, Donald, 114
Koyo, 37

Ladder Diagram, 11
Lamport, Leslie, 114
Limiting cases, 104

Masking, 75
Memory map, 49
Metacognition, 82
Moolenaar, Bram, 113
Most Significant Bit, 71
MSB, 71
Murphy, Lynn, 77

One-shot, 19
Open-source, 113

PLC, 11, 37
PLC scan cycle, 16
Precision, single, 39

Problem-solving: annotate diagrams, 103
Problem-solving: check for exceptions, 104
Problem-solving: checking work, 104
Problem-solving: dimensional analysis, 103
Problem-solving: graph values, 104
Problem-solving: identify given data, 103
Problem-solving: identify relevant principles, 103
Problem-solving: interpret intermediate results, 103
Problem-solving: limiting cases, 104
Problem-solving: qualitative to quantitative, 104
Problem-solving: quantitative to qualitative, 104
Problem-solving: reductio ad absurdum, 104
Problem-solving: simplify the system, 38, 73, 103
Problem-solving: thought experiment, 103
Problem-solving: track units of measurement, 103
Problem-solving: visually represent the system, 103
Problem-solving: work in reverse, 104
Programmable Logic Controller, 11, 37
Programming, computer, 55
Python, 60

Qualitatively approaching a quantitative problem, 104

Raw data, 38
Reading Apprenticeship, 77
Reading, active, 33, 39
Reductio ad absurdum, 104, 110, 111
Relay ladder logic, 11
RSView HMI software, Rockwell, 27

Scan cycle, PLC, 16
Schoenbach, Ruth, 77
Scientific method, 82
Signed integer, 34, 72
Simplifying a system, 38, 73, 103
Single-precision floating-point number, 39
Socrates, 110
Socratic dialogue, 111
Source code, 56, 70
SPICE, 77
Stallman, Richard, 113
Symbol, 54
Symbolic addressing, 54

Tag name, 23, 28, 29, 54
Tag name, naming conventions for, 24
Thought experiment, 103
Torvalds, Linus, 113
Two's complement, 34

Union, data, 75
Units of measurement, 103
Unsigned integer, 34, 72

Visualizing a system, 103

Whitespace, C++, 56, 57
Whitespace, Python, 63
Whole number, 34
Wonderware HMI software, 27
Work in reverse to solve a problem, 104
WYSIWYG, 113, 114