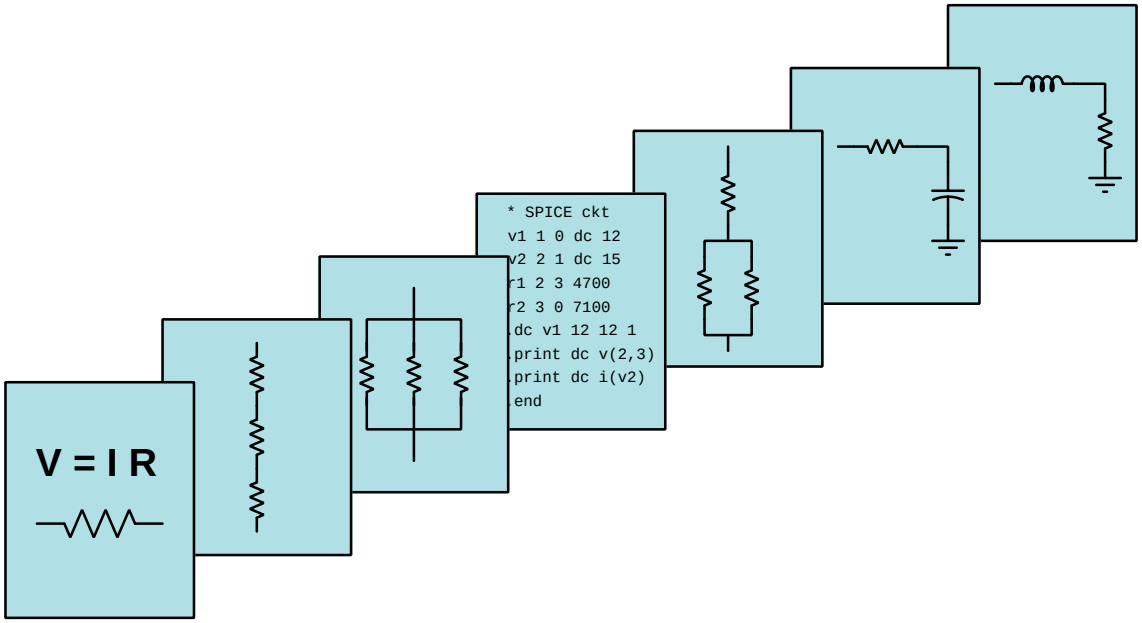
Modular Electronics Learning (ModEL) project



PLC Contact and Coil Programming

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

Last update = 10 February 2026

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

Contents

1 Introduction **3**
- 1.1 Recommendations for students . . . 3
- 1.2 Challenging concepts related to programmable logic controllers (PLCs) . . . 5
- 1.3 Recommendations for instructors . . . 6

2 Case Tutorial **9**
- 2.1 Example: NAND function in a PLC . . . 10
- 2.2 Example: simple PLC comparisons . . . 12
- 2.3 Example: high-pressure PLC-based alarm . . . 15
- 2.4 Example: PLC output statuses from process switch stimuli . . . 20
- 2.5 Example: PLC process switch statuses from contact status coloring . . . 22
- 2.6 Example: Arduino versus PLC dual-LED control . . . 25
- 2.7 Example: Arduino versus PLC logic functions . . . 27
- 2.8 Example: Arduino versus PLC on-delay timer . . . 29

3 Tutorial **31**
- 3.1 Review of basic PLC functionality . . . 31
- 3.2 Intro to Ladder Diagram (LD) programming . . . 39
- 3.3 Ladder diagram virtual elements and I/O status . . . 42
- 3.4 Ladder diagram latching functions . . . 45
- 3.5 Fault-tolerant latch programming . . . 51
- 3.6 Ladder diagram retentive coils . . . 60
- 3.7 Ladder diagram transition contacts . . . 62
- 3.8 Ladder diagram logic functions . . . 65
- 3.9 Ladder diagram combinational logic . . . 68
- 3.10 Ladder logic for arbitrary truth tables . . . 72

4 Derivations and Technical References **75**
- 4.1 Feature comparisons between PLC models . . . 76
  - 4.1.1 Viewing live values . . . 76
  - 4.1.2 Forcing live values . . . 77
  - 4.1.3 Single-bit addressing . . . 77
  - 4.1.4 Special “system” values . . . 78
  - 4.1.5 Free-running clock pulses . . . 78

4.1.6 Standard counter instructions . . . 78
4.1.7 High-speed counter instructions . . . 79
4.1.8 Timer instructions . . . 79
4.1.9 ASCII text message instructions . . . 79
4.1.10 Analog signal scaling . . . 80
4.2 Getting started with Rockwell Studio 5000 . . . 81
4.3 Legacy Allen-Bradley memory maps and I/O addressing . . . 87
4.4 SELogic control equations . . . 93
4.4.1 Basic logical functions . . . 94
4.4.2 Set-reset latch instructions . . . 96
4.4.3 One-shot instructions . . . 97
4.4.4 Counter instructions . . . 98
4.4.5 Timer instructions . . . 100
4.5 Process switch types and symbols . . . 101

5 Questions **103**
5.1 Conceptual reasoning . . . 107
5.1.1 Reading outline and reflections . . . 108
5.1.2 Foundational concepts . . . 109
5.1.3 Relay ladder logic analogy for a PLC . . . 110
5.1.4 Determining bit statuses from switch conditions . . . 113
5.1.5 Determining color highlighting from bit statuses . . . 118
5.1.6 Determining necessary switch conditions for bit statuses . . . 121
5.1.7 Determining color highlighting from switch conditions . . . 122
5.1.8 Two different motor control programs . . . 124
5.1.9 Redundant coils in a program . . . 126
5.1.10 Determining bit statuses from color highlighting . . . 127
5.1.11 Determining process switch stimuli from color highlighting . . . 131
5.1.12 Determining truth table from PLC program . . . 133
5.1.13 Design a PLC program to spec . . . 135
5.1.14 Determining necessary switch contact types . . . 137
5.1.15 PLC programs from truth tables . . . 139
5.2 Diagnostic reasoning . . . 142
5.2.1 Troubleshooting motor control program . . . 143
5.2.2 Troubleshooting motor control PLC from I/O indicators . . . 144
5.2.3 Troubleshooting motor control PLC using voltmeter . . . 145
5.2.4 Turbine low-oil trip . . . 147
5.2.5 Motor starter diagnosis from color highlighting . . . 149
5.2.6 Air compressor troubleshooting from observation . . . 152
5.2.7 Possible faults in a PLC/HMI pump control system . . . 154
5.2.8 Diagnostic tests on a failed PLC/HMI pump control system . . . 156

A Problem-Solving Strategies **157**

B Instructional philosophy **159**
B.1 First principles of learning 160
B.2 Proven strategies for instructors 161
B.3 Proven strategies for students 163
B.4 Design of these learning modules 164

C Tools used **167**

D Creative Commons License **171**

E Version history **179**

Index **179**

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.

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

- **Outcome** – 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.

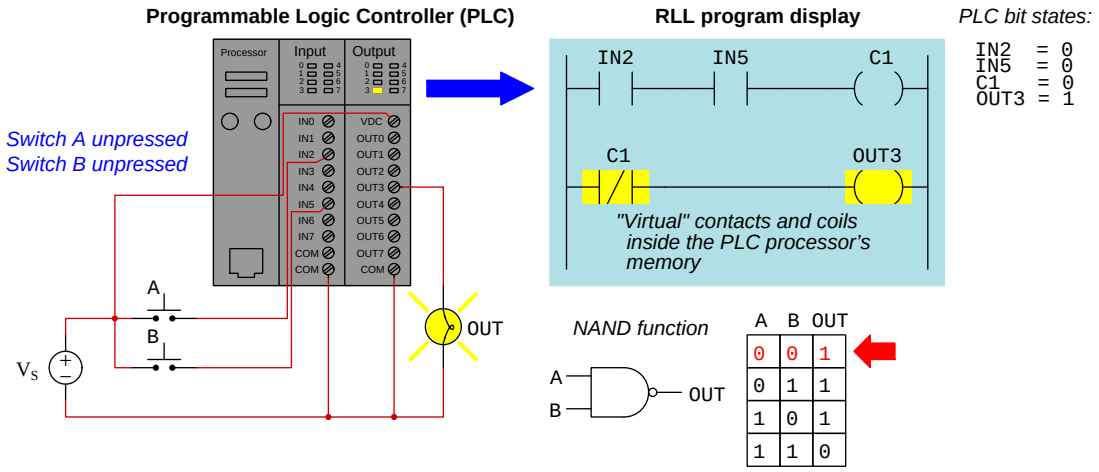
Chapter 2

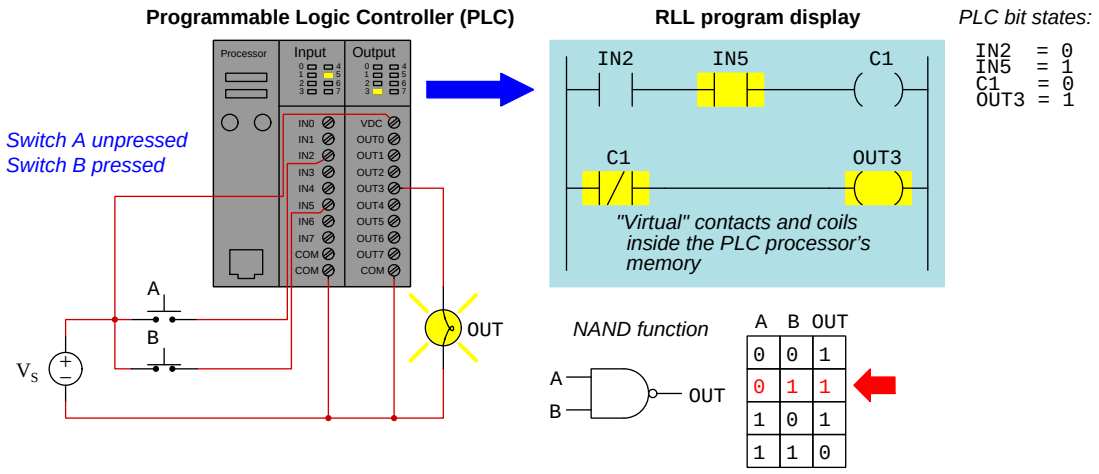
Case Tutorial

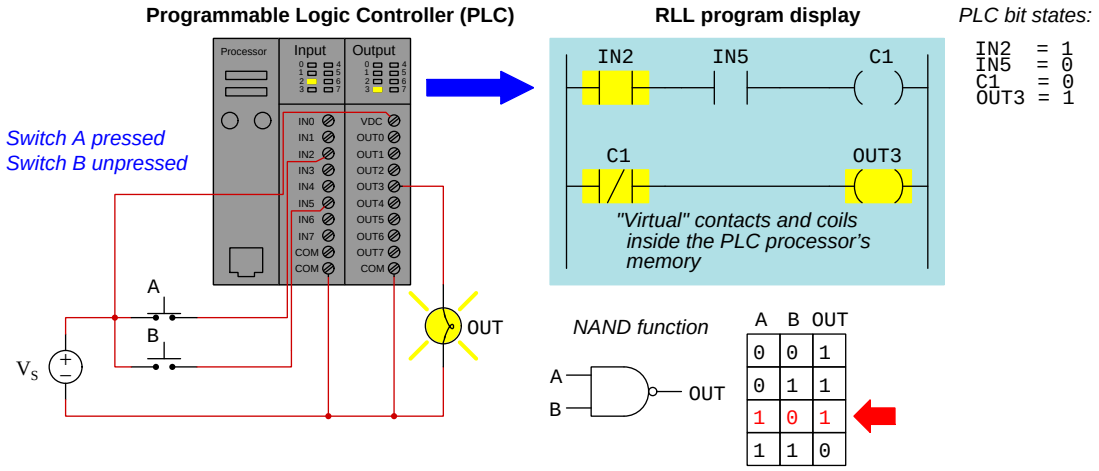
The idea behind a *Case Tutorial* is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as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

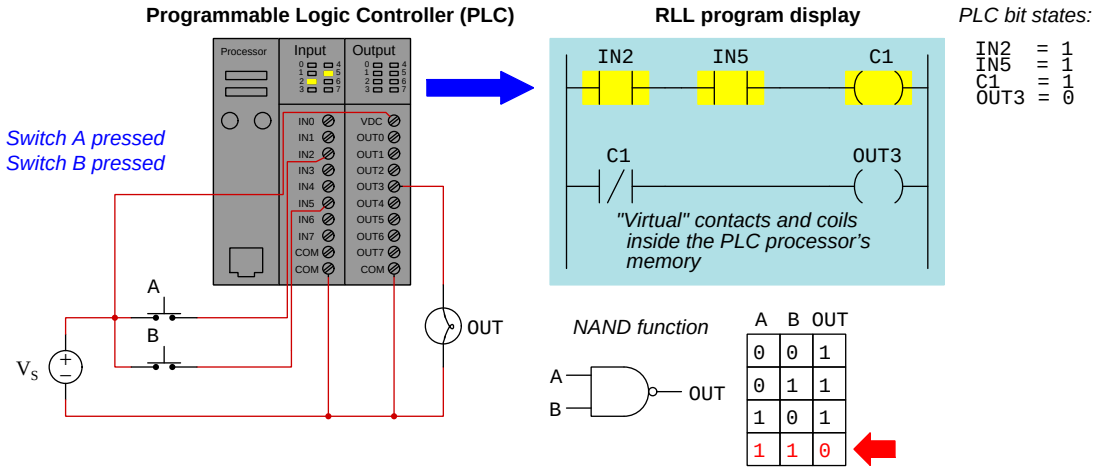
These examples also serve well as challenges following your reading of the other Tutorial(s) in this module – can you explain *why* the circuits behave as they do?

2.1 Example: NAND function in a PLC



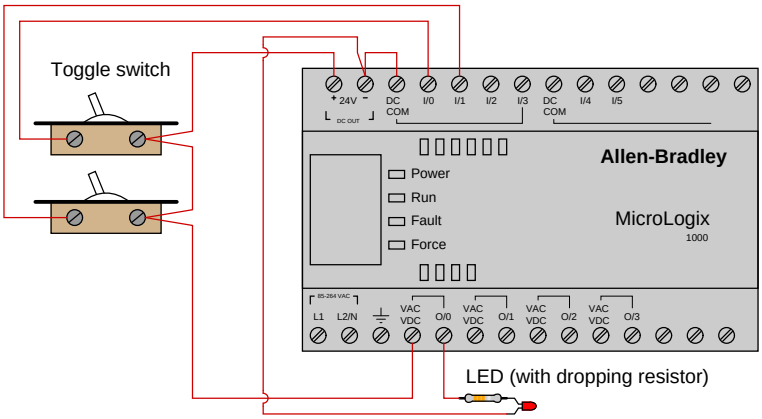



Programmable Logic Controller (PLC)
RLL program display
PLC bit states:
Processor
Input
Output
Switch A pressed
Switch B unpressed
A
B
OUT
IN2
IN5
C1
C1
OUT3
"Virtual" contacts and coils inside the PLC processor's memory
IN2 = 1
IN5 = 0
C1 = 0
OUT3 = 1
NAND function
A B OUT
0 0 1
0 1 1
1 0 1
1 1 0

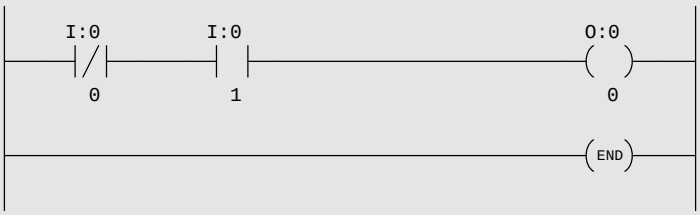

Programmable Logic Controller (PLC)
RLL program display
PLC bit states:
Processor
Input
Output
Switch A pressed
Switch B pressed
A
B
OUT
IN2
IN5
C1
C1
OUT3
"Virtual" contacts and coils inside the PLC processor's memory
IN2 = 1
IN5 = 1
C1 = 1
OUT3 = 0
NAND function
A B OUT
0 0 1
0 1 1
1 0 1
1 1 0

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:



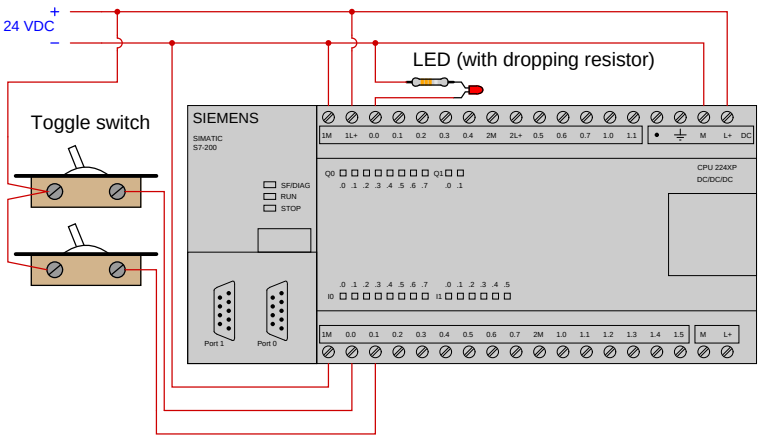
Ladder-Diagram program written to PLC:



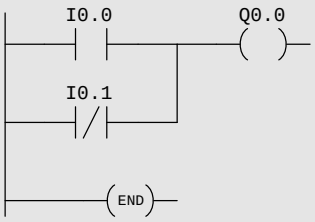
Note how Allen-Bradley I/O is labeled in the program: input bits designated by the letter `I` and output bits designated by the letter `O`.

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:



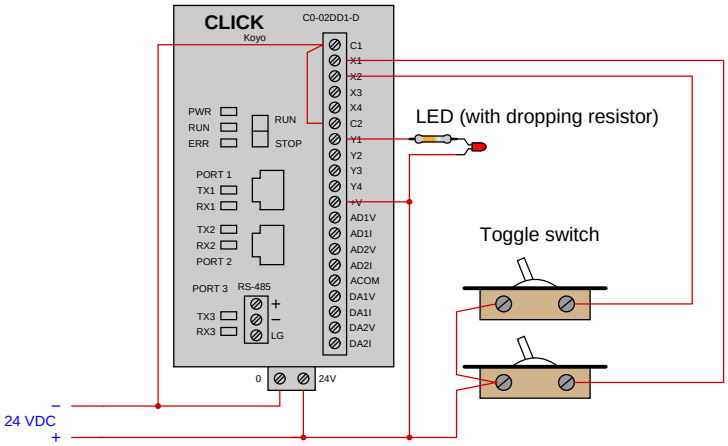
Ladder-Diagram program written to PLC:



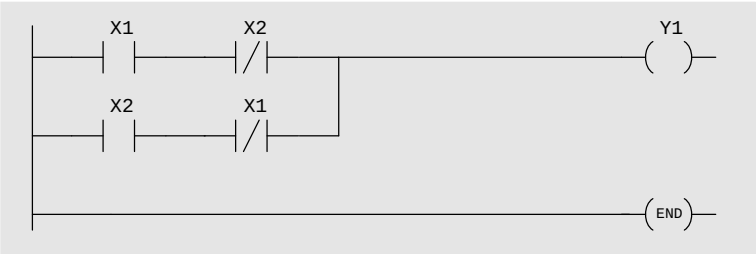
Note how Siemens I/O is labeled in the program: input bits designated by the letter `I` and output bits designated by the letter `Q`.

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`.

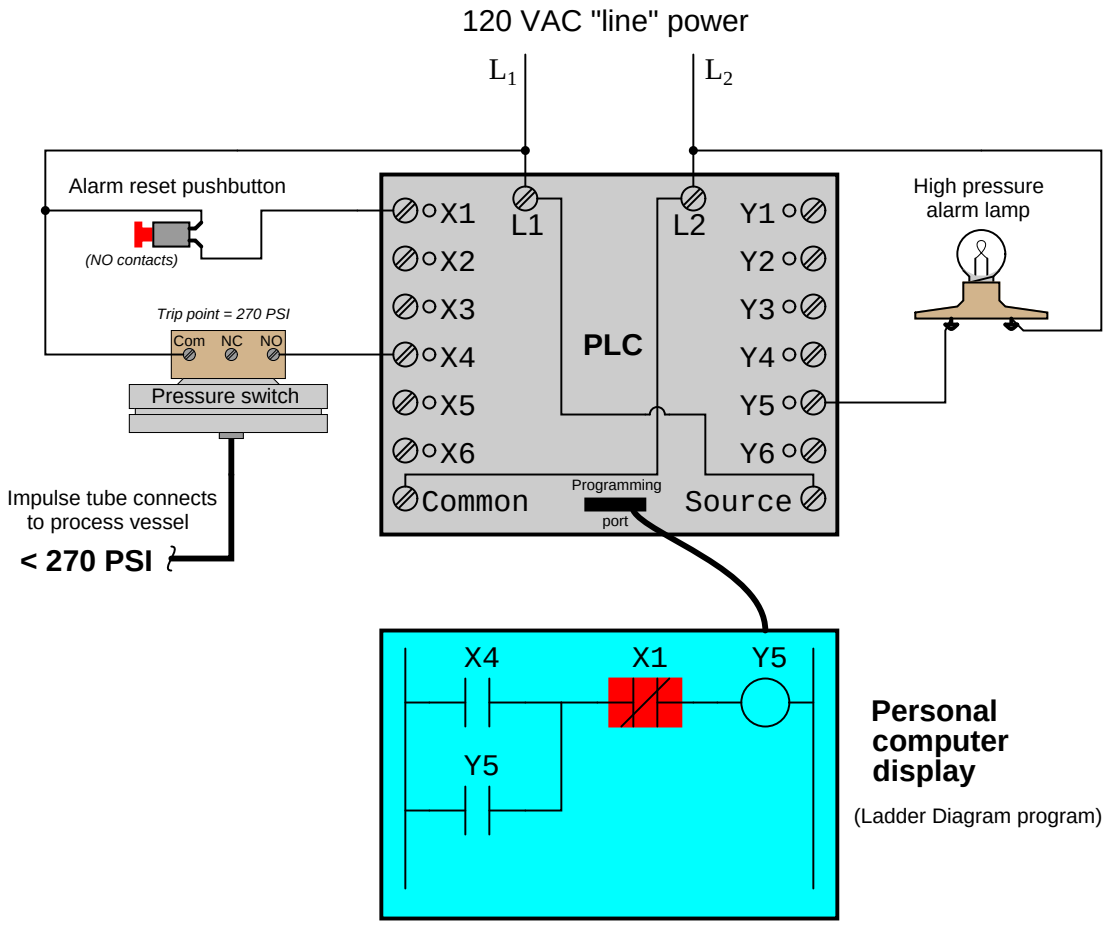
In order to energize the LED, at least one of the following conditions must be met:

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

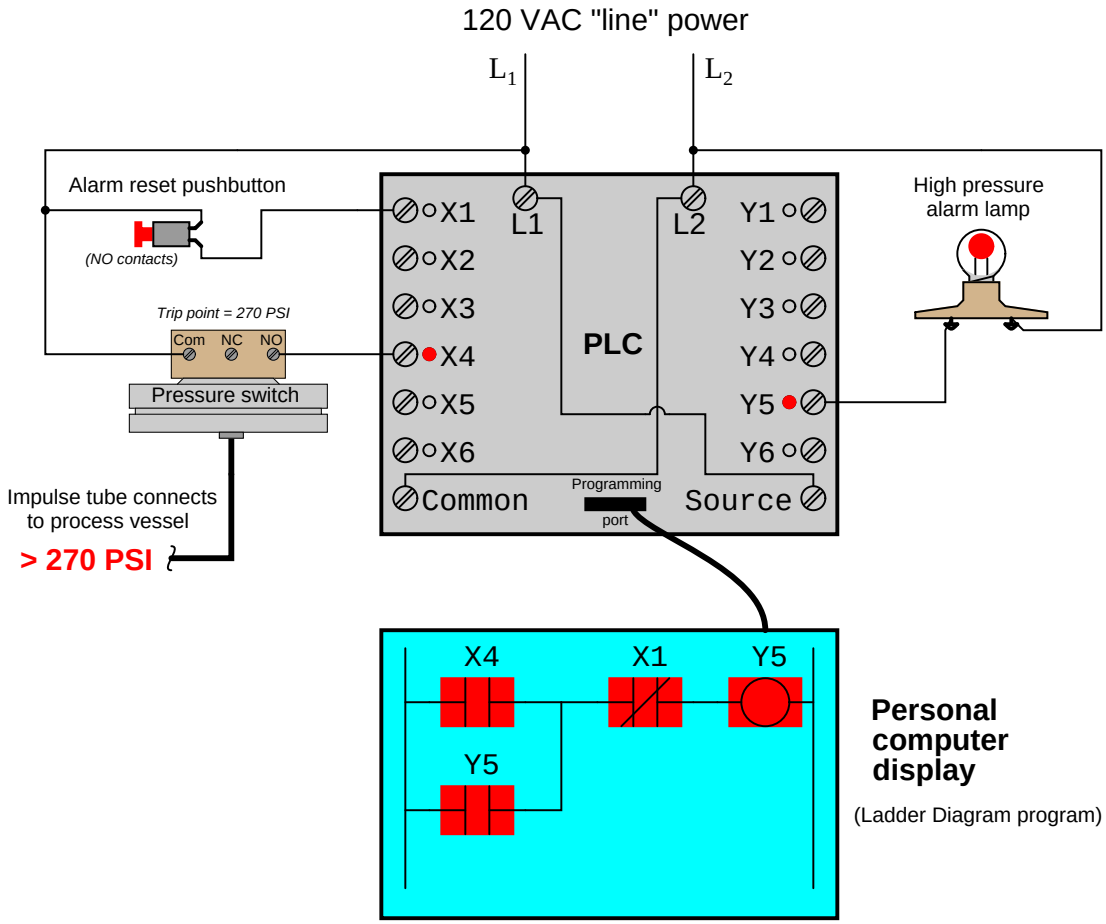
Either the switch connected to input terminal 0.0 must be on (closed) or the switch connected to input terminal 0.1 must be off (open).

2.3 Example: high-pressure PLC-based alarm

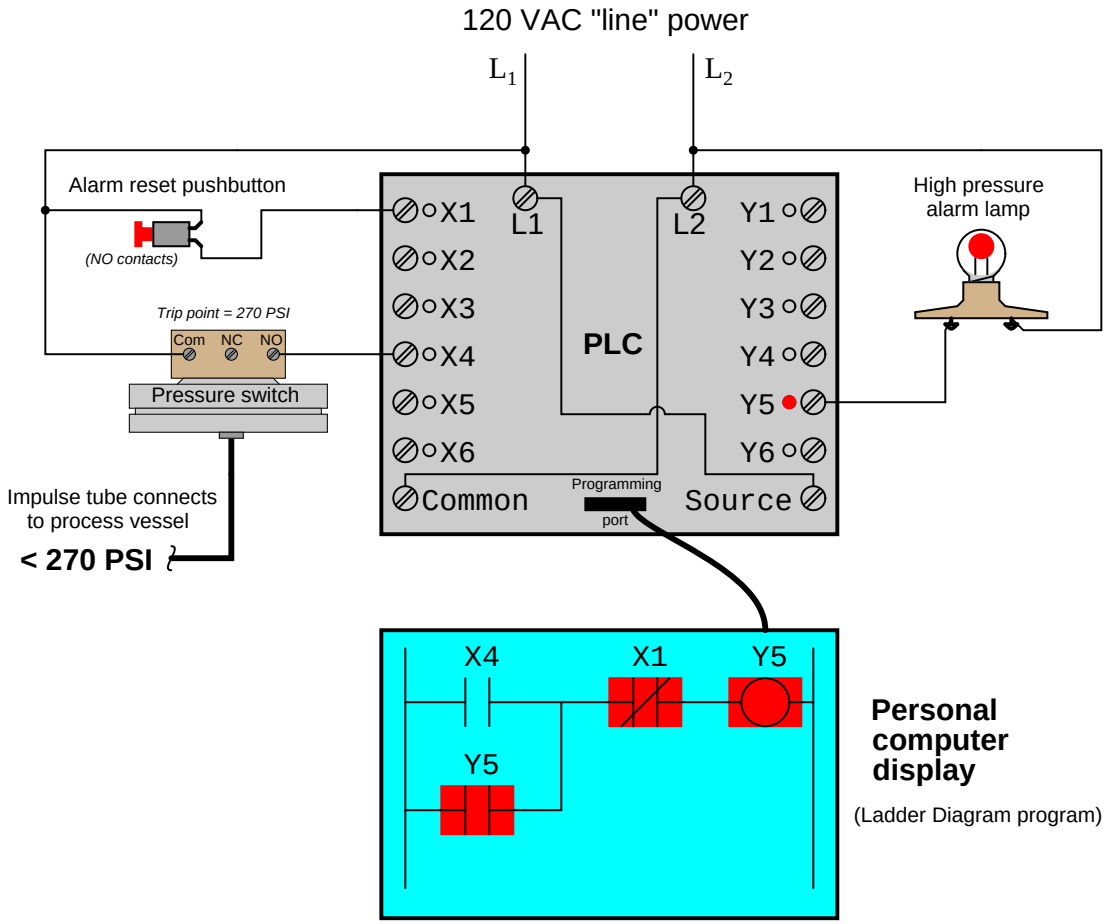
Step #1 – fluid pressure inside the process vessel is low, and the alarm lamp is de-energized:



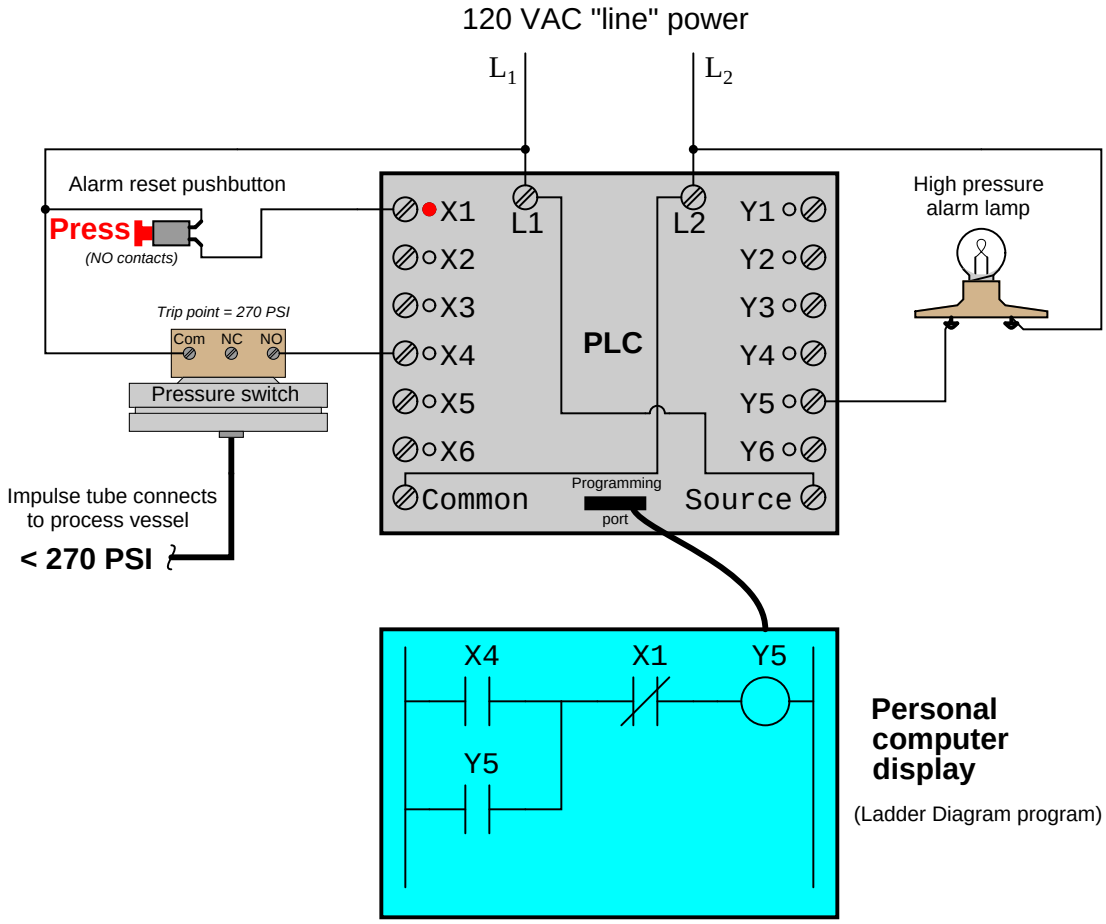
Step #2 – the pressure switch detects a high fluid pressure condition, triggering the PLC to energize the alarm lamp:



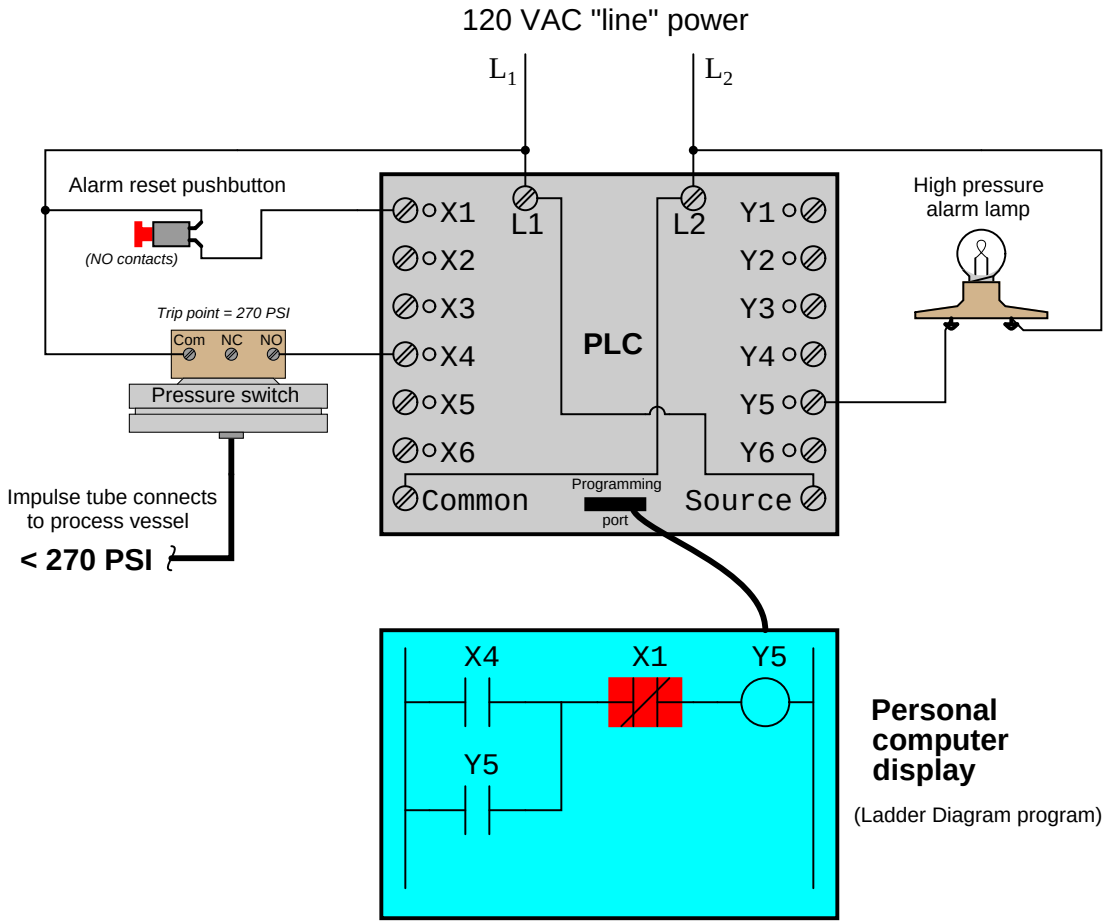
Step #3 – fluid pressure decreases below the pressure switch's trip point, but the PLC maintains power to the alarm lamp to signify that a high-pressure event happened:



Step #4 – a human operator presses the "Alarm reset" pushbutton to de-energize the alarm lamp:

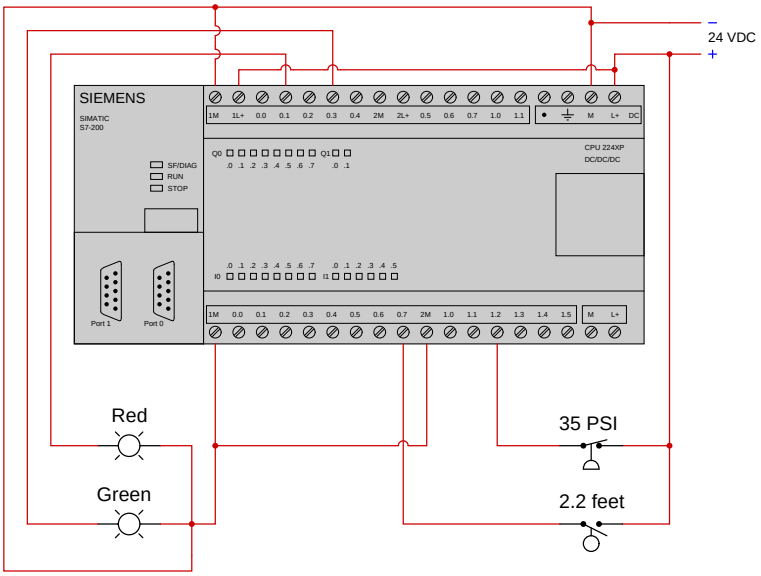


Step #5 – the lamp remains de-energized, awaiting another high-pressure event before energizing again:

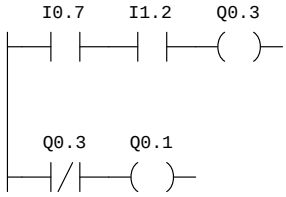


2.4 Example: PLC output statuses from process switch stimuli

Suppose we have a Siemens S7-200 PLC connected to a pair of process switches and lamps as shown in this illustration:



We happen to know that the fluid pressure is 30 PSI and the fluid level is 4 feet. The PLC's program is as follows:



With pressure being *below* the switch's trip point, we know that pressure switch will be in its "normal" (resting) state which happens to be *closed*. This closed status energizes input `I1.2`, setting that bit inside the PLC's memory to a value of one (1). This 1 bit status actuates the normally-open `I:1.2` contact instruction, causing it to close (i.e. become colored).

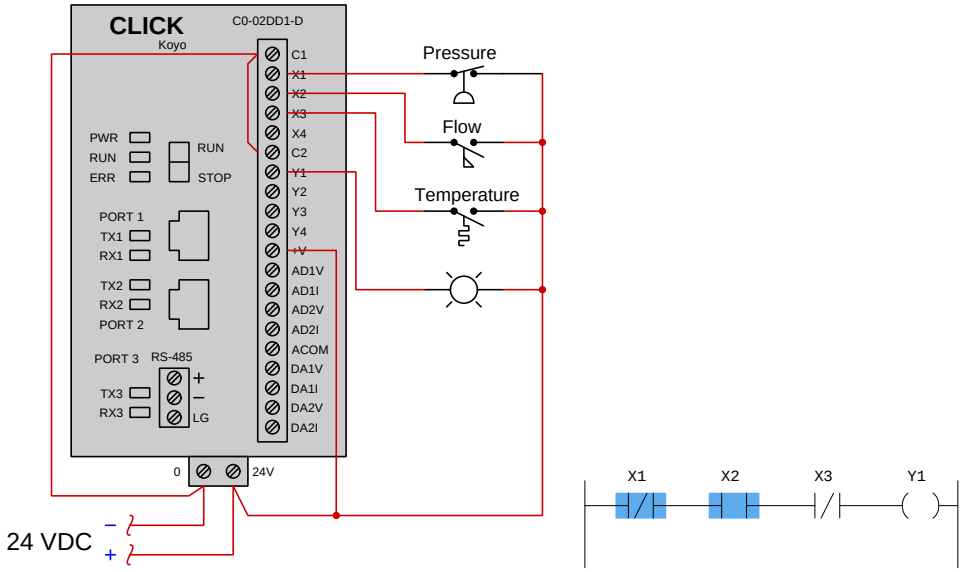
With level being *above* the switch's trip point, we know that level switch will be in its "actuated" (non-resting) state which happens to be *closed*. This closed status energizes input `I0.7`, setting that bit inside the PLC's memory to a value of one (1). This 1 bit status actuates the normally-open `I:0.7` contact instruction, causing it to close (i.e. become colored).

With both of those PLC contact instructions in the top rung being colored, color makes it to

coil `Q0.3`, setting that bit to a 1 state. Discrete output `Q0.3` becomes energized, causing the Green lamp to energize as well. The same 1 bit status of `Q0.3` prompts the normally-closed `Q0.3` contact instruction to open (i.e. become un-colored), thereby preventing color from reaching coil `Q0.1`. This de-energizes output `Q0.1` and makes the Red lamp de-energized.

2.5 Example: PLC process switch statuses from contact status coloring

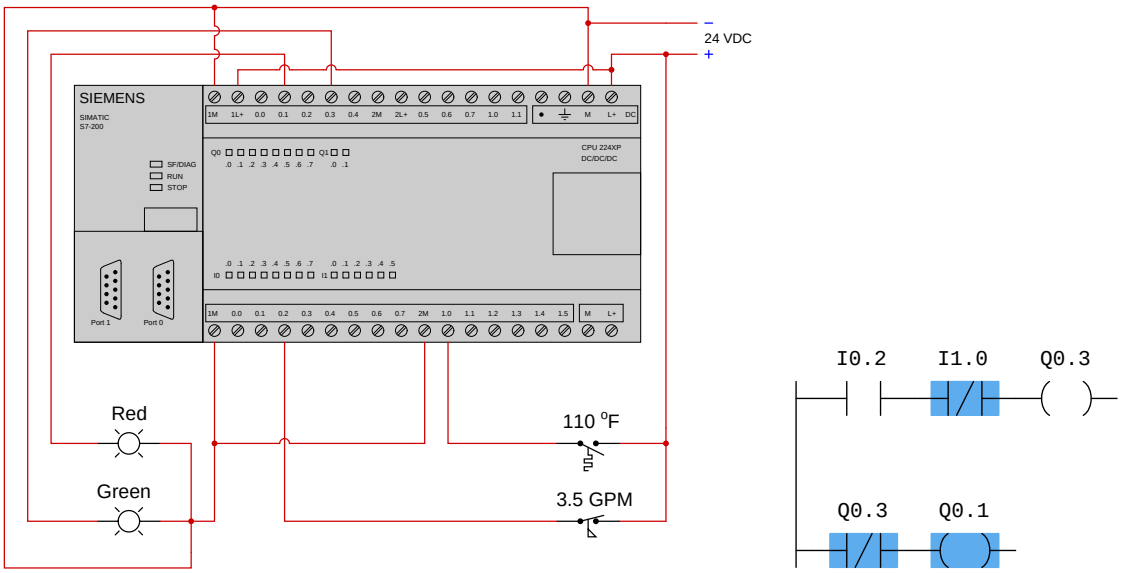
The following illustration shows a Koyo "CLICK" PLC connected to three process switches and one lamp, with the live status of each PLC instruction shown as well:



Based on the displayed color highlighting of the PLC program's virtual contacts, we may conclude the following:

- The `X1` contact instruction is colored which means it is closed. It is also drawn as normally-closed, and for a normally-closed PLC contact instruction to be closed the bit controlling it must be a zero (0). This means the `X1` input must be de-energized, which in turn means the pressure switch contact must be open. Since the pressure switch is normally-closed, in order to be in the open state we must have a **high pressure** condition.
- The `X2` contact instruction is colored which means it is closed. It is drawn as normally-open, and for a normally-open PLC contact instruction to be closed the bit controlling it must be a one (1). This means the `X2` input must be energized, which in turn means the flow switch contact must be closed. Since the flow switch is normally-open, in order to be in the closed state we must have a **high flow** condition.
- The `X3` contact instruction is un-colored which means it is open. It is drawn as normally-closed, and for a normally-closed PLC contact instruction to be open the bit controlling it must be a one (1). This means the `X3` input must be energized, which in turn means the temperature switch contact must be closed. Since the temperature switch is normally-open, in order to be in the closed state we must have a **high temperature** condition.
- No color reaches the `Y1` coil instruction which means its bit state will be a zero (0). This makes the `Y1` output de-energized, ensuring the lamp will be de-energized as well.

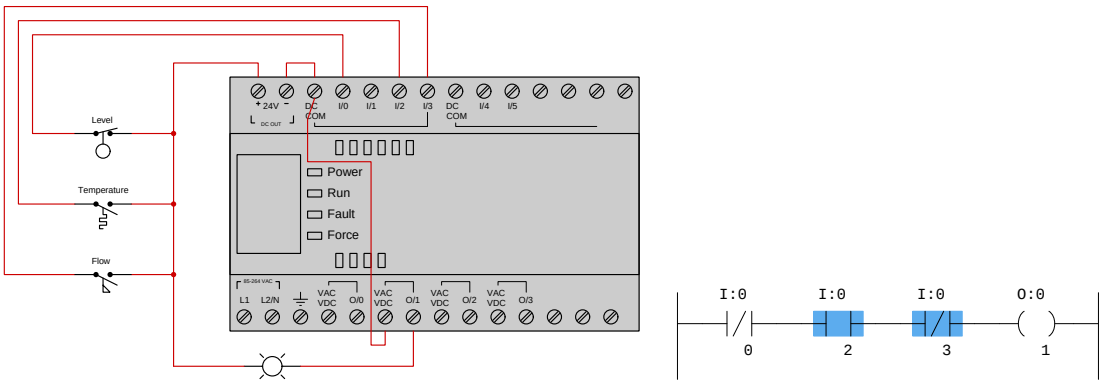
The following illustration shows a Siemens S7-200 PLC connected to two process switches and two lamps, with the live status of each PLC instruction shown as well:



Based on the displayed color highlighting of the PLC program's virtual contacts, we may conclude the following:

- The `I0.2` contact instruction is un-colored which means it is open. It is also drawn as normally-open, and for a normally-open PLC contact instruction to be open the bit controlling it must be a zero (0). This means the `I0.2` input must be de-energized, which in turn means the flow switch contact must be open. Since the flow switch is normally-closed, in order to be in the open state we must have a **high flow** condition.
- The `I1.0` contact instruction is colored which means it is closed. It is also drawn as normally-closed, and for a normally-closed PLC contact instruction to be closed the bit controlling it must be a zero (0). This means the `I1.0` input must be de-energized, which in turn means the temperature switch contact must be open. Since the temperature switch is normally-open, in order to be in the open state we must have a **low temperature** condition.
- No color reaches the `Q0.3` coil instruction which means its bit state will be a zero (0). This makes the `Q0.3` output de-energized, ensuring the Green lamp will be de-energized as well.
- Color does reach the `Q0.1` coil instruction which means its bit state will be a one (1). This makes the `Q0.1` output energized, ensuring the Red lamp will be energized as well.

The following illustration shows an Allen-Bradley MicroLogix PLC connected to three process switches and a lamp, with the live status of each PLC instruction shown as well:



Based on the displayed color highlighting of the PLC program's virtual contacts, we may conclude the following:

- The `I:0/0` contact instruction is un-colored which means it is open. It is drawn as normally-closed, and for a normally-closed PLC contact instruction to be open the bit controlling it must be a one (1). This means the `I:0/0` input must be energized, which in turn means the level switch contact must be closed. Since the level switch is normally-closed, in order to be in the closed state we must have a **low level** condition.

- The `I:0/2` contact instruction is colored which means it is closed. It is drawn as normally-open, and for a normally-open PLC contact instruction to be closed the bit controlling it must be a one (1). This means the `I:0/2` input must be energized, which in turn means the temperature switch contact must be closed. Since the temperature switch is normally-open, in order to be in the closed state we must have a **high temperature** condition.

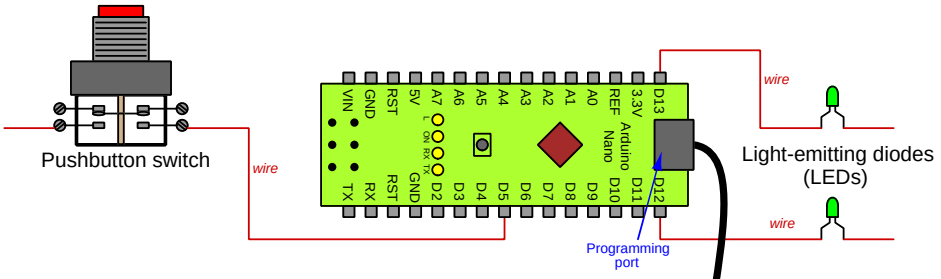
- The `I:0/3` contact instruction is colored which means it is closed. It is also drawn as normally-closed, and for a normally-closed PLC contact instruction to be closed the bit controlling it must be a zero (0). This means the `I:0/3` input must be de-energized, which in turn means the flow switch contact must be open. Since the flow switch is normally-open, in order to be in the open state we must have a **low level** condition.

- No color reaches the `O:0/1` coil instruction which means its bit state will be a zero (0). This makes the `O:0/1` output de-energized, ensuring the lamp will be de-energized as well.

2.6 Example: Arduino versus PLC dual-LED control

An *Arduino* is a popular model of microcontroller designed for student and hobbyist use, where text-based code stored in the microcontroller's memory dictates how it will respond to input signals to control devices connected to its outputs.

Here we see a partial wiring diagram and Sketch-language code causing an Arduino Nano microcontroller to control the states of two light-emitting diodes (LEDs) when a pushbutton switch is pressed and released:



Personal computer display
(Sketch code)

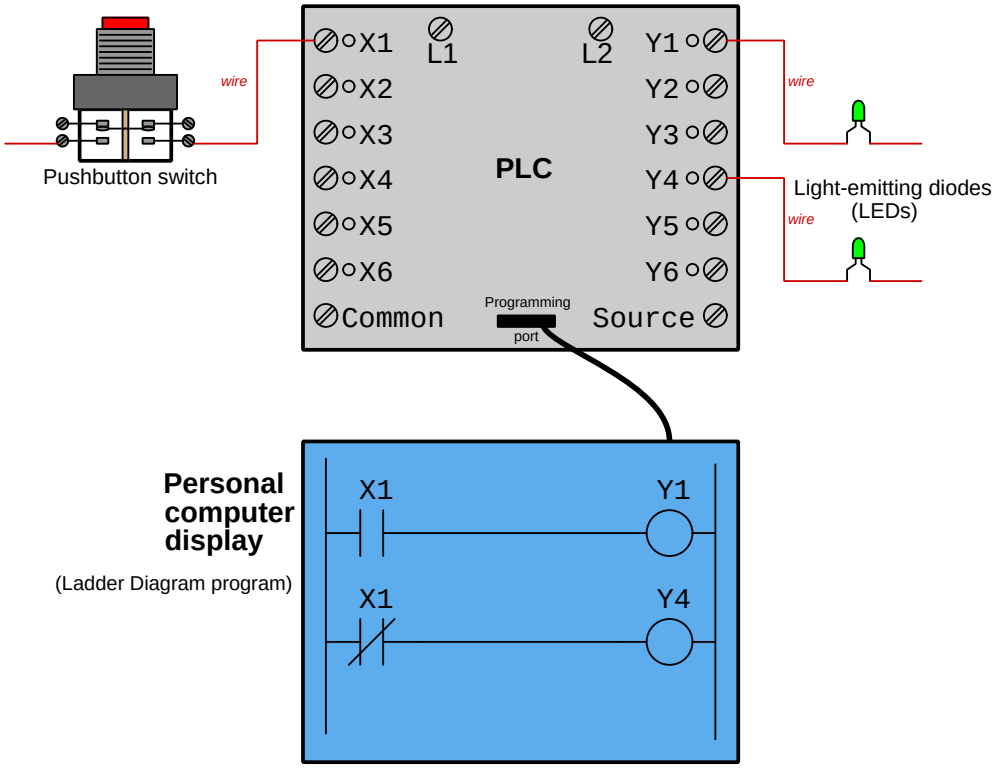
```
int button = 0;          // variable to store the read value

void setup() {
  pinMode(12, OUTPUT);  // sets digital pin 12 as an output
  pinMode(13, OUTPUT);  // sets digital pin 13 as an output
  pinMode(5, INPUT);    // sets digital pin 5 as an input
}

void loop() {
  button = digitalRead(5);   // read the input pin
  digitalWrite(13, button);  // writes pin 13 LED state same
  digitalWrite(12, !button); // writes pin 12 LED state opposite
}
```

Pressing the pushbutton switch energizes pin D5, and that electrically-energized state is read as a "1" value and stored in the *button* variable. This same "1" value gets written to pin D13 to energize that LED, while pin D12 gets the opposite value ("0") written to it to de-energize its LED.

Releasing the pushbutton switch de-energizes pin D5, causing a "0" value to be stored in the *button* variable. This same "0" value gets written to pin D13 to de-energize that LED, while pin D12 gets the opposite value ("1") written to it to energize its LED.

A *Programmable Logic Controller* or *PLC* is a special-purpose industrial computer where code stored in the PLC's memory dictates how it will respond to input signals to control devices connected to its outputs. The most common programming language for PLCs is *ladder diagram* which resembles a certain type of electrical wiring diagram, designed that way to make it familiar to electricians.

Here we see a partial wiring diagram and ladder-diagram code causing a PLC to control the states of two light-emitting diodes (LEDs) when a pushbutton switch is pressed and released:

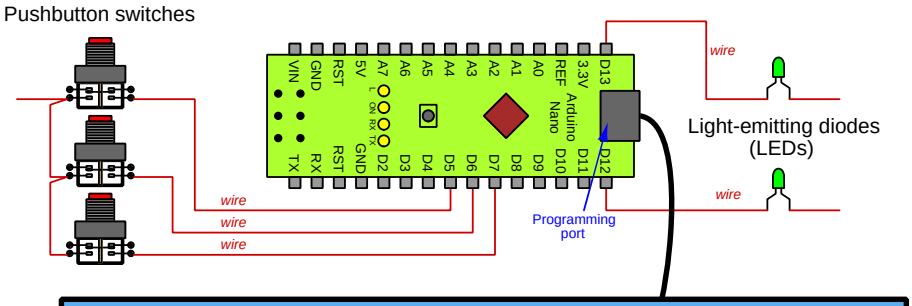


Pressing the pushbutton switch energizes input X1, and that electrically-energized state is read as a "1" value and stored in the PLC's `X1` variable. This "1" value causes the normally-open virtual switch contact to close and activate output Y1 to turn on that LED, and also causes the normally-closed virtual switch contact to open and de-activate output Y4 to turn off that LED.

Releasing the pushbutton switch de-energizes input X1 and clears the `X1` variable to a "0" value, causing both of the virtual switch contacts to return to their resting states, de-activating output Y1 and re-activating output Y4.

2.7 Example: Arduino versus PLC logic functions

An *AND* function is where multiple conditions all must be "true" in order to generate a response. An *OR* function is where just one of those multiple conditions need be "true" to make the same response. The following Arduino program implements both of these logical functions, with one LED designated for each and with three pushbutton switches providing the multiple conditions (inputs):



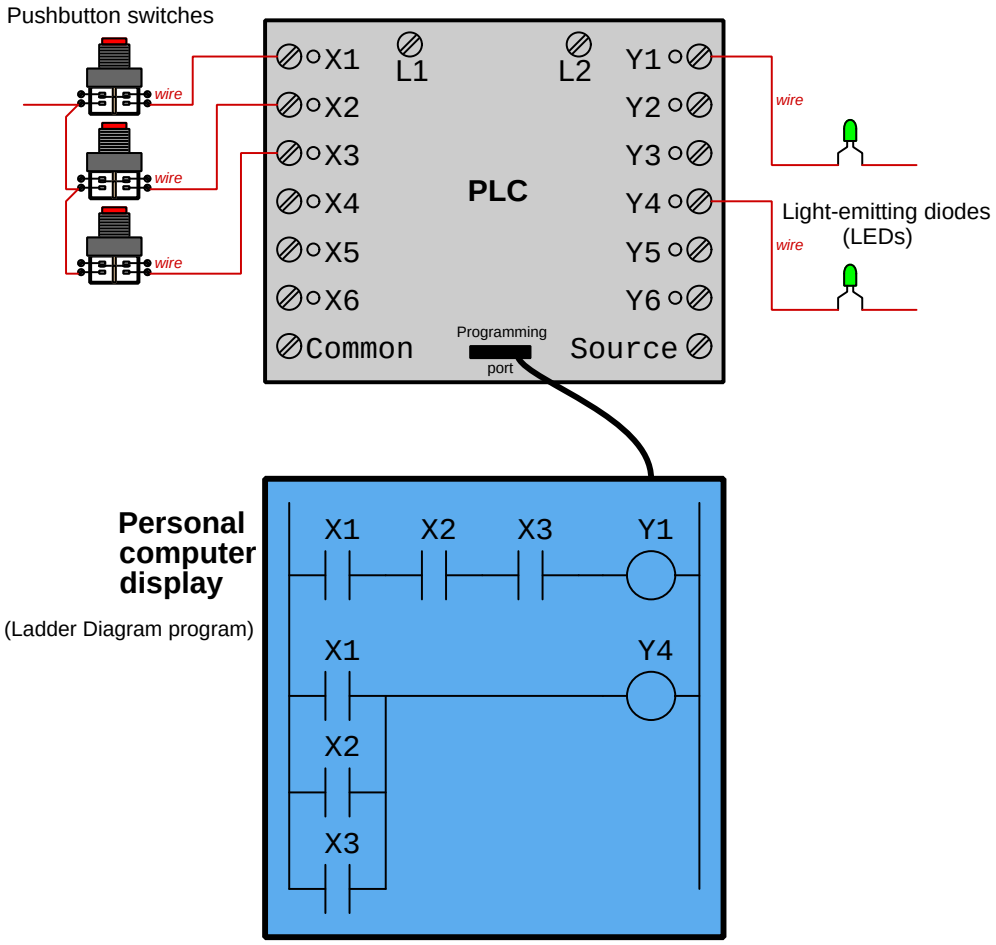
Personal computer display

(Sketch code)

```
int A, B, C;              // variables to store the read values

void setup() {
  pinMode(12, OUTPUT);  // sets digital pin 12 as an output
  pinMode(13, OUTPUT);  // sets digital pin 13 as an output
  pinMode(5, INPUT);    // sets digital pin 5 as an input
  pinMode(6, INPUT);    // sets digital pin 6 as an input
  pinMode(7, INPUT);    // sets digital pin 7 as an input
}

void loop() {
  A = digitalRead(5);         // reads input pin 5 as 'A'
  B = digitalRead(6);         // reads input pin 6 as 'B'
  C = digitalRead(7);         // reads input pin 7 as 'C'

  if (A && B && C)
    digitalWrite(13, 1);      // turns on pin 13 LED

  else
    digitalWrite(13, 0);      // turns off pin 13 LED

  if (A || B || C)
    digitalWrite(12, 1);      // turns on pin 12 LED

  else
    digitalWrite(12, 0);      // turns off pin 12 LED
}
```

In this program the LED connected to Arduino pin 13 represents the result of an AND function for the three inputs, while the LED connected to pin 12 represents the result of an OR function. Pressing any switch causes pin 12 LED to energize, but all three must be pressed simultaneously to energize pin 13.

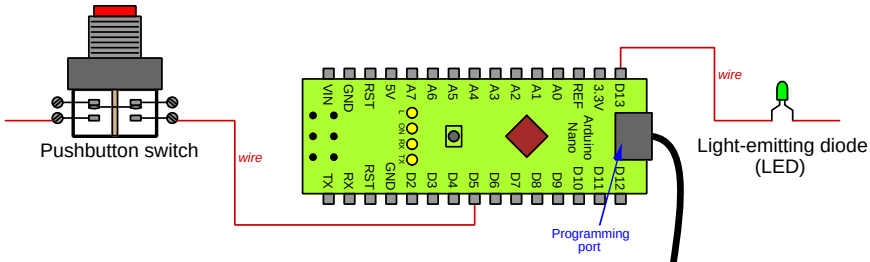
PLCs also implement *AND* and *OR* functions, but do so by connecting their virtual switches either in "series" (all in a row) or in "parallel", respectively:



In this program the LED connected to PLC output Y1 represents the output of an AND function for the three inputs, while the LED connected to output Y4 represents the output of an OR function. Pressing any switch causes the Y4 LED to energize, but all three must be pressed simultaneously to energize Y1.

2.8 Example: Arduino versus PLC on-delay timer

An "on-delay" timer is one where you electrically activate the timer, but instead of turning on an LED or other device immediately, it delays for a set time before turning it on. Here we see a partial wiring diagram and Sketch-language code causing an Arduino Nano microcontroller to delay the turn-on of a light-emitting diode (LED) when a pushbutton switch is pressed:

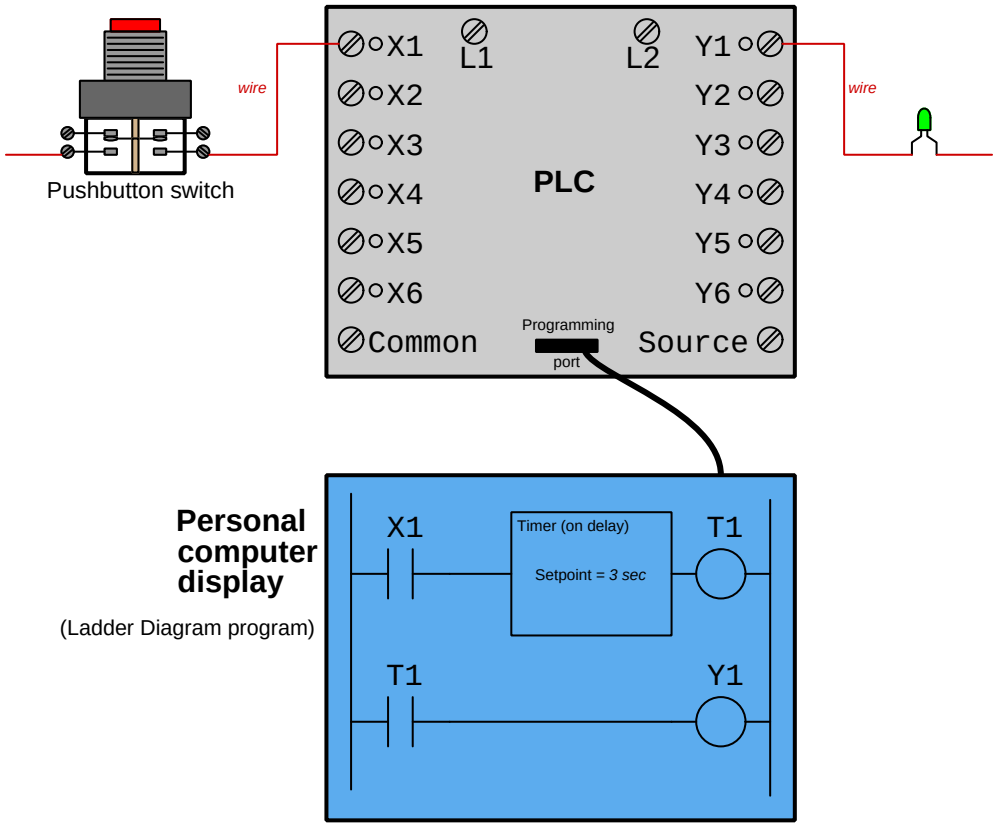


Personal computer display

(Sketch code)

```
int LastIN, IN = 0;       // variables to store switch states
unsigned long timestart;  // variable to store starting time value

void setup() {
 pinMode(13, OUTPUT);     // sets digital pin 13 as an output
 pinMode(5, INPUT);       // sets digital pin 5 as an input
}

void loop() {
 LastIN = IN;                      // update old switch status
 IN = digitalRead(5);              // read new switch status

 if (IN == 1 && LastIN == 0)       // checks for switch press
   timestart = millis();           // initializes start time

 if (IN == 1 && millis() > (timestart + 3000)) // waits 3 seconds
   digitalWrite(13, 1);            // ...to turn on D13 output pin

 else
   digitalWrite(13, 0);            // turns off D13 output pin
}
```

Here we see a partial wiring diagram and ladder-diagram code causing a PLC to implement an on-delay timing function:

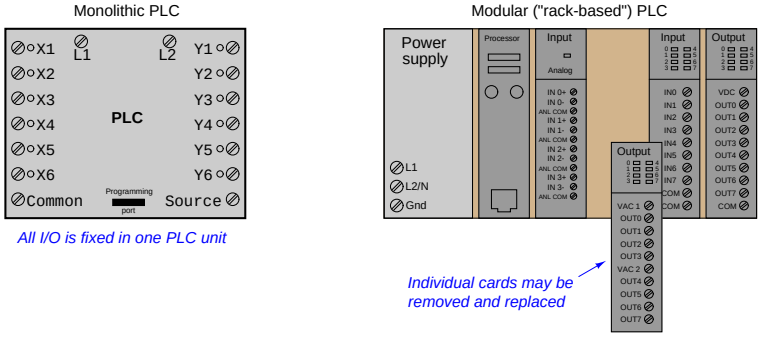


Chapter 3

Tutorial

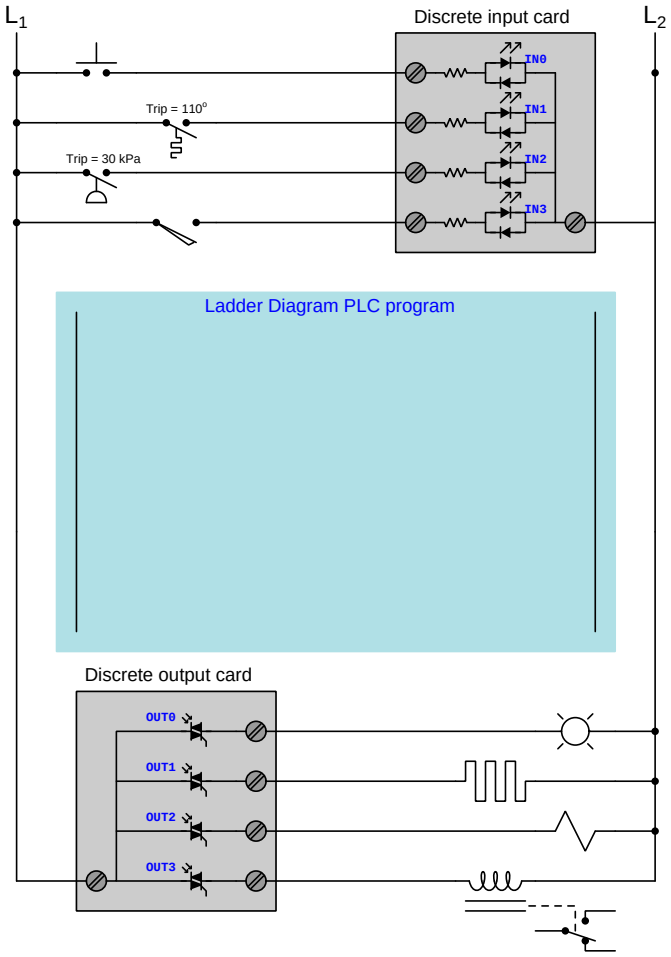
3.1 Review of basic PLC functionality

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



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

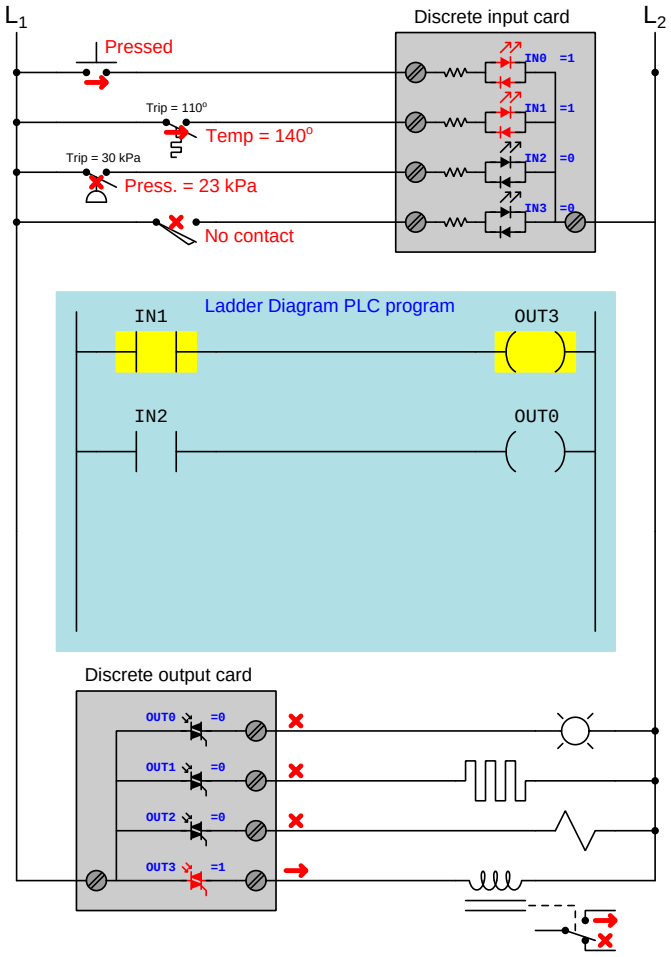
The following diagram shows a PLC separated into three sections: (1) a *discrete input card*, (2) the *program* space in the processor's memory, and (3) a *discrete output card*:



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

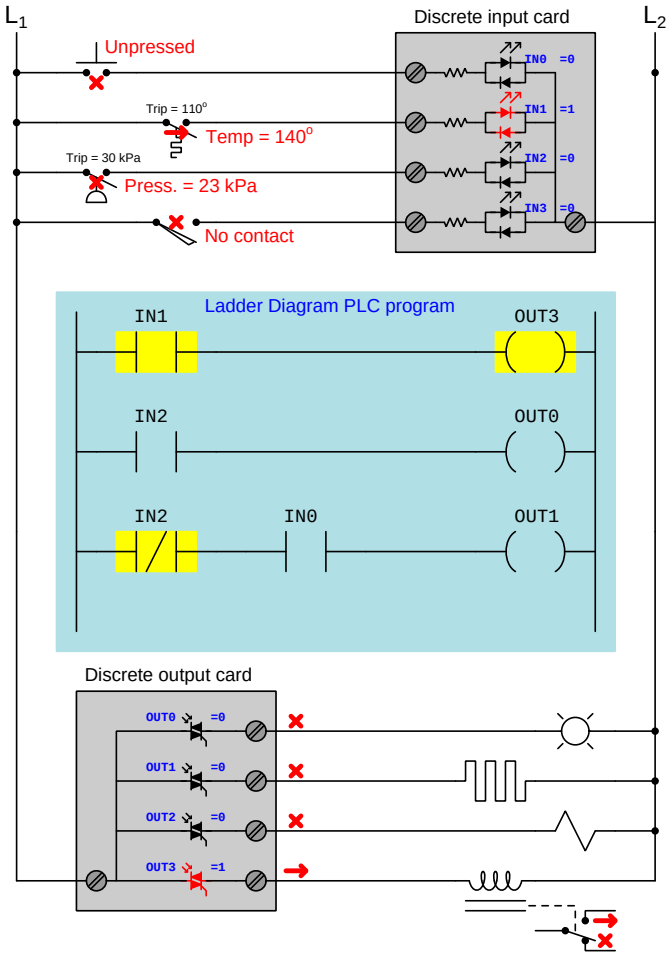
[1] Not shown in this simplified diagram are the optotransistors coupled to the LEDs inside the input card, translating each LED's state to a discrete logic level at the transistor to be interpreted by the PLC's digital processor. Likewise, another set of LEDs driven by the processor's outputs couple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

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



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

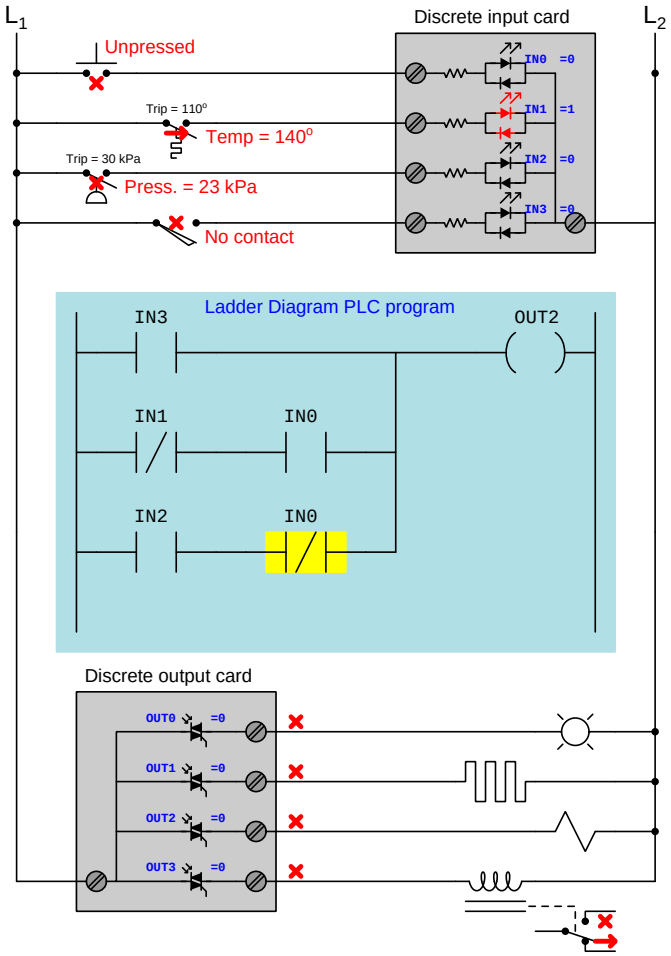
Things get more complex when we begin adding *normally-closed* contact instructions to the program. Consider this next diagram, with updated stimuli and an expanded Ladder Diagram program for the PLC to follow:



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

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

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



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

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

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

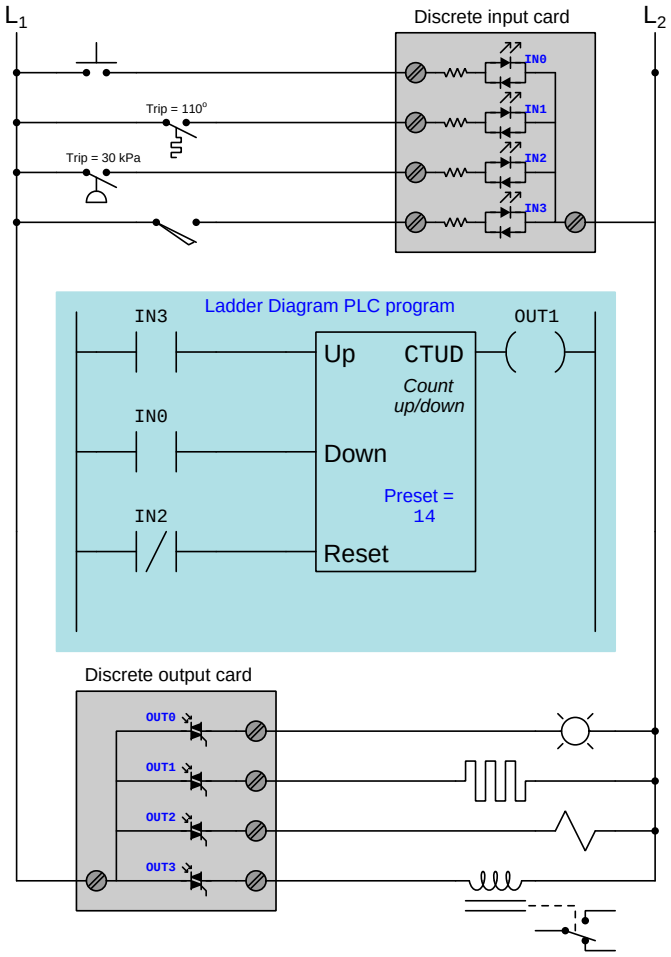
All PLCs follow this chain of logic precisely, and do so in a repeated pattern called the *scan cycle*. Each cycle begins with the PLC's processor reading the electrical states at the input terminals and updating input registers in its memory to reflect those electrical states. The processor then proceeds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

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

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

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

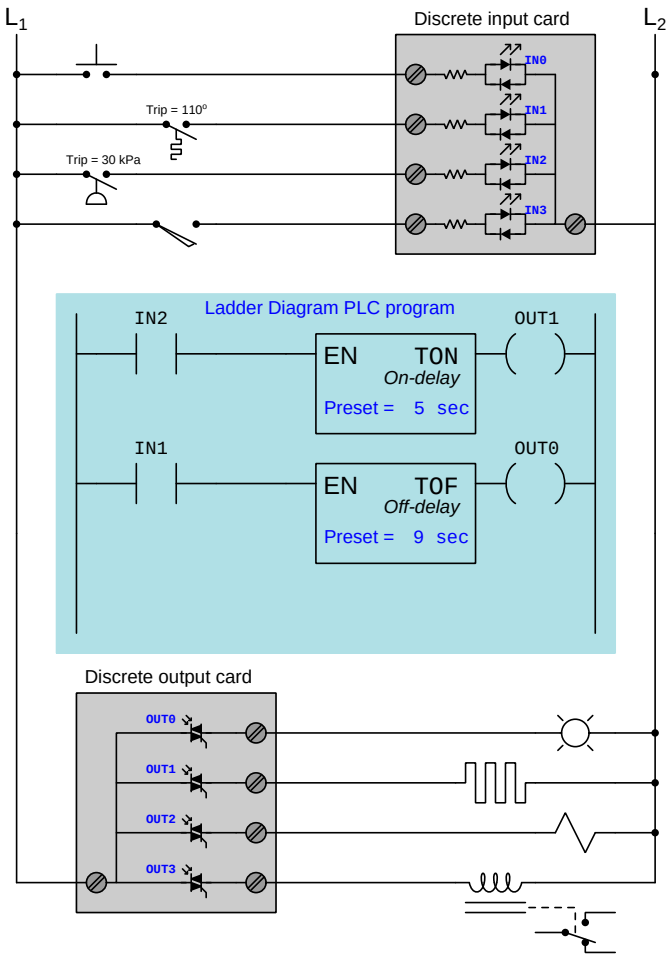
Being fully-fledged digital computers in their own right, PLCs are not limited to executing simple Boolean functions represented by "virtual relay" contacts and coils. Other digital functions include *counters* and *timers*. An example of a counter program is shown here:



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

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

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



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

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

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

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

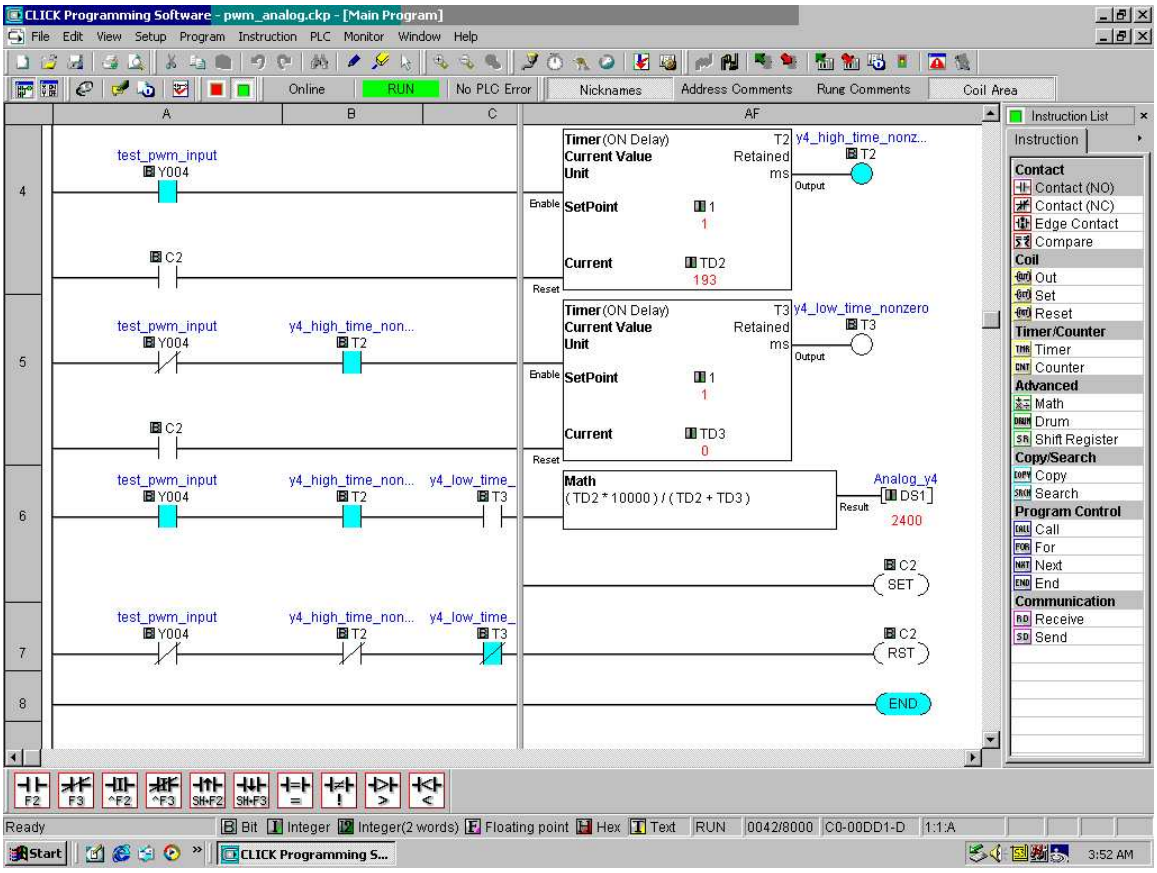
One advantage of PLCs over relay circuitry which may not be evident at first inspection is the fact that the number of virtual "contacts" and "coils" and other instructions is limited only by how much memory the PLC's processor has. The example programs shown on the previous pages were extremely short,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

3.2 Intro to Ladder Diagram (LD) programming

In the United States, the most common language used to program PLCs is *Ladder Diagram* (LD), also known as *Relay Ladder Logic* (RLL). This is a graphical language showing the logical relationships between inputs and outputs as though they were contacts and coils in a hard-wired electromechanical relay circuit. This language was invented for the express purpose of making PLC programming feel "natural" to electricians familiar with relay-based logic and control circuits. While Ladder Diagram programming has many shortcomings, it remains extremely popular and so will be the primary focus of this tutorial.

Every Ladder Diagram program is arranged to resemble an electrical diagram, making this a graphical (rather than text-based) programming language. Ladder diagrams are to be thought of as *virtual circuits*, where virtual "power" flows through virtual "contacts" (when closed) to energize virtual "relay coils" to perform logical functions. None of the contacts or coils seen in a Ladder Diagram PLC program are real; rather, they act on bits in the PLC's memory, the logical interrelationships between those bits expressed in the form of a diagram *resembling* a circuit.

The following computer screenshot shows a typical Ladder Diagram program[4] being edited on a personal computer:



Contacts appear just as they would in an electrical relay logic diagram – as short vertical line segments separated by a horizontal space. Normally-open contacts are empty within the space between the line segments, while normally-closed contacts have a diagonal line crossing through that space. Coils are somewhat different, appearing as either circles or pairs of parentheses. Other instructions appear as rectangular boxes.

Each horizontal line is referred to as a *rung*, just as each horizontal step on a stepladder is called a "rung." A common feature among Ladder Diagram program editors, as seen on this screenshot, is

[4]This particular program and editor is for the Koyo "CLICK" series of micro-PLCs.

the ability to color-highlight those virtual "components" in the virtual "circuit" ready to "conduct" virtual "power." In this particular editor, the color used to indicate "conduction" is light blue. Another form of status indication seen in this PLC program is the values of certain variables in the PLC's memory, shown in red text.

For example, you can see coil `T2` energized at the upper-right corner of the screen (filled with light blue coloring), while coil `T3` is not. Correspondingly, each normally-open `T2` contact appears colored, indicating its "closed" status, while each normally-closed `T2` contact is uncolored. By contrast, each normally-open `T3` contact is uncolored (since coil `T3` is unpowered) while each normally-closed `T3` contact is shown colored to indicate its conductive status. Likewise, the current count values of timers `T2` and `T3` are shown as 193 and 0, respectively. The output value of the math instruction box happens to be 2400, also shown in red text.

Color-highlighting of Ladder Diagram components only works, of course, when the computer running the program editing software is connected to the PLC and the PLC is in the "run" mode (and the "show status" feature of the editing software is enabled). Otherwise, the Ladder Diagram is nothing more than black symbols on a white background. Not only is status highlighting very useful in de-bugging PLC programs, but it also serves an invaluable diagnostic purpose when a technician analyzes a PLC program to check the status of real-world input and output devices connected to the PLC. This is especially true when the program's status is viewed remotely over a computer network, allowing maintenance staff to investigate system problems without even being near the PLC!

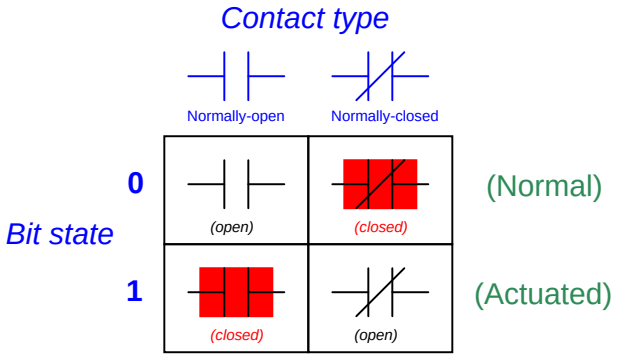
3.3 Ladder diagram virtual elements and I/O status

Perhaps the most important yet elusive concept to grasp when learning to program PLCs is the relationship between the electrical status of the PLC's I/O points and the status of variables and other "elements" in its programming. This is especially true for Ladder Diagram (LD) programming, where the program itself resembles an electrical diagram. Making the mental connection between the "real" world of the switches, contactors, and other electrical devices connected to the PLC and the "imaginary" world of the PLC's program consisting of virtual contacts and relay "coils" is most fundamental.

The first fundamental rule one should keep in mind when examining a Ladder Diagram PLC program is that **each virtual contact shown in the program *actuates* whenever it reads a "1" state in its respective bit and will be *at rest* whenever it reads a "0" state in its respective bit** (in the PLC's memory). If the contact is a normally-open (NO) type, it will open when its bit is 0 and close when its bit is 1. If the contact is a normally-closed (NC) type, it will close when its bit is 0 and open when its bit is 1. A 0 bit state causes the contact to be in its "normal" (resting) condition, while a 1 bit state *actuates* the contact, forcing it into its non-normal (actuated) state.

Another rule to remember when examining a Ladder Diagram PLC program is that the programming software offers *color highlighting*[5] to display the virtual status of each program element: **a colored contact is *closed*, while an un-colored contact is *open***. While the presence or absence of a "slash" symbol marks the *normal* status of a contact, its live color highlighting shown by PLC programming software reveals the "conductive" status of the elements *in real time*.

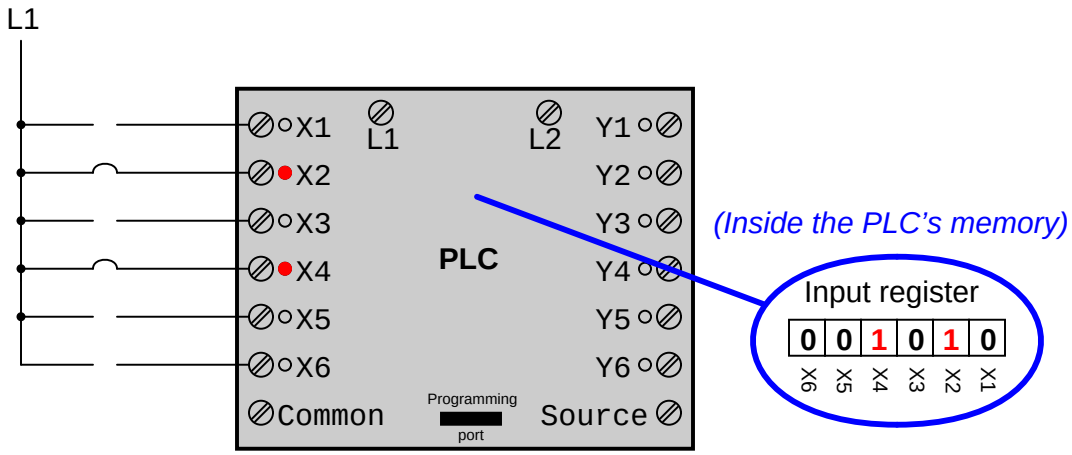
The following table shows how the two types of contacts in a PLC's Ladder Diagram program respond to bit states, using red coloring to signify each contact's virtual conductivity:



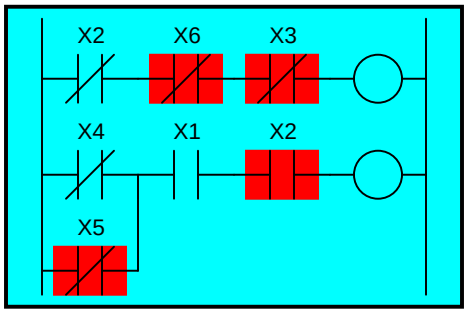
Just as a pressure switch's contacts are actuated by a high pressure condition, and a level switch's contacts are actuated by a high level condition, and a temperature switch's contacts are actuated by a high temperature condition, so a PLC's virtual contact is actuated by a high *bit* condition (1). In the context of any switch, an *actuated* condition is the opposite of its *normal* (resting) condition.

[5] It should be noted that in some situations the programming software will fail to color the contacts properly, especially if their status changes too quickly for the software communication link to keep up, and/or if the bit(s) change state multiple times within one scan of the program. However, for simple programs and situations, this rule holds true and is a great help to beginning programmers as they learn the relationship between real-world conditions and conditions within the PLC's "virtual" world.

The following simplified[6] illustration shows a small PLC with two of its discrete input channels electrically energized, causing those two bits to have "1" statuses. The color-highlighted contacts in the programming editor software's display shows a collection of contacts addressed to those input bits in various states (colored = closed ; un-colored = open). As you can see, every contact addressed to a "set" bit (1) is in its actuated state, while every contact addressed to a "cleared" bit (0) is in its normal state:



(Displayed on the editing software)



Remember that a *colored* contact is a *closed* contact. The contacts appearing as colored are either normally-closed contacts with "0" bit states, or normally-open contacts with "1" bit states. It is the combination of bit state and contact type (NO vs. NC) that determines whether the virtual contact will be open (un-colored) or closed (colored) at any given time. Correspondingly, it is a combination of colored highlighting and virtual contact type that indicates the real-world energization status of a particular PLC input at any given time.

[6] The electrical wiring shown in this diagram is incomplete, with the "Common" terminal shown unconnected for simplicity's sake.

In my teaching experience, the main problem students have comprehending PLC Ladder Diagram programs is that they over-simplify and try to directly associate real-world switches connected to the PLC with their respective contact instructions inside the PLC program. Students mistakenly think the real-world switch connecting to the PLC and the respective virtual switch contact inside the PLC program are one and the same, when this is not the case at all. Rather, the real-world switch sends power to the PLC input, *which in turn controls the state of the virtual contact(s) programmed into the PLC.* Specifically, I see students routinely fall into the following misconceptions:

- Students mistakenly think the contact instruction type (NO vs. NC) needs to match that of its associated real-world switch
- Students mistakenly think color highlighting of a contact instruction is equivalent to the electrical status of its associated real-world PLC input
- Students mistakenly think a closed real-world switch must result in a closed contact instruction in the live PLC program

To clarify, here are the fundamental rules one should keep in mind when interpreting contact instructions in Ladder Diagram PLC programs:

- **Each input bit in the PLC's memory will be a "1" when its input channel is powered, and will be a "0" when its input channel is unpowered**
- **Each virtual contact shown in the program *actuates* whenever it reads a "1" state in its respective bit, and will be *at rest* whenever it reads a "0" state in its respective bit**
- **A colored contact is *closed* (passes virtual power in the PLC program), while an un-colored contact is *open* (blocks virtual power in the PLC program)**

In trying to understand PLC Ladder Diagram programs, the importance of these rules cannot be overemphasized. The truth of the matter is a causal chain – rather than a direct equivalence – between the real-world switch and the contact instruction status. The real-world switch controls whether or not electrical power reaches the PLC input channel, which in turn controls whether the input register bit will be a "1" or a "0", which in turn controls whether the contact instruction will actuated or at rest. Virtual contacts inside the PLC program are thus *controlled* by their corresponding real-world switches, rather than simply being *identical* to their real-world counterparts as novices tend to assume. Following these rules, we see that normally-open (NO) contact instructions will mimic what their real-world switches are doing, while normally-closed (NC) contact instructions will act opposite of their real-world counterparts.

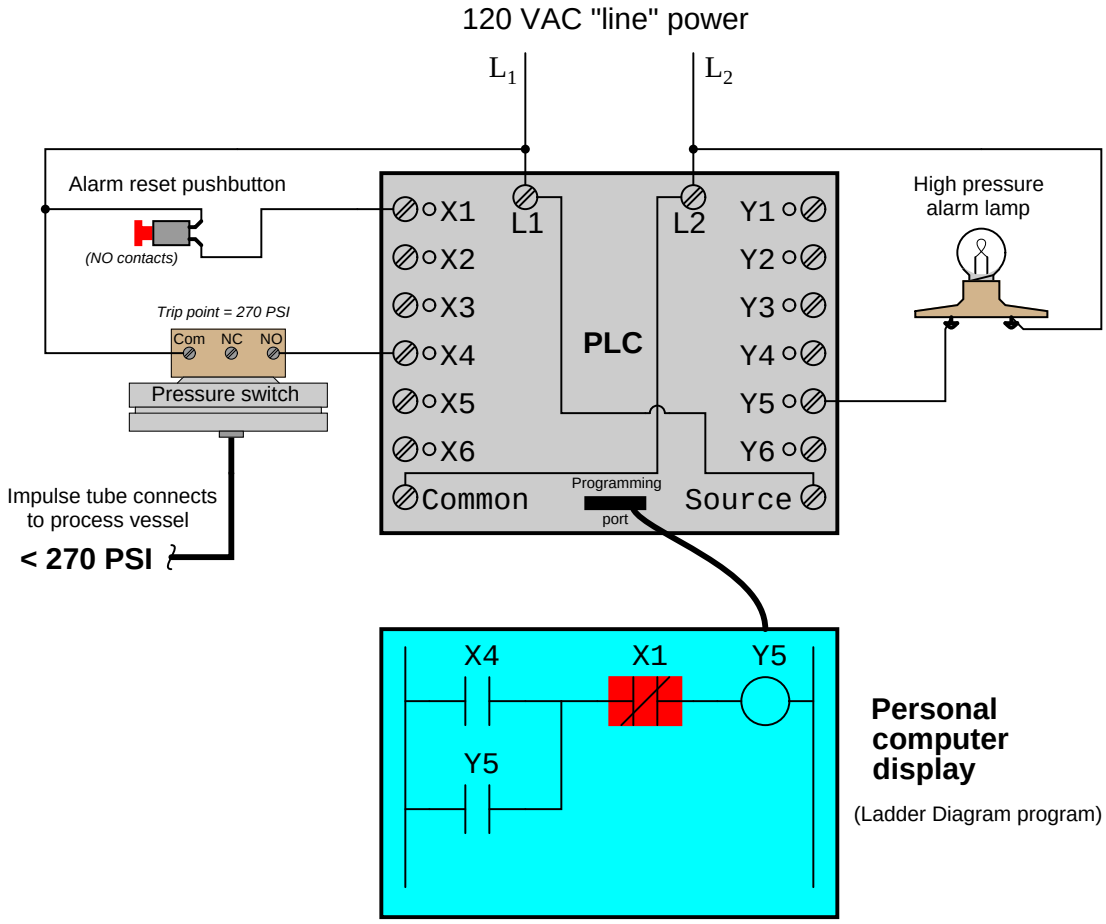
The color highlighting of *coil* instructions in a Ladder Diagram PLC program follows similar rules. A coil will be "on" (colored) when all contact instructions prior to it are closed (colored). A colored coil writes a "1" to its respective bit in memory, while an un-colored coil instruction writes a "0" to its respective bit in memory. If these bits are associated with real-world discrete output channels on the PLC, their states will control the real-world energization of devices electrically connected to those channels.

3.4 Ladder diagram latching functions

A very important function offered by PLCs is to *latch* a discrete bit in either the "on" or "off" state based on one or more momentary input signals. A practical example of this might be a motor start-stop control where a human operator presses a "Start" pushbutton to start the electric motor and at some later time presses a separate "Stop" pushbutton to halt the motor. Since each of these pushbutton controls is typically a "momentary" contact making electrical continuity only when pressed, some logic device must perform the "latch" function necessary to maintain the motor's state until the other switch is pressed. Fortunately, this kind of "latching" function is very easy to implement using Ladder Diagram programming in a PLC.

In this section we will explore the concept of latching in the context of a PLC system designed to energize a warning lamp in the event that a process vessel experiences a high fluid pressure. The PLC's task is to energize a warning lamp if the process vessel pressure ever exceeds 270 PSI, and keep that warning lamp energized even if the pressure falls below the trip point of 270 PSI. This way, human operators will be alerted to both *past* and *present* process vessel overpressure events.

Below we see an illustration of a PLC connected to a pressure-activated switch, a pushbutton switch, and an alarm lamp. 120 volt AC "line" power (L1 and L2) provides electrical energy for the PLC to operate, as well as signal potential for the input switches and power for the warning lamp. Two switches connect to the input of this PLC: one normally-open pushbutton switch acting as the alarm reset (pressing this switch "unlatches" the alarm lamp) and one normally-open pressure switch acting as the sensing element for high process vessel pressure:

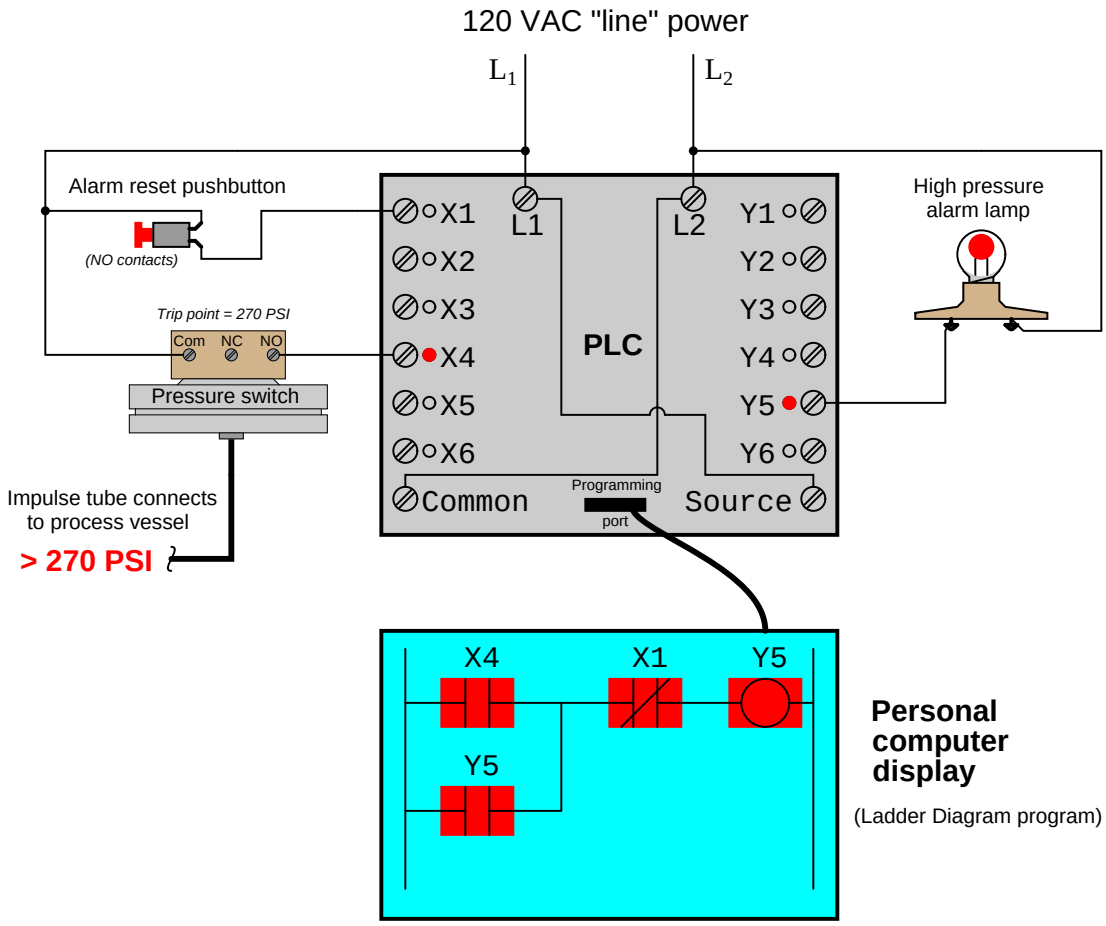


In this system the reset pushbutton connects to discrete input `X1` of the PLC, while the pressure switch connects to discrete input `X4`. The warning lamp connects to discrete output `Y5`. Red indicator LEDs next to each I/O terminal visually indicate the electrical status of the I/O points, while red-shaded highlighting shows the *virtual power*[7] status of the "contacts" and "coils" in the PLC's program, displayed on the screen of a personal computer connected to the PLC through a programming cable.

[7] For a PLC program contact, the shading represents virtual "conductivity." For a PLC program coil, the shading represents a set (1) bit.

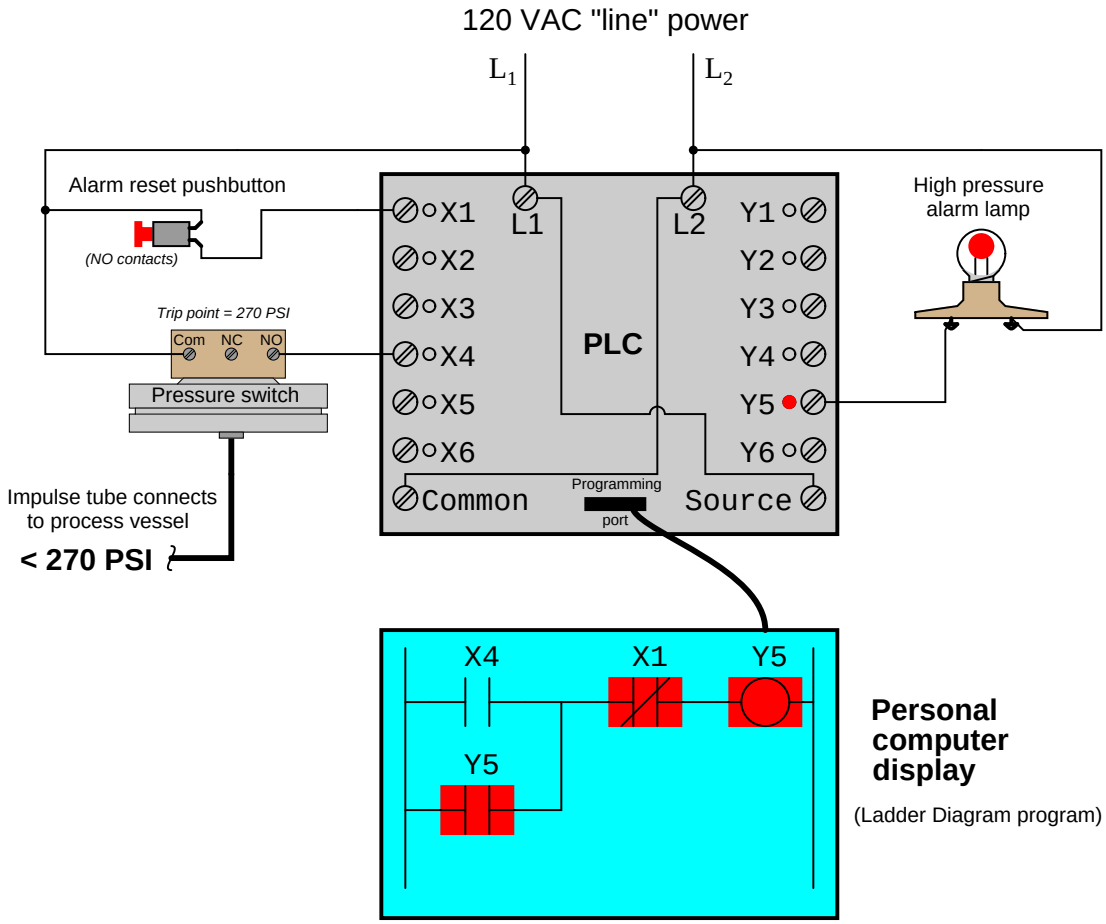
With no one pressing the reset pushbutton, that switch will be in its normal status, which for a "normally-open" switch is open. Likewise with the pressure switch: with process pressure less than the trip point of 270 PSI, the pressure switch will also be in its normal status, which for a "normally-open" switch is open. Since neither switch is conducting electricity right now, neither discrete input `X1` nor `X4` will be energized. This means the "virtual" contacts inside the PLC program will likewise be in their own normal states. Thus, any virtual contact drawn as a normally-open will be open (not passing virtual power), and any virtual contact drawn as a normally-closed (a diagonal slash mark through the contact symbol) will be closed. This is why the two normally-open virtual contacts `X4` and `Y5` have no highlighting, but the normally-closed virtual contact `X1` does – the colored highlighting representing the ability to pass virtual power.

If the process vessel experiences a high pressure (> 270 PSI), the pressure switch will actuate, closing its normally-open contact. This will energize input `X4` on the PLC, which will "close" the virtual contact `X4` in the ladder program. This sends virtual power to the virtual "coil" `Y5`, which in turn latches itself on through virtual contact `Y5`[8] and also energizes the real discrete output `Y5` to energize the warning lamp:



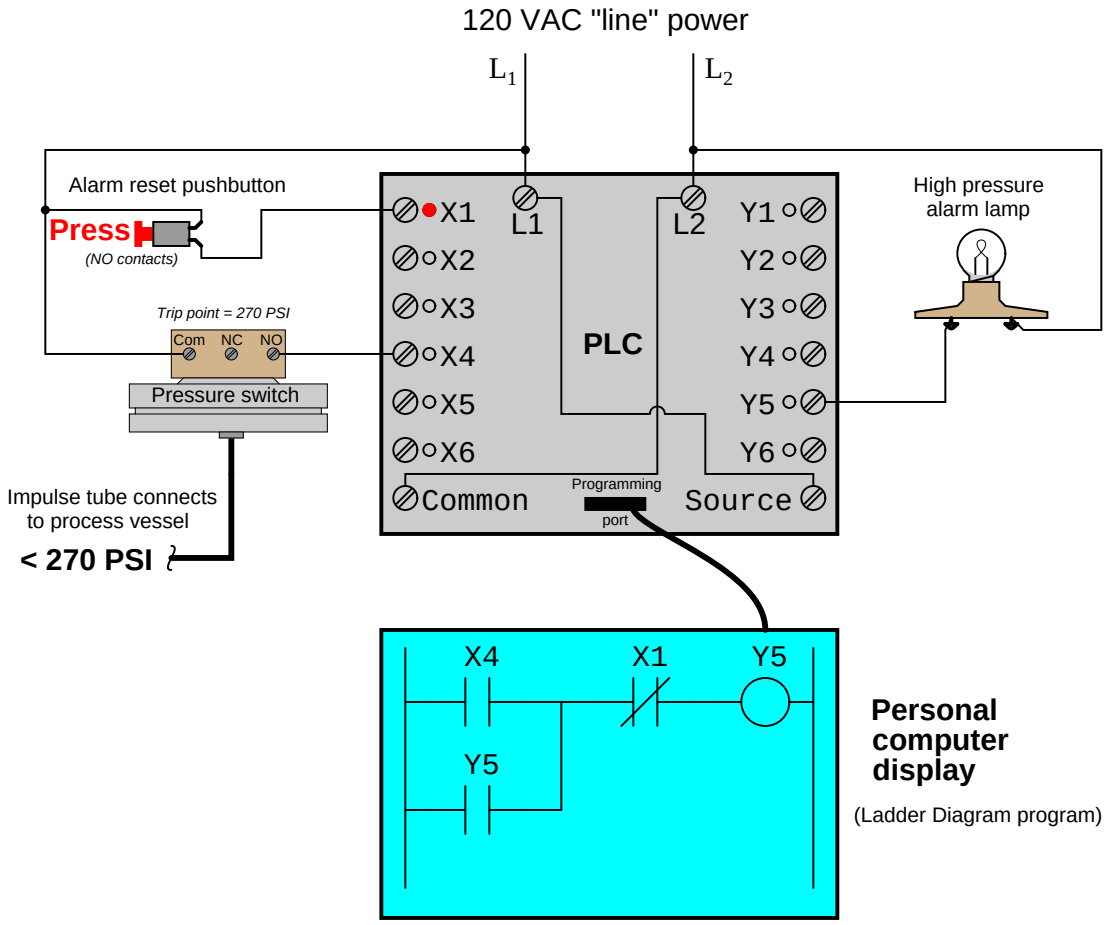
[8] It is worth noting the legitimacy of referencing virtual contacts to output bits (e.g. contact `Y5`), and not just to input bits. A "virtual contact" inside a PLC program is nothing more than an instruction to the PLC's processor to *read* the status of a bit in memory. It matters not whether that bit is associated with a physical input channel, a physical output channel, or some abstract bit in the PLC's memory. It would, however, be wrong to associate a virtual coil with an input bit, as coil instructions *write* bit values to memory, and input bits are supposed to be controlled solely by the energization states of their physical input channels.

If now the process pressure falls below 270 PSI, the pressure switch will return to its normal state (open), thus de-energizing discrete input `X4` on the PLC. Because of the latching contact `Y5` in the PLC's program, however, output `Y5` remains on to keep the warning lamp in its energized state:



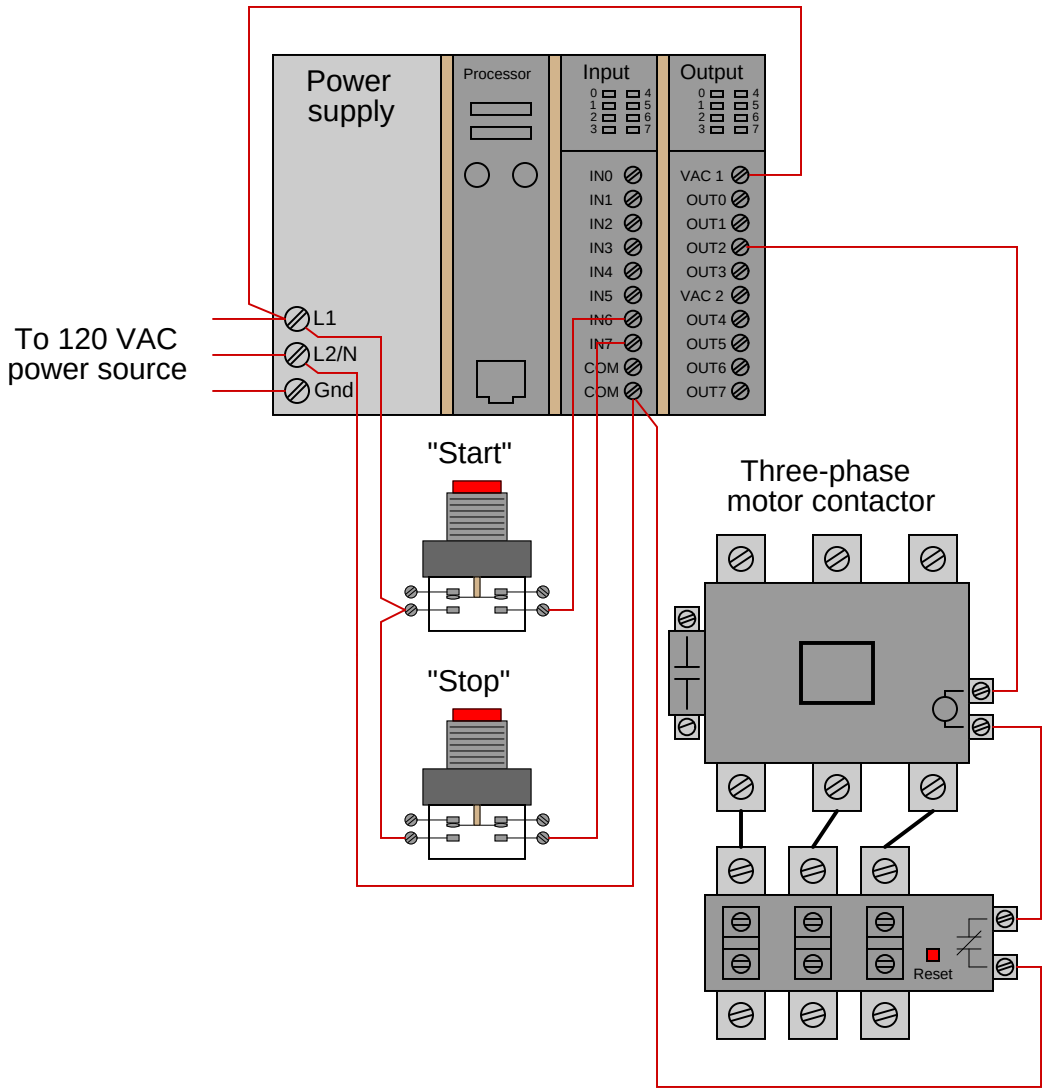
Thus, the `Y5` contact performs a *seal-in* function to keep the `Y5` bit set (1) even after the high-pressure condition clears. This is precisely the same concept as the "seal-in" auxiliary contact on a hard-wired motor starter circuit, where the electromechanical contactor keeps itself energized after the "Start" pushbutton switch has been released.

The only way for a human operator to re-set the warning lamp is to press the pushbutton. This will have the effect of energizing input `X1` on the PLC, thus opening virtual contact `X1` (normally-closed) in the program, thus interrupting virtual power to the virtual coil `Y5`, thus powering down the warning lamp and un-latching virtual power in the program:



3.5 Fault-tolerant latch programming

As mentioned previously, a common application for a latching PLC program is the control of an electric motor using momentary-contact "Start" and "Stop" pushbutton switches. We will continue our exploration of this concept by applying it to the motor-control PLC system shown below:

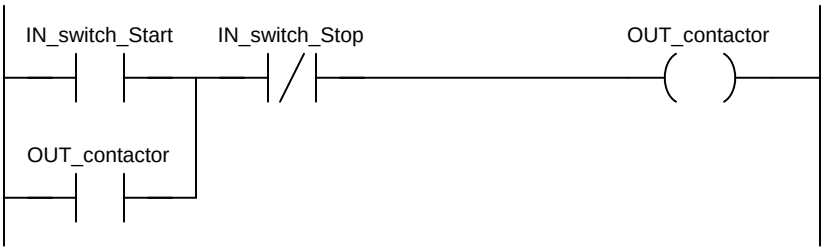


In this system, two pushbutton switches are connected to discrete inputs on a PLC, and the PLC in turn energizes the coil of a motor contactor relay by means of one of its discrete outputs[9].

[9]The particular input and output channels chosen for this example are completely arbitrary. There is no particular

An overload contact is wired directly in series with the contactor coil to provide motor overcurrent protection, even in the event of a PLC failure where the discrete output channel remains energized[10].

The ladder program for this motor control system would look like this:



Pressing the "Start" pushbutton energizes discrete input channel 6 on the PLC, which "closes" the virtual contact in the PLC program labeled `IN_switch_Start`. The normally-closed virtual contact for input channel 7 (the "Stop" pushbutton) is already closed by default when the "Stop" button is not being pressed, and so the virtual coil will receive "power" when the "Start" pushbutton is pressed and the "Stop" pushbutton is not.

Note the *seal-in* contact bearing the exact same label as the coil: `OUT_contactor`. At first it may seem strange to have both a contact and a coil in a PLC program labeled identically[11], since contacts are most commonly associated with inputs and coils with outputs, but this makes perfect sense if you realize the true meaning of contacts and coils in a PLC program: as *read* and *write* operations on bits in the PLC's memory. The coil labeled `OUT_contactor` *writes* the status of that bit, while the contact labeled `OUT_contactor` *reads* the status of that same bit. The purpose of this contact, of course, is to latch the motor in the "on" state after a human operator has released his or her finger from the "Start" pushbutton.

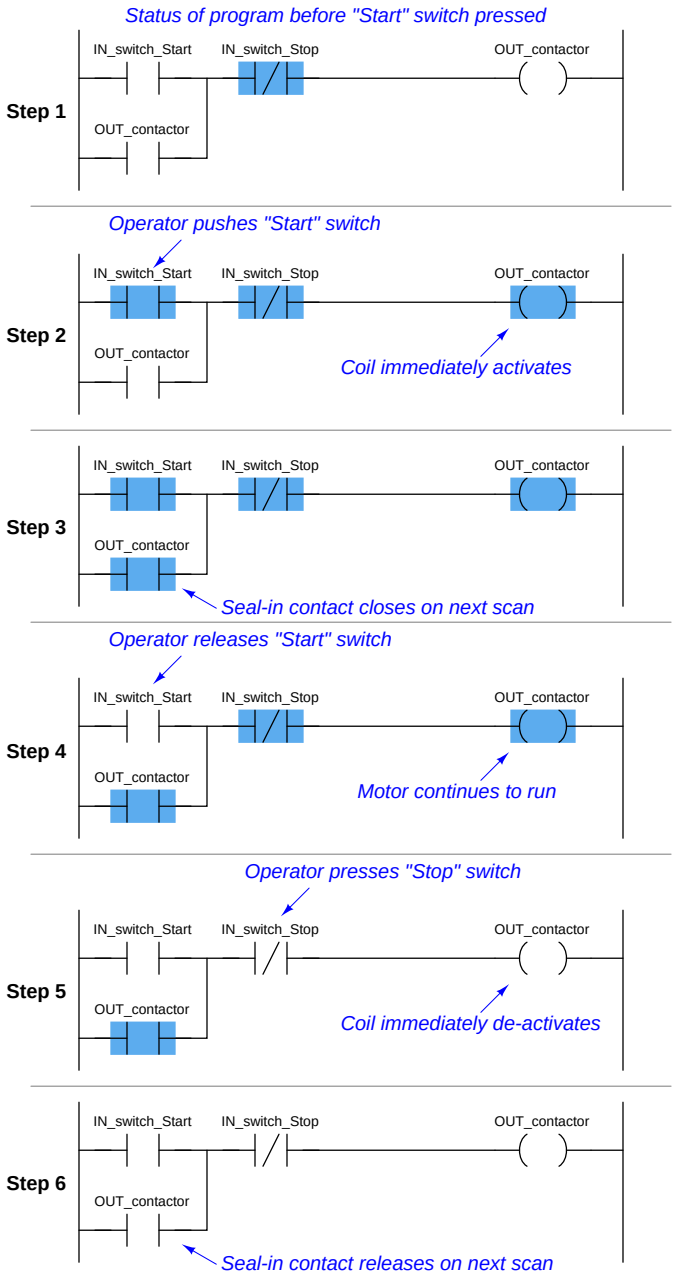
This programming technique is known as *feedback*, where an output variable of a function (in this case, the feedback variable is `OUT_contactor`) is also an input to that same function. The path of feedback is *implicit* rather than *explicit* in Ladder Diagram programming, with the only indication of feedback being the common name shared by coil and contact. Other graphical programming languages (such as Function Block) have the ability to show feedback paths as connecting lines between function outputs and inputs, but this capacity does not exist in Ladder Diagram.

reason to choose input channels 6 and 7, or output channel 2, as I have shown in the wiring diagram. Any available I/O channels will suffice.

[10] While it is possible to wire the overload contact to one of the PLC's discrete input channels and then program a *virtual* overload contact in series with the output coil to stop the motor in the event of a thermal overload, this strategy would rely on the PLC to perform a safety function which is probably better performed by hard-wired circuitry.

[11] A very common misconception among students first learning PLC Ladder Diagram programming is to always associate contacts with PLC inputs and coils with PLC outputs, thus it seems weird to have a contact bear the same label as an output. However, this is a false association. In reality, contacts and coils are *read* and *write* instructions, and thus it is possible to have the PLC read one of its own output bits as a part of some logic function. What *would* be truly strange is to label a coil with an input bit address or tag name, since the PLC is not electrically capable of setting the real-world energization status of any input channels.

A step-by-step sequence showing the operation and status of this simple program illustrates how the seal-in contact functions, through a start-up and shut-down cycle of the motor:



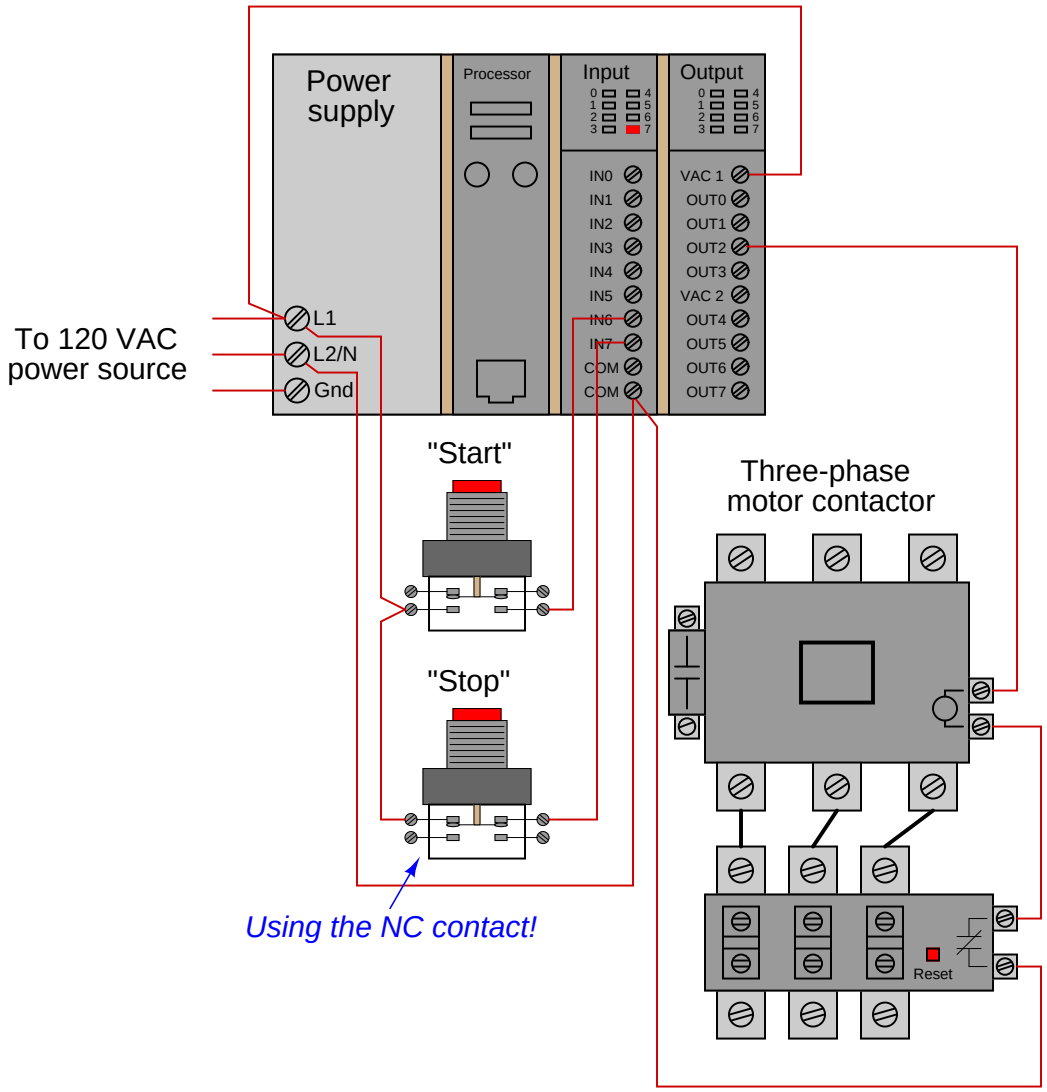
This sequence helps illustrate the *order of evaluation* or *scan order* of a Ladder Diagram program. The PLC reads a Ladder Diagram from left to right, top to bottom, in the same general order as

a human being reads sentences and paragraphs written in English. However, according to the IEC 61131-3 standard, a PLC program must evaluate (read) all inputs (contacts) to a function before determining the status of a function's output (coil or coils). In other words, the PLC does not make any decision on how to set the state of a coil until all contacts providing power to that coil have been read. Once a coil's status has been written to memory, any contacts bearing the same tag name will update with that status on subsequent rungs in the program.

Step 5 in the previous sequence is particularly illustrative. When the human operator presses the "Stop" pushbutton, the input channel for `IN_switch_Stop` becomes activated, which "opens" the normally-closed virtual contact `IN_switch_Stop`. Upon the next scan of this program rung, the PLC evaluates all input contacts (`IN_switch_Start`, `IN_switch_Stop`, and `OUT_contactor`) to check their status before deciding what status to write to the `OUT_contactor` coil. Seeing that the `IN_switch_Stop` contact has been forced open by the activation of its respective discrete input channel, the PLC writes a "0" (or "False") state to the `OUT_contactor` coil. However, the `OUT_contactor` feedback contact does not update until the next scan, which is why you still see it color-highlighted during step 5.

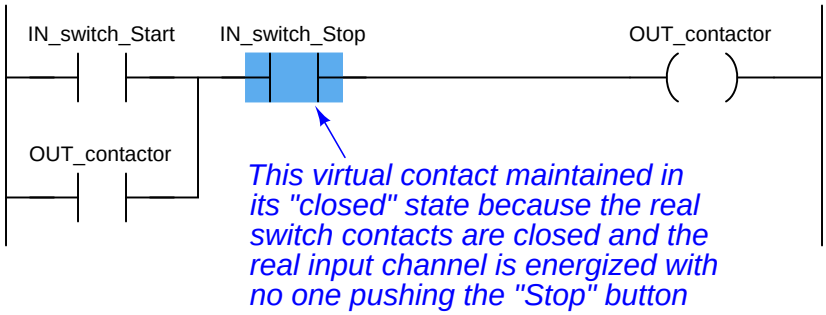
A potential problem with this system as it is designed is that the human operator loses control of the motor in the event of an "open" wiring failure in either pushbutton switch circuit. For instance, if a wire fell off a screw contact for the "Start" pushbutton switch circuit, the motor could not be started if it was already stopped. Similarly, if a wire fell off a screw contact for the "Stop" pushbutton switch circuit, the motor could not be stopped if it was already running. In either case, a broken wire connection acts the same as the pushbutton switch's "normal" status, which is to keep the motor in its present state. In some applications, this failure mode would not be a severe problem. In many applications, though, it is quite dangerous to have a running motor that cannot be stopped. For this reason, it is customary to design motor start/stop systems a bit differently from what has been shown here.

In order to build a "fail-stop" motor control system with our PLC, we must first re-wire the pushbutton switch to use its normally-closed (NC) contact:



This keeps discrete input channel 7 activated when the pushbutton is unpressed. When the operator presses the "Stop" pushbutton, the switch's contact will be forced open, and input channel 7 will de-energize. If a wire happens to fall off a screw terminal in the "Stop" switch circuit, input channel 7 will de-energize just the same as if someone pressed the "Stop" pushbutton, which will automatically shut off the motor.

In order for the PLC program to work properly with this new switch wiring, the virtual contact for `IN_switch_Stop` must be changed from a normally-closed (NC) to a normally-open (NO):



As before, the `IN_switch_Stop` virtual contact is in the "closed" state when no one presses the "Stop" switch, enabling the motor to start any time the "Start" switch is pressed. Similarly, the `IN_switch_Stop` virtual contact will open any time someone presses the "Stop" switch, thus stopping virtual "power" from flowing to the `OUT_contactor` coil.

Although this is a very common way to build PLC-controlled motor start/stop systems – with an NC pushbutton switch and an NO "Stop" virtual contact – students new to PLC programming often find this logical reversal confusing[12]. Perhaps the most common reason for this confusion is a mis-understanding of the "normal" concept for switch contacts, be they real or virtual. The `IN_switch_Stop` virtual contact is programmed to be normally-open (NO), but yet it is *typically* found in the closed state. Recall that the "normal" status of any switch is its status while in a resting condition of no stimulation, *not* necessarily its status while the process is in a "normal" operating mode. The "normally-open" virtual contact `IN_switch_Stop` is typically found in the closed state because its corresponding input channel is typically found energized, owing to the normally-closed pushbutton switch contact, which passes real electrical power to the input channel while no one presses the switch. Just because a switch is configured as normally-open does not necessarily mean it will be *typically* found in the open state! The status of any switch contact, whether real or virtual, is a function of its configuration (NO versus NC) and the stimulus applied to it.

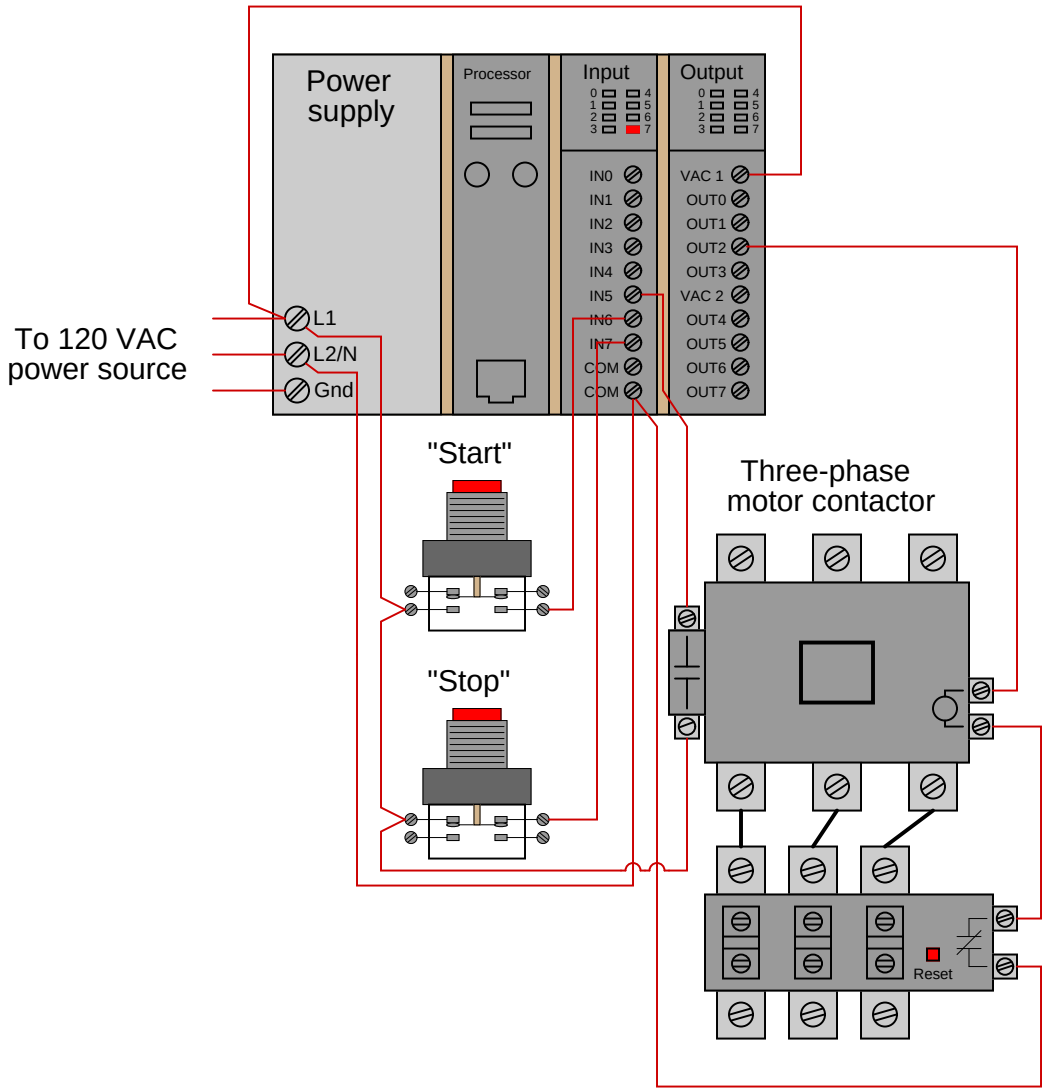
[12] In an effort to alleviate this confusion, the Allen-Bradley corporation (Rockwell) uses the terms *examine if closed* (XIC) and *examine if open* (XIO) to describe "normally open" and "normally closed" virtual contacts, respectively, in their Ladder Diagram programming. The idea here is that a virtual contact drawn as a normally-open symbol will be "examined" (declared "true") by the PLC's processor if its corresponding input channel is energized (powered by a real-life contact in the *closed* state). Conversely, a virtual contact drawn as a normally-closed symbol (with a slash mark through the middle) will be "examined" by the PLC's processor if its corresponding input channel is de-energized (if the real-life contact sending power to that terminal is in the open state). In my experience, I have found this nomenclature to be even more confusing to students than simply calling these virtual contacts "normally open" and "normally closed" like other PLC manufacturers do. The foundational concept for students to grasp here is that *the virtual contact is not a direct representation of the real-life electrical switch contact – rather, it is a read instruction for the bit set by power coming from the real-life electrical switch contact.*

Another concern surrounding real-world wiring problems is what this system will do if the motor contactor coil circuit opens for any reason. An open circuit may develop as a result of a wire falling off a screw terminal, or it may occur because the thermal overload contact tripped open due to an over-temperature event. The problem with our motor start/stop system as designed is that it is not "aware" of the contactor's real status. In other words, the PLC "thinks" the contactor will be energized any time discrete output channel 2 is energized, but that may not actually be the case if there is an open fault in the contactor's coil circuit.

This may lead to a dangerous condition if the open fault in the contactor's coil circuit is later cleared. Imagine an operator pressing the "Start" switch but noticing the motor does not actually start. Wondering why this may be, he or she goes to look at the overload relay to see if it is tripped. If it is tripped, and the operator presses the "Reset" button on the overload assembly, the motor will immediately start because the PLC's discrete output has remained energized all the time following the pressing of the "Start" switch. Having the motor start up as soon as the thermal overload is reset may come as a surprise to operations personnel, and this could be quite dangerous if anyone happens to be near the motor-powered machinery when it starts.

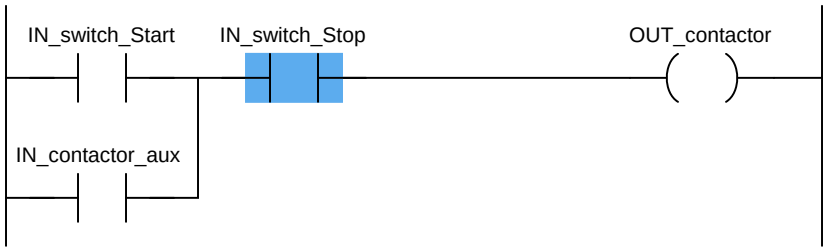
What would be safer is a motor control system that refuses to "latch" on unless the contactor actually energizes when the "Start" switch is pressed. For this to be possible, the PLC must have some way of sensing the contactor's status.

In order to make the PLC "aware" of the contactor's real status, we may connect the auxiliary switch contact to one of the unused discrete input channels on the PLC, like this:



Now, the PLC is able to sense the real-time status of the contactor via input channel 5.

We may modify the PLC program to recognize this status by assigning a new tag name to this input (`IN_contactor_aux`) and using a normally-open virtual contact of this name as the seal-in contact instead of the `OUT_contactor` bit:



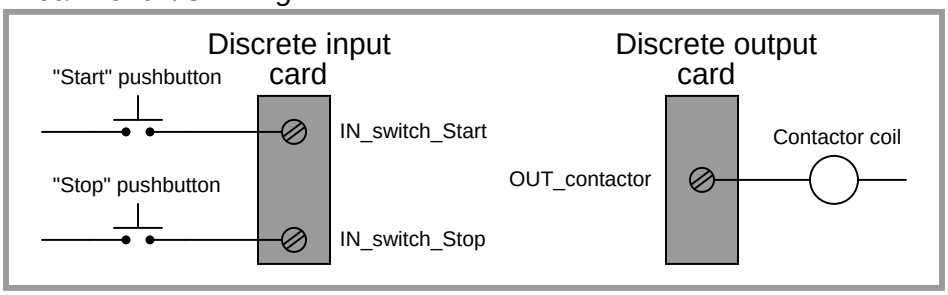
Now, if the contactor fails to energize for any reason when the operator presses the "Start" switch, the PLC's output will fail to latch when the "Start" switch is released. When the open fault in the contactor's coil circuit is cleared, the motor will *not* immediately start up, but rather wait until the operator presses the "Start" switch again, which is a much safer operating characteristic than before.

3.6 Ladder diagram retentive coils

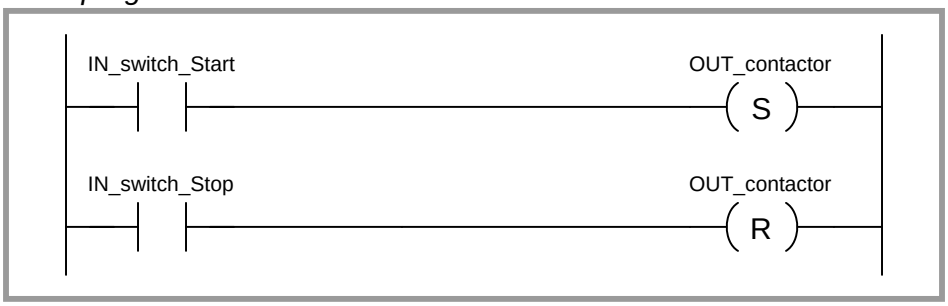
A special class of virtual "coil" used in PLC ladder programming that bears mentioning is the "latching" coil. These usually come in two forms: a *set* coil and a *reset* coil. Unlike a regular "output" coil that positively writes to a bit in the PLC's memory with every scan of the program, "set" and "reset" coils only write to a bit in memory when energized by virtual power. Otherwise, the bit is allowed to retain its last value.

A very simple motor start/stop program could be written with just two input contacts and two of these latching coils (both bearing the same tag name, writing to the same bit in memory):

Real-world I/O wiring



PLC program



Note the use of a normally-open (NO) pushbutton switch contact (again!), with no auxiliary contact providing status indication of the contactor to the PLC. This is a very minimal program, shown for the strict purpose of illustrating the use of "set" and "reset" latching coils in Ladder Diagram PLC programming.

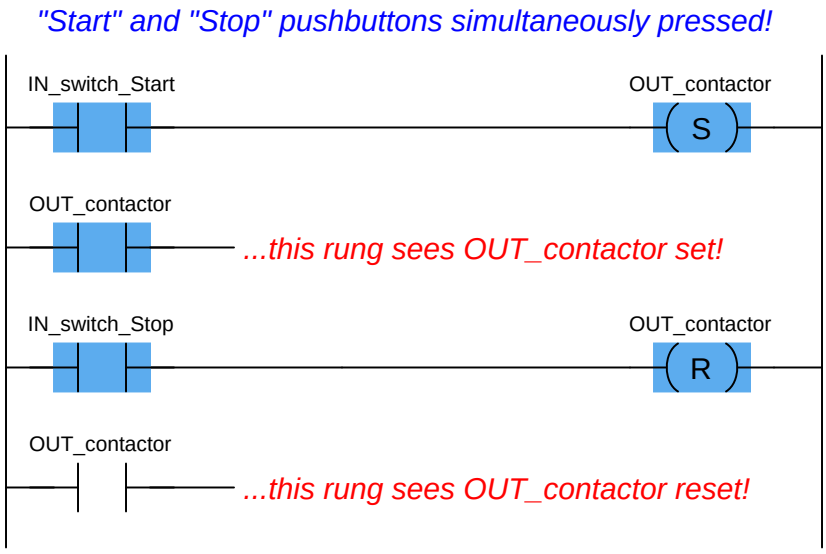
"Set" and "Reset" coils[13] are examples of what is broadly known in the world of PLC programming as *retentive instructions*. A "retentive" instruction *retains* its value after being virtually "de-energized" in the Ladder Diagram "circuit." A standard output coil is *non-retentive*, which means it does not "latch" when de-energized. The concept of retentive and non-retentive instructions will appear again as we explore PLC programming, especially in the area of *timers*.

[13] Referred to as "Output Latch" (OTL) and "Output Unlatch" (OTU) coils by Rockwell/Allen-Bradley.

Ordinarily, we try to avoid multiple coils bearing the same label in a PLC Ladder Diagram program. With each coil representing a "write" instruction, multiple coils bearing the same name represents multiple "write" operations to the same bit in the PLC's memory. Here, with latching coils, there is no conflict because each of the coils only writes to the `OUT_contactor` bit when its respective contact is energized. So long as only one of the pushbutton switches is actuated at a time, there is no conflict between the identically-named coils.

This raises the question: what would happen if *both* pushbutton switches were simultaneously pressed? What would happen if *both* "Set" and "Reset" coils were "energized" at the same time? The result is that the `OUT_contactor` bit would first be "set" (written to a value of 1) then "reset" (written to a value of 0) in that order as the two rungs of the program were scanned from top to bottom. PLCs typically do not typically update their discrete I/O registers while scanning the Ladder Diagram program (this operation takes place either before or after each program scan), so the real discrete output channel status will be whatever the *last* write operation told it to be, in this case "reset" (0, or off).

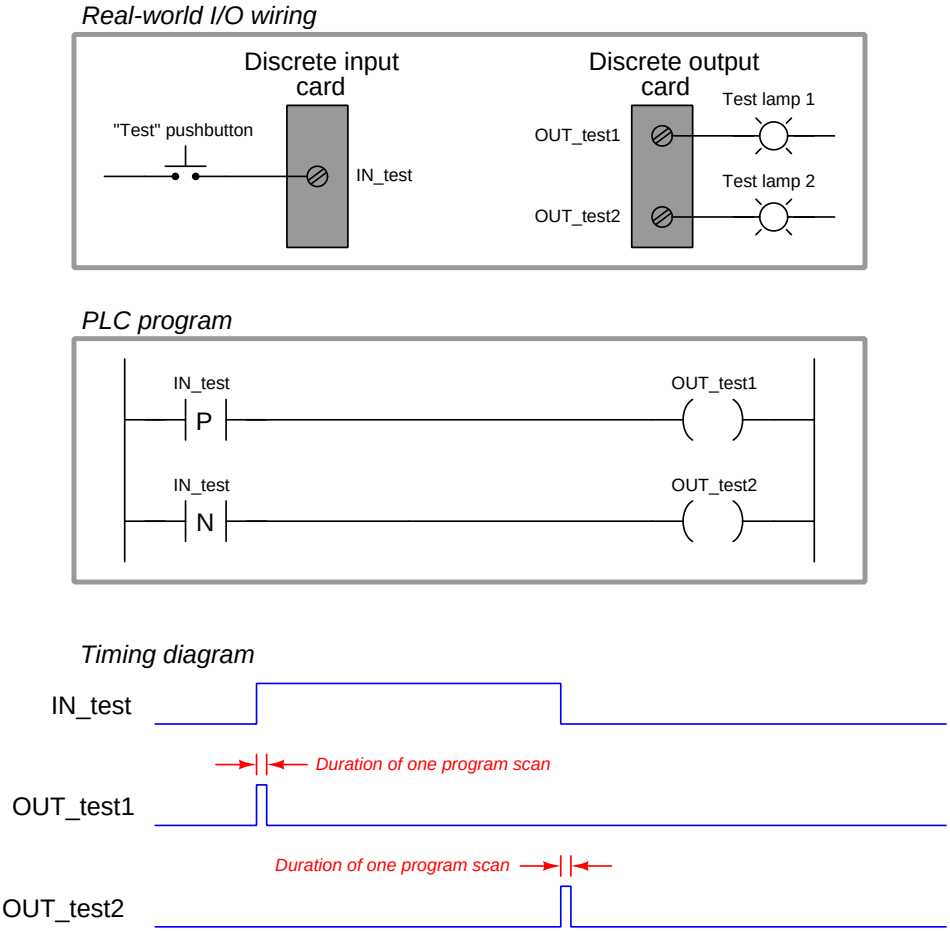
Even if the discrete output is not "confused" due to the conflicting write operations of the "Set" and "Reset" coils, other rungs of the program written between the "Set" and "Reset" rungs might be. Consider for example a case where there were other program rungs following the "Set" and "Reset" rungs reading the status of the `OUT_contactor` bit for some purpose. Those other rungs *would* indeed become "confused" because they would see the `OUT_contactor` bit in the "set" state while the actual discrete output of the PLC (and any rungs following the "Reset" rung) would see the `OUT_contactor` bit in the "reset" state:



Multiple (non-retentive) output coils with the same memory address are almost always a programming *faux pax* for this reason, but even retentive coils which are designed to be used in matched pairs can cause trouble if the implications of simultaneous energization are not anticipated. Multiple *contacts* with identical addresses are no problem whatsoever, because multiple "read" operations to the same bit in memory will never cause a conflict.

3.7 Ladder diagram transition contacts

The IEC 61131-3 PLC programming standard specifies *transition-sensing* contacts as well as the more customary "static" contacts. A transition-sensing contact actuates only for the duration of one program scan even if its corresponding bit remains active for multiple scans. Two types of transition-sensing Ladder Diagram contacts are defined in the IEC standard: one for *positive* (false-to-true) transitions and another for *negative* (true-to-false) transitions. The following example shows a wiring diagram, Ladder Diagram program, and a timing diagram (time progressing from left to right) demonstrating how each type of transition-sensing contact functions when stimulated by a real (electrical) input signal to a discrete channel:

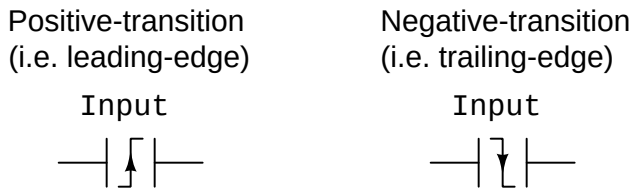


When the pushbutton switch is pressed and the discrete input energized, the first test lamp will blink "on" for exactly one scan of the PLC's program, then return to its off state. The positive-transition contact (with the letter "P" inside) activates the coil `OUT_test1` only during the scan it sees the status of `IN_test` transition from "false" to "true," even though the input remains energized for many scans after that transition. Conversely, when the pushbutton switch is released and the

discrete input de-energizes, the second test lamp will blink "on" for exactly one scan of the PLC's program then return to its off state. The negative-transition contact (with the letter "N" inside) activates the coil `OUT_test2` only during the scan it sees the status of `IN_test` transition from "true" to "false," even though the input remains de-energized for many scans after that transition.

The duration of a single PLC program scan is typically very short: usually tens of milliseconds. If this program were actually tested in a real PLC, you would probably not be able to see either test lamp light up, each pulse being so short-lived. Transition contacts are typically used any time we need to execute an instruction just one time following a "triggering" event, as opposed to executing that instruction over and over again so long as the event status is "true".

Not all PLCs follow the IEC 61131-3 programming standard, and may have alternative versions of transition instructions. For example, transition contacts may appear with "leading edge" or "falling edge" pulse symbols within as shown below:

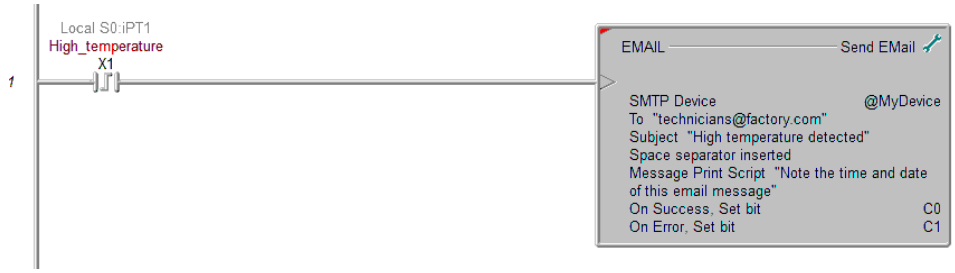


These leading-edge and trailing-edge contact instructions function exactly the same as the positive-transition and negative-transition instructions as defined by IEC 61131-3, respectively, but merely utilize different graphical symbols. Similarly, some PLCs avoid introducing new contact types and instead opt for entirely new instructions designed to be placed to the right of regular contact instructions in a rung. Most notably, Rockwell (Allen-Bradley) PLCs offer One Shot Rising (`OSR`) and One Shot Falling (`OSF`) instructions that pass virtual power through only when their rung-in condition transitions from false to true (OSR) or from true to false (OSF):

Any instruction placed to the right of the `OSR` or `OSF` will be activated for exactly one program scan every time the regular contact instruction to the left transitions from uncolored to colored (`OSR`) or from colored to uncolored (`OSF`).

Transition-sensing contact instructions find application in PLC programming where some instruction must be activated for just one program scan and not for multiple successive scans in a row. An example of an instruction that should only be executed once rather than over and over again during successive program scans is a *communication* instruction, such as transmitting data to a radio paging system or sending an email message over the internet to alert personnel of abnormal conditions. If the initiating event (e.g. a pushbutton switch, or a proximity switch detecting an out-of-position machine) persists, there is no point in transmitting a rapid series of messages from the PLC again and again with every scan – one transmitted message is enough. Such communication instructions are usually triggered by transition contacts rather than regular contact instructions in order to ensure just one transmission per event.

In the following program screenshot we see a positive-transition (leading-edge) contact instruction connected to activate an email-send function within a PLC. The contact is associated with a bit driven by a discrete input connected to a high-temperature sensing switch, so that the occurence of a high-temperature event triggers the PLC to send an email message to a group account for all technicians working at that factory:

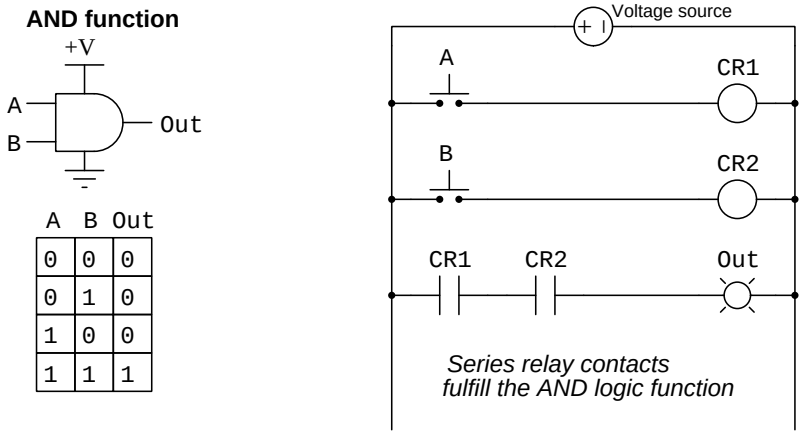


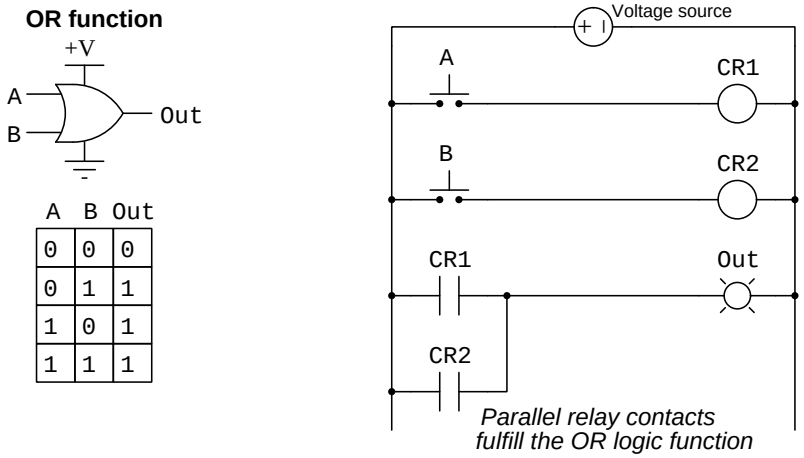
A transition contact instruction makes perfect sense for this application, as it will cause the email instruction to activate just one time for every high-temperature excursion. If the temperature drifts past the high-alarm point once, all technicians receive just one email message in their inboxes. If the temperature rises past the alarm point and then falls back below, then rises up one more time, the PLC will send one email message for every time the temperature switch closes. Multiple email messages therefore will represent multiple times when the temperature increased from nominal to too-hot.

3.8 Ladder diagram logic functions

A *logic function* is a relationship between one or more discrete input signals and an output signal. In the study of digital systems the most basic logic functions are *AND* and *OR*, their names describing the input conditions necessary to activate their outputs. In the case of an "AND" function, *all* inputs must be activated ("true") in order for the output to activate; i.e. the first input must be active *and* the second input must be active *and* etc. In the case of an "OR" function, *any* one of the inputs need be activated ("true") in order for the output to activate; i.e. either the first input must be active *or* the second input must be active *or* etc.

Such logic functions may be hardware-implemented either by using semiconductor integrated circuits called *logic gates* or by electromechanical relays with contacts wired in series for AND and in parallel for OR:

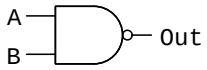




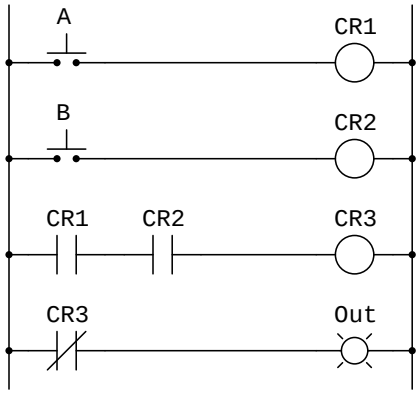
Note the *truth tables* shown by each of these functions, representing all possible combination of true (1) and false (0) states for any of these two-input functions.

Another important logical function is the *NOT*, which simply inverts the logical state of its single input. NOT functions are often combined with AND and OR functions to create NAND and NOR functions, respectively. In the case of the NAND function, all we need to do is invert the output of a standard AND function (series contacts), and we may do that by using another relay with a normally-closed contact to perform the added inversion (NOT) function. In the case of the NOR function, we invert the output of a standard OR function (parallel contacts) the same way:

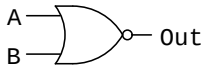
NAND function



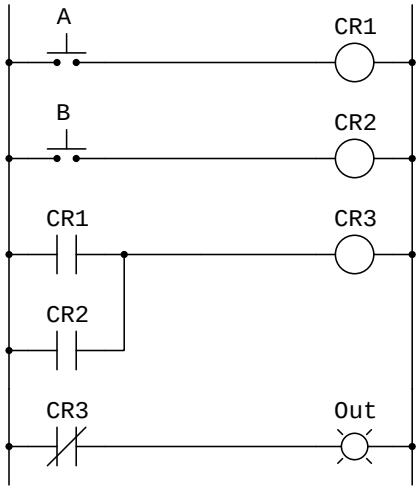
A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0



NOR function

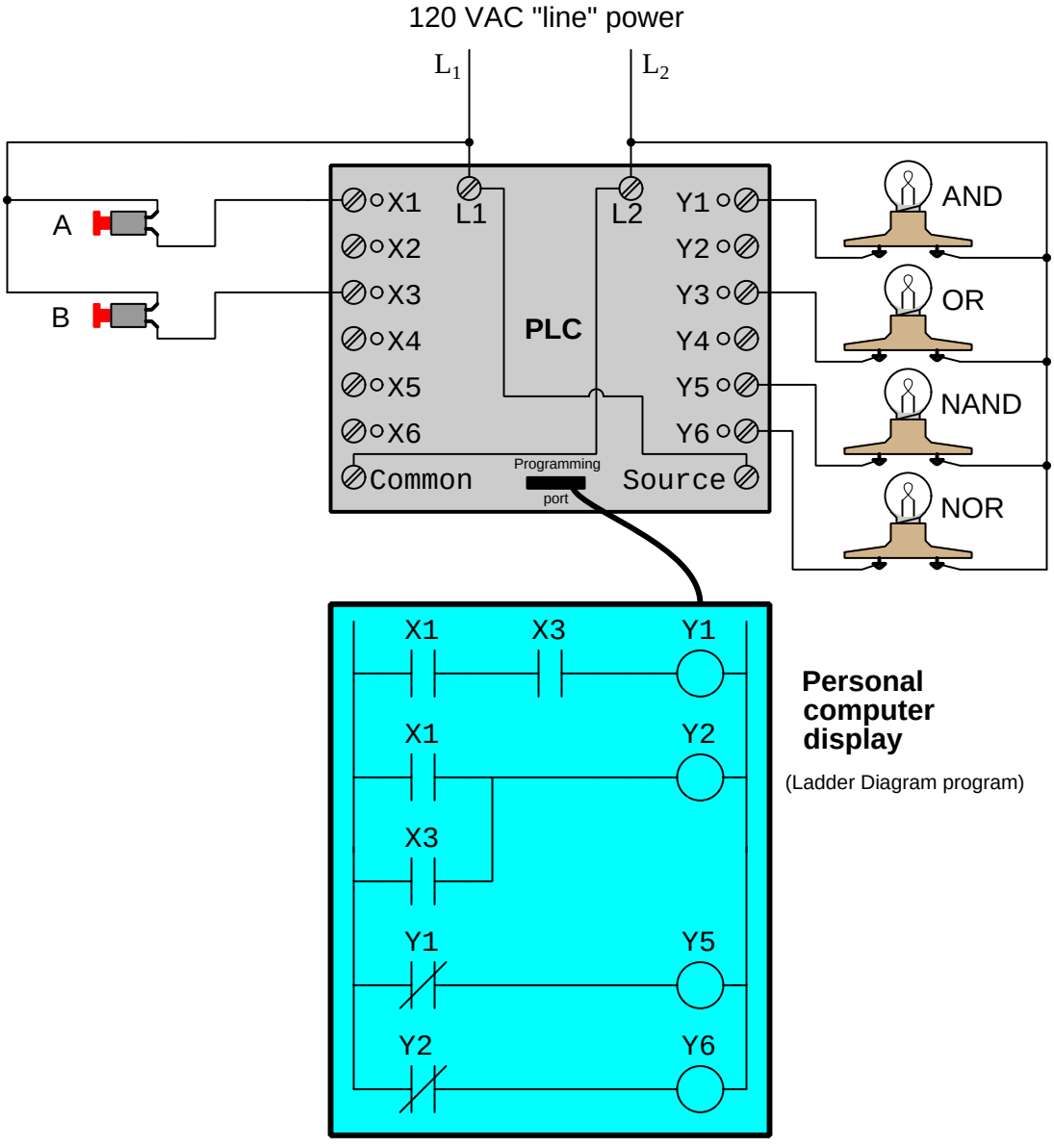


A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0



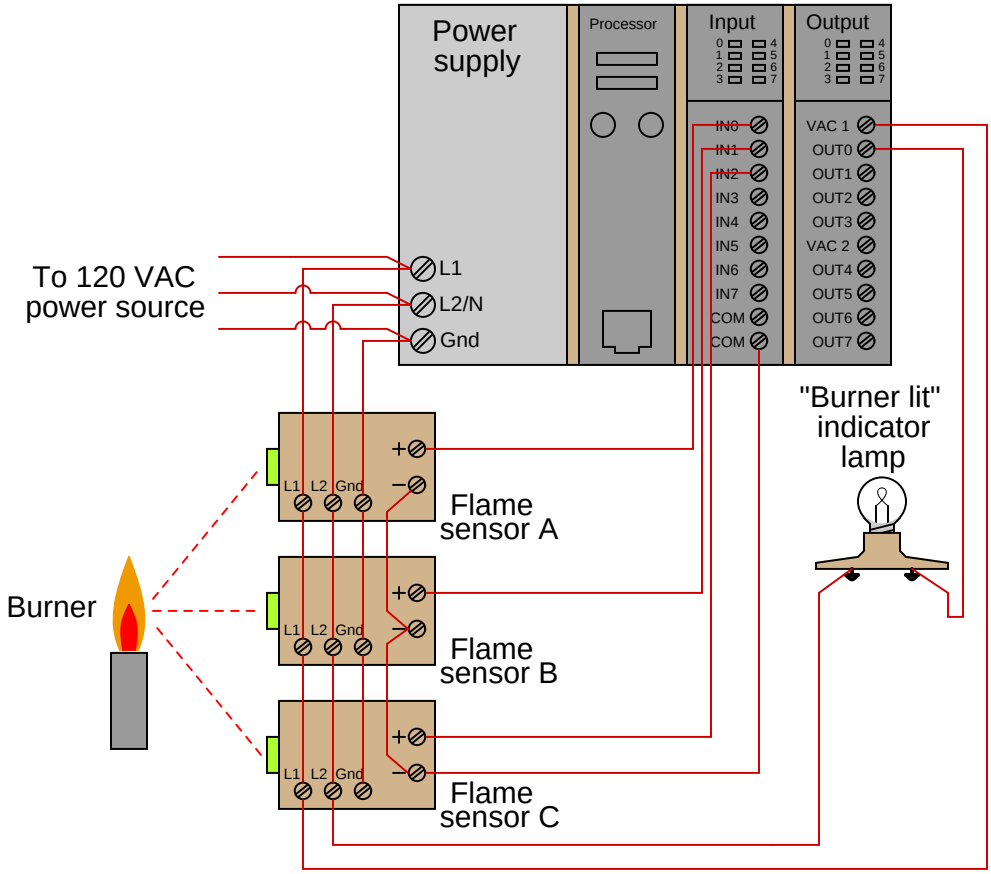
Once again we see *truth tables* calling out the relationships between input and output conditions where "true" is represented by the value 1 and "false" by the value 0.

Of course, anything we can build using electromechanical relays is easy to implement as Ladder Diagram programming in a PLC, and so it should come as no surprise that we can and often do implement a variety of logical functions in PLCs. Below we see a PLC wired and programmed to implement all four of these basic logic functions (AND, OR, NAND, NOR) with inputs provided by a pair of pushbutton switches and outputs shown by four lamps:



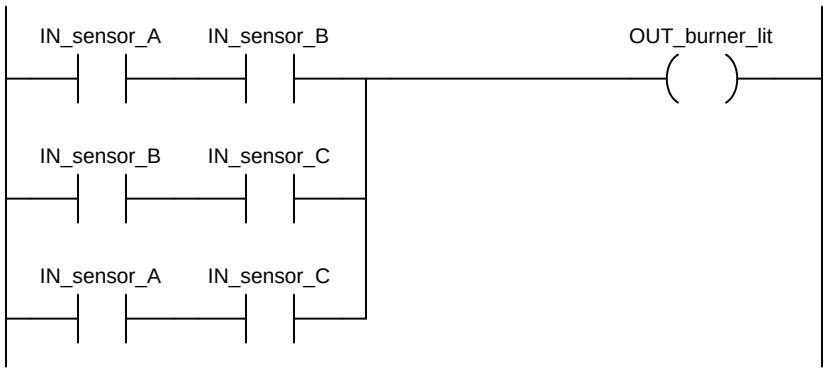
3.9 Ladder diagram combinational logic

When elementary logic functions are combined to make more complex functions, we refer to that as *combinational logic*. As you might imagine, a great many practical applications demand logical functions more complex than simple AND and OR. In this section we will explore a very practical application requiring a combination of AND and OR functions, this application employing three redundant flame sensors all monitoring the status of a burner flame. The purpose of this system will be to indicate a "lit" burner if at least two out of the three sensors indicate flame. If only one sensor indicates flame (or if no sensors indicate flame), the system will declare the burner to be un-lit. The burner's status will be visibly indicated by a lamp that human operators can readily see inside the control room area:



Each flame sensor outputs a DC voltage signal indicating the detection of flame at the burner, either on (24 Volts DC) or off (0 Volts DC). These three discrete DC voltage signals are sensed by the first three channels of the PLC's discrete input card. The indicator lamp is a 120 Volt lamp, and so must be powered by an AC discrete output card, shown here in the PLC's last slot.

A Ladder Diagram PLC program sufficient to determine if at least two out of the three sensors detect flame is shown below. To make the program more readable, we will assign tag names (symbolic addresses) to each input and output bit in the PLC, describing its real-world device in an easily-interpreted format. We will tag the first three discrete input channels as `IN_sensor_A`, `IN_sensor_B`, and `IN_sensor_C`, and the output as `OUT_burner_lit`.



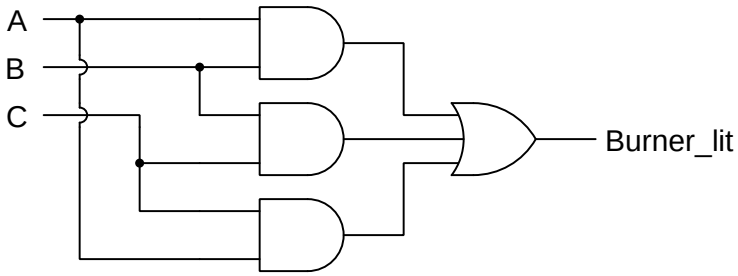
Series-connected contacts in a Ladder Diagram perform the logical `AND` function, while parallel contacts perform the logical `OR` function. Thus, this two-out-of-three flame-sensing program could be verbally described as:

"Burner is lit if either `A` *and* `B`, *or* either `B` *and* `C`, *or* either `A` *and* `C`"

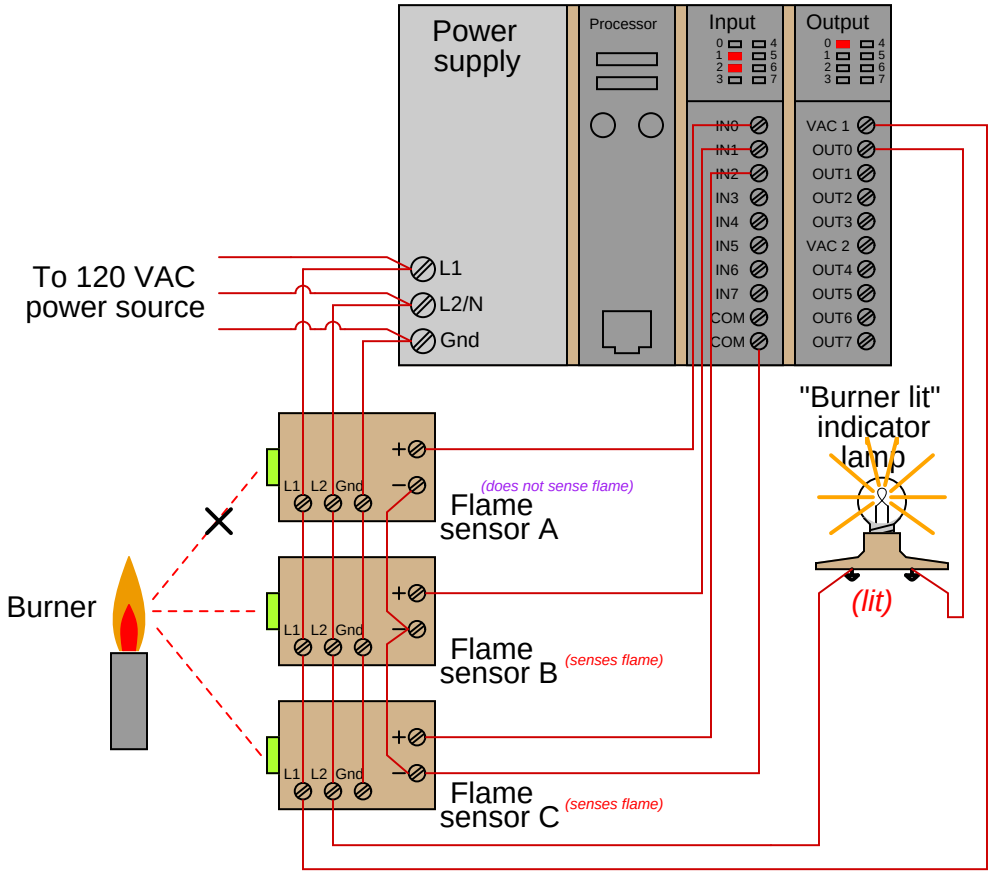
An alternate way to express this is to use the notation of *Boolean algebra*, where multiplication represents the `AND` function and addition represents the `OR` function:

$$\text{Burner_lit} = AB + BC + AC$$

Yet another way to represent this logical relationship is to use logic gate symbols:



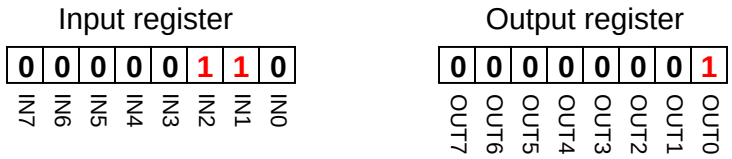
To illustrate how this program would work, we will consider a case where flame sensors B and C detect flame, but sensor A does not[14]. This represents a two-out-of-three-good condition, and so we would expect the PLC to turn on the "Burner lit" indicator light as programmed. From the perspective of the PLC's rack, we would see the indicator LEDs for sensors B and C lit up on the discrete input card, as well as the indicator LED for the lamp's output channel:



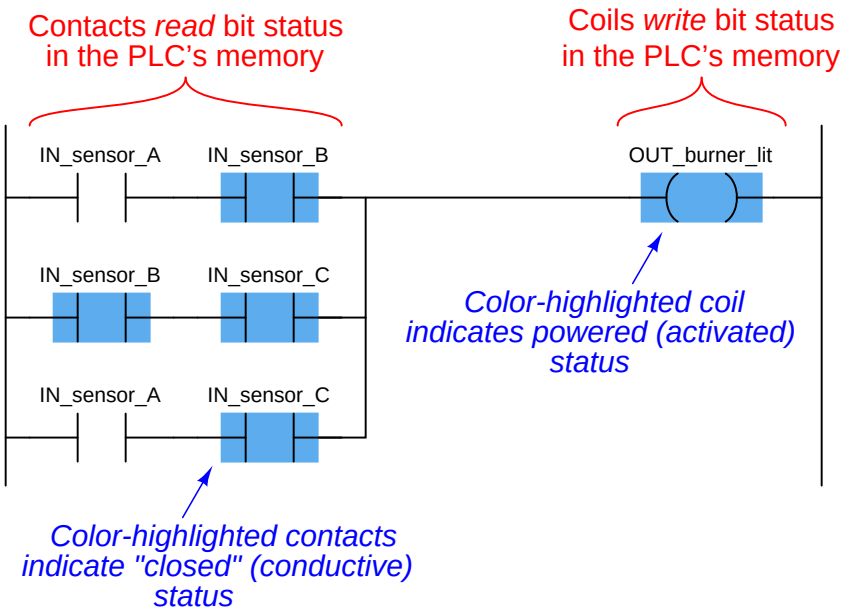
Those two energized input channels "set" bits (1 status) in the PLC's memory representing the status of flame sensors B and C. Flame sensor A's bit will be "clear" (0 status) because its corresponding input channel is de-energized. The fact that the output channel LED is energized (and the "Burner lit" indicator lamp is energized) tells us the PLC program has "set" that corresponding bit in the PLC's output memory register to a "1" state.

[14]The most likely reason why one out of two flame sensors might not detect the presence of a flame is some form of misalignment or fouling of the flame sensor. In fact, this is a good reason for using a 2-out-of-3 flame detection system rather than a simplex (1-out-of-1) detector scheme: to make the system more tolerant of occasional sensor problems without compromising burner safety.

A display of input and output register bits shows the "set" and "reset" states for the PLC at this moment in time:



Examining the Ladder Diagram program with status indication enabled, we see how only the middle contact pair is passing "virtual power" to the output coil:



Recall that the purpose of a contact in a PLC program is to *read* the status of a bit in the PLC's memory. These six "virtual contacts" read the three input bits corresponding to the three flame sensors. Each normally-open "contact" will "close" when its corresponding bit has a value of 1, and will "open" (go to its normal state) when its corresponding bit has a value of 0. Thus, we see here that the two contacts corresponding to sensor A appear without highlighting (representing no "conductivity" in the virtual relay circuit) because the bit for that input is reset (0). The two contacts corresponding to sensor B and the two contacts corresponding to sensor C all appear highlighted (representing "conductivity" in the virtual circuit) because their bits are both set (1).

Recall also that the purpose of a coil in a PLC program is to *write* the status of a bit in the PLC's memory. Here, the "energized" coil sets the bit for the PLC output 0 to a "1" state, thus activating the real-world output and sending electrical power to the "Burner lit" lamp.

Note that the color highlighting does *not* indicate a virtual contact is *conducting* virtual power, but merely that it is *able* to conduct power. Color highlighting around a virtual coil, however, *does* indicate the presence of virtual "power" at that coil.

3.10 Ladder logic for arbitrary truth tables

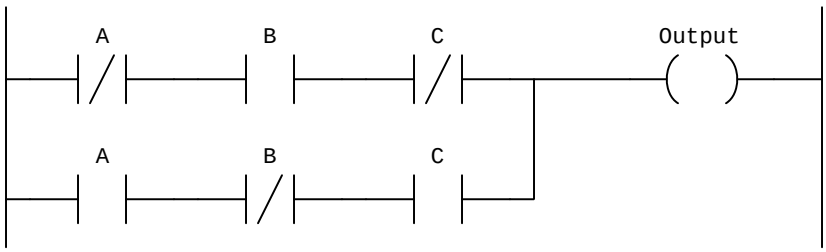
Suppose we needed to program a PLC to implement the following truth table:

A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Note how the output in this truth table is "true" (1) only during two of the eight possible input combinations:

When $A = 0$ *and* $B = 1$ *and* $C = 0$ *or* when $A = 1$ *and* $B = 0$ *and* $C = 1$

The words *and* and *or* are italicized here for a reason, as they indicate combinations of AND and OR logic that might be employed to this purpose. Since we know that AND functions are equivalent to *series* connections and OR functions are equivalent to *parallel* connections in electrical circuits, we may mimic these connections in the form of a Ladder Diagram PLC program as shown below:



The upper string of contacts will all be colored when $A = 0$ and $B = 1$ and $C = 0$, activating the output coil. Similarly, the lower string of contacts will all be colored when $A = 1$ and $B = 0$ and $C = 1$, also activating the output coil. However, for any other combination of 1 and/or 0 states on inputs A, B, and C there will not be continuous coloring across the Ladder Diagram program which means the output will be inactive during those conditions.

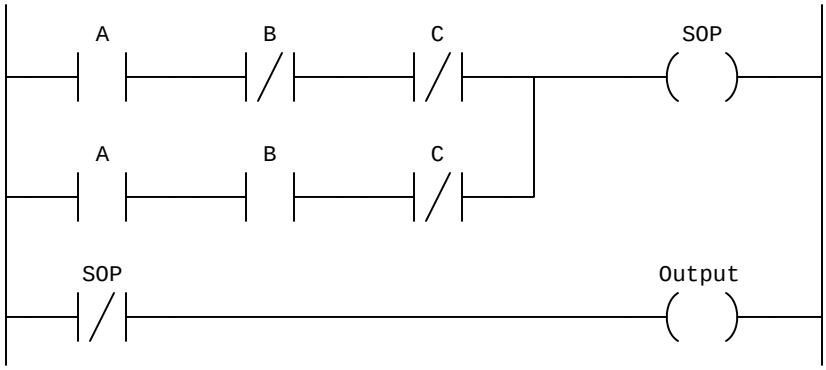
All we need to write in our Ladder Diagram program to implement any truth table is a set of series-connected contacts where each series string represents a particular combination of input states resulting in an activated output, and then parallel all of those series strings.

This technique, known as *Sum of Products* for its Boolean-algebra representation, is easy to implement and understand for truth tables with a minority of "1" output states, since the number of unique combinations resulting in an activated output is the number of series contact strings needed in our Ladder Diagram PLC program. However, for truth tables where most of the output states are "1" this becomes a clumsy solution.

Take for example the next truth table where a majority of the output states are "1" rather than "0":

A	**B**	**C**	**Output**
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

If we were to write our PLC Ladder Diagram program using the previous technique, that program would be much larger – requiring *six* series strings of contacts – and potentially more confusing to understand. However, there is a more elegant solution: rather than writing one string of Ladder Diagram logic for each of the "1" output conditions as we did before, we may instead write one string of series contacts for each of the "0" output conditions, and then use one more rung to invert ("NOT") that result to finally drive the PLC's output bit. The result looks like this:



In this program, the status of the SOP bit is that of the truth table if all its output bit states were inverted; i.e. if the two "0" output states were in fact "1" states and the other six "1" states were in fact "0". The last rung of logic simply inverts the state of the SOP bit to become the output bit.

Chapter 4

Derivations and Technical References

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

4.1 Feature comparisons between PLC models

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

4.1.1 Viewing live values

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

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

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

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

4.1.2 Forcing live values

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

4.1.3 Single-bit addressing

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

4.1.4 Special "system" values

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

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

4.1.5 Free-running clock pulses

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

4.1.6 Standard counter instructions

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

4.1.7 High-speed counter instructions

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

4.1.8 Timer instructions

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

4.1.9 ASCII text message instructions

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

4.1.10 Analog signal scaling

- Allen-Bradley Logix 5000: the I/O configuration menu (specifically, the *Module Properties* window) allows you to directly and easily scale analog input signal ranges into any arbitrary numerical range desired. Floating-point ("REAL") format is standard, but integer format may be chosen for faster processing of the analog signal.

- Allen-Bradley PLC-5, SLC 500, and MicroLogix: raw analog input values are 16-bit signed integers. The `SCL` and `SCP` instructions are custom-made for scaling these raw integer ADC count values into ranges of your choosing.

- Siemens S7-200: raw analog input values are 16-bit signed integers. Interestingly, the S7-200 PLC provides built-in potentiometers assigned to special word registers (`SMB28` and `SMB29`) with an 8-bit (0-255 count) range. These values may be used for any suitable purpose, including combination with the raw analog input register values in order to provide mechanical calibration adjustments for the analog input(s).

- Koyo (Automation Direct) DirectLogic: you must use standard math instructions (e.g. `ADD`, `MUL`) to implement a $y = mx + b$ linear equation for scaling purposes.

- Koyo (Automation Direct) CLICK: the I/O configuration menu allows you to directly and easily scale analog input signal ranges into any arbitrary numerical range desired.

4.2 Getting started with Rockwell Studio 5000

These instructions were developed using version 38.01.00 of Rockwell Automation's Studio 5000 Logix Designer software, communicating with a model 1756-L902TS ControlLogix PLC processor module.

Starting a new project online:

1. Open *Logix Designer* software.
2. Click on `File` → `New` to create a new project.
3. Click on `Communications` → `Network Browser`.
4. If you are connecting to the PLC processor using USB, expand on that item in the list; otherwise, choose the network connection you prefer.
5. Drill down on the processor icon (for ControlLogix this will be the one beginning with the 1756 part number).
6. Select `Backplane` and then click on the `Set Project Path` button.
7. Click on the `Go Online` button to begin working while connected to the PLC processor.
8. A `Connected to Go Online` window will appear, in which you will click on the `Download` button.
9. Looking in the `Controller Organizer` list on the left-hand side of the software window, drill down into `Tasks` → `MainTask` → `MainProgram` to raise the ladder-diagram programming window.
10. Right-click on `I/O Configuration` → `1756 Backplane` and choose `Discover Modules`.
11. Under the `Action` column, click on the `Create` button for each I/O module you wish to include in this project.
12. A `Device Definition` window will pop up for each module as you "create" it, where you confirm proper rack slot number and Revision number; here you may also write custom names and descriptions for each module if you prefer.
13. When prompted with a `Create new module online` dialogue box, choose `Create and Uninhibit` if you wish that I/O module to be active. Note that the default is `Create and Inhibit` which will render the module unworkable for your project!
14. Note that some modules will open up additional dialogue windows.
15. Go to the `MainProgram -- MainRoutine` window and start building your ladder-diagram program for the PLC to run, dragging and dropping new instructions into rungs.

16. To create tags as you write your program, right-click on the tagname placeholder (`?`) appearing at the top of each new placed instruction. If the new tagname must map to a real I/O point on the controller, be sure to choose `Alias` rather than `Base`, and then under `Alias For` you may choose the specific I/O point. Note that for basic inputs and outputs you will find the bits and words under "Data". For example, `Local:1:I.Data.0` represents channel 0 on a discrete input module plugged into slot 1 of the PLC chassis, and `Local:3:O.Data.5` represents channel 5 of a discrete output module plugged into slot 3.

17. When online-editing, all new lines and edits will show themselves as new on the program editing view by a dashed vertical line over the left-hand rail. Right-clicking on the rung number of that new/edited rung will allow you to `Accept Pending Rung Edits`, or there is also a green check-mark icon at the top of the editing window to do the same. This compiles the ladder code and displays any syntax errors found.

18. An icon with a fat right-pointing green arrow called `Finalize All Edits in Program` (hot-key sequence *Ctrl + Shift + F*) updates these edits in the running PLC processor. This is analogous to downloading a program to the PLC processor when the PLC is already in Run mode.

Offline programming differences:

1. After making any edits to the ladder-diagram program while offline, right-click on the rung number and select `Verify Rung`. This, like `Accept Pending Rung Edits` for online editing, compiles your code and lets you know if any syntax errors exist.

2. Click on `Communications` → `Download` to send this code to the PLC.

Rockwell PLC tags:

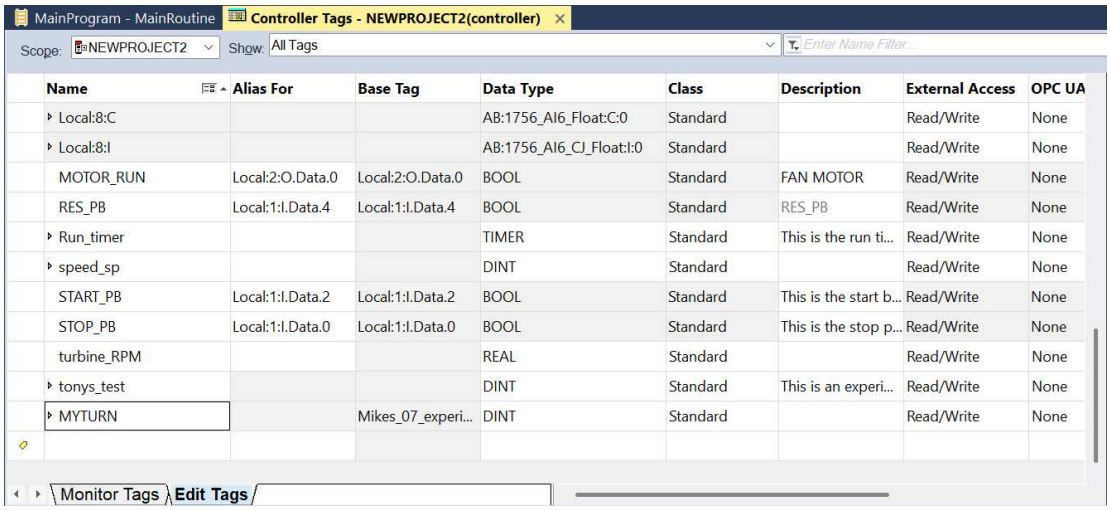
Rockwell Logix5000 programming is designed around the concept of alphanumerically-named *tags* rather than the fixed "file" designations used for bits and words in legacy Rockwell/Allen-Bradley controller products. Much about Logix5000 programming borrows from text-based programming languages such as C and C++, where tagnames serve the same purpose as *variable names* in those typed languages. For example, tags may be of a certain data type such as `BOOL` (bit) or `REAL` (floating-point) or `DINT` (double-width signed integer), or they may be comprised of multiple data elements (arrays) or multiple data types (structures) just like in C or C++, and just like variable names in C or C++ nearly any tagname is permitted using alphanumeric characters (with no spaces). Also, every tagname has a *scope* defining which levels of the PLC's programming it may be accessed. A tagname with scope set for `MainProgram` will only be readable and writable to that section of the program, just like a variable in C is local to a function if declared within that function. By contrast, a tagname with scope set at the project level is considered *global* and may be referenced by any portion of that PLC's program just like a variable in C declared at the start of the program before the `main` function.

Whenever possible, begin any new Rockwell Logix5000 PLC programming project by building out a comprehensive list of tags before writing any ladder-diagram code, making sure each tagname is clearly descriptive and not ambiguous. This is analogous to declaring all variables in C or C++ before writing any functions. Doing so helps organize your thoughts and helps make the writing of logic more self-documenting as each and every tag you use has a sensible name.

Four major categories of tags exist within Logix5000 systems:

- **Base tags** – these are tags belonging directly to a particular project on a particular PLC, and this is the default category for all new tags you as the programmer will create.
- **Alias tags** – these are tags linked to other existing tags with the sole purpose of accessing that data via a different tagname. Alias tags are commonly linked to hardware I/O addresses (e.g. `Local:3:I.Data.5` may be aliased to `Start_PB_switch` so when it appears near a contact instruction its purpose (i.e. a discrete bit controlled by a pushbutton switch wired to an input module) is obvious to anyone reading that ladder-diagram code.)
- **Produced tags** – a category of tags used in cyclic controller-to-controller communication, a produced tag residing in a controller for the purpose of broadcasting that data onto a digital network where it may be read into one or more tags within other controllers.
- **Consumed tags** – a category of tags used in cyclic controller-to-controller communication, a consumed tag residing in a controller for the purpose of receiving data broadcast (published) by another controller. Every consumed tag reads from *one* produced tag in some other controller, and this produced $\rightarrow$ consumed communication occurs at regular intervals without requiring any ladder-diagram code execution. The regular interval time is known as the *Requested Packet Interval* or *RPI*.

The following screenshot shows a partial tagname list configured into one particular Rockwell PLC used to monitor and control a turbine engine:



Name	Alias For	Base Tag	Data Type	Class	Description	External Access	OPC UA
Local:8:C			AB:1756_AI6_Float:C:0	Standard		Read/Write	None
Local:8:I			AB:1756_AI6_CJ_Float:I:0	Standard		Read/Write	None
MOTOR_RUN	Local:2:O.Data.0	Local:2:O.Data.0	BOOL	Standard	FAN MOTOR	Read/Write	None
RES_PB	Local:1:I.Data.4	Local:1:I.Data.4	BOOL	Standard	RES_PB	Read/Write	None
Run_timer			TIMER	Standard	This is the run ti...	Read/Write	None
speed_sp			DINT	Standard		Read/Write	None
START_PB	Local:1:I.Data.2	Local:1:I.Data.2	BOOL	Standard	This is the start b...	Read/Write	None
STOP_PB	Local:1:I.Data.0	Local:1:I.Data.0	BOOL	Standard	This is the stop p...	Read/Write	None
turbine_RPM			REAL	Standard		Read/Write	None
tonys_test			DINT	Standard	This is an experi...	Read/Write	None
MYTURN		Mikes_07_experi...	DINT	Standard		Read/Write	None

In this list we see multiple tags with empty `Alias For` and `Base Tag` fields. These are actually *base* tags, but they were arbitrarily created during the programming process: `turbine_RPM` is one of these, receiving the a calculated result performed by a math instruction in the program; `speed_sp` is another. Some of the other tags in this list contain hardware I/O addresses in both the `Alias For` and `Base Tag` fields, and these are *alias* tags: `RES_PB` is one of these, being the user-named tag for a "reset" pushbutton switch wired to channel 4 of a discrete input card inserted into slot 1 of the local PLC chassis. The tag named `MYTURN` is actually a *consumed* tag, receiving data at regular intervals from a produced tag residing inside a remote PLC named `Mikes_07_experiment`, which we know by noting that PLC's name in the `Base Tag` field. A single produced tag exists here but unfortunately bears no hint of being such: `tonys_test`. This tag may be identified as a produced tag only by viewing its properties in a different window.

Configuring produced and consumed tags:

- *Produced* and *Consumed* tags are special tags in the PLC where the PLC broadcasts data (Produce) or subscribes to broadcasted data (Consume) data. So, the data contained in a Produced tag in one PLC gets sent over a network to become a Consumed tag in one or more other PLCs. With Rockwell PLCs, this exchange happens automatically without need for additional rungs of logic in the program.
- Go offline. Creating a new Produced tag or a new Consumed tag requires you be offline!
- An easy way to create a new tag is to right-click on `Controller Tags` in the Controller Organizer list. This opens up a pop-up window with *all* the tag's parameters in one place which is generally more convenient than scrolling horizontally in the tag browser window and looking for hidden drop-down menu options.
- Under the `Type` option in the `New Tag` window, you will find the option to make the tag `Produced` or `Consumed`. If the tag is `Produced`, there isn't much more to specify except for tag `Name` and `Data Type`. If the tag is `Consumed`, you must also specify which PLC is producing it, and you do that by clicking the `Connection` button to the right of the `Type` field.
- When the `Consumed Tag Connection` pop-up window appears, there will be a `Producer` field where you specify the name of the PLC producing that tag. You will also specify the Produced tag name inside of that producer PLC in the `Remote Data` field.
- To be able to specify that producer PLC by name within the consumer PLC, it must already be listed in the consumer PLC's `Controller Organizer` list. This is done by going to the `I/O Configuration` category within that list, then drill down to the specific communications port (e.g. the `A1 Ethernet` port), and then right-click on that port to create a `New Module`. A `Select Module Type` pop-up window will appear, with a list of all Rockwell product numbers, one for each PLC module in the Logix5000 family. Choose the module number for the producer PLC's processor card (e.g. `1756-L902TS`) and then another window pops up called `New Module` where you specify a name for this PLC (which will be the producer parameter from the previous step) as well as its communication parameters (e.g. IP address). Once the name of this producer PLC is set and its IP address also set, then you can download all of these changes to your PLC, and it will begin looking for data to put into the new consumed tag.

Analog input module configuration:

- Ensure that the I/O module is not inhibited. You can tell by checking that the "OK" status light is solid green, rather than blinking.
- Go offline. You cannot change I/O module parameters while online!
- Right-click on the particular I/O module under `I/O Configuration` in the left-hand pane of Studio 5000 and select `Properties`. This will open up a new tab/window where you can tell the module how to function.
- Under `Overview`, click on `Device definition` and start selecting your desired options for the analog module. For example, here you would select either single-ended or differential channel type if both are supported.
- Download/accept all changes, then with the controller in Remote Run mode, view values in Controller Tags. Note that under the `View` menu you must have the `Value` display option checked to see any live values.
- Under `Local:X:I.ChYData` you should see the live analog input value (where `X` is the slot number into which the module has been plugged and `Y` is the channel number on the module).
- Sometimes Studio 5000 software does strange and unexpected things. For example, sometimes you have have to go offline then online again just to see live analog data for the channel (otherwise it may appear frozen at a fixed value).

Modbus communication:

As popular as Modbus is and has been for many years as an industrial communication standard, the lack of convenient support within Logix5000 PLCs for Modbus suggests the Rockwell philosophy is biased against its use. Nevertheless, one can download special Modbus instructions from Rockwell (called *Add-On Instructions* or *AOI*s) for Modbus/TCP communication. Alternatively, Rockwell manufactures *gateway* devices to map Modbus/TCP registers to EtherNet/IP points.

General tips:

- When placing instructions on the ladder-diagram MainRoutine area, you will need to drop that new instruction where the green dot appears on the run, otherwise it won't stick.
- Click on `Windows` → `Reset Default Layout` to restore the editor layout to its factory-default state, in case you do something that makes the viewing areas and toolbars unusable.
- Click on `View` → `Toolbars` → `Language Element` to toggle the quick-reference toolbox for instructions on and off.
- Right-clicking on any tagname appearing in the ladder logic program gives you the option to `Edit` properties of that tagname, opening up the same dialogue window used to create it.

4.3 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]:

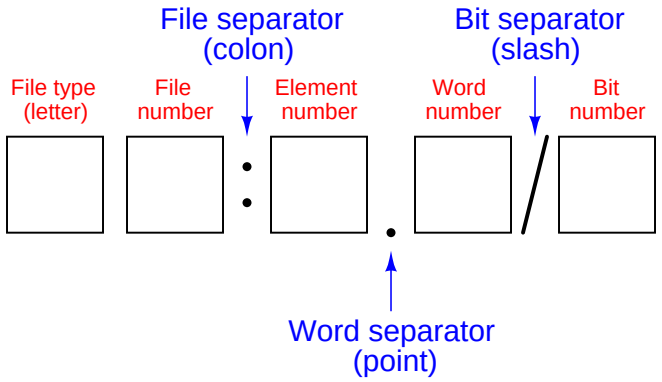
File number	File type	Logical address range
0	Output image	O:0 to O:30
1	Input image	I:0 to I:30
2	Status	S:0 to S:n
3	Binary	B3:0 to B3:255
4	Timers	T4:0 to T4:255
5	Counters	C5:0 to C5:255
6	Control	R6:0 to R6:255
7	Integer	N7:0 to N7:255
8	Floating-point	F8:0 to F8:255
9	Network	x9:0 to x9:255
10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

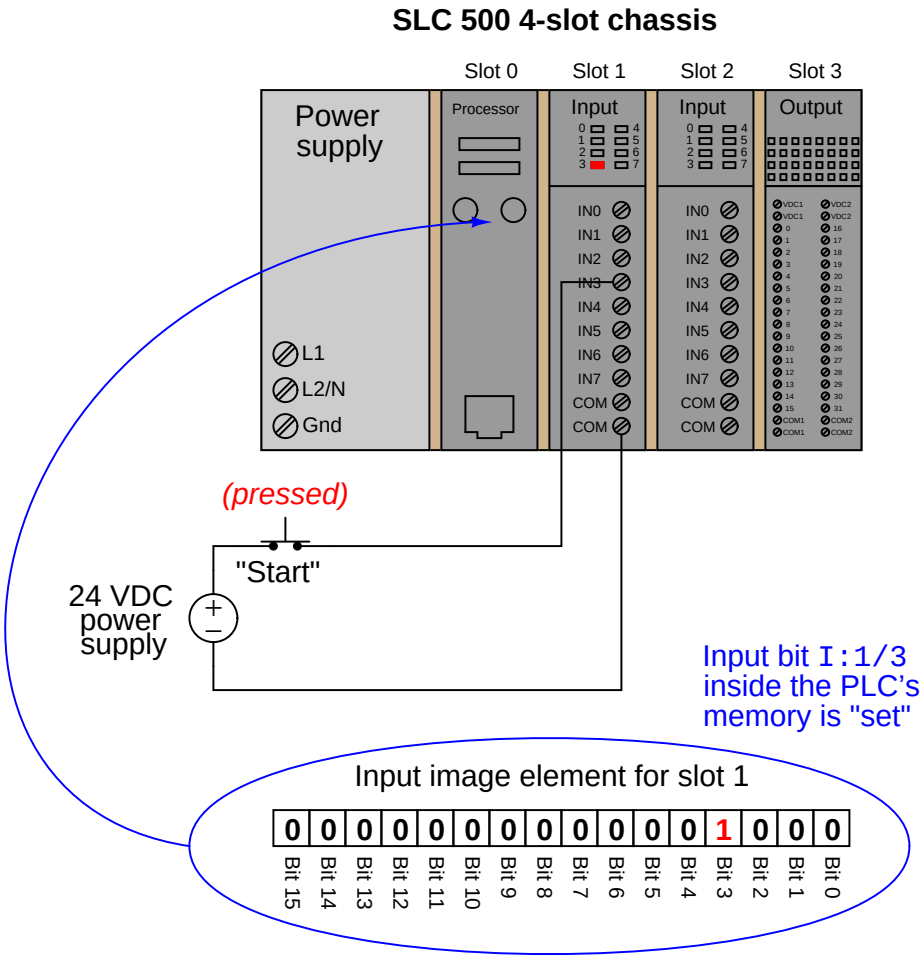
[3]This is not to say one *cannot* specify a particular bit in an otherwise whole word. In fact, this is one of the powerful advantages of Allen-Bradley's addressing scheme: it gives you the ability to precisely specify portions of data, even if that data is not generally intended to be portioned into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

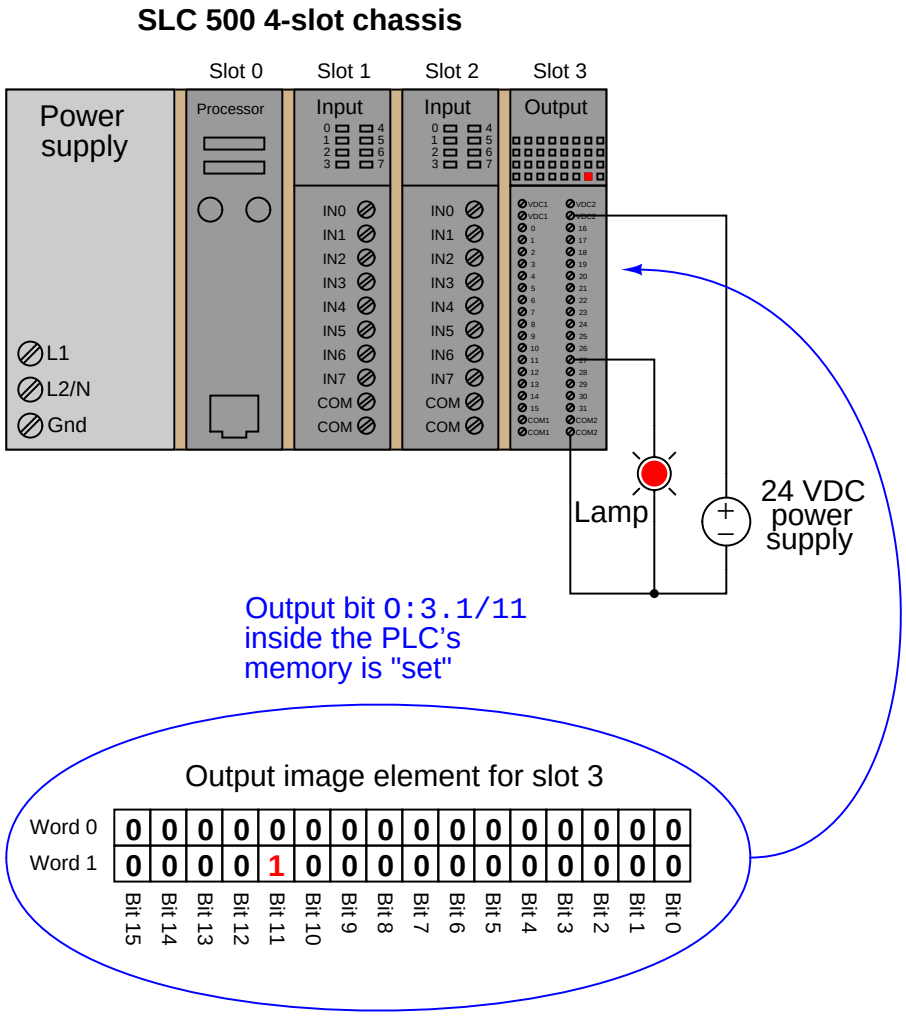
[5]Referencing the Allen-Bradley engineering literature, we see that the accumulator word may alternatively be addressed by number rather than by mnemonic, `T4:2.2` (word 2 being the accumulator word in the timer data structure), and that the "done" bit may be alternatively addressed as `T4:2.0/13` (bit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

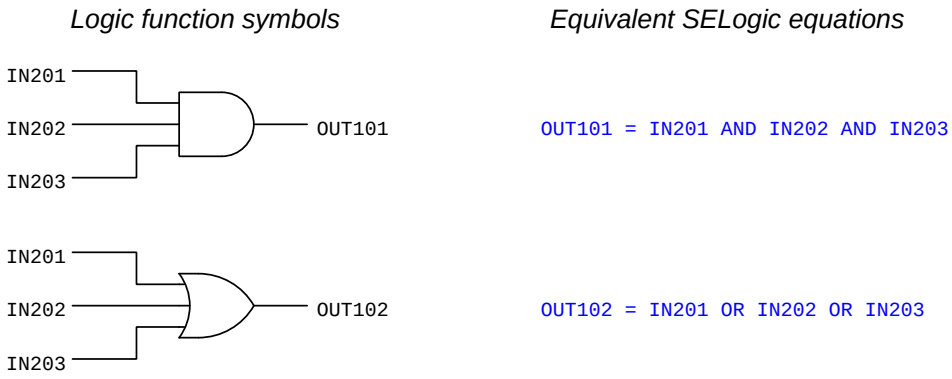
The modern trend in PLC addressing is to avoid the use of direct addresses such as `I:1/3` altogether, so they do not appear anywhere in the programming code. The Allen-Bradley "Logix" series of programmable logic controllers is the most prominent example of this new convention at the time of this writing.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

4.4 SELogic control equations

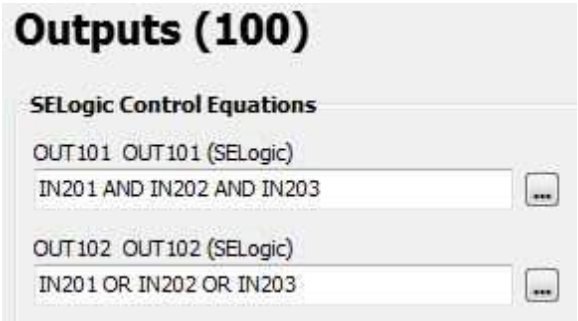
Some programmable logic controllers use Boolean-style equations to describe logical functions. A company called Schweitzer Engineering Laboratories (SEL) manufactures control equipment for the electric power industry, including a wide range of *protective relays* and *programmable automation controllers*,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

4.4.1 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:



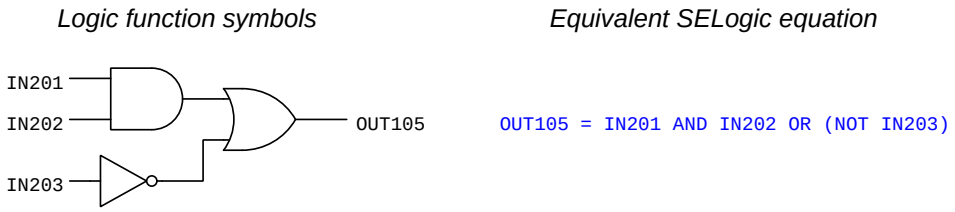
SEL programming software[7] provides text-entry fields for typing these control equations. A more primitive interface makes use of the built-in serial terminal server capability of SEL controllers, allowing programming edits to be made with nothing more than a serial terminal (e.g. personal computer 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.

4.4.2 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
```

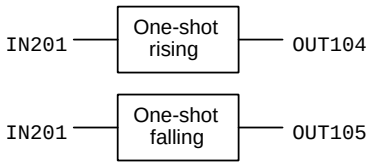
In SELogic, the name of the latch (e.g. `LT01`) is its output bit. Using the example just shown, we could direct output `OUT108` to be controlled by `LT01`'s output with an additional line of SELogic code:

```
OUT108 = LT01
```

4.4.3 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

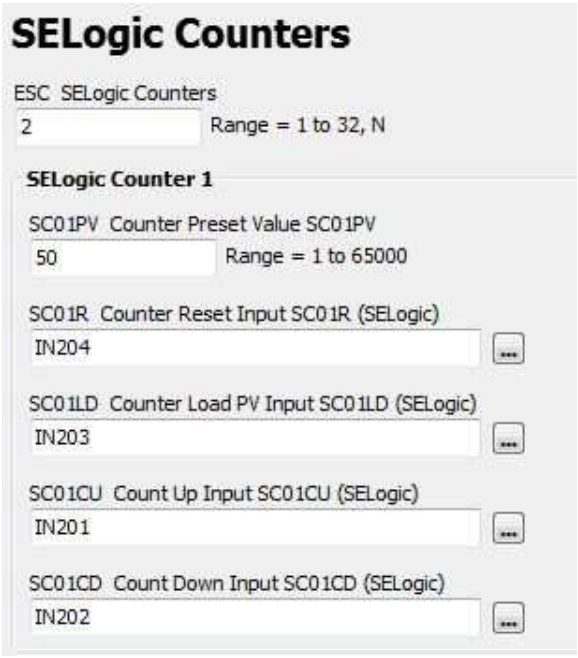


Equivalent SELogic equations

```
OUT104 = R_TRIG IN201

OUT105 = F_TRIG IN201
```

4.4.4 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
```

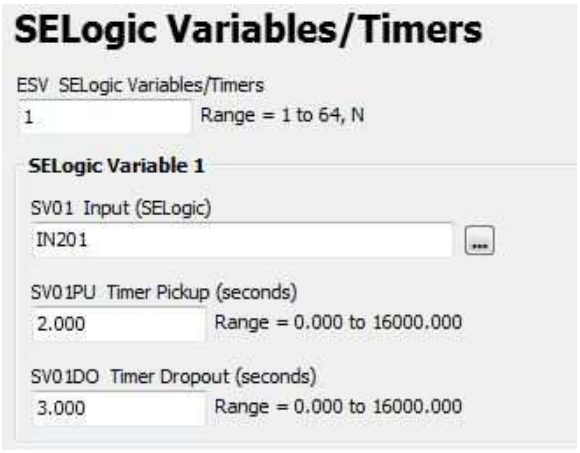
Once configured, the accumulated value of the first counter instruction is addressed simply as SC01. Discrete outputs based on the attainment of certain count values may be driven by comparison statements such as =, >, and <. For example, to activate output OUT107 whenever the first counter's accumulated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

4.4.5 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
```

4.5 Process switch types and symbols

A "process" switch is one actuated by some physical parameter that is not human. A pushbutton switch or a toggle switch would *not* be a process switch, but a fluid pressure switch, a temperature switch, a fluid flow switch, etc., would all be examples of process switches.

As with all other types of electrical switches, the "normal" state of a process switch is a state of minimum or no stimulus. The illustration on the following page shows schematic diagram symbols for several different types of process switch, along with their open/closed states depending on each switch's "normal" type and the applied stimulus.

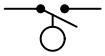
NO pressure switch

Open when there is zero pressure (minimum stimulus)
Closed when pressure increases beyond threshold

NC pressure switch

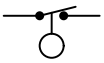
Closed when there is zero pressure (minimum stimulus)
Open when pressure increases beyond threshold

NO level switch



Open when there is zero level (minimum stimulus)
Closed when level increases beyond threshold

NC level switch



Closed when there is zero level (minimum stimulus)
Open when level increases beyond threshold

NO temperature switch

Open when temperature is cold (minimum stimulus)
Closed when temperature increases beyond threshold

NC temperature switch

Closed when temperature is cold (minimum stimulus)
Open when temperature increases beyond threshold

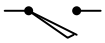
NO flow switch

Open when there is zero flow (minimum stimulus)
Closed when flow increases beyond threshold

NC flow switch

Closed when there is zero flow (minimum stimulus)
Open when flow increases beyond threshold

NO limit switch



Open when nothing touches (minimum stimulus)
Closed when physical contact made

NC limit switch

Closed when nothing touches (minimum stimulus)
Open when physical contact made

NO proximity switch

Open when nothing is near (minimum stimulus)
Closed with object is near

NC proximity switch

Closed when nothing is near (minimum stimulus)
Open when object is near

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.

???

???

???

???

???

???

???

???

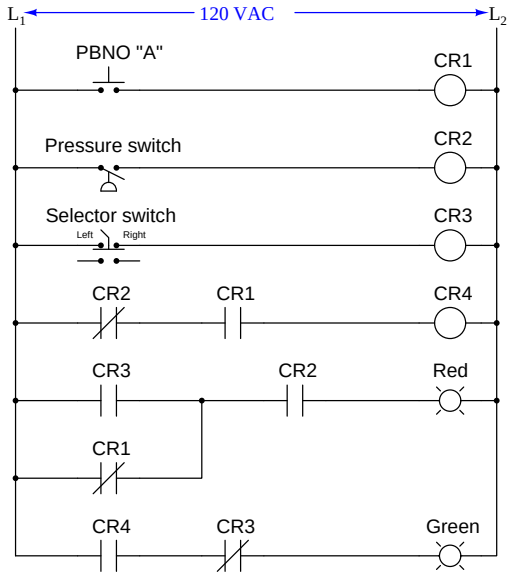
???

???

???

5.1.3 Relay ladder logic analogy for a PLC

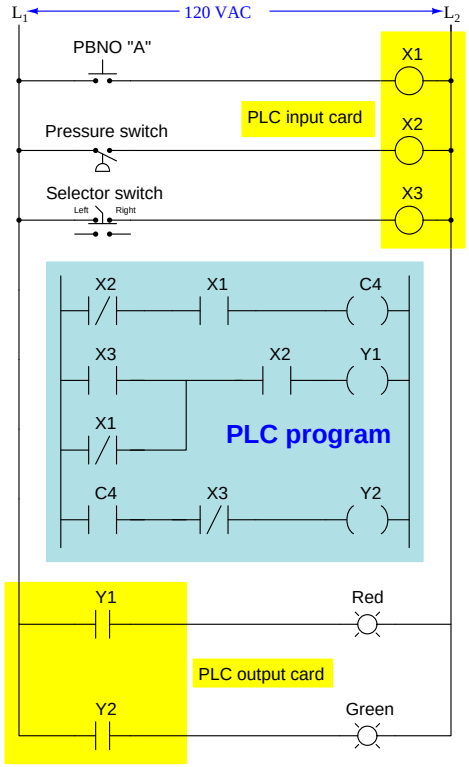
Analyze the status of all relay contacts and lamps in this hard-wired relay "ladder logic" control circuit, annotating the diagram to show electrical conductivity and non-conductivity of all contacts and energization statuses of all relay coils and loads:



Assume the following input conditions:

- Pushbutton switch *unpressed*
- Pressure *above* trip threshold
- Selector switch in its *right-hand* position

Now, analyze the status of this PLC-controlled system assuming the same input conditions. Note the distinction between the 120 VAC circuitry and the "virtual circuit" in the blue-shaded area representing the program executed by the PLC's microprocessor. As with the relay-based system, annotate all real-world contact conductivities and coil energization statuses, as well as show which of the PLC program's elements should be "colored" to reflect virtual conductivity and virtual energization:



Assume the same input conditions:

- Pushbutton switch *unpressed*
- Pressure *above* trip threshold
- Selector switch in its *right-hand* position

How is the PLC-controlled system similar to the hard-wired relay control system? How is it different?

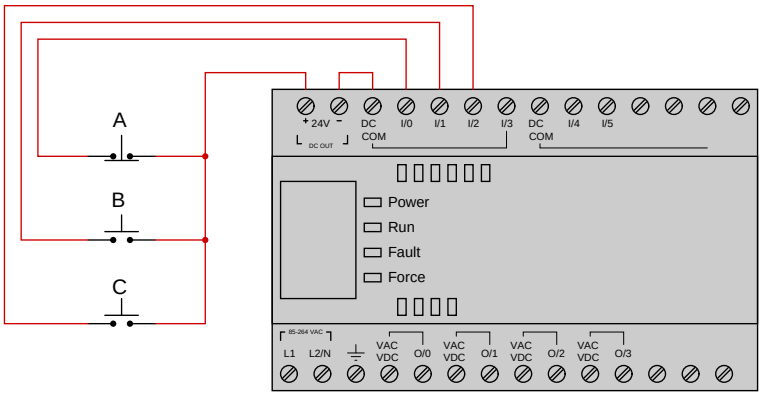
Challenges

- What advantages does the PLC enjoy over the hard-wired relay control system?
- What advantages does the electromechanical relay design enjoy over the PLC control system?

5.1.4 Determining bit statuses from switch conditions

PLC #1

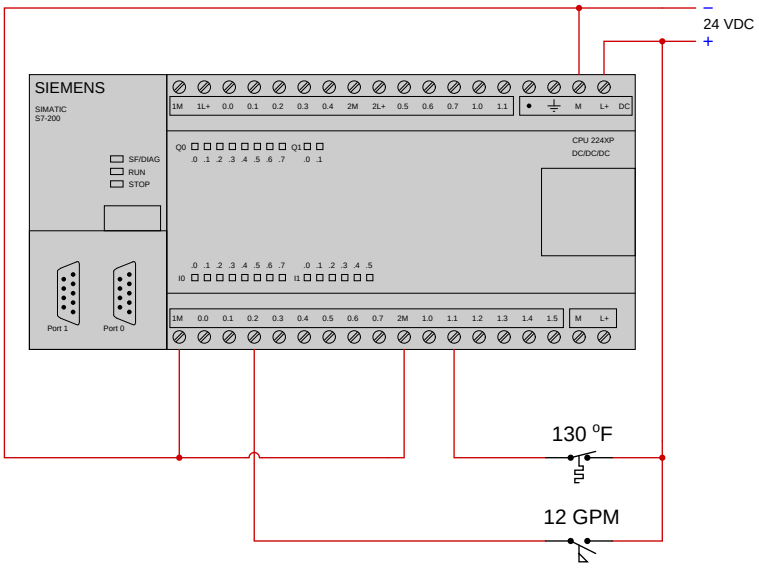
Suppose we have an Allen-Bradley MicroLogix 1000 PLC connected to three momentary-contact pushbutton switches as shown in this illustration:



Determine the bit statuses of `I:0/0`, `I:0/1`, and `I:0/2` when switch A is unpressed (released), switch B is unpressed (released), and switch C is pressed.

PLC #2

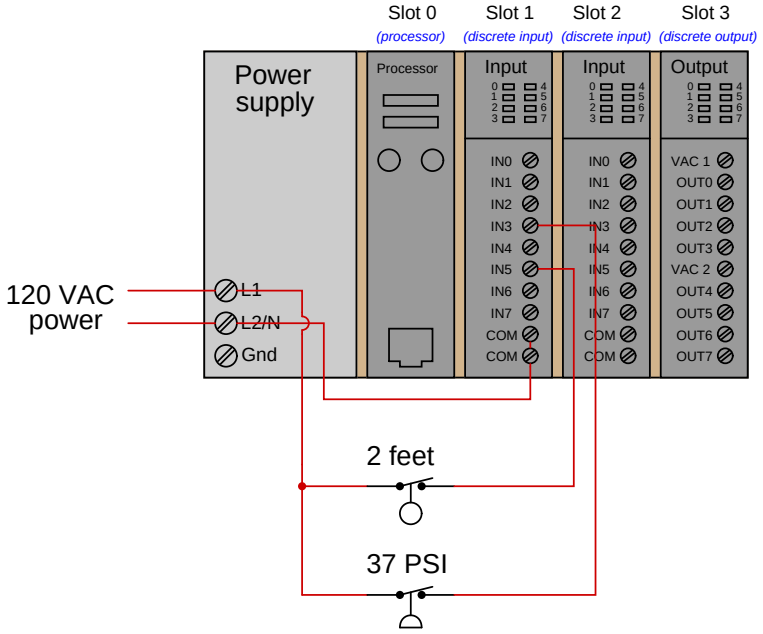
Suppose we have a Siemens S7-200 PLC connected to two process switches as shown in this illustration:



Determine the bit statuses of `I0.2` and `I1.1` when the temperature switch senses 194 ^{o}F and the flow switches senses 19 GPM.

PLC #3

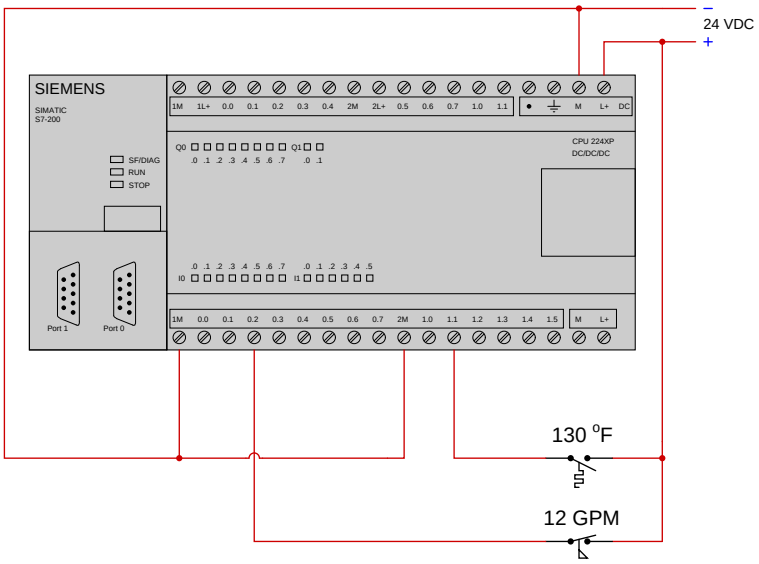
Suppose we have an Allen-Bradley SLC 500 PLC connected to two process switches as shown in this illustration:



Determine the bit statuses of `I:1/3` and `I:1/5` when the level switch senses 3 feet and the pressure switch senses 14 PSI.

PLC #4

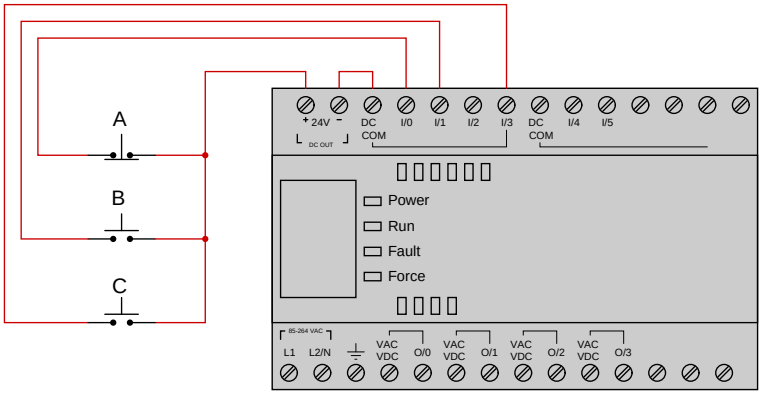
Suppose we have a Siemens S7-200 PLC connected to two process switches as shown in this illustration:



Determine the bit statuses of `I0.2` and `I1.1` when the temperature switch senses 122 ^{o}F and the flow switches senses 15 GPM.

PLC #5

Suppose we have an Allen-Bradley MicroLogix 1000 PLC connected to three momentary-contact pushbutton switches as shown in this illustration:



Determine the bit statuses of `I:0/0`, `I:0/1`, and `I:0/3` when switch A is pressed, switch B is unpressed (released), and switch C is pressed.

Challenges

- Explain the meaning of the word "normal" as it applies to a switch contact.

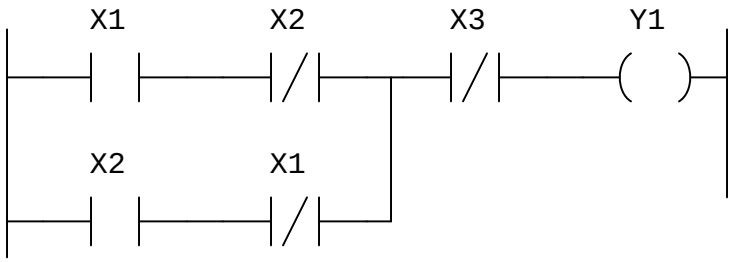
5.1.5 Determining color highlighting from bit statuses

PLC #1

Suppose a Siemens 545 PLC has the following input bit states:

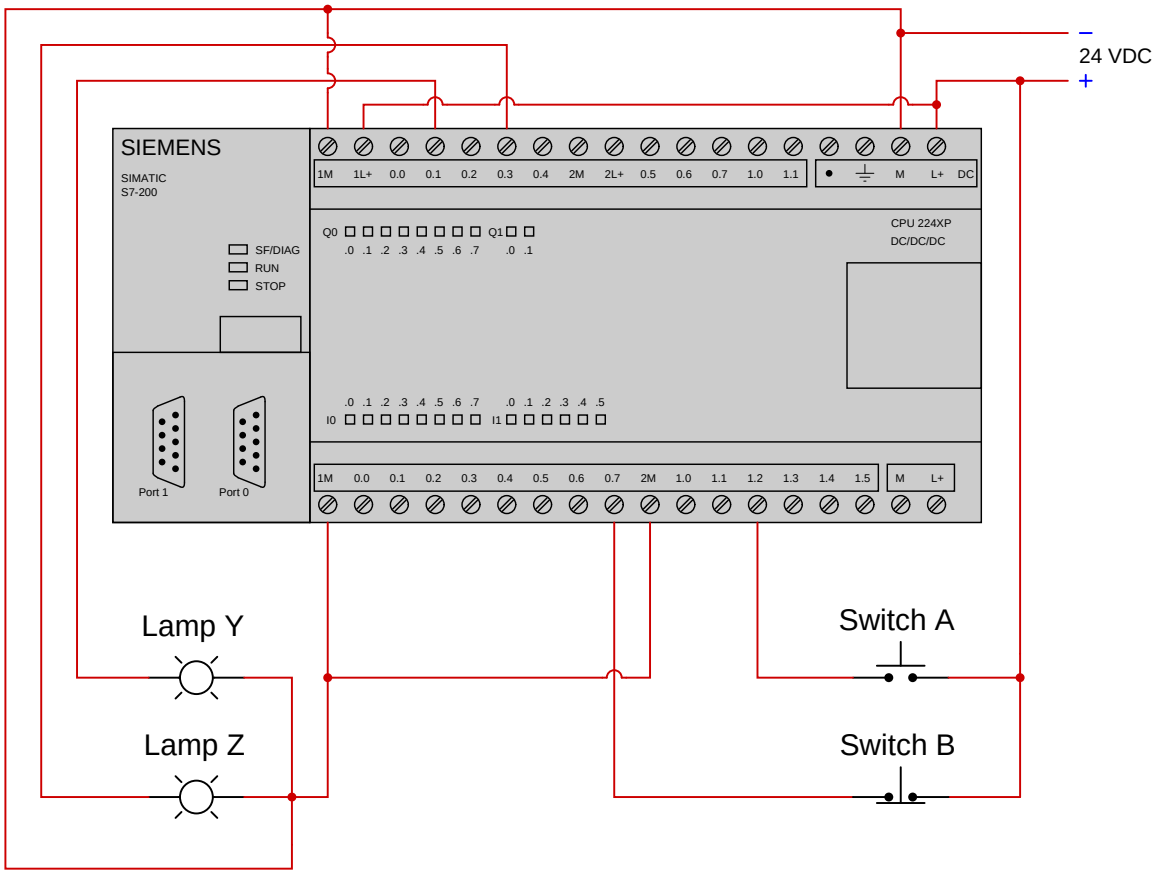
- `X1` = 0
- `X2` = 1
- `X3` = 0

Sketch color highlighting for the contacts and coils in the PLC's program given these bit statuses, also determining the status of output bit `Y1`:

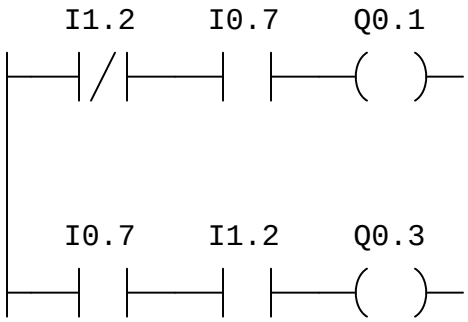


PLC #2

Suppose we have a Siemens S7-200 PLC connected to a pair of momentary-contact pushbutton switches and lamps as shown in this illustration:



Examine the following relay ladder logic (RLL) program for this Siemens PLC, determining the statuses of the two lamps provided both switches are simultaneously pressed by a human operator:



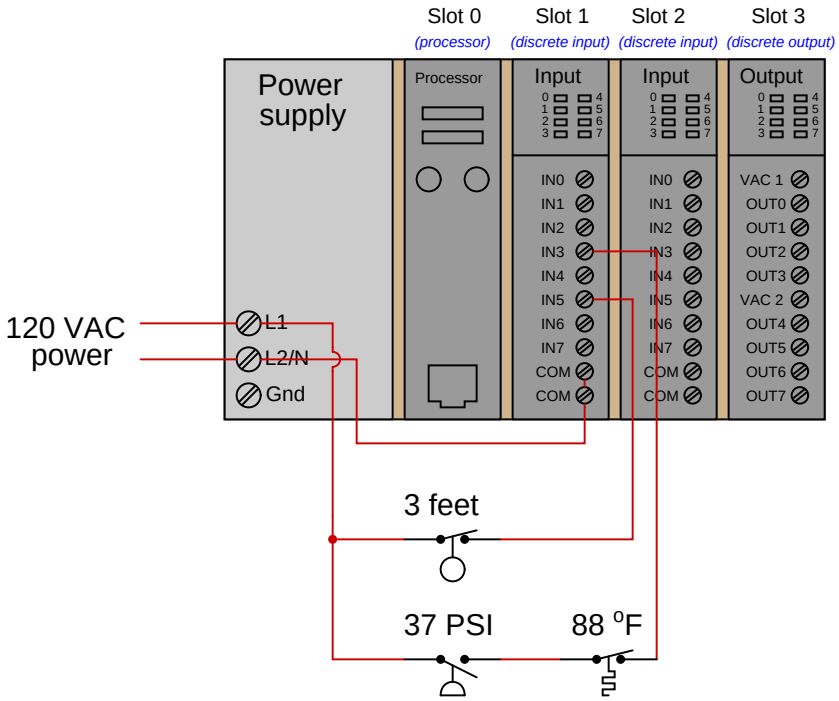
Finally, draw color highlighting showing how these "contact" instructions will appear in an online editor program given the stated input conditions.

Challenges

- Sketch a logic gate diagram implementing the same function as the first PLC program.
- Identify the significance of the labels "X" and "Y" for this PLC's bits. What do you suppose "X" signifies? What do you suppose "Y" signifies?

5.1.6 Determining necessary switch conditions for bit statuses

Suppose we have an Allen-Bradley SLC 500 PLC connected to two process switches as shown in this illustration:



Determine the process conditions necessary to generate the following input bit statuses in the PLC's memory:

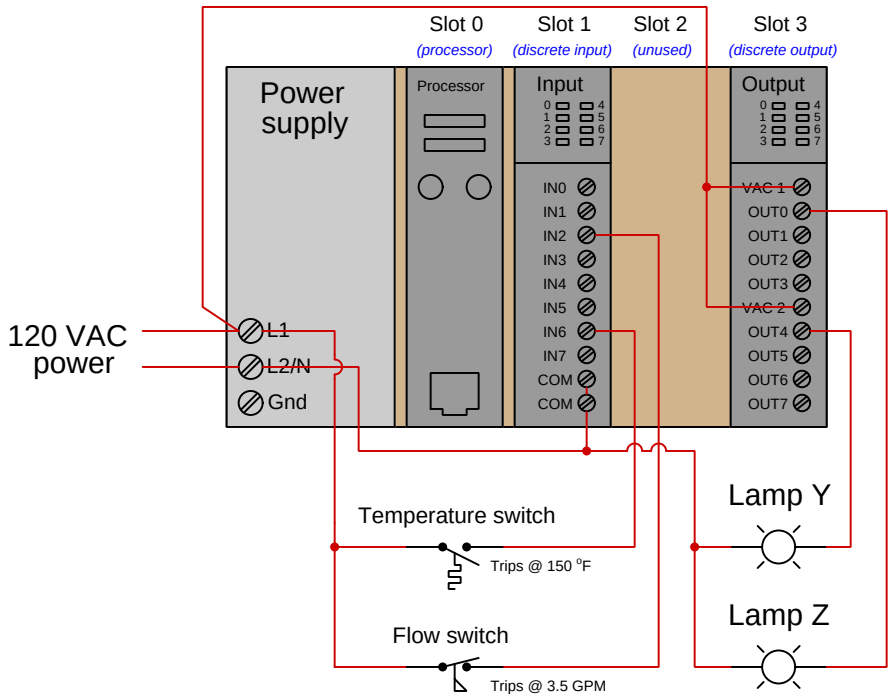
- `I:1/3` = 1
- `I:1/5` = 0

Challenges

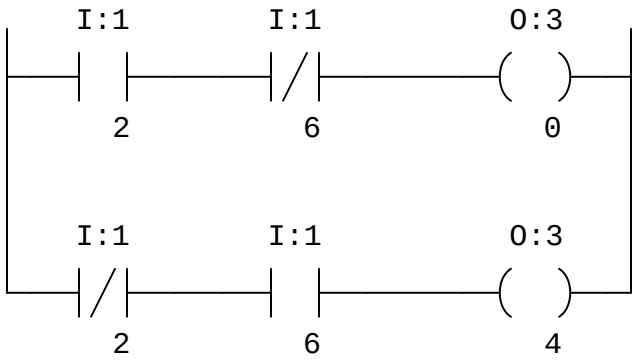
- What do the numbers "1" and "3" mean for input bit `I:1/3` in this Allen-Bradley PLC?
- What do the numbers "1" and "5" mean for input bit `I:1/5` in this Allen-Bradley PLC?
- How would you specify the bit address for the fourth channel on the other input card of this PLC?

5.1.7 Determining color highlighting from switch conditions

Suppose we have an Allen-Bradley model "SLC 500" PLC connected to a pair of process switches and lamps as shown in this illustration:



Examine the following "offline-view" (i.e. non-color-highlighted) relay ladder logic (RLL) program for this Allen-Bradley PLC, determining the statuses of the two lamps given a temperature of 172 ^{o}F and a flow of 5.1 GPM:



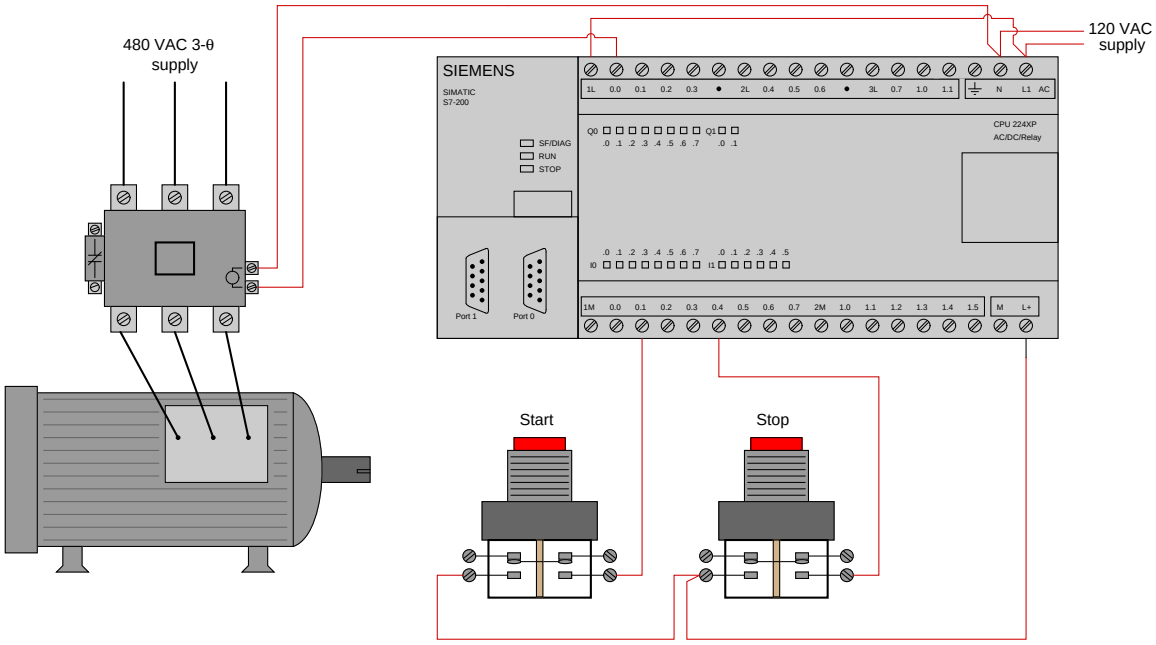
Finally, draw color highlighting showing how these "contact" instructions will appear in an online editor program given the stated input conditions.

Challenges

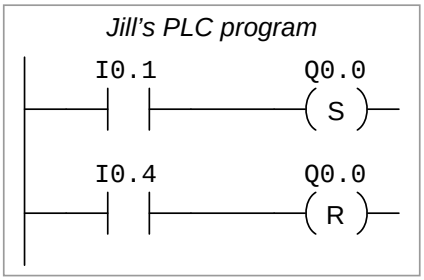
- Identify the significance of the labels "I" and "O" for this PLC's bits.
- Identify the significance of the first and second numbers in each bit label (e.g. the numbers "1" and "2" in the bit address `I:1/2`, for example). What pattern do you see as you compare the I/O connections with the respective contact instructions in the PLC program?

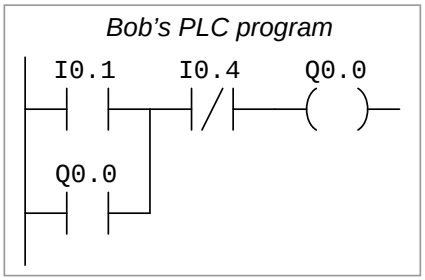
5.1.8 Two different motor control programs

Two technicians, Jill and Bob, work on programming Siemens S7-200 PLCs to control the starting and stopping of electric motors. Both PLCs are wired identically, as shown:



However, despite being wired identically, the two technicians' PLC programs are quite different. Jill's program uses *retentive coil* instructions ("Set" and "Reset" coils) while Bob's uses a "seal-in" contact instruction to perform the function of latching the motor on and off:





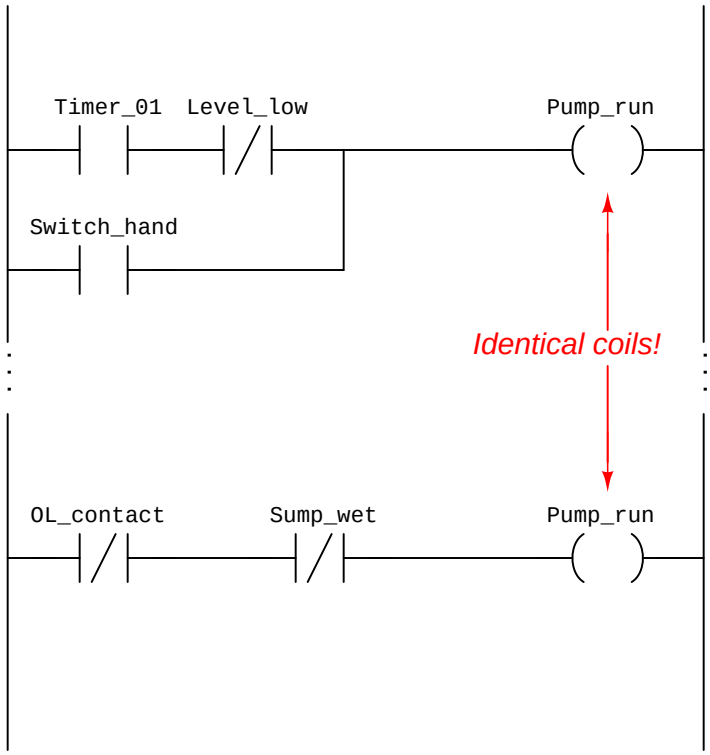
Explain how both of these PLC programs function properly to control the starting and stopping of the electric motor.

Challenges

- It is ordinarily a bad thing to assign identical bit addresses to multiple coil instructions in a PLC program. With Jill's retentive coil program, however, this is not only permissible but in fact necessary for its proper operation. Explain why this is.
- A common misconception of students first learning PLC programming is to think that the type of contact instruction used in the PLC program must match the type of switch contact connected to that input (e.g. "A N.O. PLC instruction must go with a N.O. switch"). Explain why this is incorrect.
- Explain how both PLC programs will react if both the "start" and "stop" pushbuttons are simultaneously pressed.
- Alter both PLC programs to be "fail-safe" (i.e. shut the motor off) if ever the stop pushbutton switch fails circuit open.

5.1.9 Redundant coils in a program

In relay ladder logic (RLL) programming, it is considered bad practice to have multiple instances of an identical (standard) "relay" coil in a program:



Explain why this is considered poor practice in PLC programming. Next, determine the status of the `Pump_run` output channel given the following bit states:

- `Timer_01` = `1`
- `Level_low` = `1`
- `Switch_hand` = `0`
- `OL_contact` = `0`
- `Sump_wet` = `0`

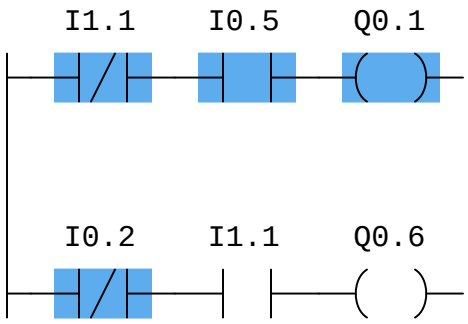
Challenges

- Explain why it *is* acceptable to have redundant retentive coils in a PLC Ladder Diagram program.

5.1.10 Determining bit statuses from color highlighting

PLC #1

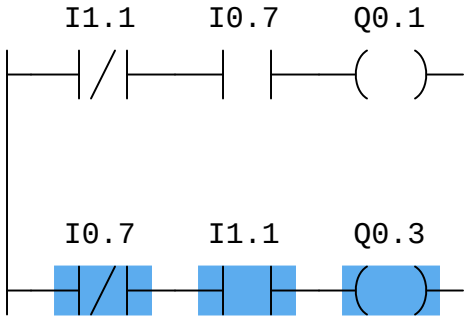
Examine this "live" display of a Siemens S7-300 PLC's program, and from this determine all bit statuses represented by the color highlighting in this ladder logic program:



- I0.2 =
- I0.5 =
- I1.1 =
- Q0.1 =
- Q0.6 =

PLC #2

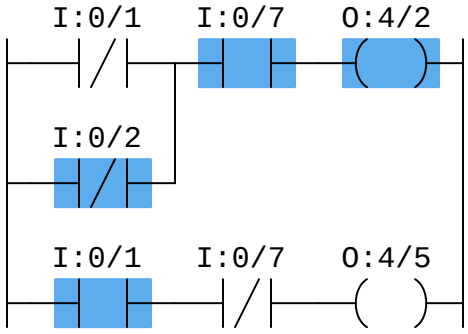
Examine this "live" display of a Siemens S7-300 PLC's program, and from this determine all bit statuses represented by the color highlighting in this ladder logic program:



- I0.7 =
- I1.1 =
- Q0.1 =
- Q0.3 =

PLC #3

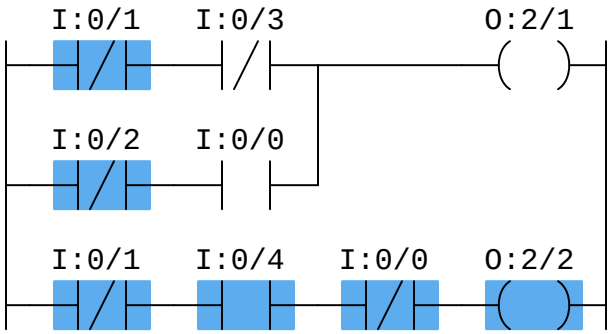
Examine this "live" display of an Allen-Bradley PLC's program, and from this determine all bit statuses represented by the color highlighting in this ladder logic program:



- I:0/1 =
- I:0/2 =
- I:0/7 =
- O:4/2 =
- O:4/5 =

PLC #4

Examine this "live" display of an Allen-Bradley PLC's program, and from this determine all bit statuses represented by the color highlighting in this ladder logic program:



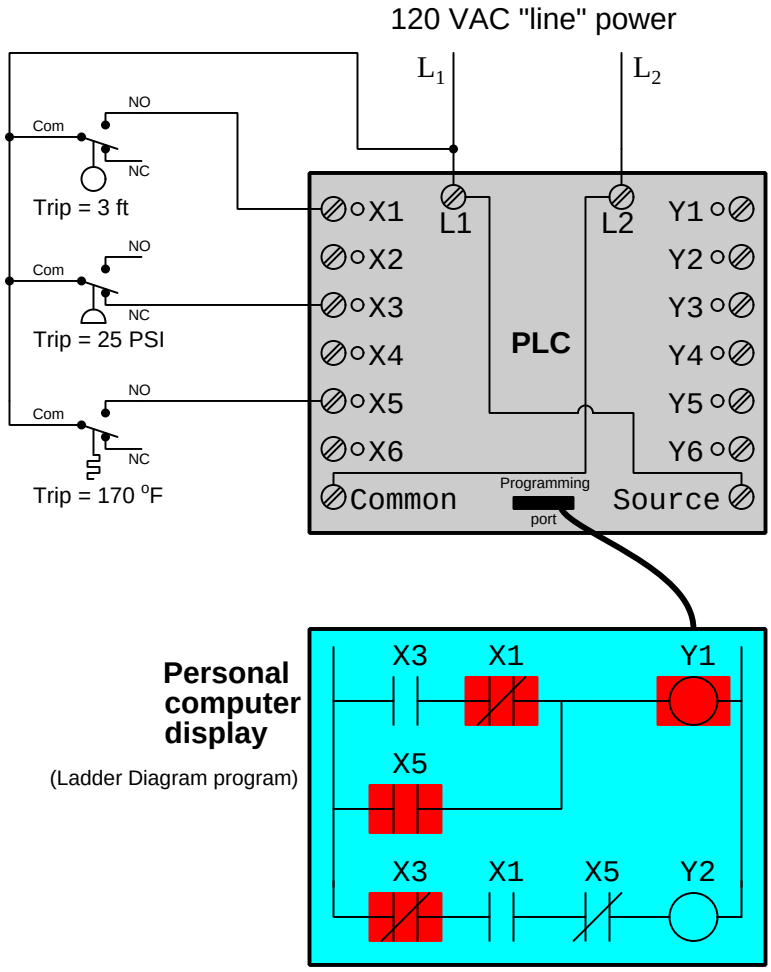
- I:0/0 =
- I:0/1 =
- I:0/2 =
- I:0/3 =
- I:0/4 =
- O:2/1 =
- O:2/2 =

Challenges

- PLC training expert Ron Beaufort teaches students to think of a "normally-open" PLC program contact instruction as a command to the PLC's processor to *"Go look for a 1"*. Conversely, he teaches students to think of a "normally-closed" instruction as a command to *"Go look for a 0"*. Explain what Mr. Beaufort means by these phrases, and how this wisdom relates to this particular problem. Incidentally, Mr. Beaufort's excellent instructional videos (available freely on YouTube) are quite valuable to watch!
- Identify the significance of the labels "I" and "Q" for this PLC's bits. What do you suppose "I" signifies? What do you suppose "Q" signifies?

5.1.11 Determining process switch stimuli from color highlighting

Suppose we have a PLC connected to three process switches as shown in this illustration:



Based *solely* on the color-highlighting you see in the "live" PLC program display, determine as best you can the pressure and temperature stimulating each switch. Pay no heed to the status LEDs on the front face of the PLC itself, as their lit/unlit states are not shown in this illustration.

Also, determine the following bit states within the PLC's memory corresponding to the same color-highlighting:

- `X1` =
- `X3` =
- `X5` =

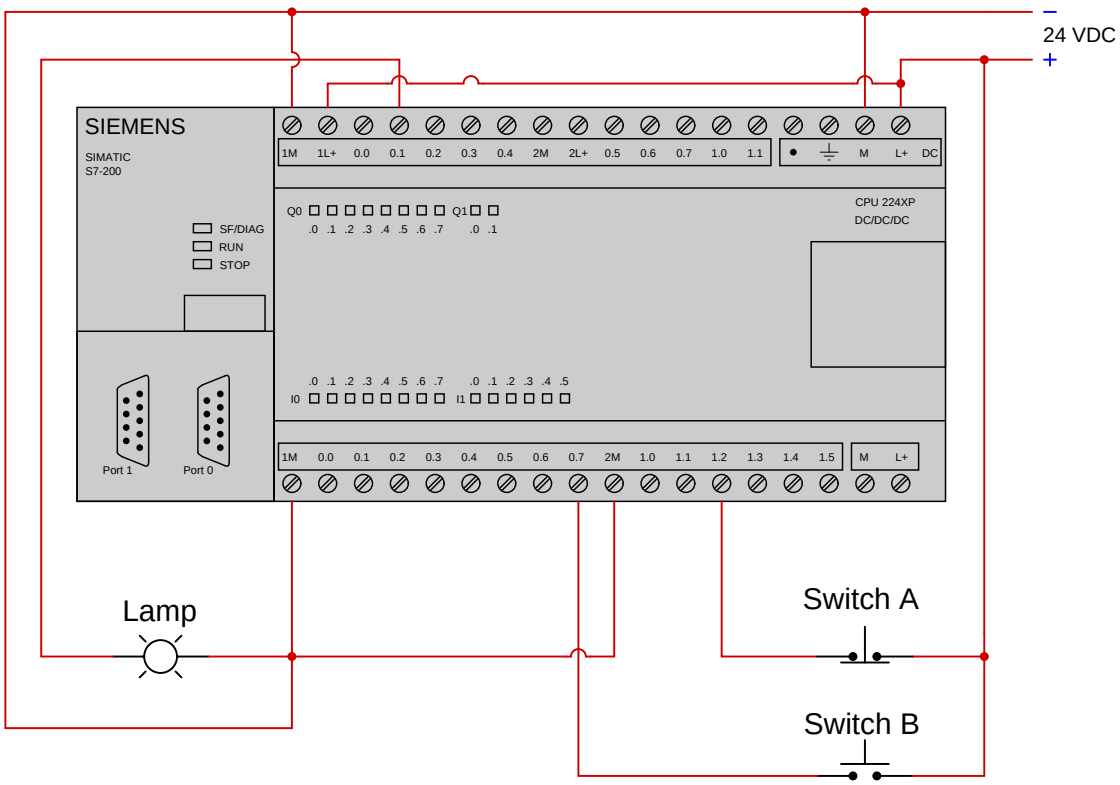
- `Y1` =
- `Y2` =

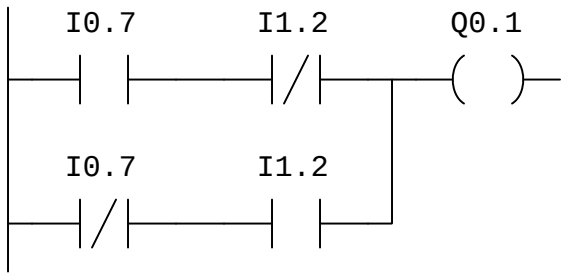
Challenges

- Identify which LED indicators on the PLC's face would be lit in this condition.

5.1.12 Determining truth table from PLC program

Suppose we have a Siemens S7-200 PLC connected to a pair of momentary-contact pushbutton switches and a lamp as shown in this illustration, with a relay ladder logic (RLL) program shown below that:





Complete the following "truth table" showing the status of the lamp given all possible switch status combinations:

Switch A	**Switch B**	**Lamp**
Unpressed	Unpressed	
Unpressed	Pressed	
Pressed	Unpressed	
Pressed	Pressed	

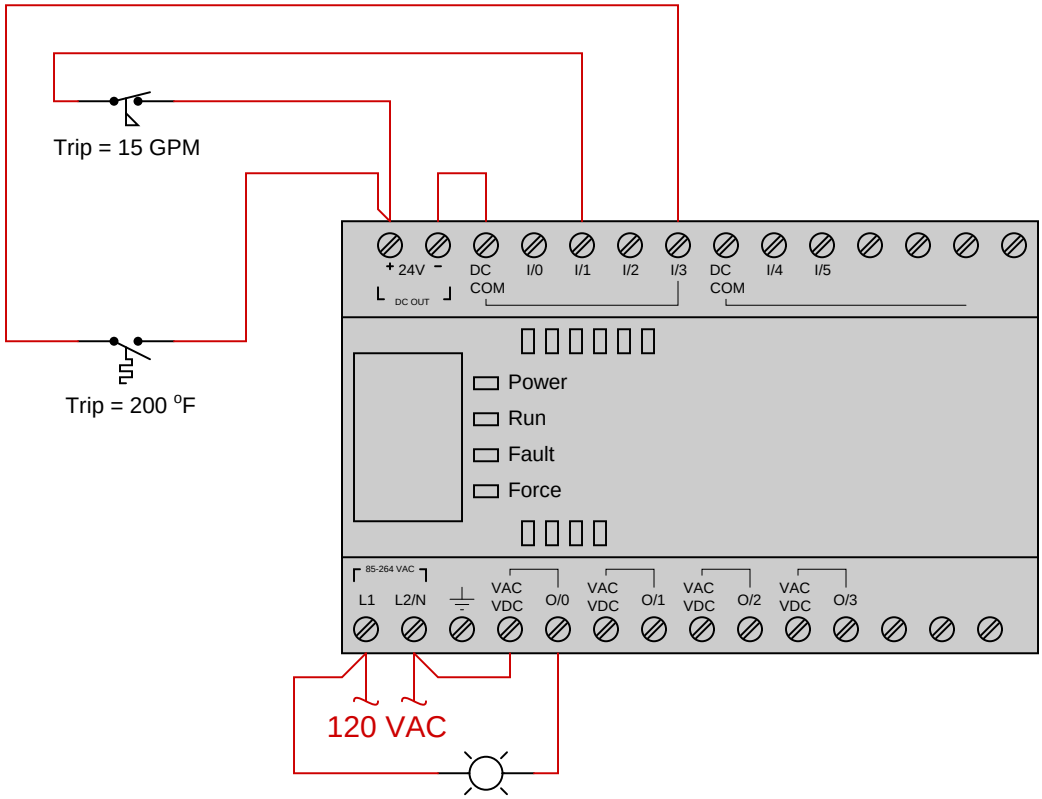
Also, show color highlighting of the contact and coil instructions in the PLC's program for each of the four rows in the truth table.

Challenges

- Which two-input logic function is being implemented by this PLC program?

5.1.13 Design a PLC program to spec

Suppose we have an Allen-Bradley MicroLogix 1000 PLC connected to a temperature switch and a flow switch as shown below:



Design #1

We wish for the lamp to come on only when the temperature is below 200 degrees F and the flow rate is below 15 GPM. Write a relay ladder-logic (RLL) program for the PLC to fulfill this function:

Design #2

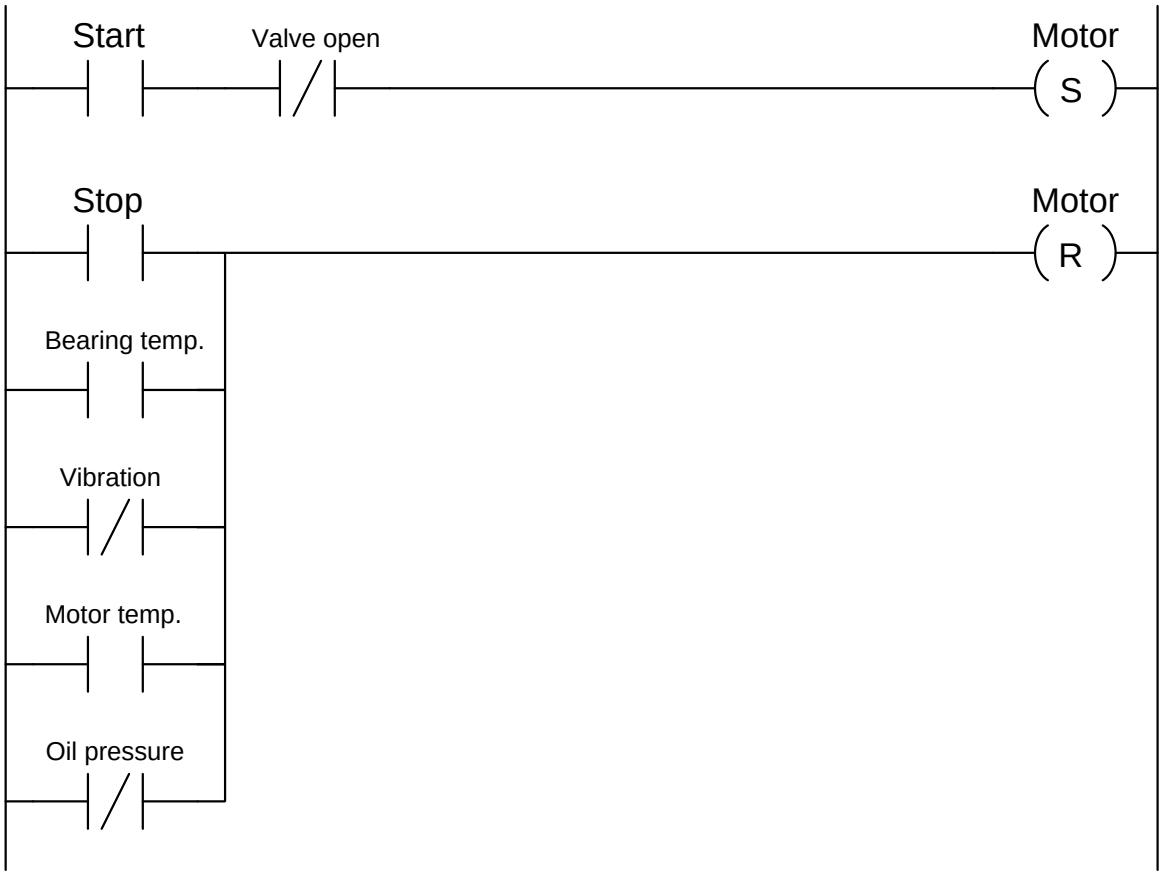
Now we wish for the lamp to come on when the temperature is above 200 degrees F or the flow rate is below 15 GPM. Write a relay ladder-logic (RLL) program for the PLC to fulfill this function:

Challenges

- What would happen if the flow switch failed open?
- What would happen if the temperature switch failed open?

5.1.14 Determining necessary switch contact types

The following PLC program was written to control the operation of a large electric motor-driven pump. A variety of "permissive" inputs protect the pump from damage under abnormal conditions, and one permissive in particular ("valve open") will only allow the pump to start up if one of the valves in the piping system is in the full-open position:



Identify the type of contact (either NO or NC) necessary for each of these electrical switch contacts, based on the trip condition (either *high* or *low*) and how each input is applied in the PLC program:

- Start pushbutton = *NO* or *NC*?
- Throttling valve open limit = *NO* or *NC*?
- Stop pushbutton = *NO* or *NC*?
- High bearing temperature = *NO* or *NC*?

- High vibration = *NO* or *NC*?
- High motor temperature = *NO* or *NC*?
- Low oil pressure = *NO* or *NC*?

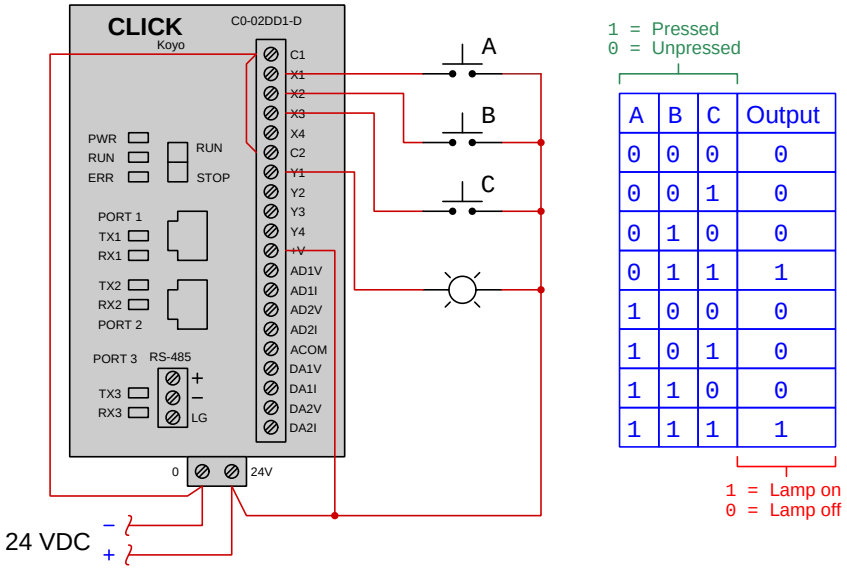
Challenges

- Explain why it is a helpful problem-solving strategy to first identify the necessary virtual contact *coloring* which will allow the motor to start up and keep running (i.e. the condition of all permissives during correct operating conditions).

5.1.15 PLC programs from truth tables

For each of the PLCs and truth tables shown, write a Ladder Diagram program to implement. Consider every "1" input state shown in the table to mean the pushbutton switch is being *pressed* and every "0" to mean the switch is released (unpressed). Pay close attention to the "normal" state of each electrical switch, as you will need to know that in order to determine whether a PLC input will be energized or not when the switch is pressed!

System #1

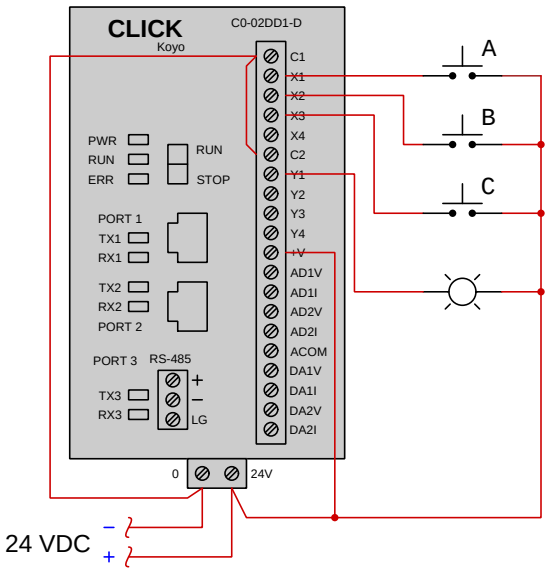


1 = Pressed
0 = Unpressed

A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

1 = Lamp on
0 = Lamp off

System #2

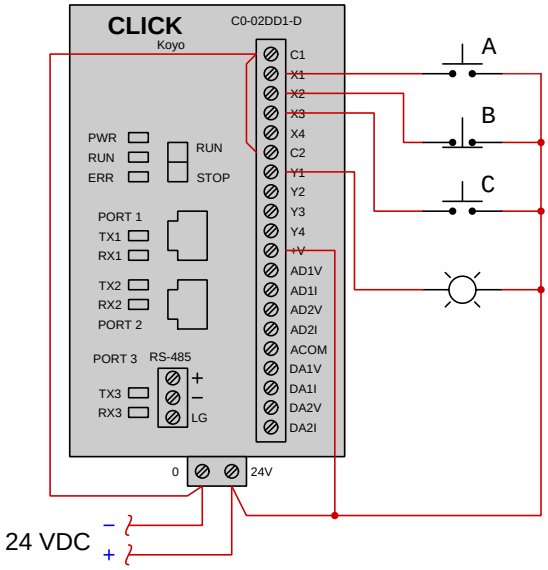


1 = Pressed
0 = Unpressed

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

1 = Lamp on
0 = Lamp off

System #3

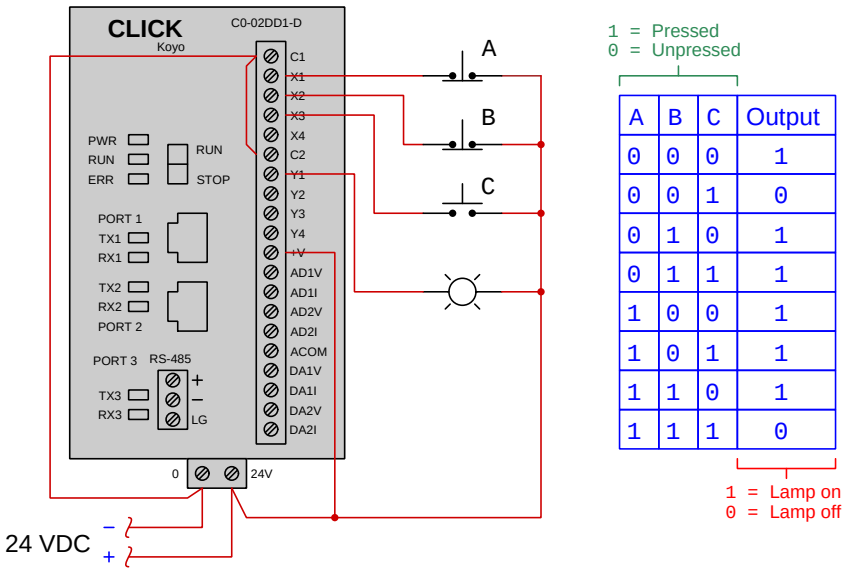


1 = Pressed
0 = Unpressed

A	B	C	Output
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

1 = Lamp on
0 = Lamp off

System #4



Challenges

- Identify programming solutions using the fewest contact instructions possible to implement the truth table.

5.2 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.

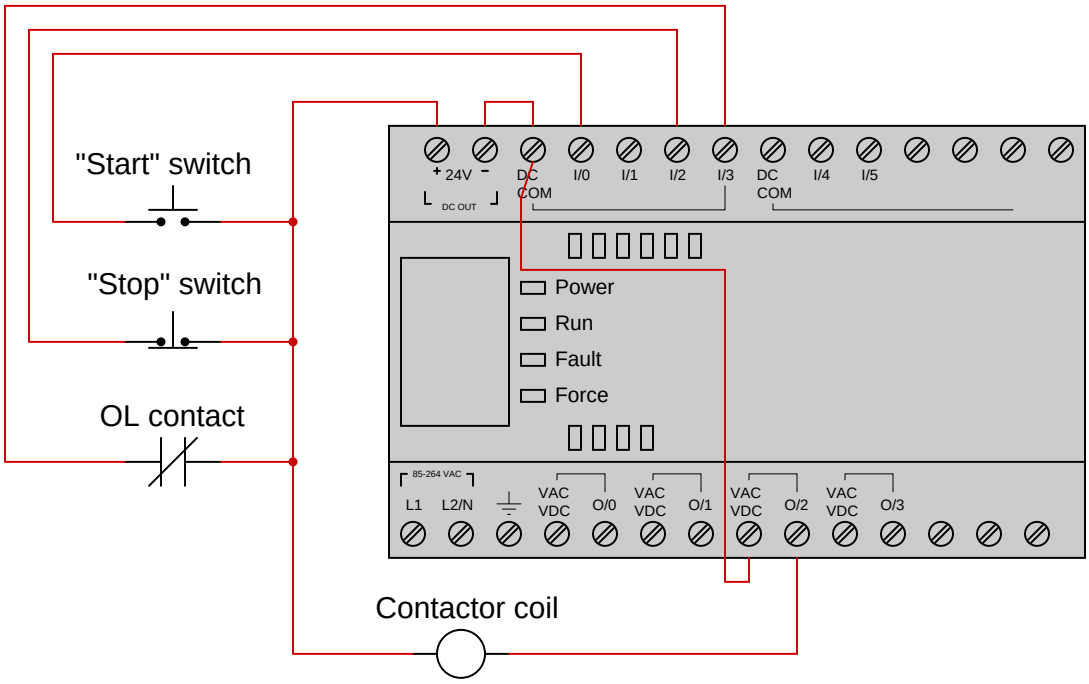
As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of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

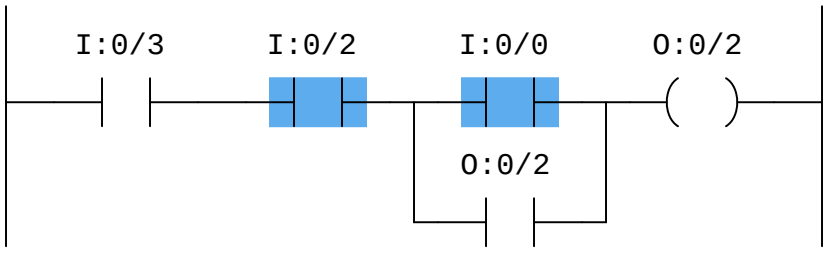
Another means of checking your diagnostic answers, where applicable, is to use circuit simulation software to explore the effects of faults placed in circuits. For example, if one of these diagnostic questions requires that you predict the effect of an open or a short in a circuit, you may check the 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.

5.2.1 Troubleshooting motor control program

Suppose we have an Allen-Bradley MicroLogix 1000 controller connected to a pair of momentary-contact pushbutton switches and contactor controlling power to an electric motor as shown in this illustration:



This motor control system has a problem, though: the motor refuses to start when the "Start" pushbutton is pressed. Examine the "live" display of the ladder logic program inside this Allen-Bradley PLC to determine what the problem is, assuming an operator is continuously pressing the "Start" pushbutton as you examine the program:



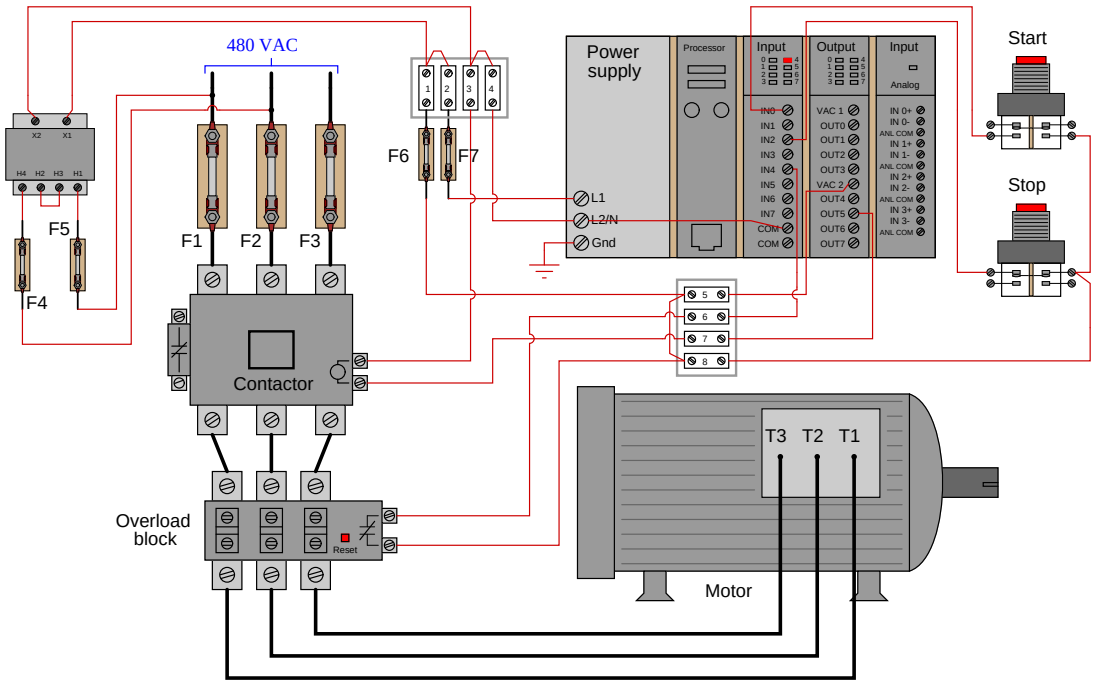
Identify at least two causes that could account for all you see here.

Challenges

- Identify the symptoms resulting from the "Start" switch being failed open.
- Identify the symptoms resulting from the "Stop" switch being failed open.
- Identify the symptoms resulting from the contactor coil being failed open.

5.2.2 Troubleshooting motor control PLC from I/O indicators

Suppose we have an Allen-Bradley SLC 500 controller connected to a pair of momentary-contact pushbutton switches and contactor controlling power to an electric motor as shown in this illustration:



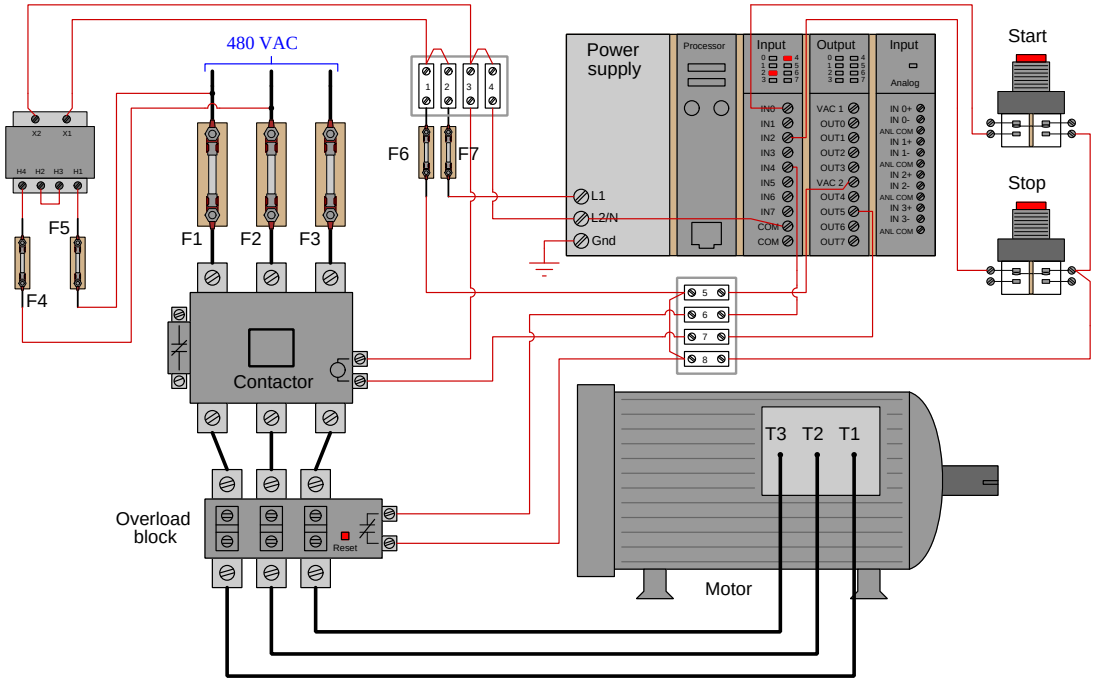
This motor control system has a problem, though: the motor refuses to start when the "Start" pushbutton is pressed. Closely examine the pictorial diagram (including the status LEDs on the PLC's I/O cards), then identify at least two faults that could account for the motor's refusal to start.

Challenges

- Explain why knowledge of the PLC program is not necessary to diagnose this fault.

5.2.3 Troubleshooting motor control PLC using voltmeter

Suppose we have an Allen-Bradley SLC 500 controller connected to a pair of momentary-contact pushbutton switches and contactor controlling power to an electric motor as shown in this illustration:



This motor control system has a problem, though: the motor refuses to start when the "Start" pushbutton is pressed. A voltage measurement taken between terminals 1 and 4 reveals 119.4 Volts AC. I/O LED statuses are shown in the above illustration.

Identify the likelihood of each specified fault listed below. Consider each fault one at a time (i.e. no coincidental faults), determining whether or not each fault could independently account for *all* observations and symptoms in this circuit.

- Failed-open Start pushbutton switch
- Failed-open Stop pushbutton switch
- Blown fuse F4
- Blown fuse F6
- Blown fuse F7
- Overload contact tripped (open)

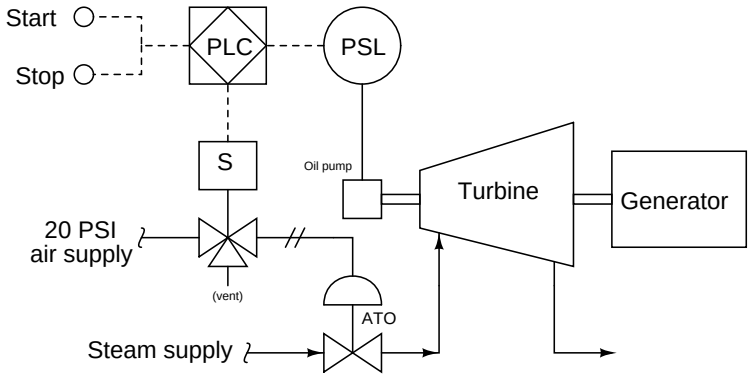
- Failed-open contactor coil

Challenges

- Explain why knowledge of the PLC program is not necessary to diagnose this fault.

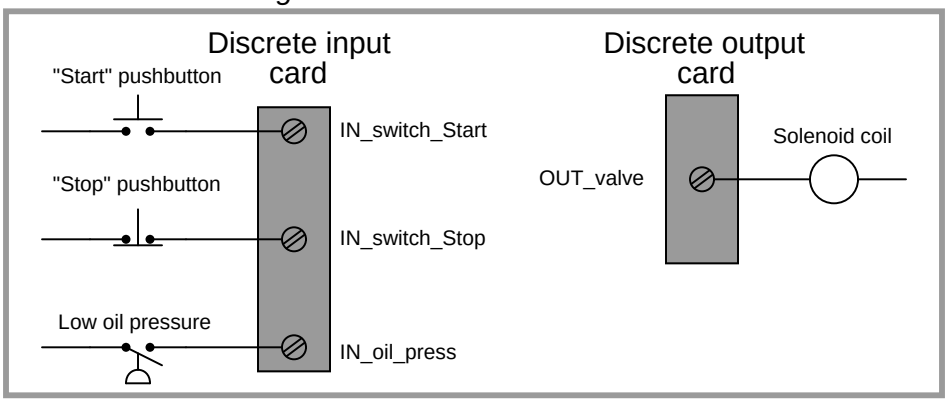
5.2.4 Turbine low-oil trip

A PLC is being used to monitor the oil pressure for a steam turbine driving an electrical generator, shutting steam off to the turbine if ever the oil pressure drops below a 10 PSI limit. The turbine's lubrication oil pump is driven by the turbine shaft itself, supplying itself with pressurized lubricating oil to keep all the turbine bearings properly lubricated and cooled:

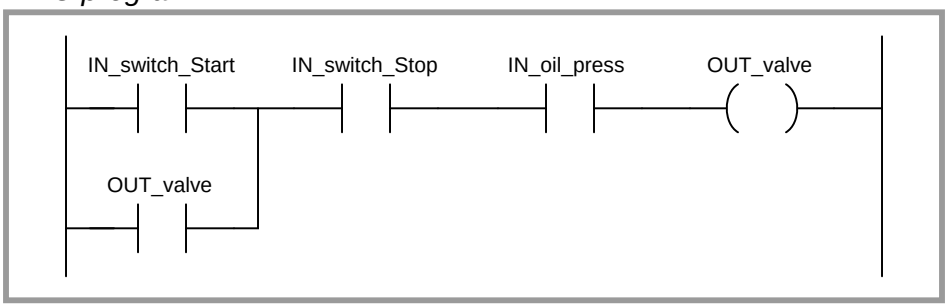


Another technician programmed the PLC for the start/stop function, but this program has a problem:

Real-world I/O wiring



PLC program



Identify what this problem is, and fix it! Hint: the oil pump is driven by the turbine, and as such cannot generate any oil pressure until the turbine begins to spin.

Challenges

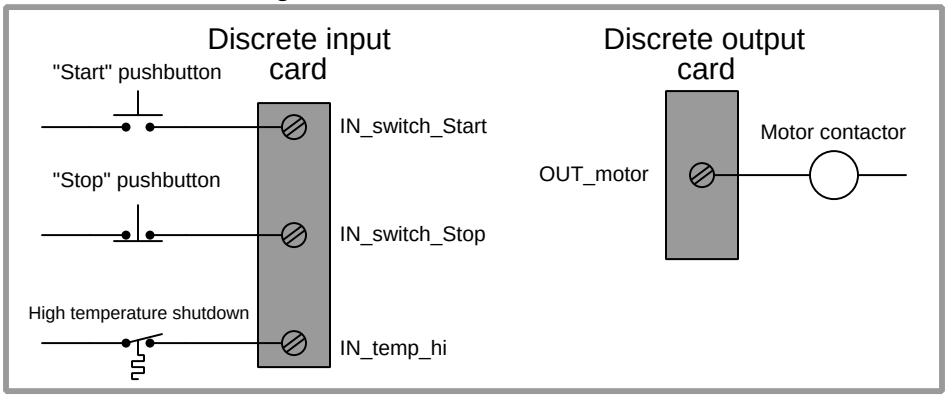
- Explain how this problem could be fixed by the addition of a *timer* instruction to the PLC program.

5.2.5 Motor starter diagnosis from color highlighting

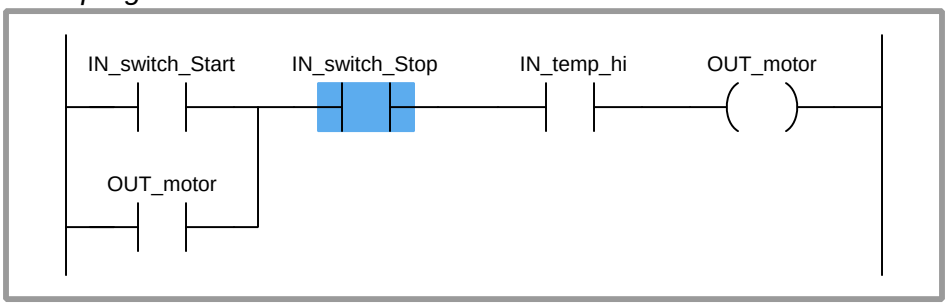
PLC #1

Identify one plausible fault to explain why this PLC-controlled motor refuses to start up when the "Start" button is pressed, given the following wiring diagram and online PLC program display shown below. Note: no one is pushing any buttons at this time.

Real-world I/O wiring



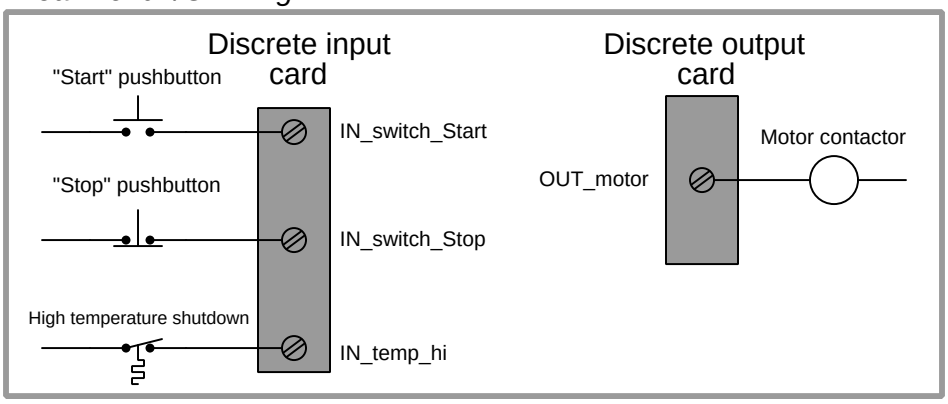
PLC program



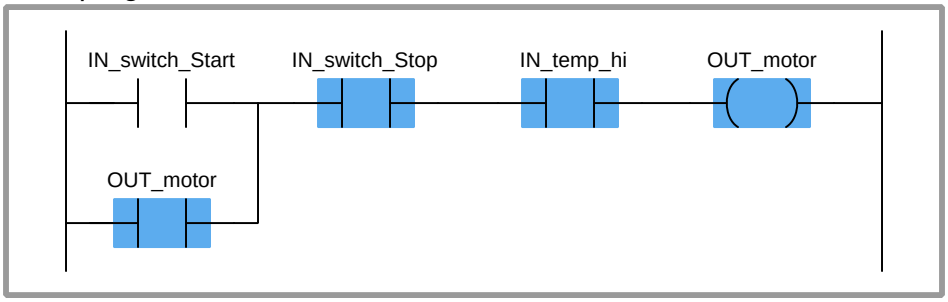
PLC #2

Identify one plausible fault to explain why this PLC-controlled motor refuses to start up when the "Start" button is pressed, given the following wiring diagram and online PLC program display shown below. Note: no one is pushing any buttons at this time.

Real-world I/O wiring



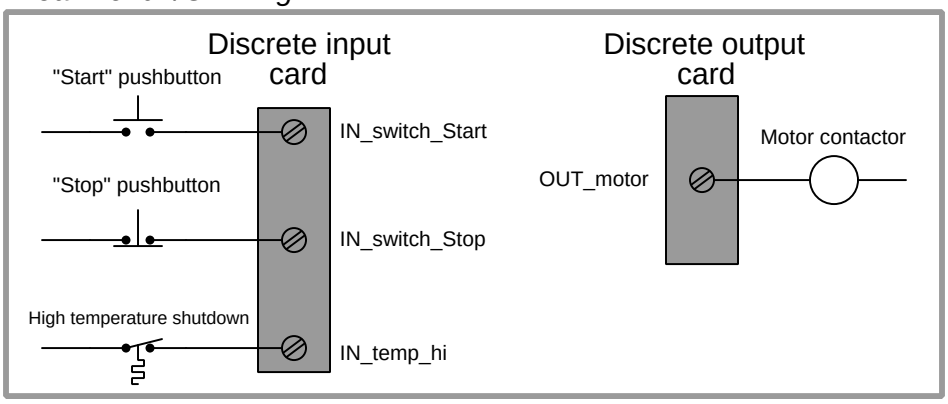
PLC program



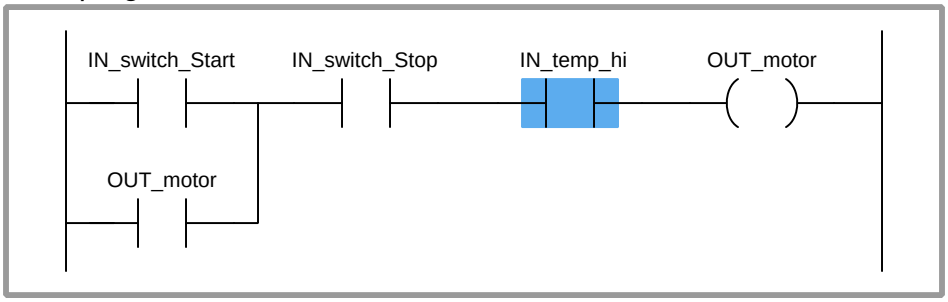
PLC #3

Identify one plausible fault to explain why this PLC-controlled motor refuses to start up when the "Start" button is pressed, given the following wiring diagram and online PLC program display shown below. Note: no one is pushing any buttons at this time.

Real-world I/O wiring



PLC program

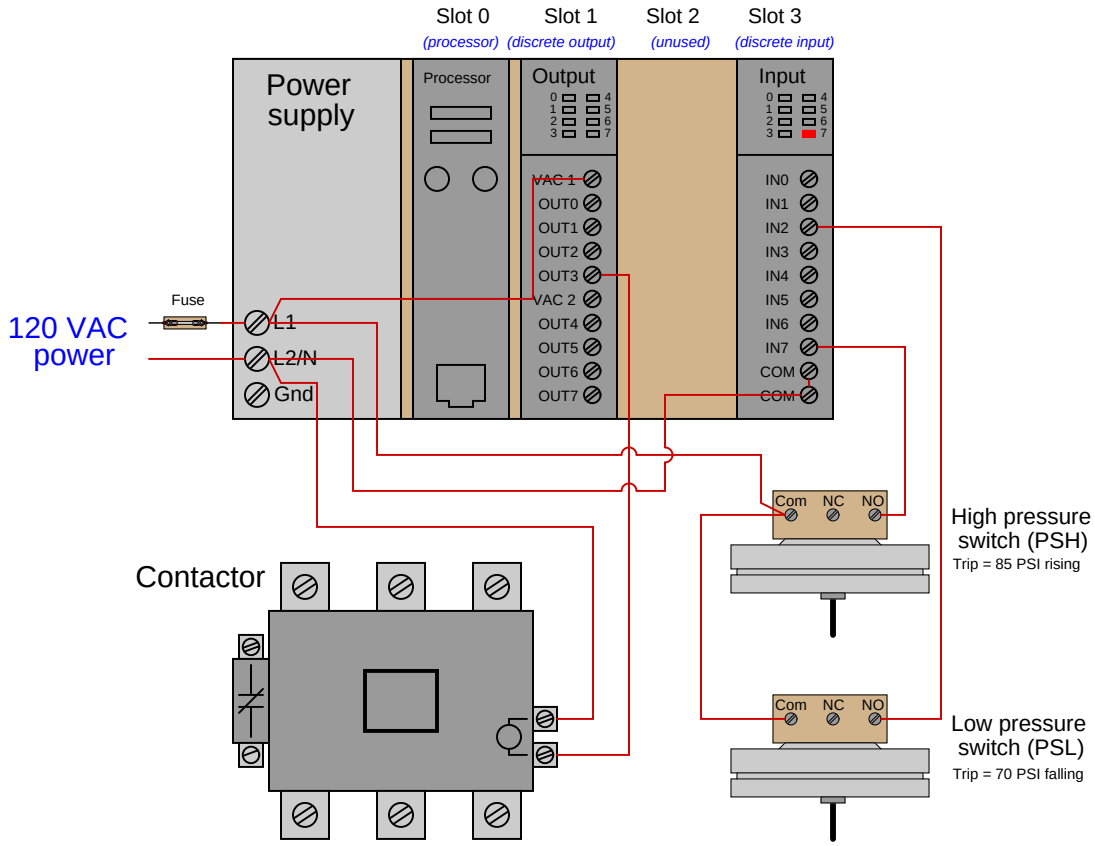


Challenges

- If you did not have access to a computer to view the PLC's live status, how could you diagnose these faults?

5.2.6 Air compressor troubleshooting from observation

An Allen-Bradley SLC 500 PLC is used to control an air compressor, using an across-the-line motor starter. All 480 volt power wiring has been omitted for simplicity:



The system is broken, though: the compressor refuses to start even though the air tank is empty (no air pressure). You have no test equipment with you – all you have is what you see in the above illustration.

Identify the likelihood of each specified fault listed below. Consider each fault one at a time (i.e. no coincidental faults), determining whether or not each fault could independently account for *all* observations and symptoms in this circuit.

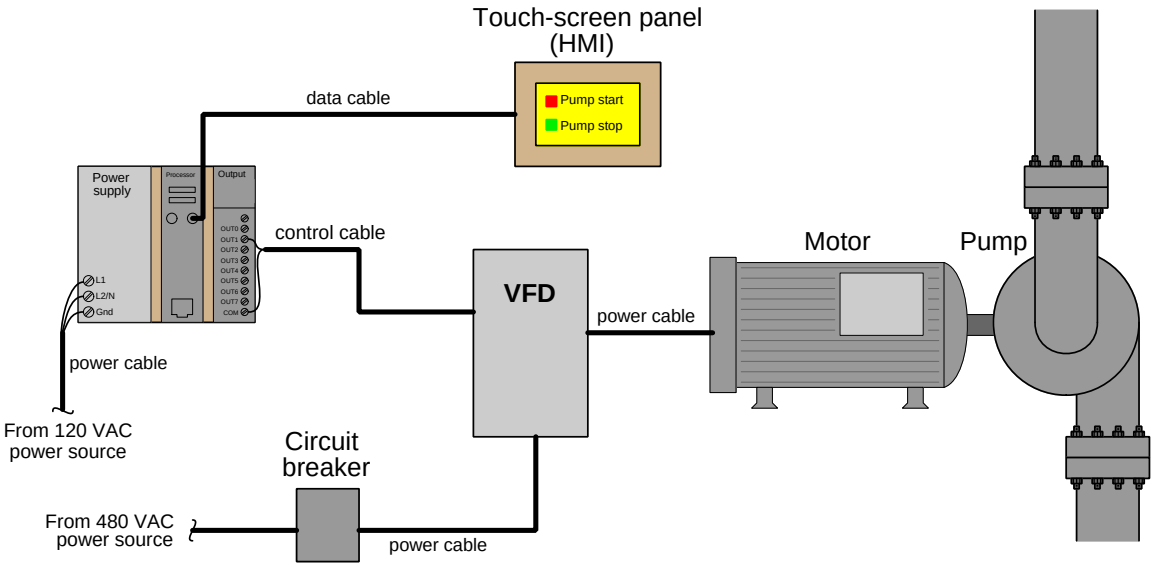
- Failed-open wire between PSL and IN2 terminal
- Failed-open wire between PSH and IN7 terminal
- Failed-open contactor coil
- Failed-shorted contactor coil
- PSL stuck (as though $P > 70$ PSI)
- PSH stuck (as though $P > 85$ PSI)
- Failed input card
- Failed output card
- PLC in "stop" mode
- Blown fuse

Challenges

- If you had access to the PLC's ladder-diagram program with colored status indication (i.e. a "live" view of the program), what would you look for to aid in your troubleshooting.

5.2.7 Possible faults in a PLC/HMI pump control system

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



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

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

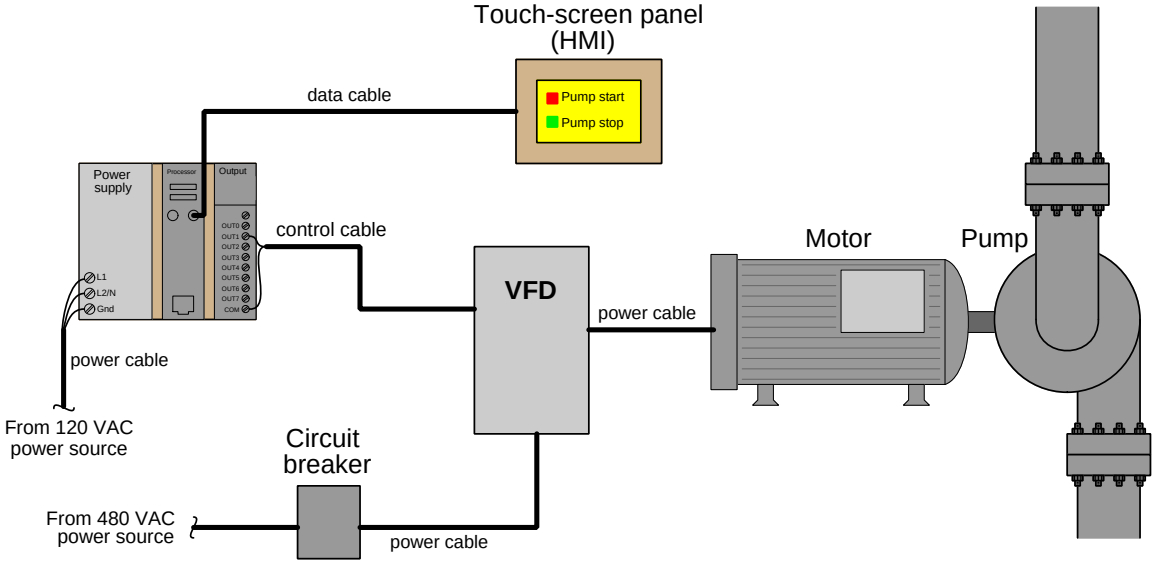
- Circuit breaker off
- Touch-screen panel malfunctioning
- Programming error in PLC
- Faulted power cable between VFD and motor
- Faulted power cable between breaker and VFD
- PLC output card malfunctioning
- Open control cable
- Shorted control cable
- Open data cable

Challenges

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

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

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



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

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

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

Challenges

- What, exactly, makes a diagnostic test useful?

Appendix A

Problem-Solving Strategies

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

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

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

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

Appendix B

Instructional philosophy

B.1 First principles of learning

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

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

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

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

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

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

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

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

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

B.2 Proven strategies for instructors

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

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

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

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

B.3 Proven strategies for students

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

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

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

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

B.4 Design of these learning modules

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

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

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

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

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

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

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

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

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

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

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

To high standards of education,

Tony R. Kuphaldt

Appendix C

Tools used

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

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

The `GNU/Linux` computer operating system

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

Bram Moolenaar's `Vim` text editor

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

Donald Knuth's TEX typesetting system

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

Leslie Lamport's LATEX extensions to TEX

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

Tim Edwards' `Xcircuit` drafting program

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

`Gimp` graphic image manipulation program

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

`SPICE` circuit simulation program

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

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

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

`gnuplot` mathematical visualization software

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

`Python` programming language

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

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

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

Section 1 – Definitions.

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

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

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

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

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

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

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

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

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

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

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

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

Section 2 – Scope.

a. License grant.

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

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

B. produce, reproduce, and Share Adapted Material.

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

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

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

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

5. Downstream recipients.

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

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

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

b. Other rights.

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

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

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

Section 3 – License Conditions.

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

a. Attribution.

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

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

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

ii. a copyright notice;

iii. a notice that refers to this Public License;

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

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

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

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

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

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

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

Section 4 – Sui Generis Database Rights.

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

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

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

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

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

Section 5 – Disclaimer of Warranties and Limitation of Liability.

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

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

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

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

Section 6 – Term and Termination.

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

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

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

2. upon express reinstatement by the Licensor.

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

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

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

Section 7 – Other Terms and Conditions.

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

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

Section 8 – Interpretation.

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

be made without permission under this Public License.

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

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

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

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

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

Appendix E

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.

9-10 February 2026 – edited some instructor notes and added a page break to one of the multi-part questions. One of the instructor notes had to do with the last answer on the "Troubleshooting motor control PLC using voltmeter" Diagnostic Reasoning question and was suggested by Trent Johnson.

1-4 February 2026 – minor edits to the Tutorial chapter, including more explanation in the "Ladder diagram transition contacts" section about transition contacts. Also made some corrections to instructor notes. Also clarified the unlit status LED display in the "Determining process switch stimuli from color highlighting" Conceptual Reasoning question.

31 January 2026 – added more Diagnostic Reasoning questions from the SINST project, as well as some explanatory text to the Tutorial chapter on transition contacts.

22 January 2026 – added more design scenarios to the "Design a PLC program to spec" Conceptual Reasoning question. Also added a new Technical Reference section on process switch types and symbols.

29-30 May 2025 – divided the Tutorial chapter into more sections, and added questions.

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

Index

Absolute addressing, 92
Adding quantities to a qualitative problem, 158
AND function, 65
Annotating diagrams, 157
Auxiliary contact, 49

Beaufort, Ron, 130

Card, I/O, 31
Checking for exceptions, 158
Checking your work, 158
Code, computer, 167
Coil, immediate, 36
Consumed tag, 83, 84
Contactor, 49, 52

Dimensional analysis, 157
Direct addressing, 92

Edwards, Tim, 168
Examine if closed (XIC), 56
Examine if open (XIO), 56

Graph values to solve a problem, 158
Greenleaf, Cynthia, 103

How to teach with these modules, 165
Hwang, Andrew D., 169
Hyperterminal software, 94

I/O, 31
Identify given data, 157
Identify relevant principles, 157
Immediate coil, 36
Intermediate results, 157
Inverted instruction, 165

Knuth, Donald, 168

Koyo CLICK PLC, 40

Ladder Diagram, 3, 31
Ladder Diagram programming, 39
Lamport, Leslie, 168
LD, 39
Limiting cases, 158

Memory map, 88
Metacognition, 108
Moolenaar, Bram, 167
Murphy, Lynn, 103

NAND function, 66
Non-retentive instruction, PLC program, 60
NOR function, 66
Normal state of a PLC program contact, 41, 42, 56, 71
NOT function, 66

One-shot, 39, 97
Open-source, 167
OR function, 65
Out coil, PLC programming, 60

PLC, 3, 31
PLC scan cycle, 36
Problem-solving: annotate diagrams, 157
Problem-solving: check for exceptions, 158
Problem-solving: checking work, 158
Problem-solving: dimensional analysis, 157
Problem-solving: graph values, 158
Problem-solving: identify given data, 157
Problem-solving: identify relevant principles, 157
Problem-solving: interpret intermediate results, 157
Problem-solving: limiting cases, 158
Problem-solving: qualitative to quantitative, 158

Problem-solving: quantitative to qualitative, 158
Problem-solving: reductio ad absurdum, 158
Problem-solving: simplify the system, 157
Problem-solving: thought experiment, 157
Problem-solving: track units of measurement, 157
Problem-solving: visually represent the system, 157
Problem-solving: work in reverse, 158
Produced tag, 83, 84
Programmable Logic Controller, 3, 31
PuTTY software, 94

Qualitatively approaching a quantitative problem, 158

Reading Apprenticeship, 103
Reductio ad absurdum, 158, 164, 165
Relay ladder logic, 31
Relay Ladder Logic programming, 39
Requested Packet Interval, 83, 84
Reset coil, PLC programming, 60
Retentive instruction, PLC programming, 60
RLL, 39
RPI, 83, 84
Rung, PLC programming, 41

Scan cycle, PLC, 36
Schoenbach, Ruth, 103
Schweitzer Engineering Laboratories, 93
Scientific method, 108
Seal-in contact, 49
SEL, 93
SEL-2440 DPAC, 94
SEL-5030 AcSELerator QuickSet software, 94
SELogic, 93
Set coil, PLC programming, 60
Set-Reset latch, 96
Simplifying a system, 157
Socrates, 164
Socratic dialogue, 165
SPICE, 103
Stallman, Richard, 167
Static contact, PLC programming, 62
Symbol, 93
Symbolic addressing, 93

Tag name, 93
Tag names, 69
Tag, consumed, 83, 84
Tag, produced, 83, 84
Terminal emulator software, 94
Termite software, 94
Thought experiment, 157
Torvalds, Linus, 167
Transition-sensing contact, PLC programming, 62

Units of measurement, 157

Visualizing a system, 157

Work in reverse to solve a problem, 158
WYSIWYG, 167, 168

XIC (examine if closed), 56
XIO (examine if open), 56