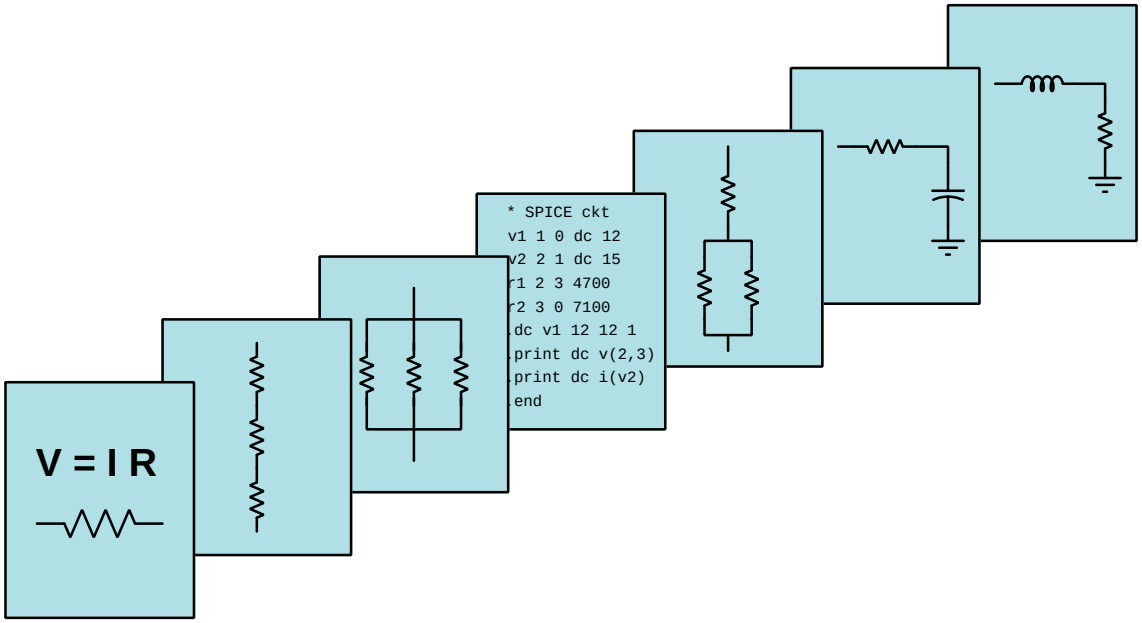
Modular Electronics Learning (ModEL) project



PLC Sequencer Programming

Last update = 20 May 2025

Contents

Chapter 1

Introduction

1.1 Recommendations for students

A *programmable logic controller* or *PLC* is a specialized form of industrial computer, designed to be programmed by the end user for many different types of discrete and continuous process control applications. The word "programmable" in its name reveals just why PLCs are so useful: the end-user is able to program, or instruct, the PLC to execute virtually any control function imaginable.

PLCs were introduced to industry as electronic replacements for electromechanical relay controls. In applications where relays typically control the starting and stopping of electric motors and other discrete output devices, the reliability of an electronic PLC meant fewer system failures and longer operating life. The re-programmability of a PLC also meant changes could be implemented to the control system strategy must easier than with relay circuits, where re-wiring was the only way to alter the system's function. Additionally, the computer-based nature of a PLC meant that process control data could now be communicated by the PLC over *networks*, allowing process conditions to be monitored in distant locations, and by multiple operator stations.

The legacy of PLCs as relay-replacements is probably most evident in their traditional programming language: a graphical convention known as a *Ladder Diagram*. Ladder Diagram PLC programs resemble ladder-style electrical schematics, where vertical power "rails" convey control power to a set of parallel "rung" circuits containing switch contacts and relay coils. A human being programming a PLC literally draws the diagram on the screen, using relay-contact symbols to represent instructions to read data bits in the PLC's memory, and relay-coil symbols to represent instructions writing data bits to the PLC's memory. When executing, these graphical elements become colored when "conductive" to virtual electricity, thereby indicating their status to any human observer of the program. This style of programming was developed to make it easier for industrial electricians to adapt to the new technology of PLCs. While Ladder Diagram programming definitely has limitations compared to other computer programming languages, it is relatively easy to learn and diagnose, which is why it remains popular as a PLC programming language today.

While ladder diagram programming was designed to be simple by virtue of its resemblance to relay ladder-logic schematic diagrams, this very same resemblance often creates problems for students encountering it for the very first time. A very common misconception is to think that the contact and

coil symbols shown on the editing screen of the PLC programming software are somehow *identical* or at least *directly representative* of real-world contacts and coils wired to the PLC. This is not true. Contacts and coils shown on the screen of PLC programming software applications are *instructions* for the PLC to follow, and their logical states depend both on how they are drawn in the program *and* upon their related bit states in the PLC's memory.

The relationship between a discrete sensor (e.g. switch) and the colored state of a ladder diagram element inside of a PLC follows a step-by-step chain of causation:

1. Physical closure of the discrete switch causes electricity to flow through the switch's contact.
2. This electrical current flows through the PLC input terminal wired to that switch.
3. This energization causes a corresponding bit in the PLC's memory to become "high" (1).
4. Any Ladder Diagram "contact" instruction associated with that bit will become "actuated". If the contact instruction is normally-open, the "1" bit state will "close" the contact instruction and cause it to be colored. If the contact instruction is normally-closed, the "1" bit state will "open" is and cause it to be un-colored.
5. If all elements in a rung of the Ladder Diagram program are colored, the final instruction (at the far right end of the rung) will become activated and will cause it to fulfill its function.

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

- How might an experiment be designed and conducted to test whether a switch is normally-open or normally-closed?
- How might an experiment be designed and conducted to test the "scan time" of a PLC program?
- What are some practical applications of PLCs?
- What is the "normal" status of a switch?
- How does a "two-out-of-three" alarm or shutdown system function?
- What is a "nuisance trip"?
- Are there applications where a hard-wired relay control system might actually be better than a system using a PLC?
- What purpose is served by the color highlighting feature of PLC program editing software?
- How might you alter one of the example analyses shown in the text, and then determine the behavior of that altered circuit?
- Devise your own question based on the text, suitable for posing to someone encountering this subject for the first time

1.2 Challenging concepts related to programmable logic controllers (PLCs)

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

- **Normal status of a switch** – to say that a switch is "normally open" or "normally closed" refers to its electrical state *when at rest*, and not necessarily the state you will typically find the switch in. The root of the confusion here is the word *normal*, which most people take to mean "typical" or "ordinary". In the case of switches, though, it refers to the zero-stimulation status of the switch as it has been manufactured. In the case of PLCs it refers to the status of a "virtual switch" when its controlling bit is 0.

- **Seal-in contacts** – the simplest and most common way to make momentary-contact "Start" and "Stop" switches latch a motor's state on and off is to wire an auxiliary contact from the contactor in parallel with the Start switch to "seal in" that circuit once the contactor energizes. The same logical structure may be implemented in PLC ladder-diagram programming by placement of a virtual contact in parallel with the initiating contact, that additional contact being controlled by the coil of that same rung. This, however, complicates analysis of the circuit by granting it *states*. State-based logic is more complex than combinational logic because the status of state-based logic depends not only on input conditions but also on past history. This means a person must analyze the PLC program before and after input condition changes to determine how it will respond.

- **Sourcing versus Sinking output currents** – a common misconception is that since PLC outputs are called "outputs" it must mean that current only ever *exits* those terminals. This is untrue. All that "output" actually signifies is the fact that the PLC is outputting *information* consisting of voltage values measured between that terminal and ground. Sometimes the assertion of an "on" state requires that the PLC output channel actually draws current *in* through its "output" terminal!

1.3 Recommendations for instructors

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

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

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

  Assessment – Interpret elements of a real PLC ladder-logic program such as those shown in the Simplified and Full Tutorial chapters, including identification of virtual coils, virtual contacts (normally-open and normally-closed), color-highlighting, and associated statuses of those elements based on color highlighting.

- **Outcome** – Apply the concept of "normal" contact status to real and virtual contacts in PLC systems

  Assessment – Predict the status of every PLC program virtual contact and PLC program coil in a ladder-style diagram for various input switch state combinations. Good starting points include the Case Tutorial section "Example: NAND function in a PLC" as well as the two-out-of-three water flow alarm system shown in the Simplified Tutorial chapter.

- **Outcome** – Properly associate physical switch states with their associated PLC bit states, PLC virtual contact states, PLC virtual coil states, and PLC output states in a PLC-controlled system.

  Assessment – Determine PLC bit states based on given physical switch stimuli in a schematic diagram; e.g. pose problems in the form of the "Determining bit statuses from switch conditions" Conceptual Reasoning question.

  Assessment – Determine physical switch stimuli based on given PLC bit states in a schematic diagram; e.g. pose problems in the form of the "Determining necessary switch conditions for bit statuses" Conceptual Reasoning question.

  Assessment – Determine PLC ladder-diagram program element coloring based on given physical switch stimuli in a schematic diagram; e.g. pose problems in the form of the "Determining color highlighting from switch conditions" Conceptual Reasoning question.

  Assessment – Determine PLC bit states based on color-highlighting shown in a live view of a PLC's ladder-diagram program; e.g. pose problems in the form of the "Determining bit statuses from color highlighting".

  Assessment – Determine physical switch stimuli based on color-highlighting shown in a live view of a PLC's ladder-diagram program; e.g. pose problems in the form of the "Determining process switch stimuli from color highlighting" Conceptual Reasoning question.

- **Outcome** – Sketch wire connections necessary to interface a PLC to a controlled process

  Assessment – Sketch wire placement in a pictorial diagram showing how switches would connect to the input channels of a PLC and how loads would connect to the output channels of the same PLC; e.g. pose problems in the form of the "Sketching wires to PLC discrete I/O" Conceptual Reasoning question.

- **Outcome** – Diagnose a fault within a PLC-controlled system

  Assessment – Identify possible faults to account for a system's improper function based on an examination of the color highlighting in a live view of the PLC's ladder-diagram program; e.g. pose problems in the form of the "Troubleshooting motor control program" and "Motor starter diagnosis from color highlighting" Diagnostic Reasoning questions.

  Assessment – Identify possible faults to account for a system's improper function based on an examination of the I/O status indicators on the front of the PLC; e.g. pose problems in the form of the "Troubleshooting motor control PLC from I/O indicators" Diagnostic Reasoning question.

- **Outcome** – Independent research

  Assessment – Locate PLC I/O module datasheets and properly interpret some of the information contained in those documents including number of I/O channels, voltage and current limitations, sourcing versus sinking capability, etc.

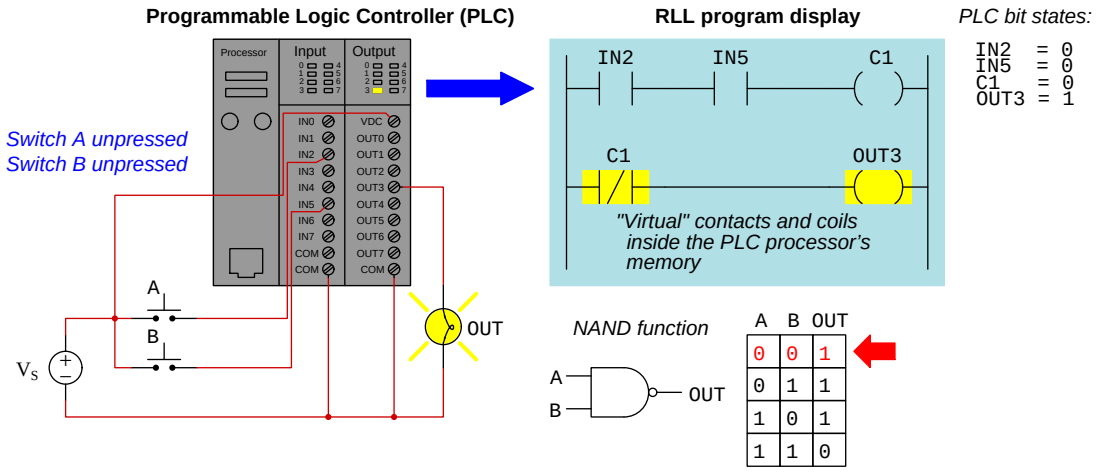
Chapter 2

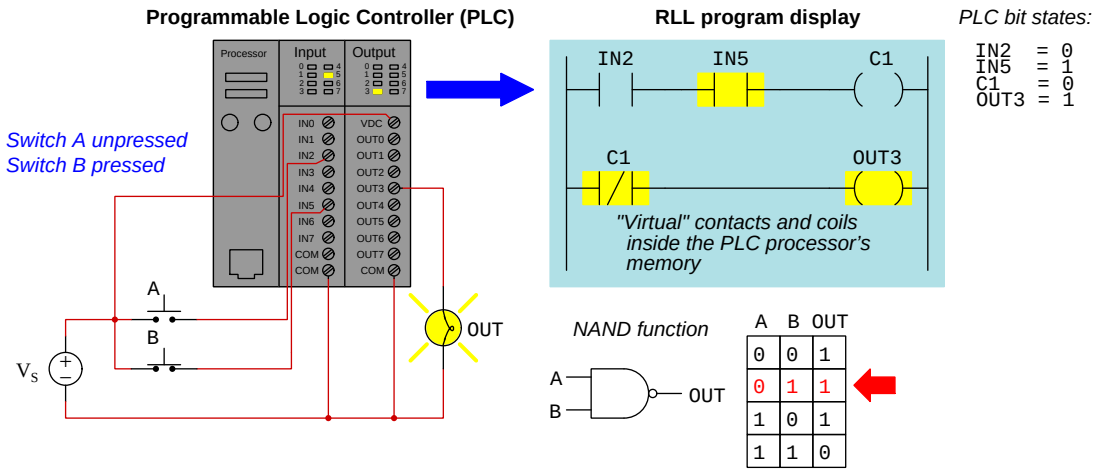
Case Tutorial

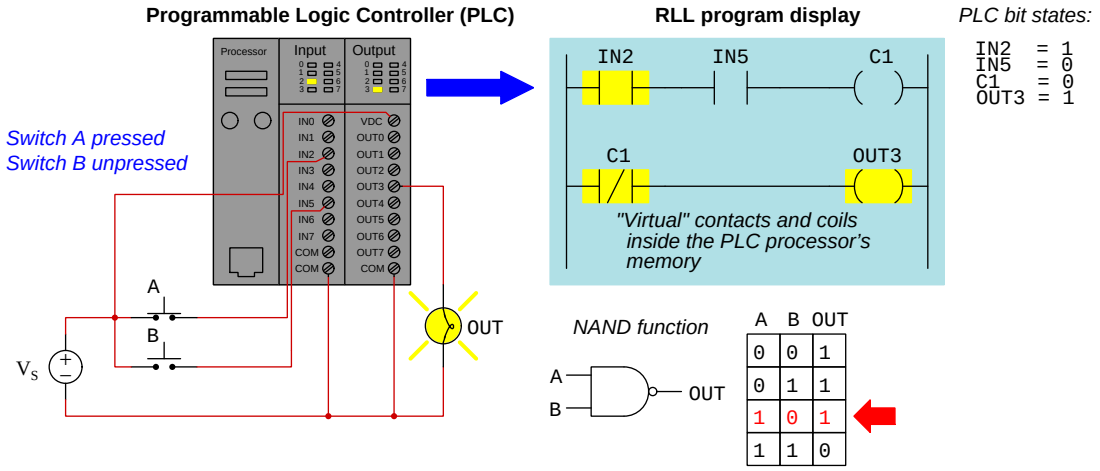
The idea behind a *Case Tutorial* is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as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

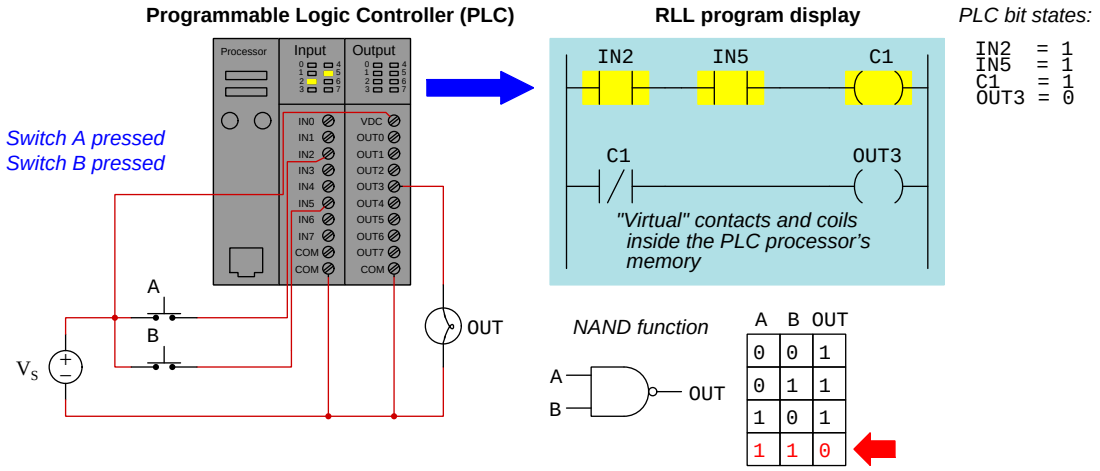
These examples also serve well as challenges following your reading of the other Tutorial(s) in this module – can you explain *why* the circuits behave as they do?

2.1 Example: NAND function in a PLC



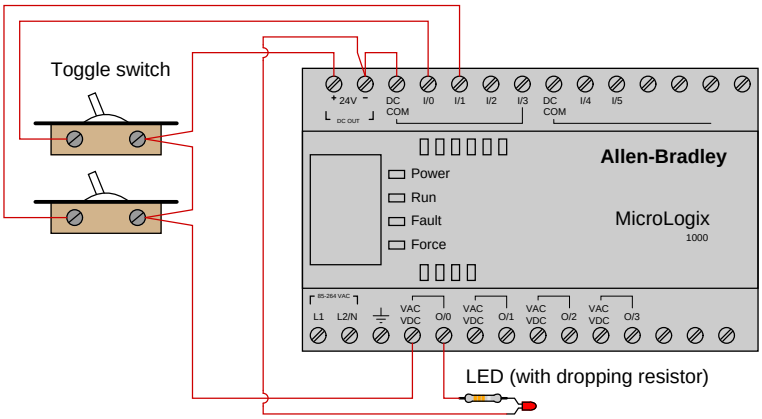



Programmable Logic Controller (PLC)
RLL program display
PLC bit states:
Processor
Input
Output
Switch A pressed
Switch B unpressed
A
B
OUT
IN2
IN5
C1
C1
OUT3
"Virtual" contacts and coils inside the PLC processor's memory
IN2 = 1
IN5 = 0
C1 = 0
OUT3 = 1
NAND function
A B OUT
0 0 1
0 1 1
1 0 1
1 1 0

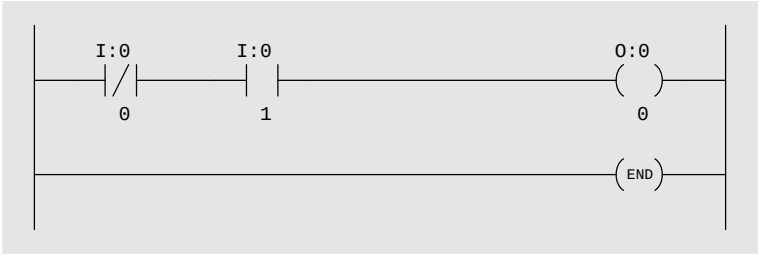

Programmable Logic Controller (PLC)
RLL program display
PLC bit states:
Processor
Input
Output
Switch A pressed
Switch B pressed
A
B
OUT
IN2
IN5
C1
C1
OUT3
"Virtual" contacts and coils inside the PLC processor's memory
IN2 = 1
IN5 = 1
C1 = 1
OUT3 = 0
NAND function
A B OUT
0 0 1
0 1 1
1 0 1
1 1 0

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:



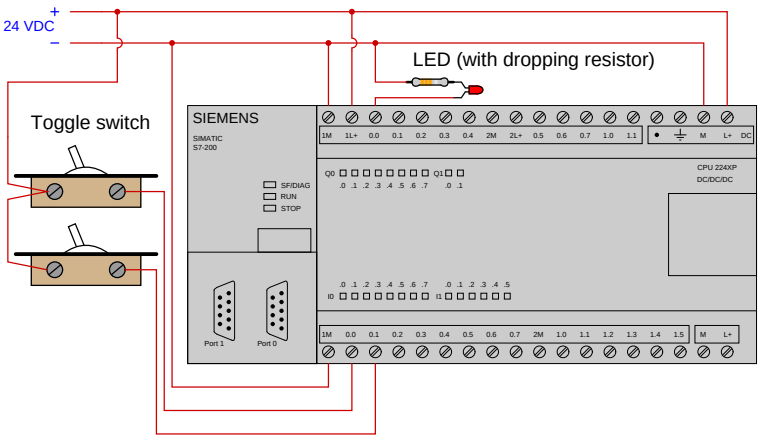
Ladder-Diagram program written to PLC:



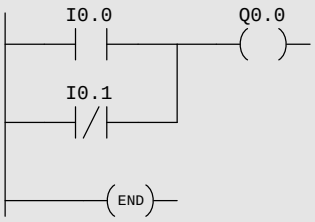
Note how Allen-Bradley I/O is labeled in the program: input bits designated by the letter `I` and output bits designated by the letter `O`.

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:



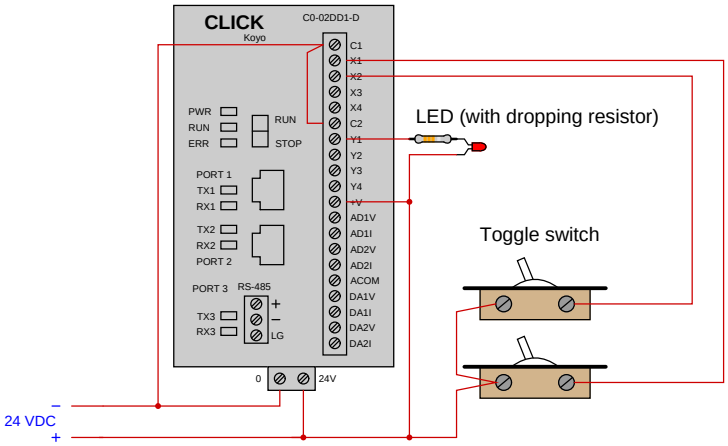
Ladder-Diagram program written to PLC:



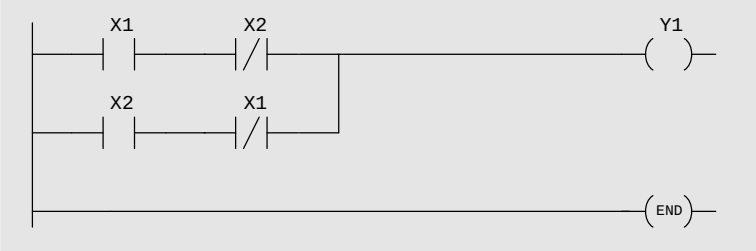
Note how Siemens I/O is labeled in the program: input bits designated by the letter `I` and output bits designated by the letter `Q`.

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)

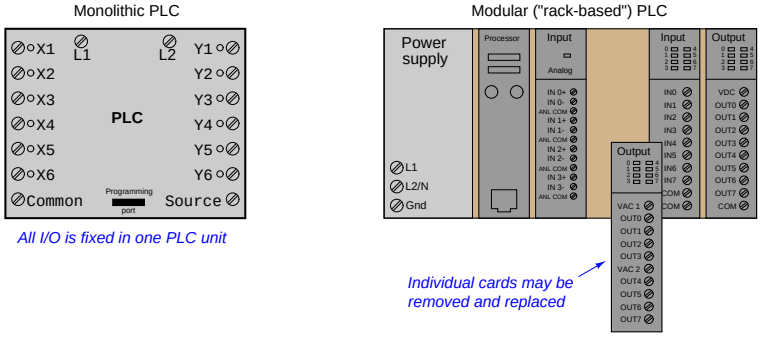
Either the switch connected to input terminal 0.0 must be on (closed) or the switch connected to input terminal 0.1 must be off (open).

Chapter 3

Tutorial

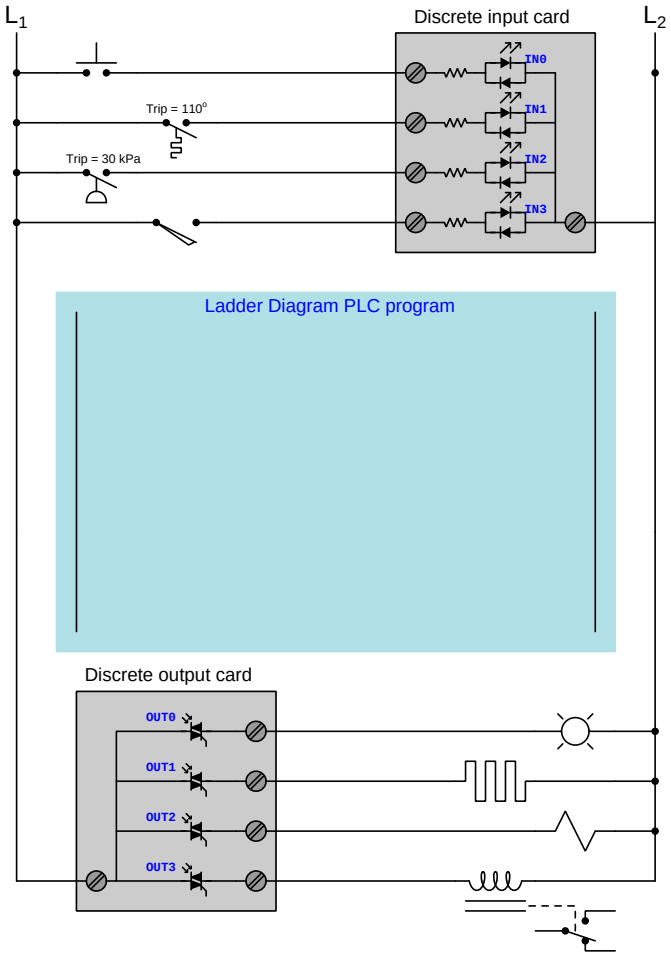
3.1 Review of basic PLC functionality

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.

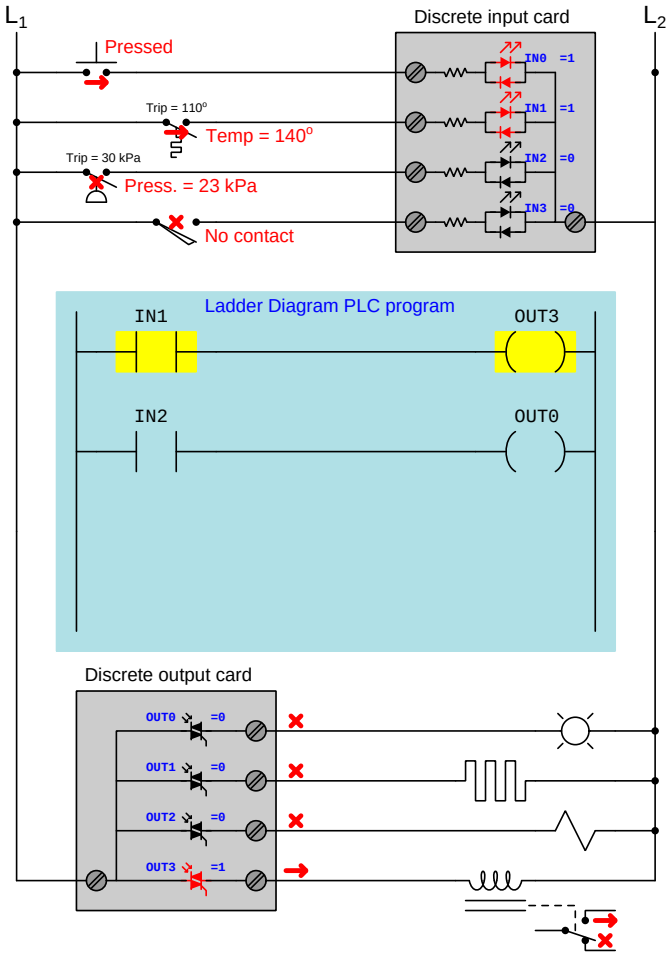
The following diagram shows a PLC separated into three sections: (1) a *discrete input card*, (2) the *program* space in the processor's memory, and (3) a *discrete output card*:



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).

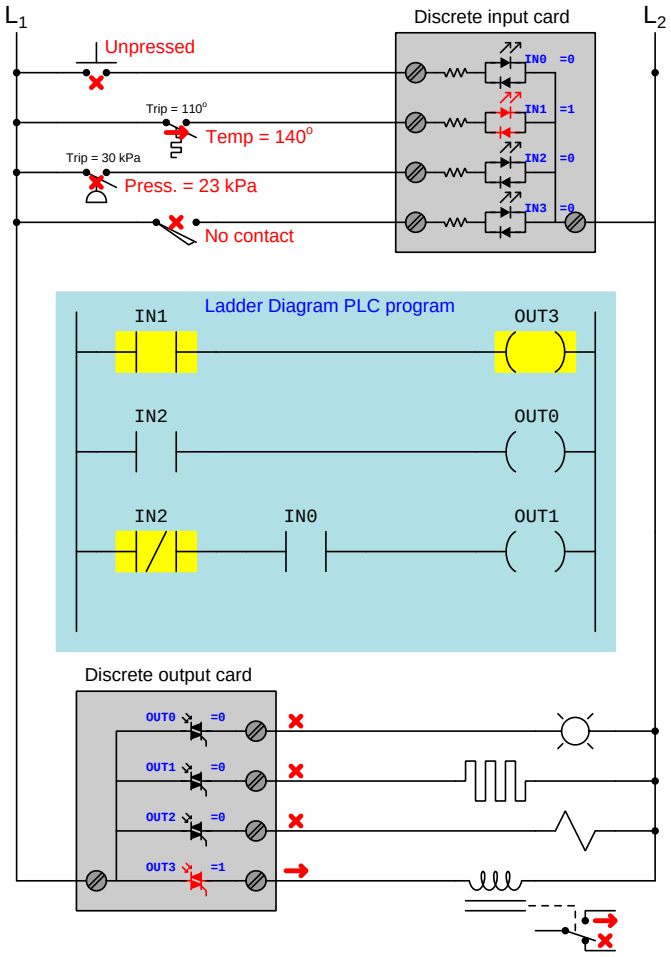
[1] Not shown in this simplified diagram are the optotransistors coupled to the LEDs inside the input card, translating each LED's state to a discrete logic level at the transistor to be interpreted by the PLC's digital processor. Likewise, another set of LEDs driven by the processor's outputs couple 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.

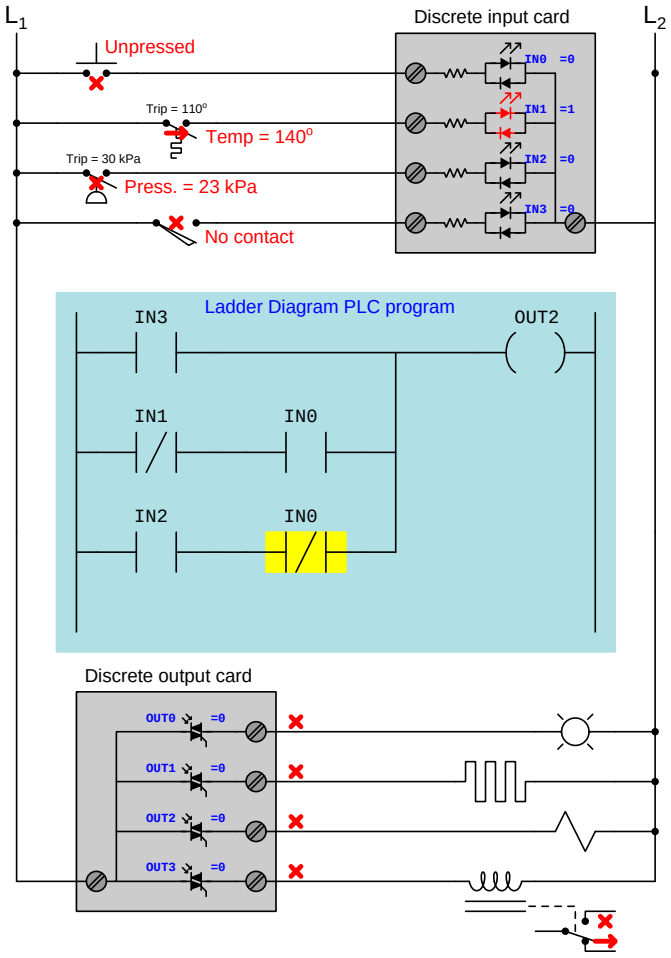
Things get more complex when we begin adding *normally-closed* contact instructions to the program. Consider this next diagram, with updated stimuli and an expanded Ladder Diagram program for the PLC to follow:



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.

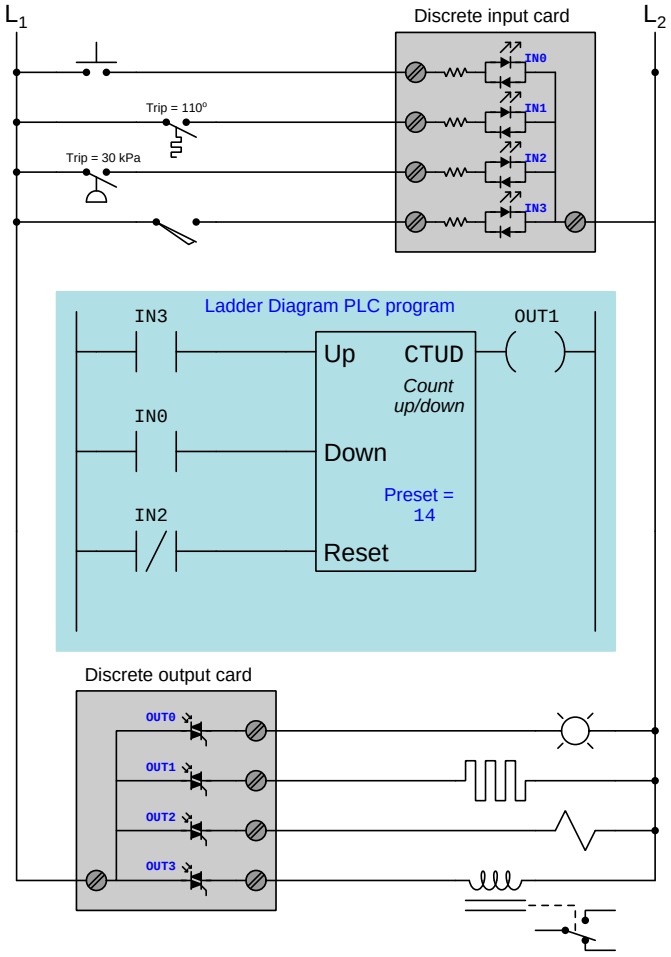
The logical chain of causality from input to output on a PLC is very important to understand, and will be represented here by a sequence of numbered statements:

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 this same causality *must* be mentally tracked 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[2] 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 are rooted in the same error, namely not deliberately 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).

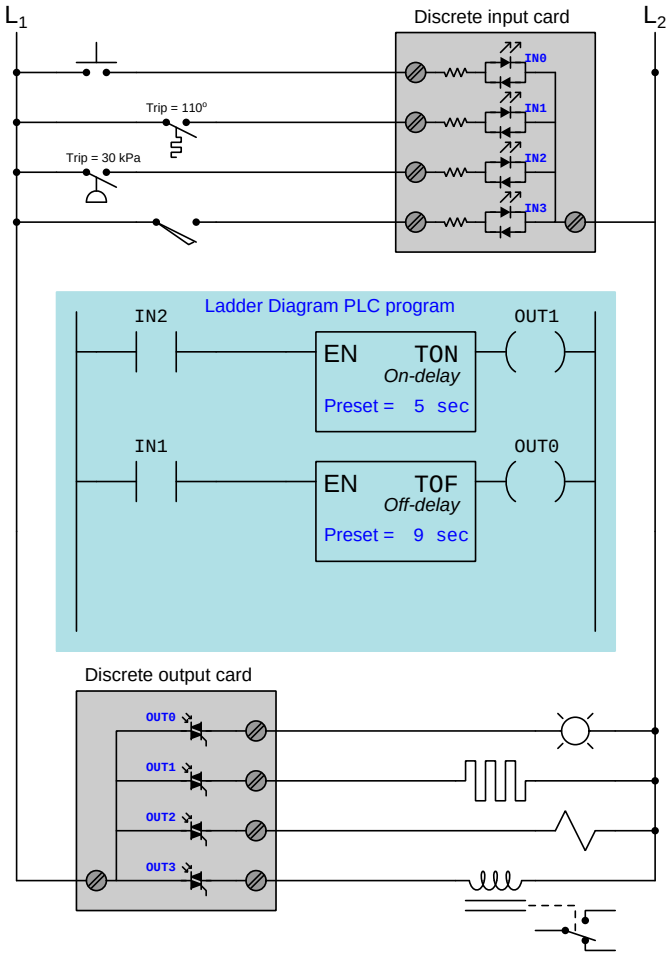
[2]For a normally-open contact instruction, this association is direct. However, for a normally-closed contact instruction it is inverted!

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.

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.

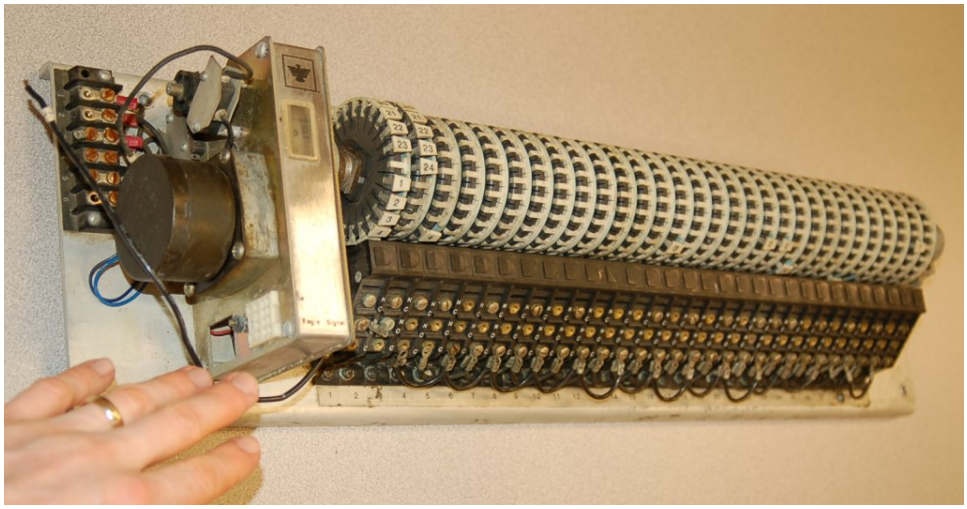
PLCs are not limited to contact, coil, counter, and timer instructions, either. A typical PLC literally offers dozens of instruction types in its set, which may be applied and combined in nearly limitless fashion. Other types of PLC programming instructions include *latch instructions* (offering bistable "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 Sequencer drum relays

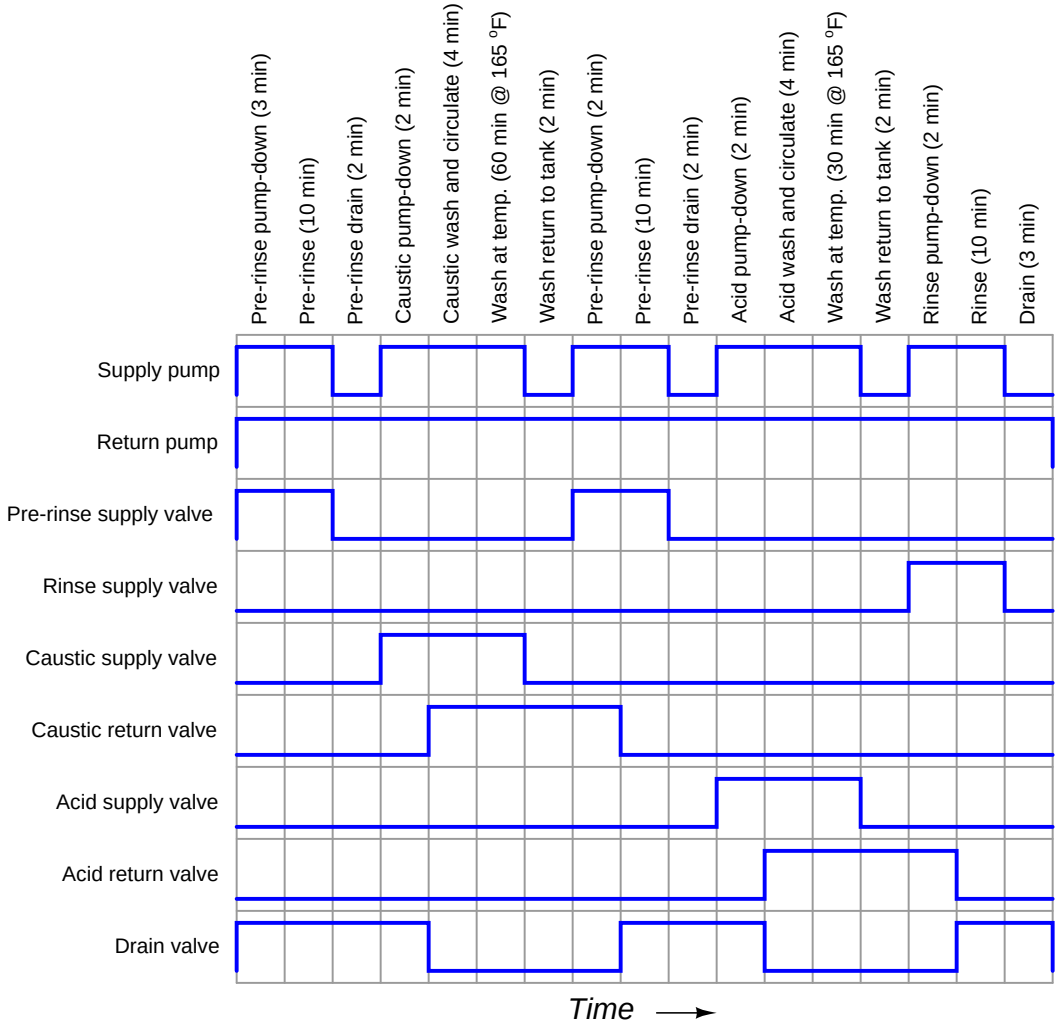
Many industrial processes require control actions to take place in certain, predefined sequences. Batch processes are perhaps the most striking example of this, where materials for making a batch must be loaded into the process vessels, parameters such as temperature and pressure controlled during the batch processing, and then discharge of the product monitored and controlled. Before the advent of reliable programmable logic devices, this form of sequenced control was usually managed by an electromechanical device known as a *drum sequencer*. This device worked on the principle of a rotating cylinder (drum) equipped with tabs to actuate switches as the drum rotated into certain positions. If the drum rotated at a constant speed (turned by a clock motor), those switches would actuate according to a timed schedule[3].

The following photograph shows a drum sequencer with 30 switches. Numbered tabs on the circumference of the drum mark the drum's rotary position in one of 24 increments. With this number of switches and tabs, the drum can control up to thirty discrete (on/off) devices over a series of twenty-four sequenced steps:



[3]The operation of the drum is not unlike that of an old *player piano*, where a strip of paper punched with holes caused hammers in the piano to automatically strike their respective strings as the strip was moved along at a set speed, thus playing a pre-programmed song.

A typical application for a sequencer is to control a *Clean In Place* (*CIP*) system for a food processing vessel, where a process vessel must undergo a cleaning cycle to purge it of any biological matter between food processing cycles. The steps required to clean the vessel are well-defined and must always occur in the same sequence in order to ensure hygienic conditions. An example timing chart is shown here:



In this example, there are nine discrete outputs – one for each of the nine final control elements (pumps and valves) – and seventeen steps to the sequence, each one of them timed. In this particular sequence, the only input is the discrete signal to commence the CIP cycle. From the initiation of the CIP to its conclusion two and a half hours (150 minutes) later, the sequencer simply steps through the programmed routine.

Another practical application for a sequencer is to implement a *Burner Management System* (BMS), also called a *flame safety system*. Here, the sequencer manages the safe start-up of a combustion burner: beginning by "purging" the combustion chamber with fresh air to sweep out any residual fuel vapors, waiting for the command to light the fire, energizing a spark ignition system on command, and then continuously monitoring for presence of good flame and proper fuel supply pressure once the burner is lit.

In a general sense, the operation of a drum sequencer is that of a *state machine*: the output of the system depends on the condition of the machine's internal state (the drum position), not just the conditions of the input signals. Digital computers are very adept at implementing state functions, and so the general function of a drum sequencer should be (and is) easy to implement in a PLC. Other PLC functions we have seen ("latches" and timers in particular) are similar, in that the PLC's output at any given time is a function of both its present input condition(s) and its past input condition(s). Sequencing functions expand upon this concept to define a much larger number of possible states ("positions" of a "drum"), some of which may even be timed.

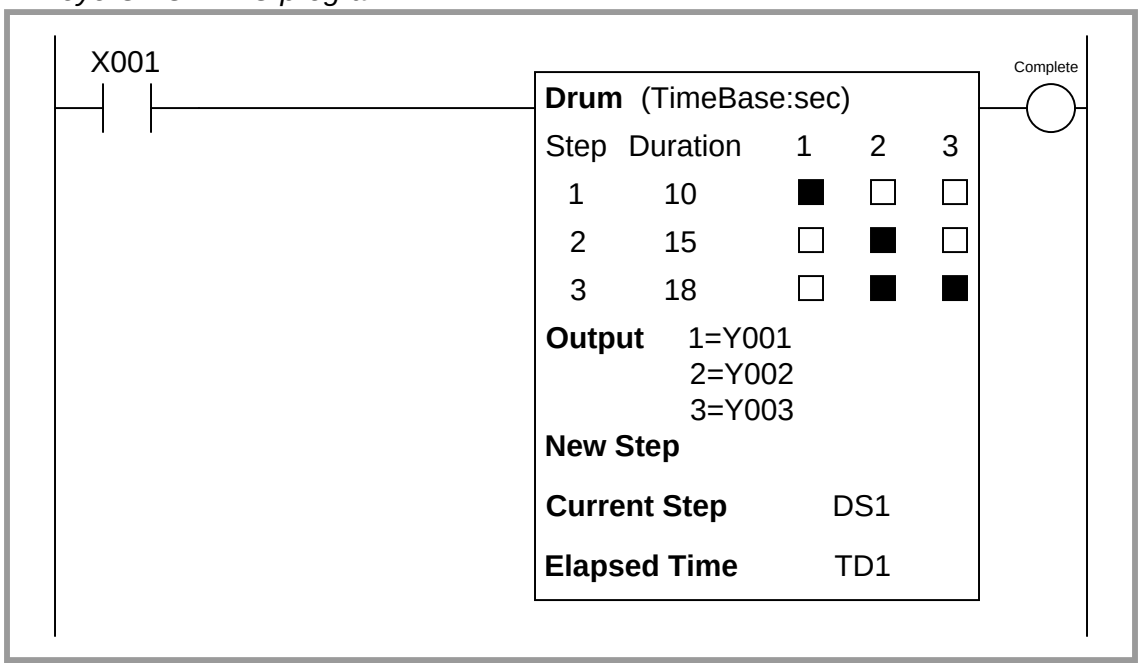
Unfortunately, despite the utility of drum sequence functions and their ease of implementation in digital form, there seems to be very little standardization between PLC manufacturers regarding sequencing instructions. Sadly, the IEC 61131-3 standard (at least at the time of this writing, in 2009) does not specifically define a sequencing function suitable for Ladder Diagram programming. PLC manufacturers are left to invent sequencing instructions of their own design. What follows here is an exploration of some different sequencer instructions offered by PLC manufacturers.

3.3 Sequence control using counters and timers

3.4 Koyo "drum" sequencer instructions

The *drum* instruction offered in Koyo PLCs is a model of simplicity itself. This instruction is practically self-explanatory, as shown in the following example:

Koyo CLICK PLC program



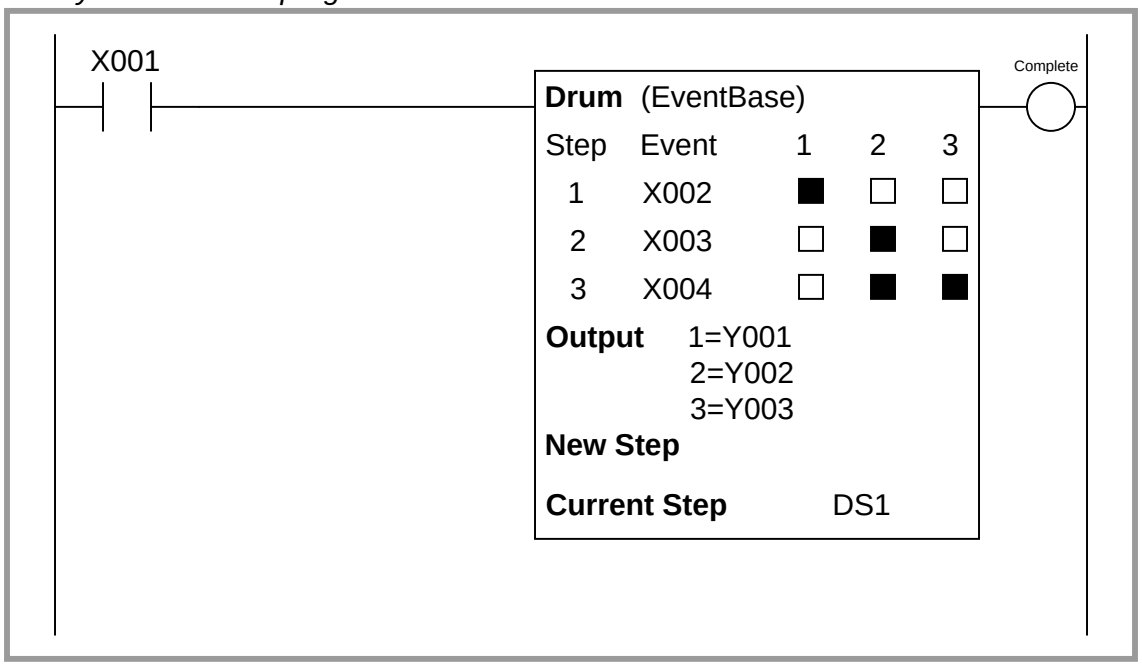
The three-by-three grid of squares represent steps in the sequence and bit states for each step. Rows represent steps, while columns represent output bits written by the drum instruction. In this particular example, a three-step sequence proceeds at the command of a single input (`X001`), and the drum instruction's advance from one step to the next proceeds strictly on the basis of elapsed time (a *time base* orientation). When the input is active, the drum proceeds through its timed sequence. When the input is inactive, the drum halts wherever it left off, and resumes timing as soon as the input becomes active again.

Being based on time, each step in the drum instruction has a set time duration for completion. The first step in this particular example has a duration of 10 seconds, the second step 15 seconds, and the third step 18 seconds. At the first step, only output bit `Y001` is set. In the second step, only output bit `Y002` is set. In the third step, output bits `Y002` and `Y003` are set (1), while bit `Y001` is reset (0). The colored versus uncolored boxes reveal which output bits are set and reset with each step. The current step number is held in memory register `DS1`, while the elapsed time (in seconds) is stored in timer register `TD1`. A "complete" bit is set at the conclusion of the three-step sequence.

Koyo drum instructions may be expanded to include more than three steps and more than three output bits, with each of those step times independently adjustable and each of the output bits arbitrarily assigned to any writable bit addresses in the PLC's memory.

This next example of a Koyo drum instruction shows how it may be set up to trigger on *events* rather than on elapsed times. This orientation is called an *event base*:

Koyo CLICK PLC program

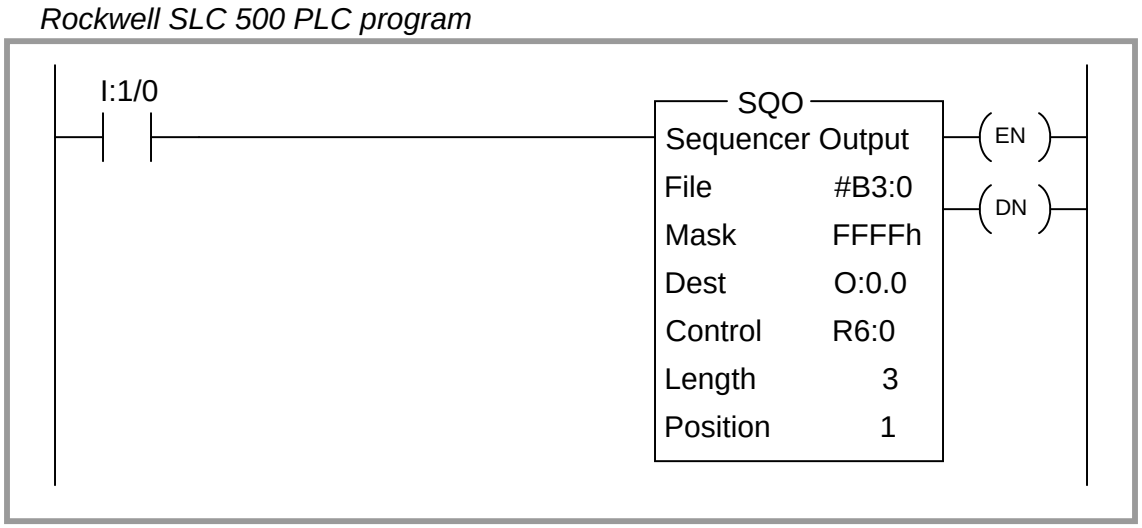


Here, a three-step sequence proceeds when enabled by a single input (`X001`), with the drum instruction's advance from one step to the next proceeding only as the different event condition bits become set. When the input is active, the drum proceeds through its sequence when each event condition is met. When the input is inactive, the drum halts wherever it left off regardless of the event bit states.

For example, during the first step (when only output bit `Y001` is set), the drum instruction waits for the first condition input bit `X002` to become set (1) before proceeding to step 2, with time being irrelevant. When this happens, the drum immediately advances to step 2 and waits for input bit `X003` to be set, and so forth. If all three event conditions were met simultaneously (`X002`, `X003`, and `X004` all set to 1), the drum would skip through all steps as fast as it could (one step per PLC program scan) with no appreciable time elapsed for each step. Conversely, the drum instruction will wait as long as it must for the right condition to be met before advancing, whether that event takes place in milliseconds or in days.

3.5 Allen-Bradley sequencer instructions

Rockwell (Allen-Bradley) PLCs use a more sophisticated set of instructions to implement sequences. The closest equivalent to Koyo's *drum* instruction is the Allen-Bradley *SQO* (Sequencer Output) instruction, shown here:



You will notice there are no colored squares inside the SQO instruction box to specify when certain bits are set or reset throughout the sequence, in contrast to the simplicity of the Koyo PLC's drum instruction. Instead, the Allen-Bradley SQO instruction is told to read a set of 16-bit words beginning at a location in the PLC's memory arbitrarily specified by the programmer, one word at a time. It steps to the next word in that set of words with each new position (step) value. This means Allen-Bradley sequencer instructions rely on the programmer already having pre-loaded an area of the PLC's memory with the necessary 1's and 0's defining the sequence. This makes the Allen-Bradley sequencer instruction more challenging for a human programmer to interpret because the bit states are not explicitly shown inside the SQO instruction box, but it also makes the sequencer far more flexible in that these bits are not fixed parameters of the SQO instruction and therefore may be dynamically altered as the PLC runs. With the Koyo drum instruction, the assigned output states are part of the instruction itself, and are therefore fixed once the program is downloaded to the PLC (i.e. they cannot be altered without editing and re-loading the PLC's program). With the Allen-Bradley, the on-or-off bit states for the sequence may be freely altered[4] during run-time. This is a very useful feature in recipe-control applications, where the recipe is subject to change at the whim of production personnel, and they would rather not have to rely on a technician or an engineer to re-program the PLC for each new recipe.

[4]Perhaps the most practical way to give production personnel access to these bits without having them learn and use PLC programming software is to program an HMI panel to write to those memory areas of the PLC. This way, the operators may edit the sequence at any time simply by pressing "buttons" on the screen of the HMI panel, and the PLC need not have its program altered in any "hard" way by a technician or engineer.

The "Length" parameter tells the SQO instruction how many words will be read (i.e. how many steps are in the entire sequence). The sequencer advances to each new position when its enabling input transitions from inactive to active (from "false" to "true"), just like a count-up (CTU) instruction increments its accumulator value with each new false-to-true transition of the input. Here we see another important difference between the Allen-Bradley SQO instruction and the Koyo drum instruction: the Allen-Bradley instruction is fundamentally *event-driven*, and does not proceed on its own like the Koyo drum instruction is able to when configured for a *time* base.

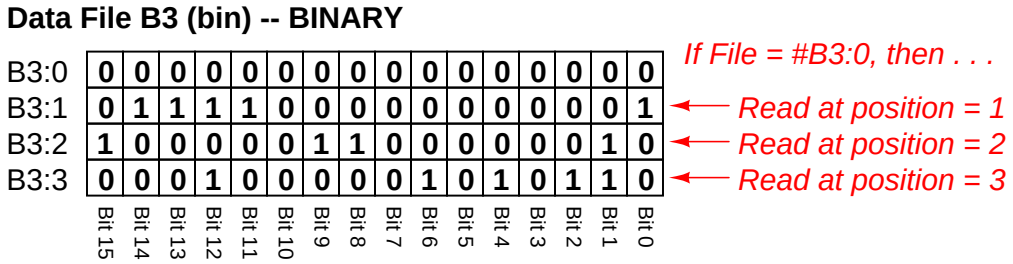
Sequencer instructions in Allen-Bradley PLCs use a notation called *indexed addressing* to specify the locations in memory for the set of 16-bit words it will read. In the example shown above, we see the "File" parameter specified as `#B3:0`. The "#" symbol tells the instruction that this is a *starting* location in memory for the first 16-bit word, when the instruction's position value is zero. As the position value increments, the SQO instruction reads 16-bit words from successive addresses in the PLC's memory. If `B3:0` is the word referenced at position 0, then `B3:1` will be the memory address read at position 1, `B3:2` will be the memory address read at position 2, etc. Thus, the "position" value causes the SQO instruction to "point" or "index" to successive memory locations.

The bits read from each indexed word in the sequence are compared against a static mask[5] specifying which bits in the indexed word are relevant. At each position, only these bits are written to the destination address.

As with most other Allen-Bradley instructions, the sequencer requires the human programmer to declare a special area in memory reserved for the instruction's internal use. The "`R6`" file exists just for this purpose, each element in that file holding bit and integer values associated with a sequencer instruction (e.g. the "enable" and "done" bits, the array length, the current position, etc.).

[5] In this particular example, the mask value is FFFF hexadecimal, which means all 1's in a 16-bit field. This mask value tells the sequencer instruction to regard *all* bits of each `B3` word that is read. To contrast, if the mask were set to a value of 000F hexadecimal instead, the sequencer would only pay attention to the four least-significant bits of each `B3` word that is read, while ignoring the 12 more-significant bits of each 16-bit word. The mask allows the SQO instruction to only write to selected bits of the destination word, rather than always writing all 16 bits of the indexed word to the destination word.

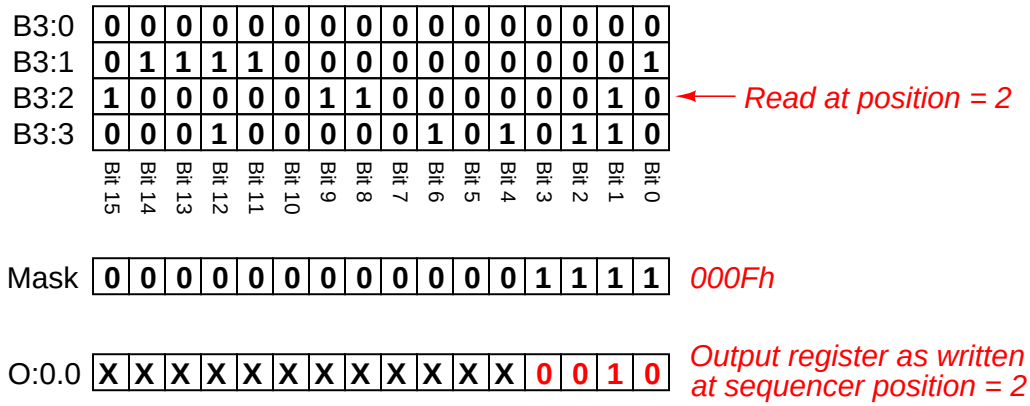
To illustrate, let us examine a set of bits held in the `B3` file of an Allen-Bradley SLC 500 PLC, showing how each row (element) of this data file would be read by an SQO instruction as it stepped through its positions:



The sequencer's position number is added to the file reference address as an *offset*. Thus, if the data file is specified in the SQO instruction box as `#B3:0`, then `B3:1` will be the row of bits read when the sequencer's position value is 1, `B3:2` will be the row of bits read when the position value is 2, and so on.

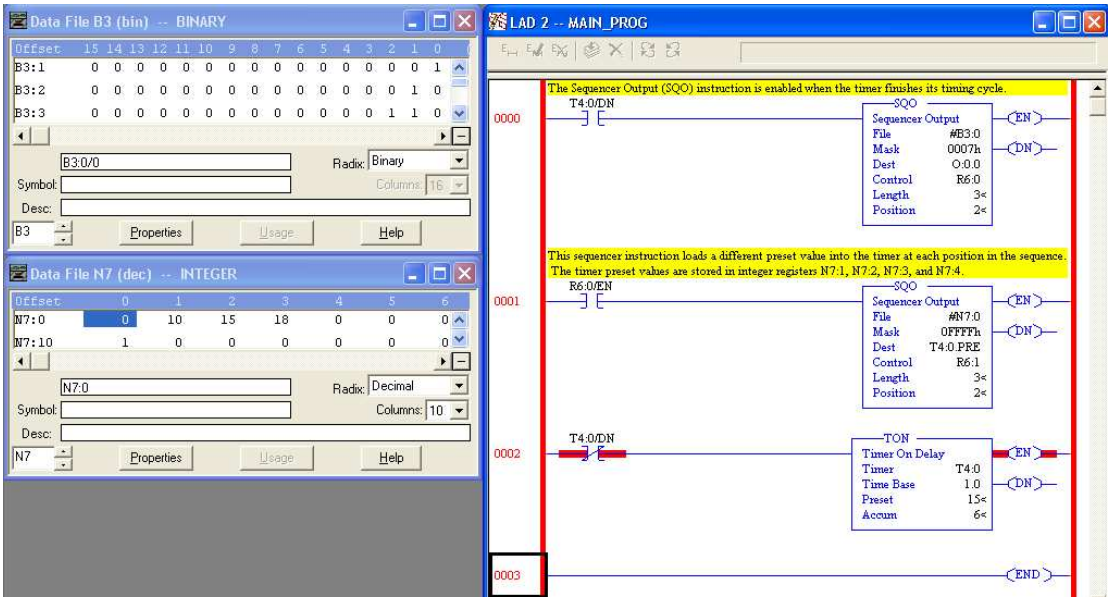
The *mask* value specified in the SQO instruction tells the instruction which bits out of each row will be copied to the destination address. A mask value of FFFFh (FFFF in *hexadecimal* format) means all 16 bits of each `B3` word will be read and written to the destination. A mask value of 0001h means only the first (least-significant) bit will be read and written, with the rest being ignored.

Let's see what would happen with an SQO instruction having a mask value of 000Fh, starting from file index `#B3:0`, and writing to a destination that is output register `O:0.0`, given the bit array values in file `B3` shown above:



When this SQO instruction is at position 2, it reads the bit values `0010` from `B3:2` and writes only those four bits to `O:0.0`. The "X" symbols shown in the illustration mean that all the other bits in that output register are untouched – the SQO instruction does not write to those bits because they are "masked off" from being written. You may think of the mask's zero bits inhibiting source bits from being written to the destination word in the same sense that *masking tape* prevents paint from being applied to a surface.

The following Allen-Bradley SLC 500 PLC program shows how a pair of SQO instructions plus an on-delay timer instruction may be used to duplicate the exact same functionality as the "time base" Koyo drum instruction presented earlier:



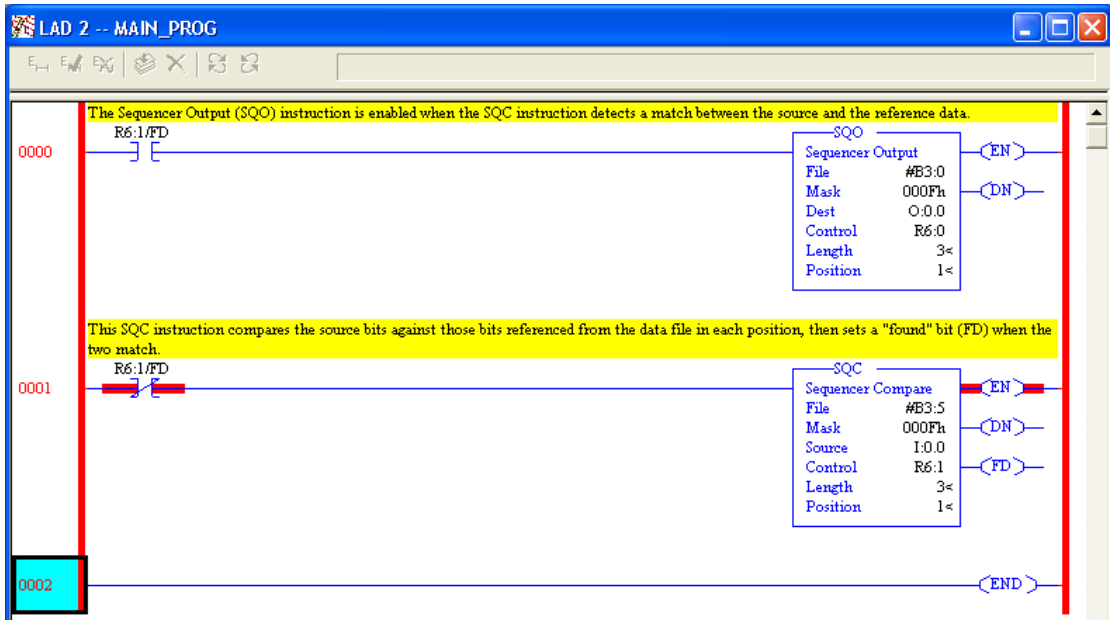
The first SQO instruction reads bits in the `B3` file array, sending only the three least-significant of them to the output register `O:0.0` (as specified by the 0007h mask value). The second SQO instruction reads integer number values from elements of the `N7` integer file and places them into the "preset" register of timer `T4:0`, so as to dynamically update the timer's preset value with each step of the sequence. The timer, in turn, counts off each of the time delays and then enables both sequencers to advance to the next position when the specified time has elapsed. Here we see a tremendous benefit of the SQO instruction's indexed memory addressing: the fact that the SQO instruction reads its bits from arbitrarily-specified memory addresses means we may use SQO instructions to sequence *any type of data existing in the PLC's memory!* We are not limited to turning on and off individual bits as we are with the Koyo drum instruction, but rather are free to index whole integer numbers, ASCII characters, or any other forms of binary data resident in the PLC's memory.

Data file windows appear on the computer screen showing the bit array held in the `B3` file as well as the timer values held in the `N7` file. In this live screenshot, we see both sequencer instructions at position 2, with the second SQO instruction having loaded a value of 15 seconds from register `N7:2` to the timer's preset register `T4:0.PRE`.

Note how the enabling contact address for the second SQO instruction is the "enable" bit of the first instruction, ensuring both instructions are enabled simultaneously. This keeps the two separate sequencers synchronized (on the same step).

Event-based transitions may be implemented in Allen-Bradley PLCs using a complementary sequencing instruction called SQC (Sequencer Compare). The SQC instruction is set up very similar to the SQO instruction, with an indexed file reference address to read from, a reserved memory structure for internal use, a set length, and a position value. The purpose of the SQC instruction is to read a data register and compare it against another data register, setting a "found" (FD) bit if the two match. Thus, the SQC instruction is ideally suited for detecting when certain conditions have been met, and thus may be used to enable an SQO instruction to proceed to the next step in its sequence.

The following program example shows an Allen-Bradley MicroLogix 1100 PLC programmed with both an SQO and an SQC instruction:



The three-position SQO (Sequencer Output) instruction reads data from B3:1, B3:2, and B3:3, writing the four least-significant of those bits to output register O:0.0. The three-position SQC (Sequencer Compare) instruction reads data from B3:6, B3:7, and B3:8, comparing the four least-significant of those bits against input bits in register I:0.0. When the four input bit conditions match the selected bits in the B3 file, the SQC instruction's FD bit is set, causing both the SQO instruction and the SQC instruction to advance to the next step.

Lastly, Allen-Bradley PLCs offer a third sequencing instruction called *Sequencer Load* (SQL), which performs the opposite function as the Sequencer Output (SQO). An SQL instruction takes data from a designated source and writes it into an indexed register according to a position count value, rather than reading data from an indexed register and sending it to a designated destination as does the SQO instruction. SQL instructions are useful for reading data from a live process and storing it in different registers within the PLC's memory at different times, such as when a PLC is used for *datalogging* (recording process data).

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 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.

4.1.4 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.

4.1.5 Standard counter instructions

- Allen-Bradley Logix 5000: `CTU` count-up, `CTD` count-down, and `CTUD` count-up/down instructions.

- Allen-Bradley SLC 500: `CTU` and `CTD` instructions.

- Siemens S7-200: `CTU` count-up, `CTD` count-down, and `CTUD` count-up/down instructions.

- Koyo (Automation Direct) DirectLogic: `UDC` counter instruction.

4.1.6 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.

4.1.7 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.

4.1.8 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.

4.1.9 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]:

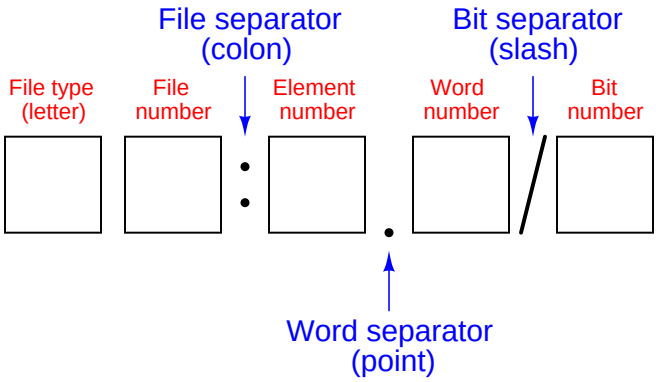
File number	File type	Logical address range
0	Output image	`O:0` to `O:30`
1	Input image	`I:0` to `I:30`
2	Status	`S:0` to `S:`n
3	Binary	`B3:0` to `B3:255`
4	Timers	`T4:0` to `T4:255`
5	Counters	`C5:0` to `C5:255`
6	Control	`R6:0` to `R6:255`
7	Integer	`N7:0` to `N7:255`
8	Floating-point	`F8:0` to `F8:255`
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].

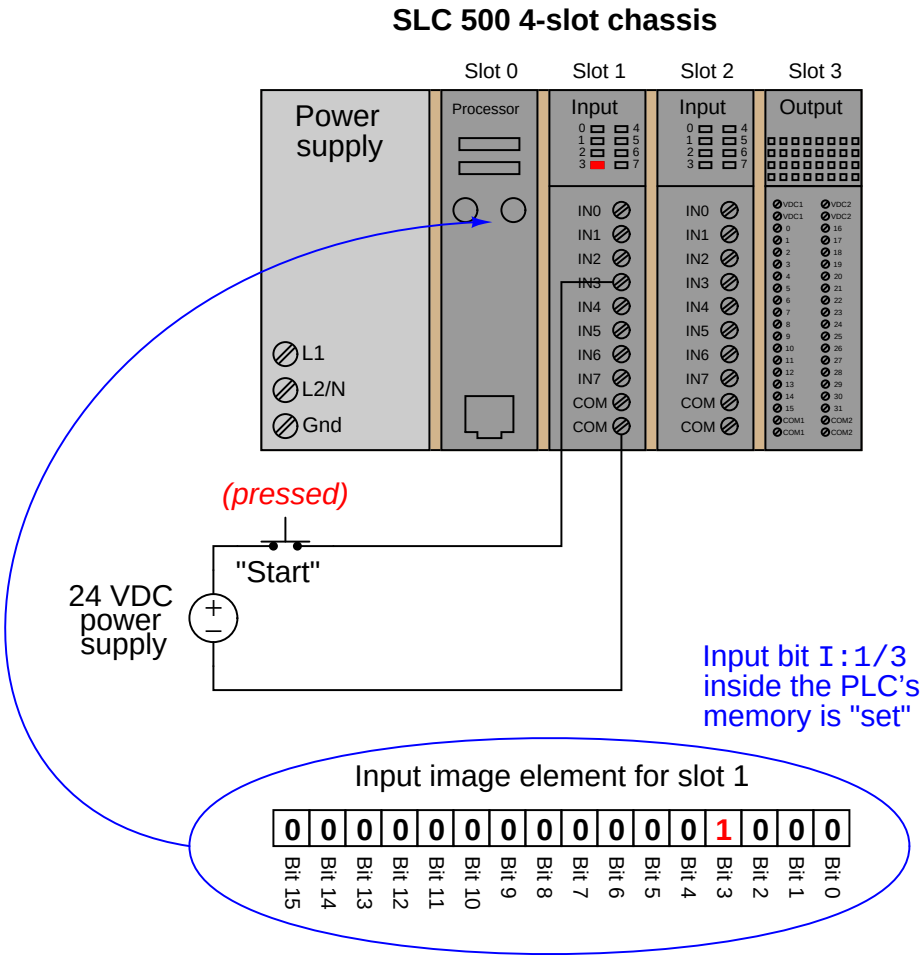
[3] This is not to say one *cannot* specify a particular bit in an otherwise whole word. In fact, this is one of the powerful advantages of Allen-Bradley's addressing scheme: it gives you the ability to precisely specify portions of data, even if that data is not generally intended to be portioned into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

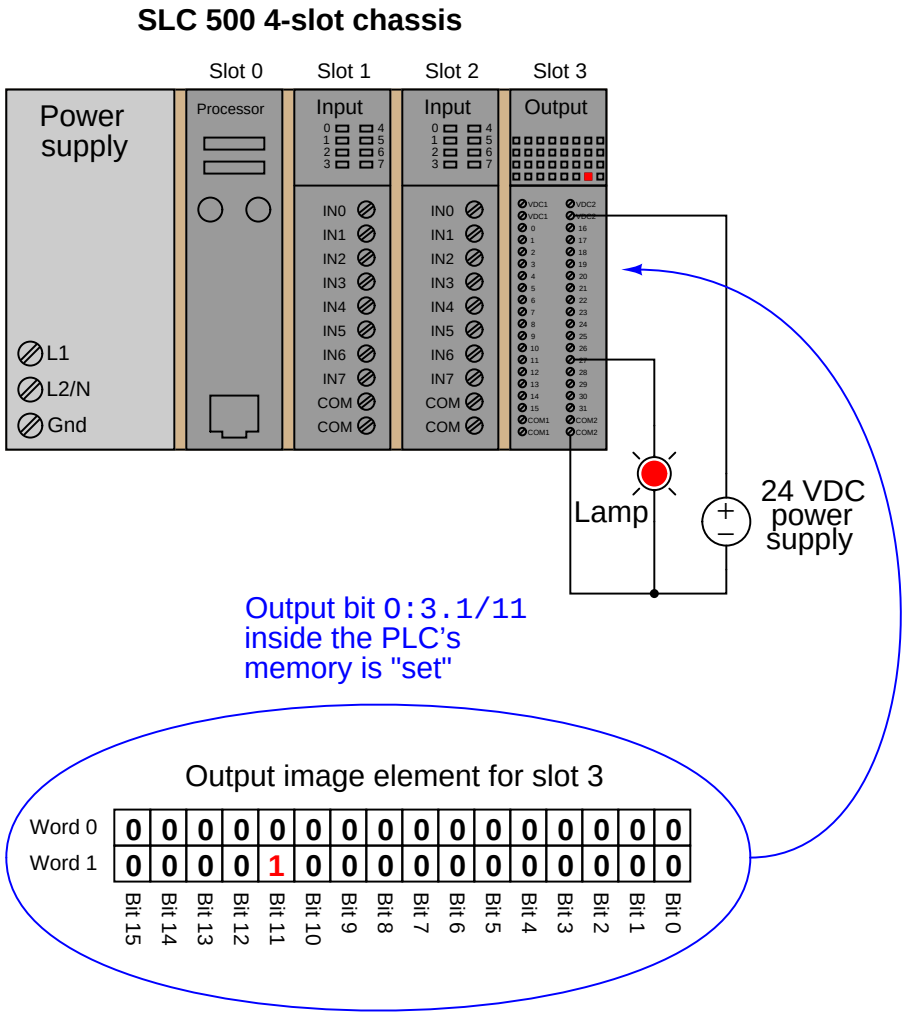
[5] Referencing the Allen-Bradley engineering literature, we see that the accumulator word may alternatively be addressed by number rather than by mnemonic, `T4:2.2` (word 2 being the accumulator word in the timer data structure), and that the "done" bit may be alternatively addressed as `T4:2.0/13` (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.

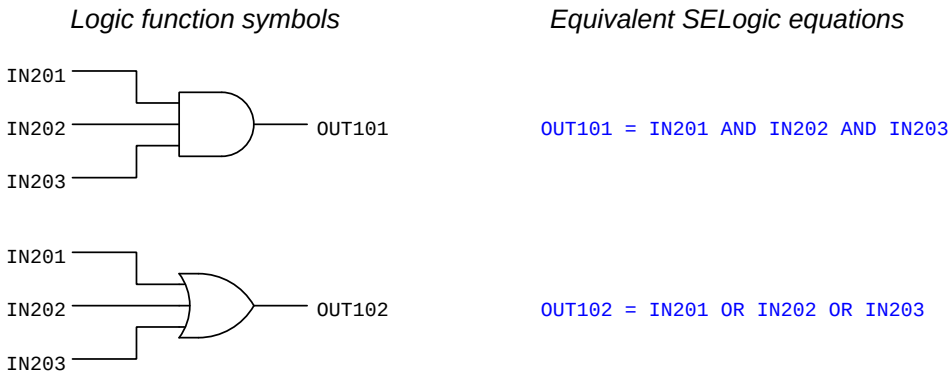
The modern trend in PLC addressing is to avoid the use of direct addresses such as `I:1/3` altogether, so they do not appear anywhere in the programming code. The Allen-Bradley "Logix" series of programmable logic controllers is the most prominent example of this new convention at the time of this writing. 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).

4.3 SELogic control equations

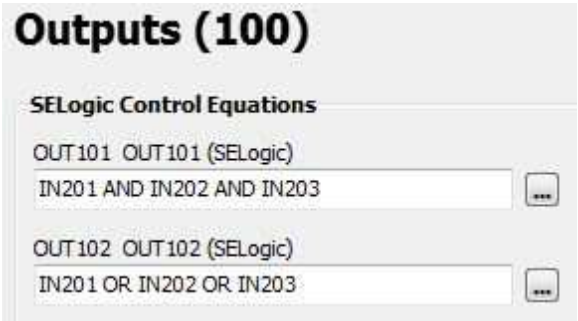
Some programmable logic controllers use Boolean-style equations to describe logical functions. A company called Schweitzer Engineering Laboratories (SEL) manufactures control equipment for the electric power industry, including a wide range of *protective relays* and *programmable automation controllers*, which use Boolean equations to describe logic. Their brand name for this is *SELogic*.

4.3.1 Basic logical functions

For example, suppose we wished to implement a three-input AND function as well as a three-input OR function in a SEL controller with the three inputs being `IN201`, `IN202`, and `IN203`. The symbolic functions and their equivalent SELogic equations are shown in the following illustration:



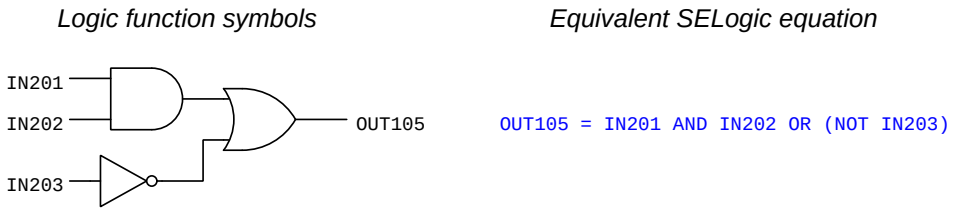
SEL programming software[7] provides text-entry fields for typing these control equations. A more primitive interface makes use of the built-in serial terminal server capability of SEL controllers, allowing programming edits to be made with nothing more than a serial terminal (e.g. personal computer running terminal emulator software such as `Termite` or `PuTTY` or `Hyperterminal`) on a command-line interface. The following screenshot shows these two SELogic control equations being edited in a window:



In this particular controller (an SEL-2440 "DPAC" Discrete Programmable Automation Controller) all inputs are numbered beginning with 201 and all outputs beginning with 101.

[7] SEL-5030 AcSELerator QuickSet software was used for all of these examples.

A simple combinational logic function is shown in this next illustration:



All SELogic equations support text *comments*, which is an important detail for any non-trivial coding. Comments are ignored by the controller, but serve as notes for any future programmers examining the code. In SELogic, comments are preceded by the hashtag symbol (**#**). For example, here is a comment as it would appear following an SELogic equation for a simple AND function:

```
OUT102 = IN205 AND IN208 # THIS IS AN AND FUNCTION
```

Characters to the left of the **#** are regarded as executable code by the controller, while text to the right of the **#** are merely comments.

4.3.2 Set-reset latch instructions

Set-Reset (or S-R) latch instructions are useful for applications such as motor starter controls, where the controller must "latch" the motor on after momentary closure of the "Start" pushbutton switch, and latch the motor off following momentary actuation of the "Stop" pushbutton. PLCs programmed in Ladder Diagram code typically use either *retentive coils* ("Set" and "Reset" coils) or seal-in contacts "wired" in parallel with the Start contact instruction. SELogic provides a dedicated function for latching called a *Latch Bit*.

Latch instructions in SELogic require a number of different parameters to be specified. First and foremost is the allocation of usable latch instructions in the controller's memory. Like legacy-style industrial PLCs with limited memory resources, the programmer first needs to declare to the PLC how many counters, timers, etc. will be used in the program so that the PLC may allocate limited memory to those functions. For SELogic latches, this is done through the `ELAT` parameter. For example the equation `ELAT = 3` allocates three latches which are called `LT01`, `LT02`, and `LT03`. After that, each latch requires assignment of its *Set* (SET) and `Reset` (RST) inputs. A screenshot showing the Set and Reset inputs of a single latch instruction appears here:

Listed as plain ASCII text, the SELogic equations would appear as follows to declare a single latch instruction and then configure its two inputs as shown in the previous screenshot, where `IN201` causes the latch to set and `IN202` causes the latch to reset:

```
ELAT = 1
SET01 = IN201
RST01 = IN202
```

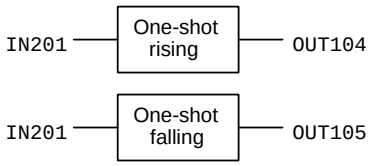
In SELogic, the name of the latch (e.g. `LT01`) is its output bit. Using the example just shown, we could direct output `OUT108` to be controlled by `LT01`'s output with an additional line of SELogic code:

```
OUT108 = LT01
```

4.3.3 One-shot instructions

One-shot functionality also is provided within SELogic by the `R_TRIG` and `F_TRIG` instructions, referring to *rising-edge* and *falling-edge*, respectively. The purpose of a "one-shot" instruction is to activate the output for a single scan of the controller's program upon a false-to-true (rising edge) or true-to-false (falling edge) transition of the input signal. This next illustration shows examples of each:

Logic function symbols



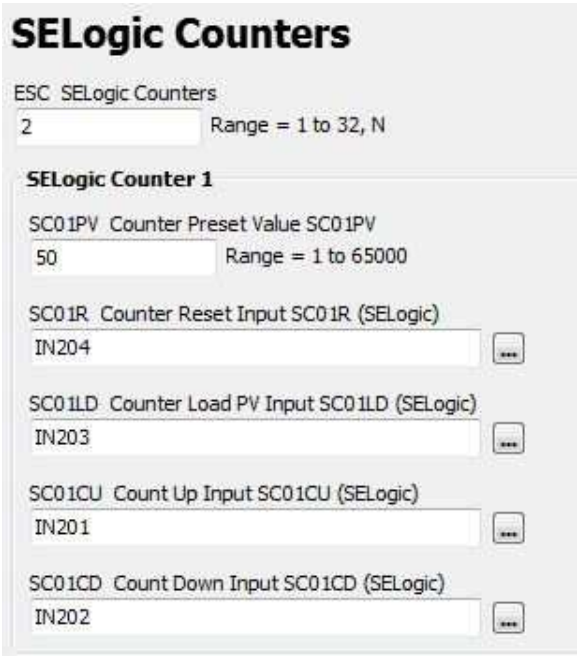
Equivalent SELogic equations

```
OUT104 = R_TRIG IN201

OUT105 = F_TRIG IN201
```

4.3.4 Counter instructions

Counter instructions in SELogic require a number of different parameters to be specified. First and foremost is the allocation of usable counter instructions in the controller's memory. Like legacy-style industrial PLCs with limited memory resources, the programmer first needs to declare to the PLC how many counters, timers, etc. will be used in the program so that the PLC may allocate limited memory to those functions. For SELogic counters, this is done through the `ESC` parameter. For example the equation `ESC = 2` allocates two counters which are called `SC01` and `SC02`. After that, each counter requires assignment of input variables such as *Preset Value* (PV), `Reset` (R), `Load PV` (LD), *Count up input* (CU) and `Count down input` (CD). A screenshot showing all of these variables (enabling two counters, followed by the parameters for Counter 1) appears here:



Listed as plain ASCII text, the SELogic equations would appear as follows to declare two counter instructions and then configure the first counter's inputs: a preset value of 50, `IN204` causing it to reset, `IN203` causing it to load the preset value, `IN201` causing it to increment, and `IN202` causing it to decrement:

```
ESC = 2
SC01PV = 50
SC01R = IN204
SC01LD = IN203
SC01CU = IN201
SC02CD = IN202
```

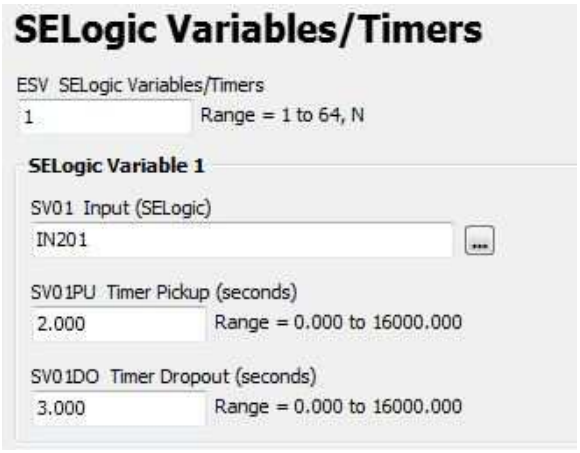
Once configured, the accumulated value of the first counter instruction is addressed simply as `SC01`. Discrete outputs based on the attainment of certain count values may be driven by comparison statements such as `=`, `>`, and `<`. For example, to activate output `OUT107` whenever the first counter's accumulated value exceeds 12, you would need to write the following SELogic equation:

```
OUT107 = SC01 > 12
```

The comparison statement `SC01 > 12` is either true or false – that is to say, it is a *Boolean* quantity – and so its truth-value fits well with the discrete status of output `OUT107`. When the counter's value exceeds 12, `OUT107` activates; if equal to or less than 12, `OUT107` de-activates.

4.3.5 Timer instructions

Like counter instructions, *timer* instructions in SELogic require a number of different parameters to be specified. First and foremost is the allocation of usable timer instructions in the controller's memory. Like legacy-style industrial PLCs with limited memory resources, the programmer first needs to declare to the PLC how many counters, timers, etc. will be used in the program so that the PLC may allocate limited memory to those functions. For SELogic timers, this is done through the `ESV` parameter. For example the equation `ESV = 1` allocates memory space for a single timer called `SV01`. After that, the timer requires assignment of input variables such as *Timer Pickup* (PU), *Timer Dropout* (DO), and *Input*. A screenshot showing all of these variables for a single timer appears here:



The SELogic timer instruction is capable of both on-delay and off-delay. This is the meaning of the "pickup" and "dropout" time delays: the *pickup* time refers to a time delay on activation of the timer (i.e. on-delay) while the *dropout* time refers to a time delay on de-activation (i.e. off-delay). For example, if we only wished to have an on-delay `SV01` timer with a delay time of 5 seconds we would set `SV01PU = 5` and `SV01DO = 0`. If we merely wished for an off-delay timer with a delay time of 8 seconds, we would set `SV01PU = 0` and `SV01DO = 8`. If we wanted a timer to exhibit *both* an on-delay of 2 seconds *and* an off-delay of 3 seconds, we would set the pickup and dropout parameters exactly as shown in the above screenshot image. Note the resolution of these time settings: down to 0.001 seconds, or 1 millisecond.

In SELogic, the name of the timer (e.g. `SV01`) is its input, or controlling, bit. The time-delayed output bit of the timer is addressed by adding the suffix "T" to the timer name. Using our example of timer `SV01`, its time-delayed output bit would be `SV01T`. Below is an example of this timer's output being set to control output `OUT105`:

```
OUT105 = SV01T
```

Chapter 5

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?

5.1 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.

5.1.1 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.

5.1.2 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.

???

???

???

???

???

???

???

???

???

???

???

5.2 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.

5.2.1 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.

5.2.2 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.*

5.3 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.

Appendix A

Problem-Solving Strategies

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

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

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

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

Appendix B

Instructional philosophy

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

To high standards of education,

Tony R. Kuphaldt

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

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

Appendix C

Tools used

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

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

The `GNU/Linux` computer operating system

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

Bram Moolenaar's `Vim` text editor

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

Donald Knuth's TEX typesetting system

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

Leslie Lamport's LATEX extensions to TEX

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

Tim Edwards' `Xcircuit` drafting program

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

`Gimp` graphic image manipulation program

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

`SPICE` circuit simulation program

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

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

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

`gnuplot` mathematical visualization software

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

`Python` programming language

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

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

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

Section 1 – Definitions.

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

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

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

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

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

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

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

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

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

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

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

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

Section 2 – Scope.

a. License grant.

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

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

B. produce, reproduce, and Share Adapted Material.

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

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

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

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

5. Downstream recipients.

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

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

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

b. Other rights.

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

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

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

Section 3 – License Conditions.

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

a. Attribution.

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

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

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

ii. a copyright notice;

iii. a notice that refers to this Public License;

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

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

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

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

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

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

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

Section 4 – Sui Generis Database Rights.

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

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

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

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

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

Section 5 – Disclaimer of Warranties and Limitation of Liability.

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

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

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

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

Section 6 – Term and Termination.

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

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

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

2. upon express reinstatement by the Licensor.

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

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

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

Section 7 – Other Terms and Conditions.

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

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

Section 8 – Interpretation.

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

be made without permission under this Public License.

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

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

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

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

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

Appendix E

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.

20 May 2025 – document first created, taking content from the legacy "Introduction to PLCs" (mod_plc) module.

Index