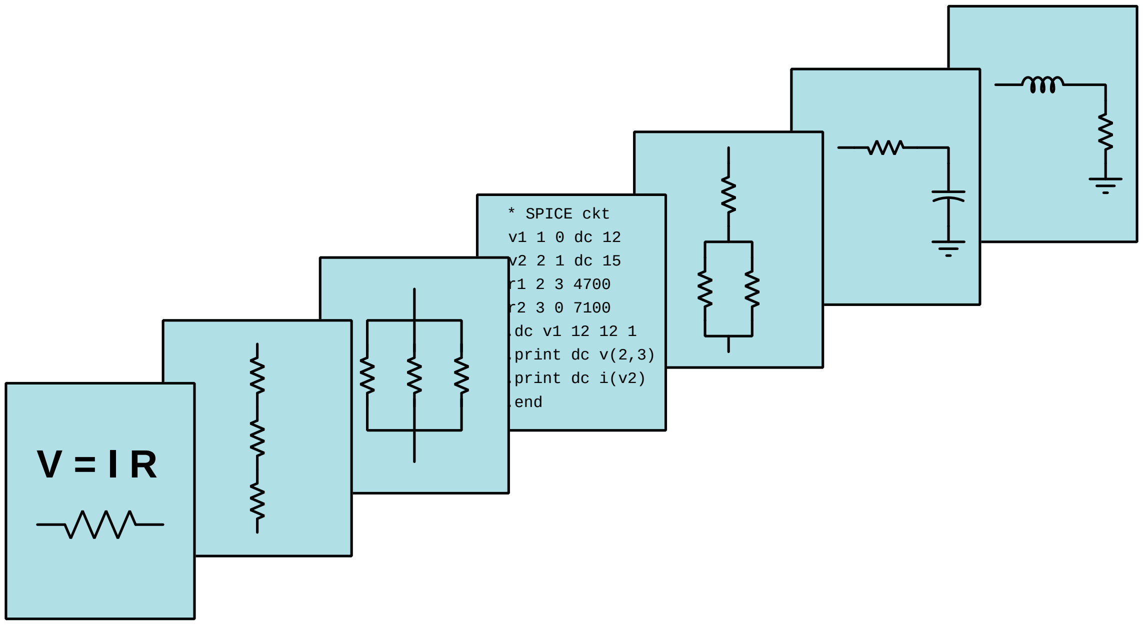
Modular Electronics Learning (ModEL) project



Introduction to PLCs

Last update = 6 December 2022

Contents

Chapter 1

Introduction

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?
- What is an HMI panel, and where would one be useful?
- 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

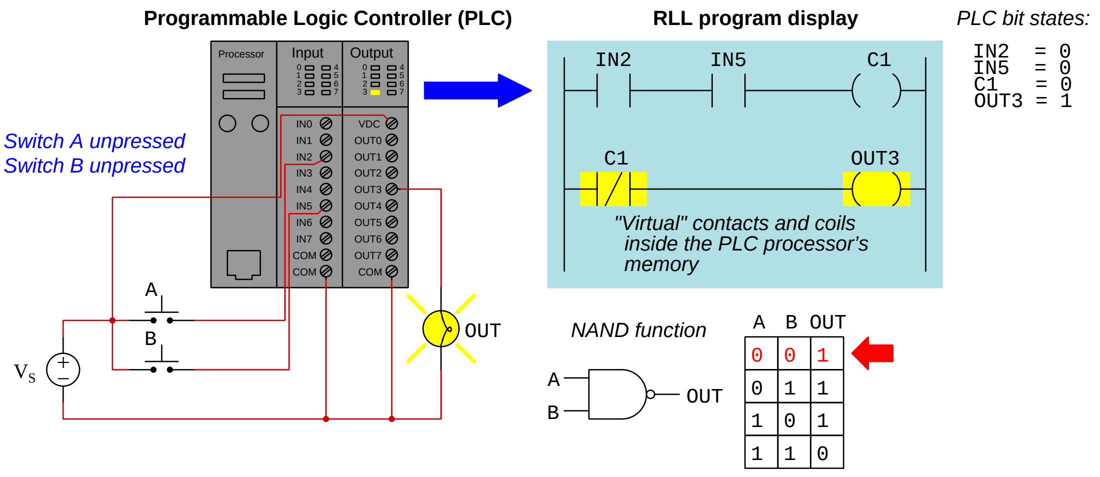
Chapter 2

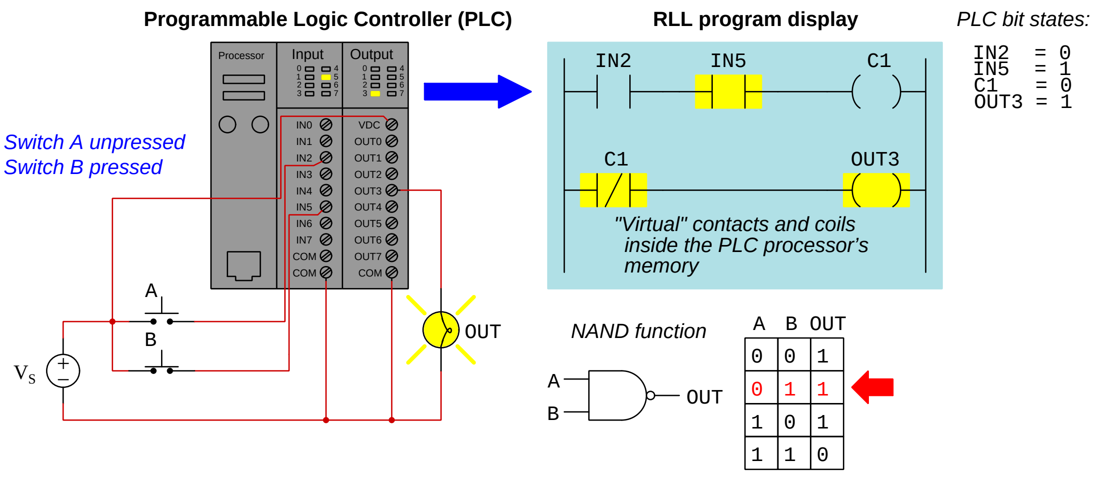
Case Tutorial

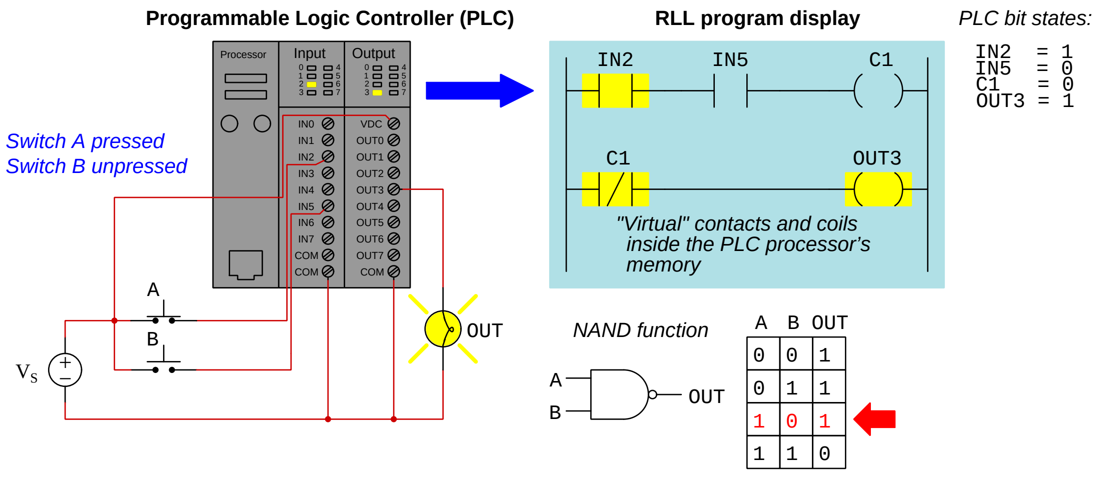
The idea behind a *Case Tutorial* is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as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

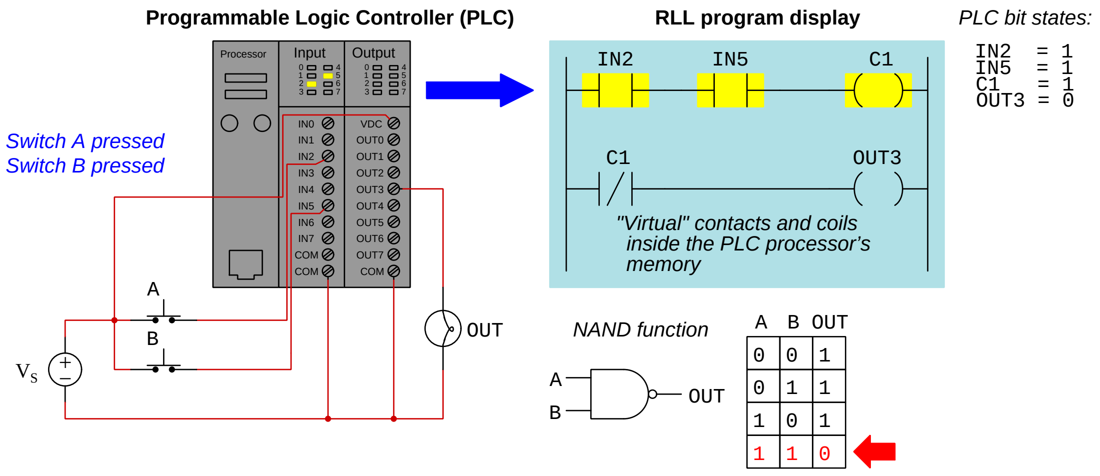
These examples also serve well as challenges following your reading of the other Tutorial(s) in this module – can you explain *why* the circuits behave as they do?

2.1 Example: NAND function in a PLC



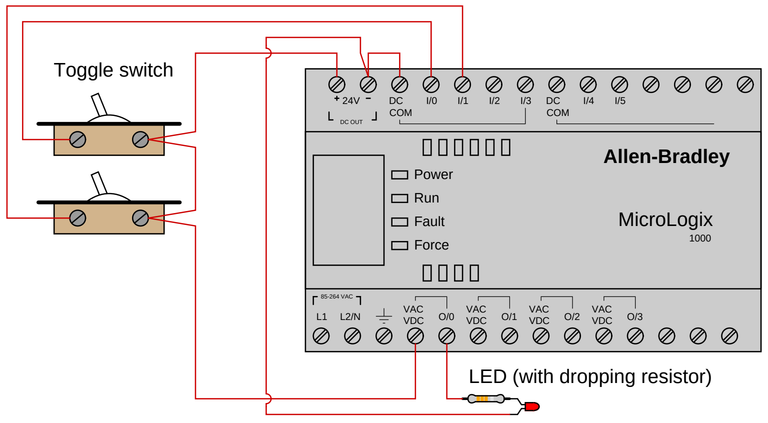


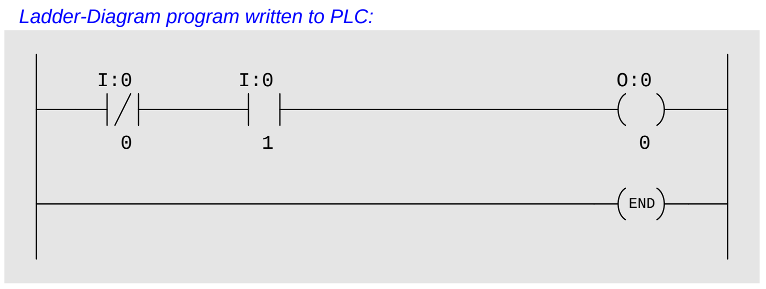

Programmable Logic Controller (PLC)
RLL program display
PLC bit states:
IN2 = 1
IN5 = 0
C1 = 0
OUT3 = 1
Switch A pressed
Switch B unpressed
Processor
Input
Output
IN0 IN1 IN2 IN3 IN4 IN5 IN6 IN7 COM COM
VDC OUT0 OUT1 OUT2 OUT3 OUT4 OUT5 OUT6 OUT7 COM
A
B
V_S
OUT
IN2 IN5 C1
C1 OUT3
"Virtual" contacts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

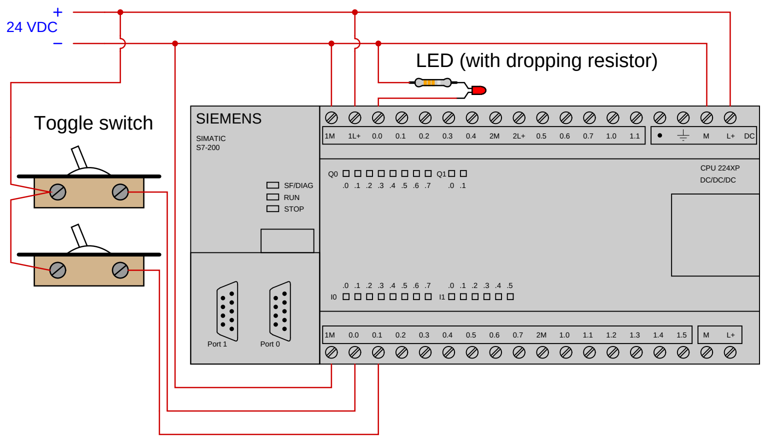


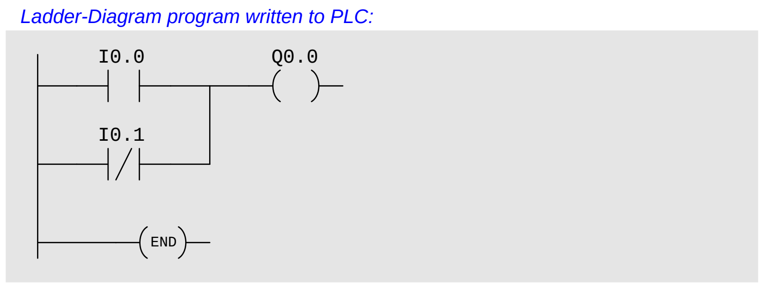


Note how Allen-Bradley I/O is labeled in the program: input bits designated by the letter `I` and output bits designated by the letter `O`.

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:

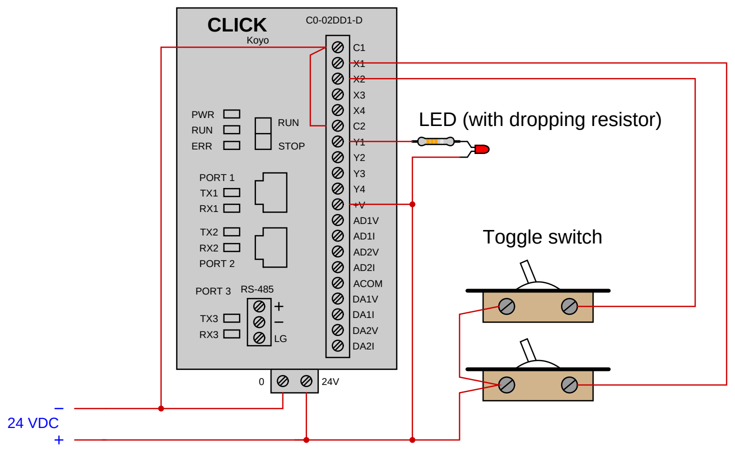


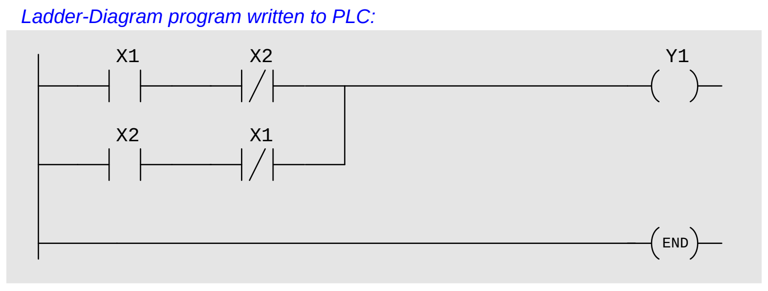


Note how Siemens I/O is labeled in the program: input bits designated by the letter `I` and output bits designated by the letter `Q`.

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:





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`.

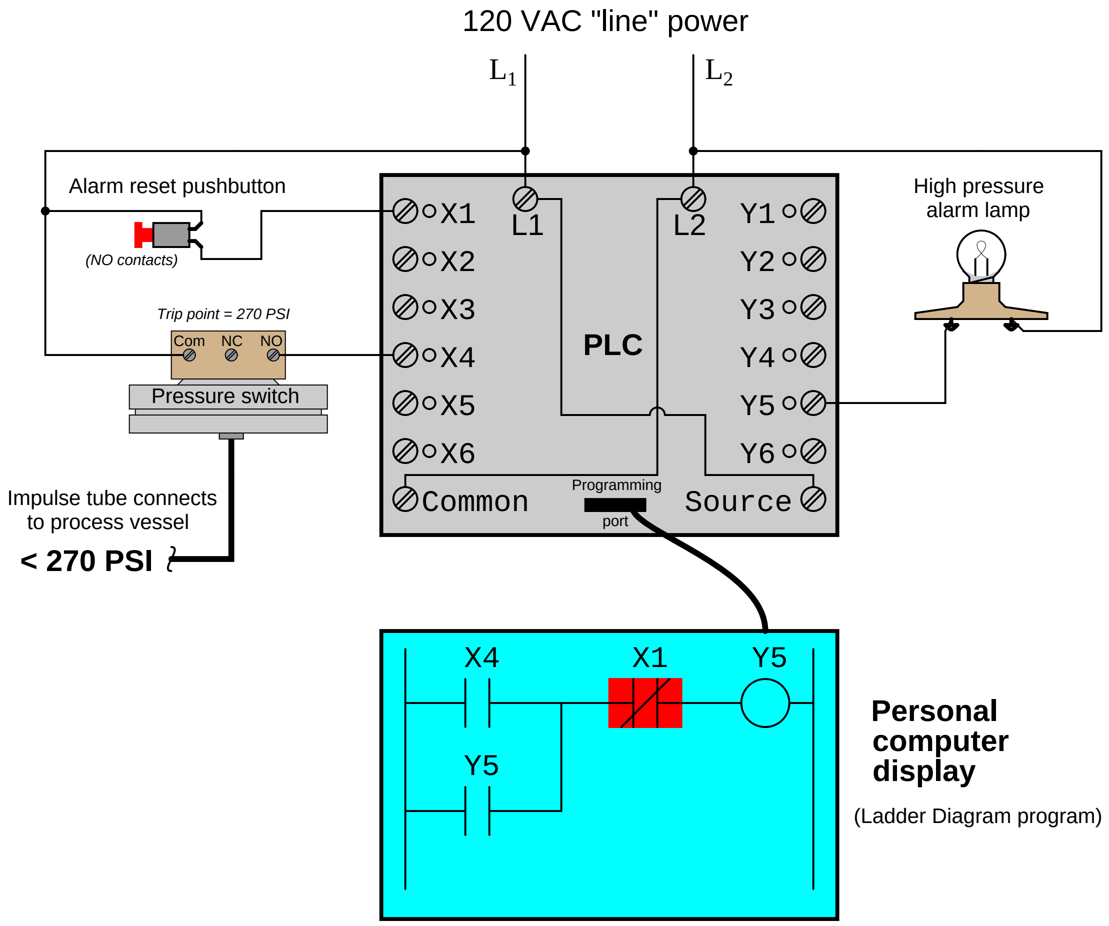
In order to energize the LED, at least one of the following conditions must be met:

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

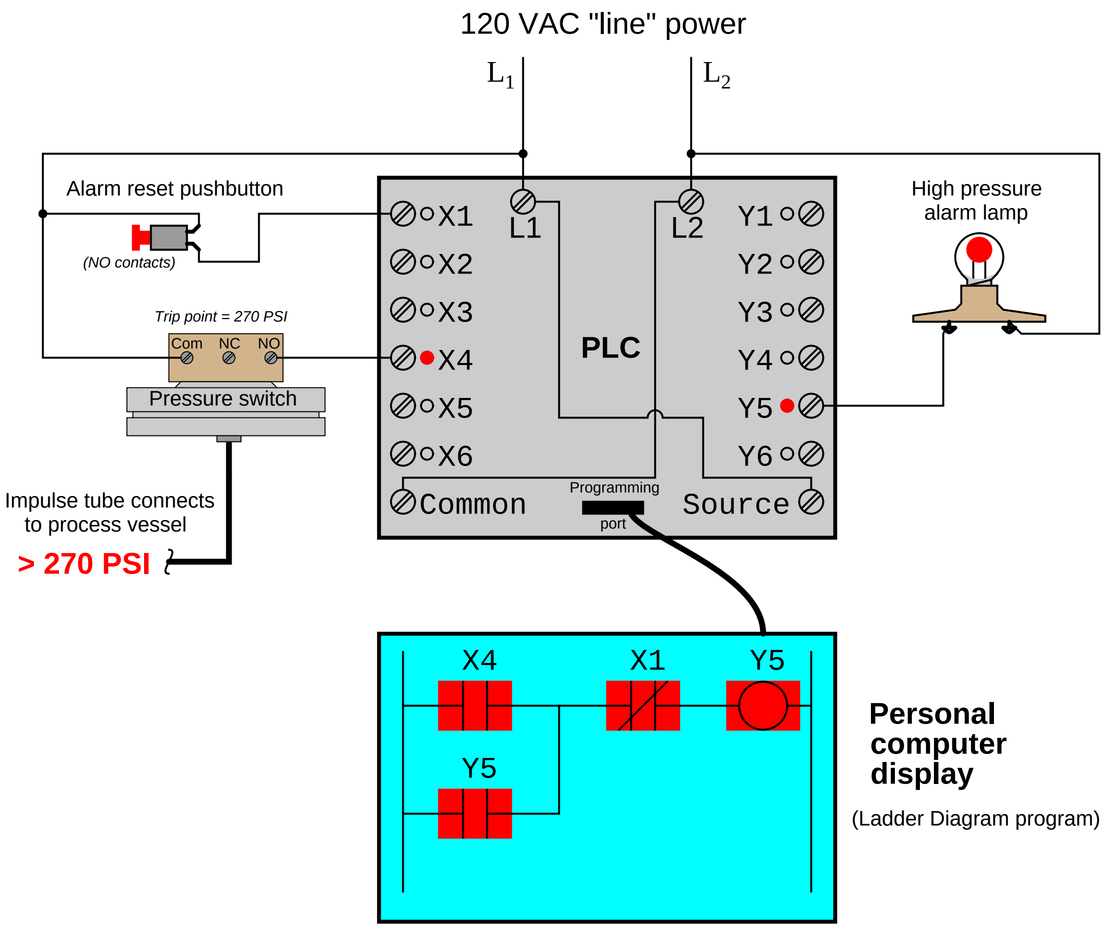
either the switch connected to input terminal 0.0 must be on (closed) or the switch connected to input terminal 0.1 must be off (open).

2.3 Example: high-pressure PLC-based alarm

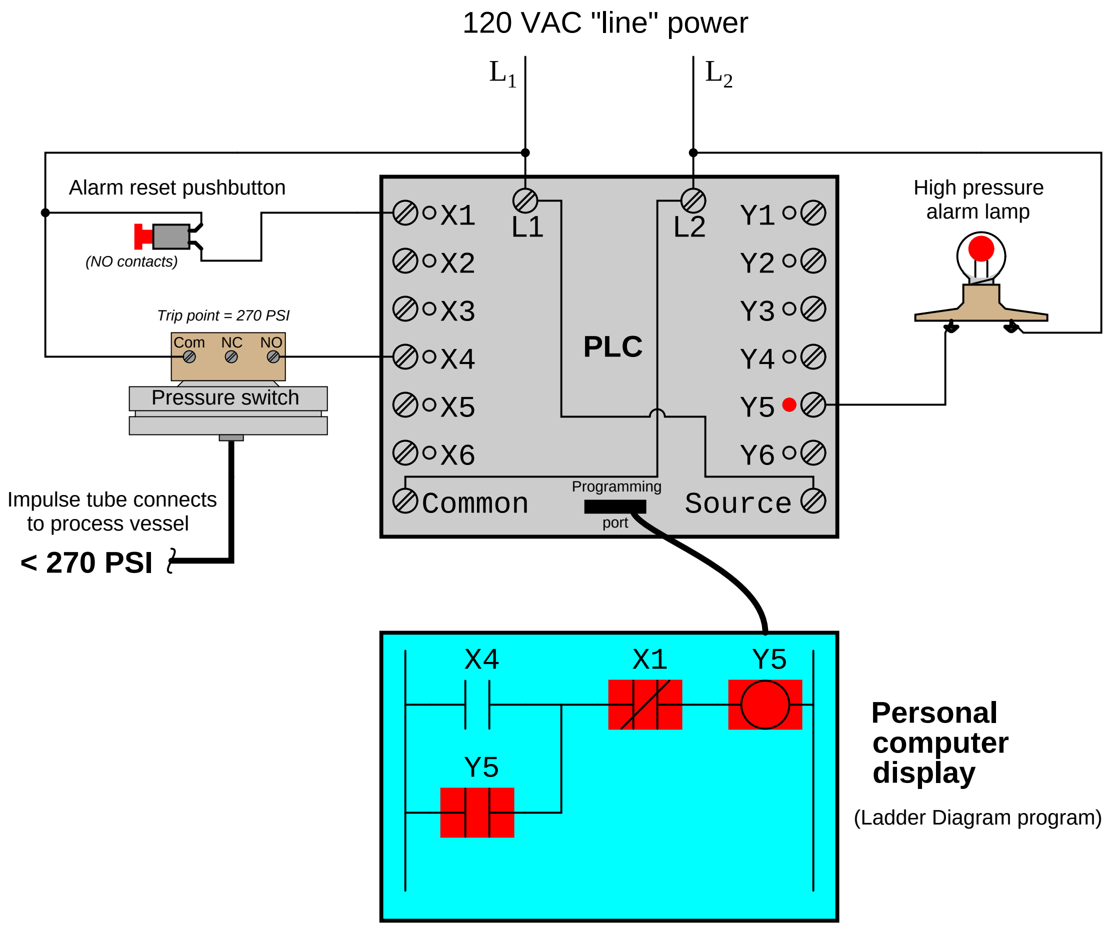
Step #1 – fluid pressure inside the process vessel is low, and the alarm lamp is de-energized:



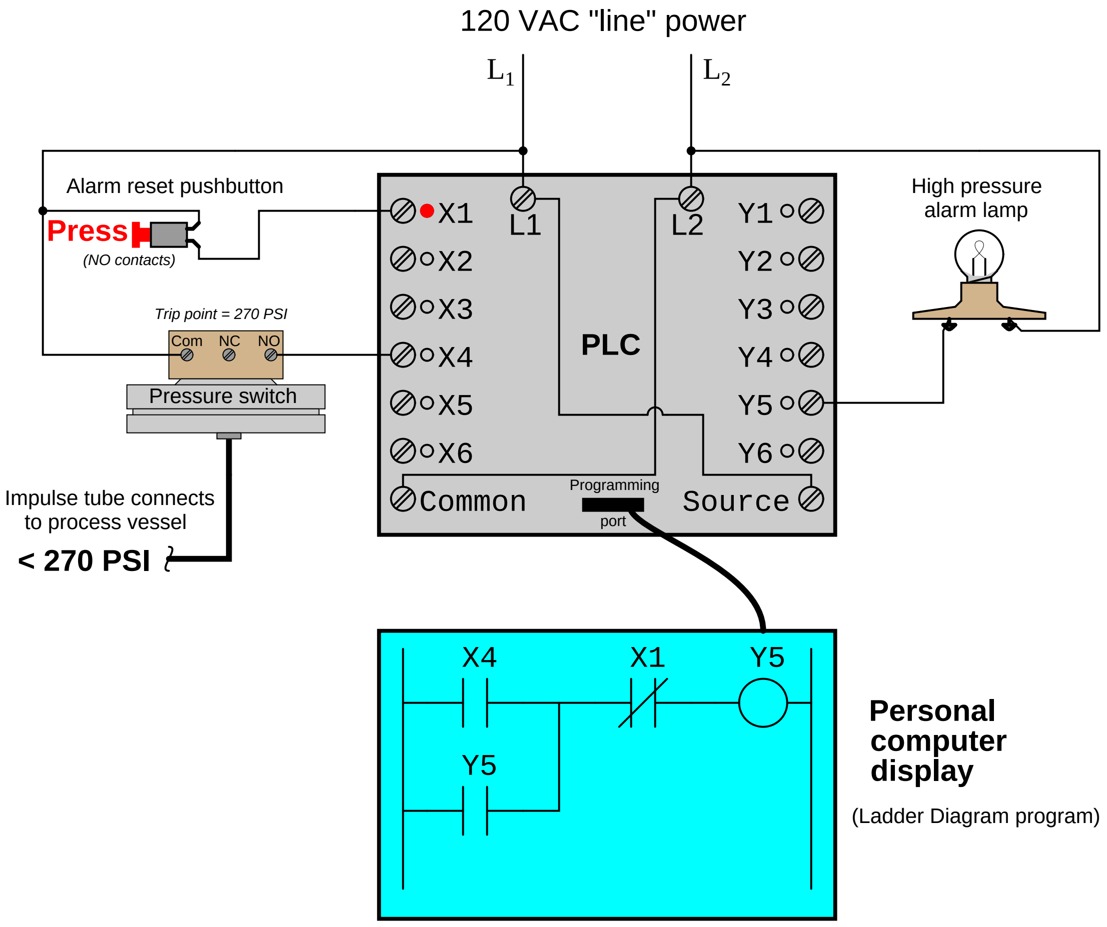
Step #2 – the pressure switch detects a high fluid pressure condition, triggering the PLC to energize the alarm lamp:



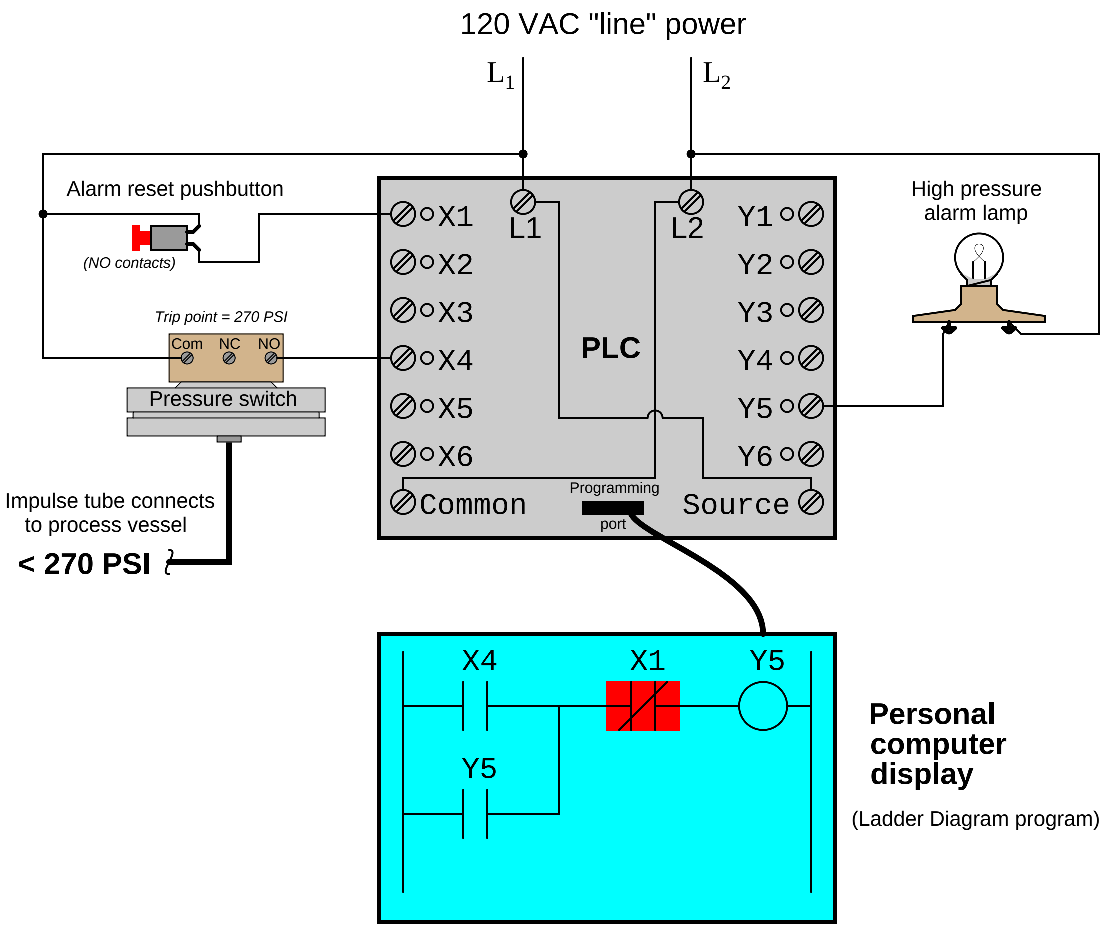
Step #3 – fluid pressure decreases below the pressure switch's trip point, but the PLC maintains power to the alarm lamp to signify that a high-pressure event happened:



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



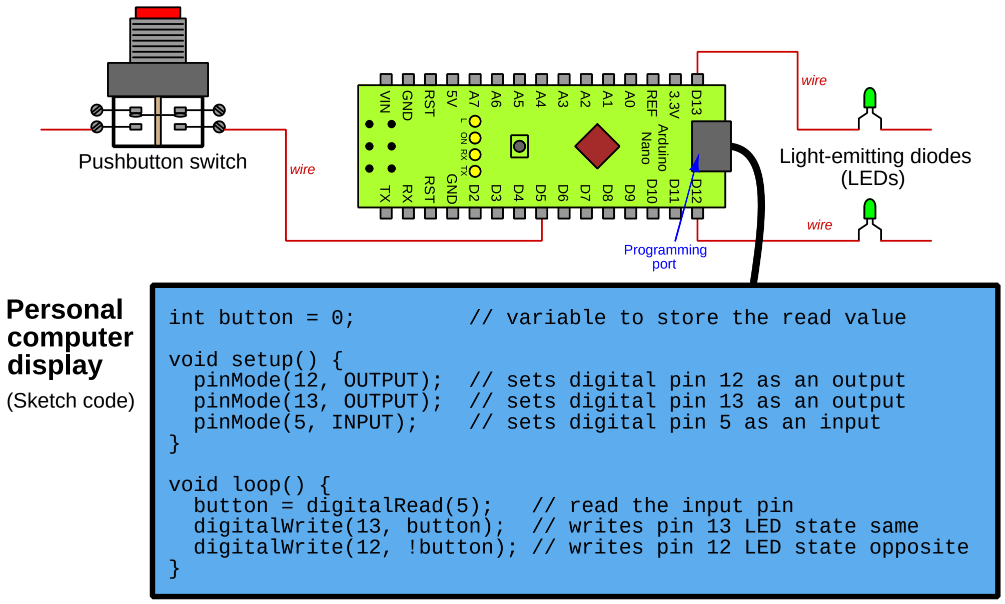
Step #5 – the lamp remains de-energized, awaiting another high-pressure event before energizing again:



2.4 Example: 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:


Pushbutton switch
wire
wire
wire
Arduino Nano
VIN GND RST 5V A7 A6 A5 A4 A3 A2 A1 A0 REF 3.3V D13
TX RX RST GND D2 D3 D4 D5 D6 D7 D8 D9 D10 D11 D12
L ON RX TX
Programming port
Light-emitting diodes (LEDs)
Personal computer display
(Sketch code)

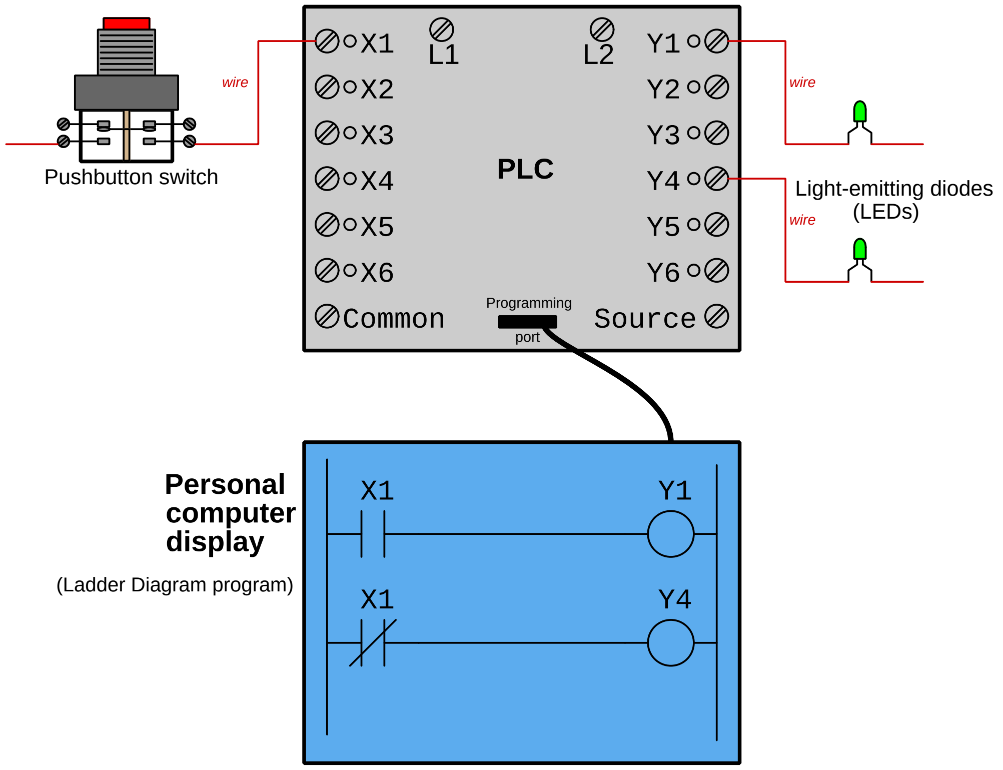
```
int button = 0;          // variable to store the read value

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:

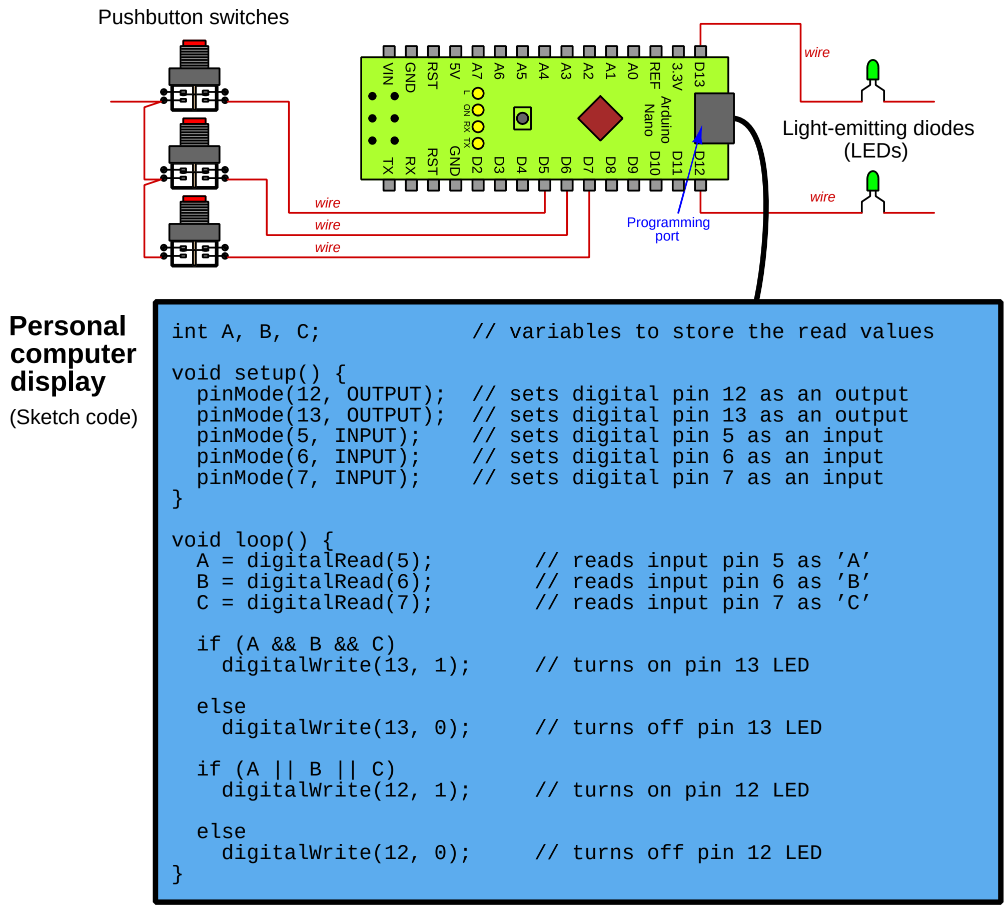


Pressing the pushbutton switch energizes input X1, and that electrically-energized state is read as a "1" value and stored in the PLC's `X1` variable. This "1" value causes the normally-open virtual switch contact to close and activate output Y1 to turn on that LED, and also causes the normally-closed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

2.5 Example: Arduino versus PLC logic functions

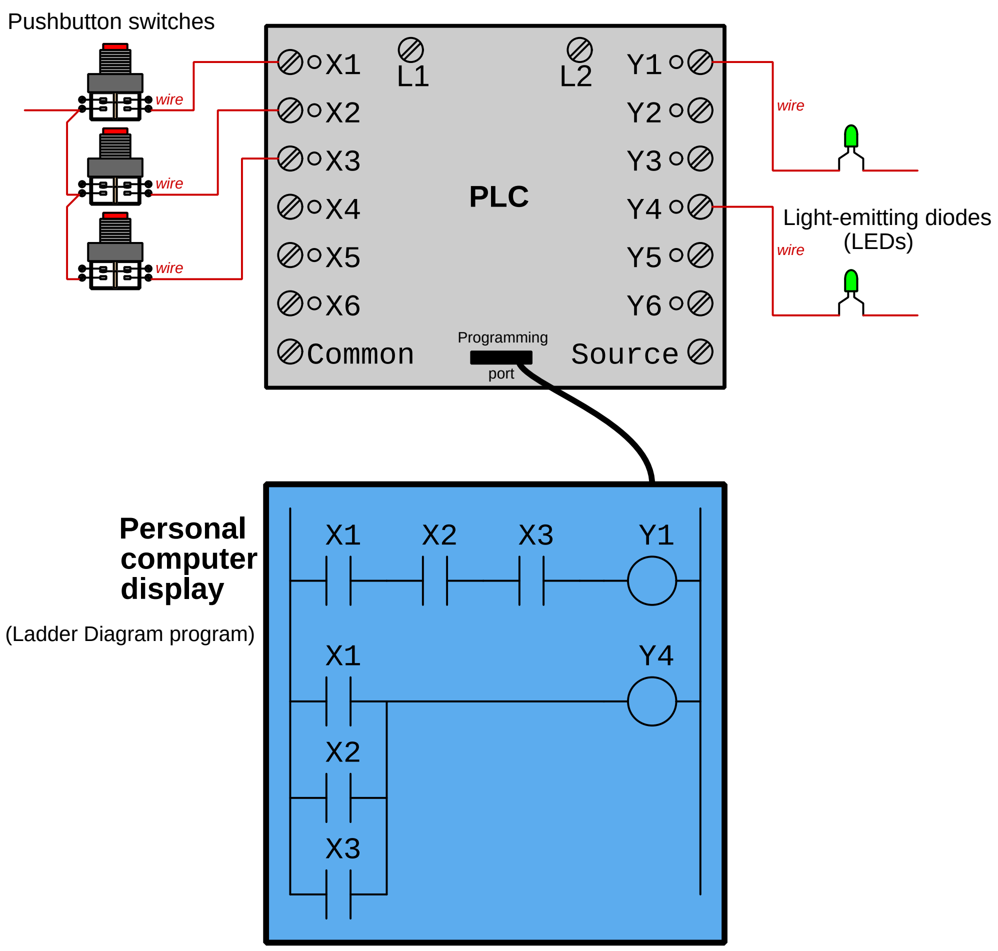
An *AND* function is where multiple conditions all must be “true” in order to generate a response. An *OR* function is where just one of those multiple conditions need be “true” to make the same response. The following Arduino program implements both of these logical functions, with one LED designated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


Pushbutton switches
wire
wire
wire
wire
wire
Arduino Nano
VIN GND RST 5V A7 A6 A5 A4 A3 A2 A1 A0 REF 3.3V D13
TX RX RST GND D2 D3 D4 D5 D6 D7 D8 D9 D10 D11 D12
L ON RX TX
Programming port
Light-emitting diodes (LEDs)
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.

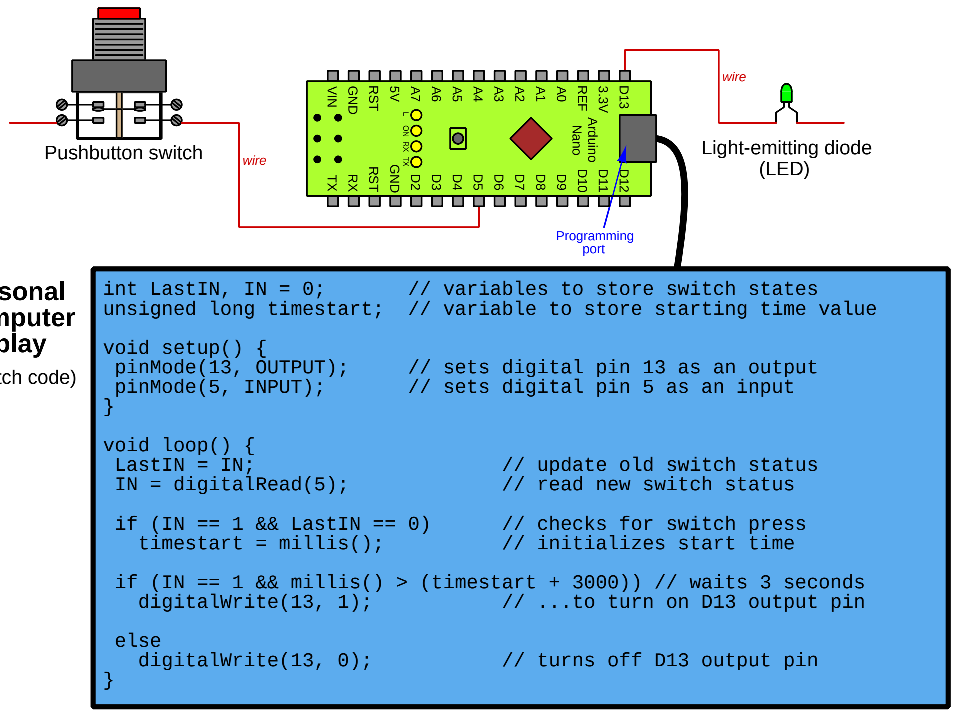
PLCs also implement *AND* and *OR* functions, but do so by connecting their virtual switches either in "series" (all in a row) or in "parallel", respectively:



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.

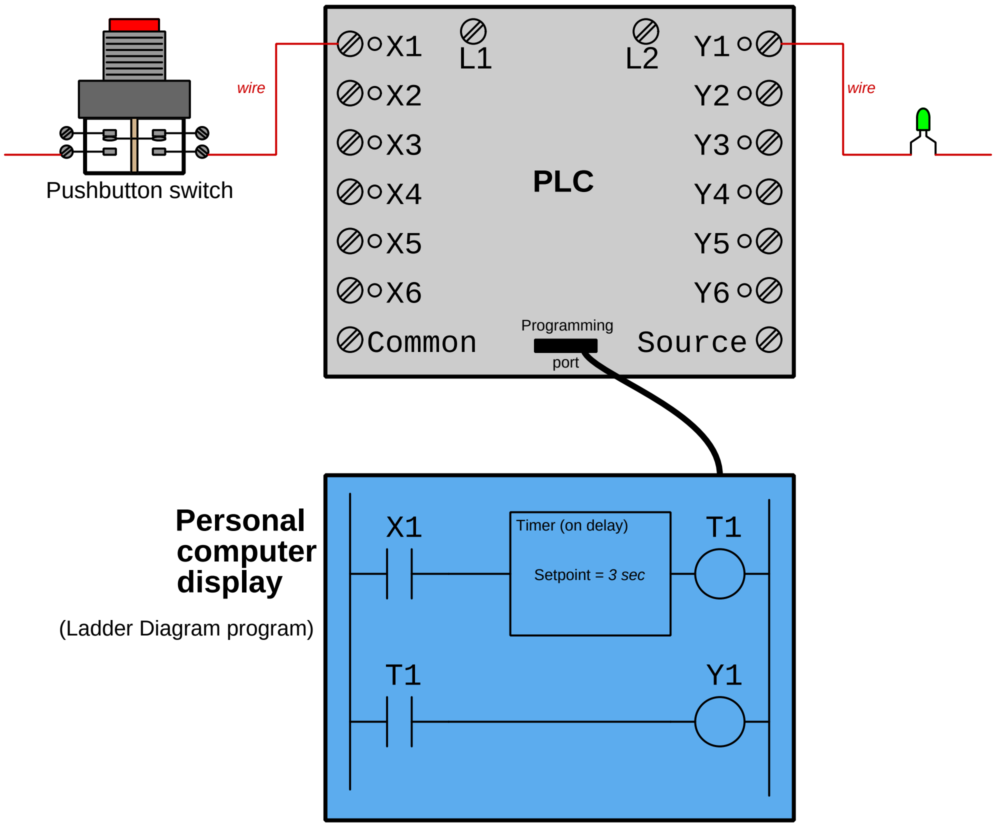
2.6 Example: Arduino versus PLC on-delay timer

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:



```
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

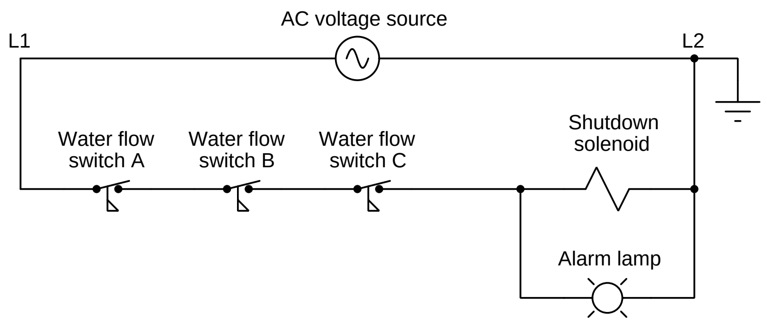
Simplified Tutorial

To understand what a *Programmable Logic Controller* or *PLC* does, it is helpful to study industrial control technology pre-dating PLCs. Like many technologies, PLCs were invented as a solution to a problem, that problem being *how to easily configure electrical control systems for industrial machines and processes.*

Prior to the advent of PLCs, most industrial control circuits used hard-wired switches and electromechanical relays to implement the necessary AND and OR and NOT logical functions. The PLC was invented as a means to implement the same logical functionality but in such a way that the circuitry did not have to be re-wired whenever a change was necessary. Being a programmable digital device like any digital computer, a PLC could be re-configured for a different control logic scheme simply by altering bit-states in its digital memory.

Let's begin with a practical example. Suppose we have a water cooling system for a large and expensive machine such as an industrial engine, and need an alarm system to monitor the flow of cooling water because a loss of cooling water would mean destruction for this machine. Coolant is so critically important that three different pumps provide cooling water through separate pipes to this same machine, the redundancy of pumps and pipes ensuring a greater level of cooling reliability. However, if water flow to this machine ceases for any reason, we want our electrical alarm system to activate a solenoid valve that will shut down the machine, as well as an indicator light to alert people what happened.

If we install three flow-sensing switches with normally-closed[1] contacts, one switch per water pipe, and connect them in series with each other to feed electrical power to the solenoid valve and to the alarm light, we will have a practical alarm-and-shutdown system for this expensive machine. The necessary connections between these components are shown in the following "ladder diagram" which is a common form of documentation for electrical control systems:



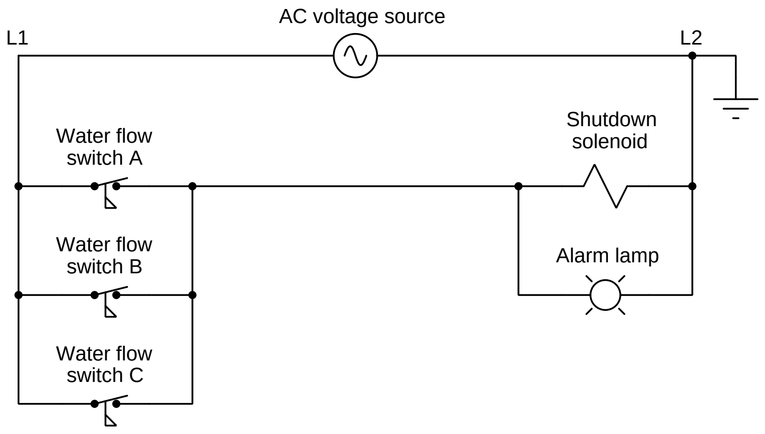
All three switches will need to close in order to energize the shutdown solenoid and the alarm lamp. This means all three water pipes' flows would have to cease in order to have all three of these flow switches return to their resting ("normal") states. Thus, this system will not take action unless and until cooling water flow stops through all three pipes. If we happen to lose cooling water flow through any one or any two pipes at a time, the machine will still be allowed to operate and the alarm lamp will not energize.

Such a system seems perfectly appropriate if the flow of water through just one pipe will be enough to sufficiently cool the machine, since the only real emergency would be a loss of cooling water flow for *all* three pipes.

[1]Recall that the "normal" status of an electrical switch is its electrical status in a condition of no physical stimulus. That is, a "normally-closed" (NC) switch will exhibit closed contacts when it is at rest. For a water flow switch, "rest" means a condition of no water flow. Since we want our flow switches to conduct electricity to the shutdown solenoid when water flow stops, we need switches that close with no water flow – i.e. normally-closed flow switch contacts. These NC contacts will be held in their "open" states by the presence of adequate water flow, which means under *regular operating conditions* they will actually be open. This is what is so confusing about the "normal" status of switches: a switch's "normal" status may or may not happen to match its *typical* status when everything is operating as it should!

However, suppose one of these three flow switches happens to *fail* in an open state, remaining open even when water flow stops. If water flow through all three pipes happens to cease while the switch is faulted like this, the system will neither shut down the machine nor turn on the alarm lamp as intended! By designing the circuit such that all three switches must indicate low flow before initiating a shutdown of the machine, we have left that machine vulnerable to any of the water flow switches failing open.

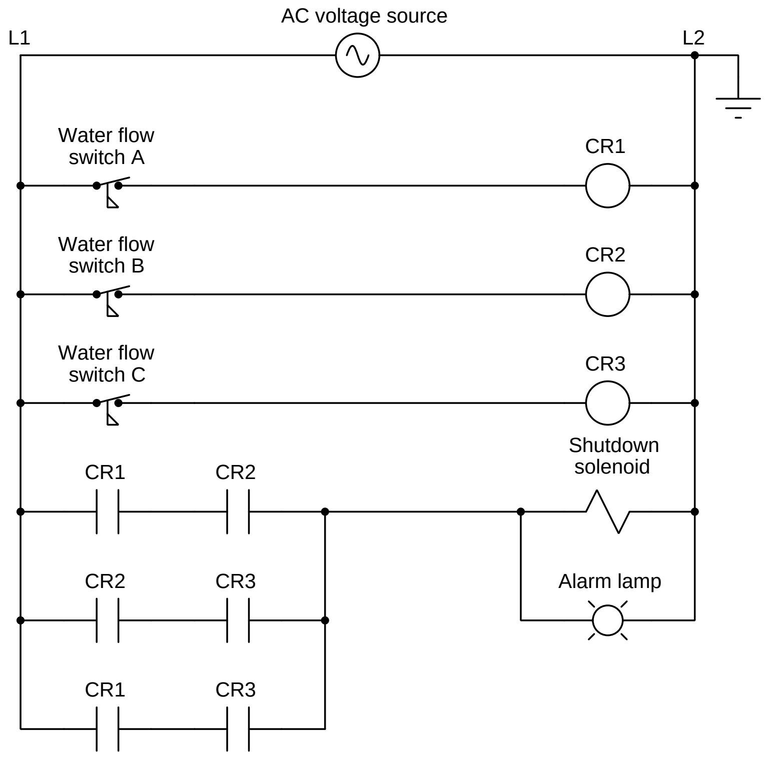
One solution to this problem is to re-configure the circuit so that the three switch contacts are wired in parallel with each other rather than in series. This way, the machine will shut down and the alarm lamp will energize if *any one or more* of the switches indicate low cooling water flow:



No longer is the machine vulnerable to a single water flow switch failing open – now, *all three switches* would have to fail open to leave the machine unprotected in the event of a total cooling water outage. However, the trade-off with the parallel-wired design is that now the circuit will shut down the machine and activate the alarm lamp even if just a single water pipe's flow is too low, even if there is still sufficient water flow through the other two to cool it. In other words, this newly-wired system errs on the side of paranoia, and therefore is prone to *nuisance trip events* where the machine shuts down unnecessarily.

A compromise solution to this new problem is to design the circuit in such a way that it takes *two out of three* water flow switches agreeing to either let the machine run or to shut it down. This way, a single failed-open flow switch will not jeopardize the machine, and neither will a loss of cooling water through just one pipe result in a "nuisance trip". However, it is electrically impossible to wire just three switch contacts to fulfill this two-out-of-three logic function.

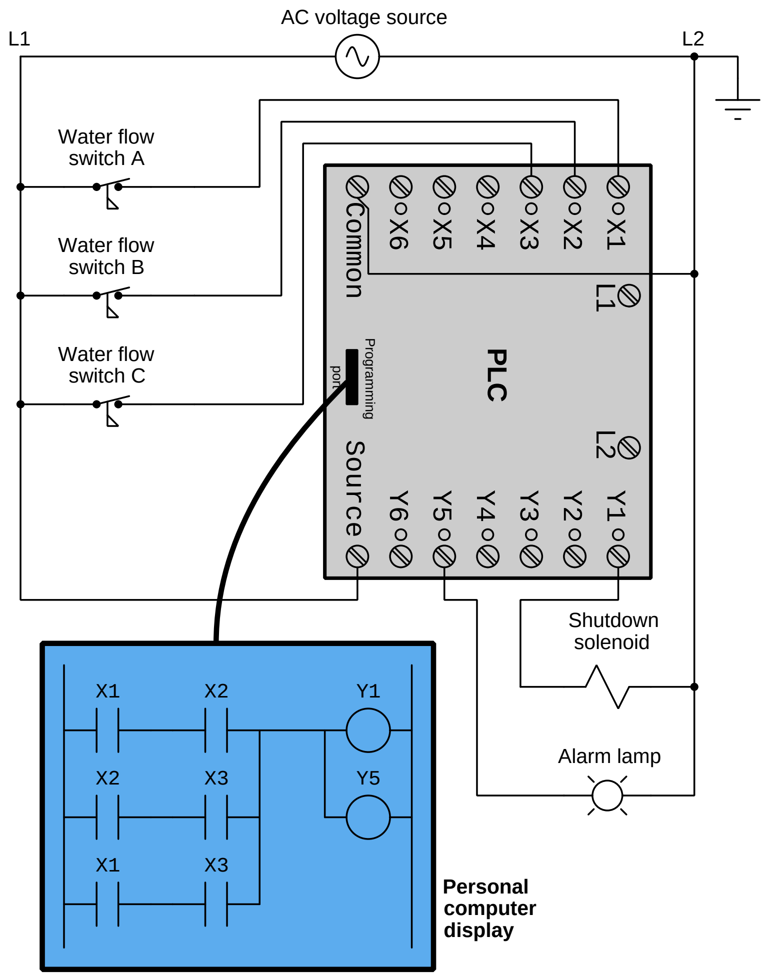
We can implement a two-out-of-three shutdown/alarm system by using three electromechanical relays, letting each flow switch activate one relay coil and then using multiple switches in each relay wired in a series-parallel configuration to form the two-out-of-three logic:



Each of these control relays (CR1, CR2, and CR3) has one coil (symbolized by a circle) and two normally-open contacts (symbolized by parallel vertical line segments), the association between coil and contact(s) indicated by label as is standard in "ladder logic" diagrams. In the configuration shown, the shutdown solenoid and alarm lamp energize if any two or more flow switches close, and conversely the machine will still run if any two or more flow switches remain open. This is the desired two-out-of-three shutdown logic.

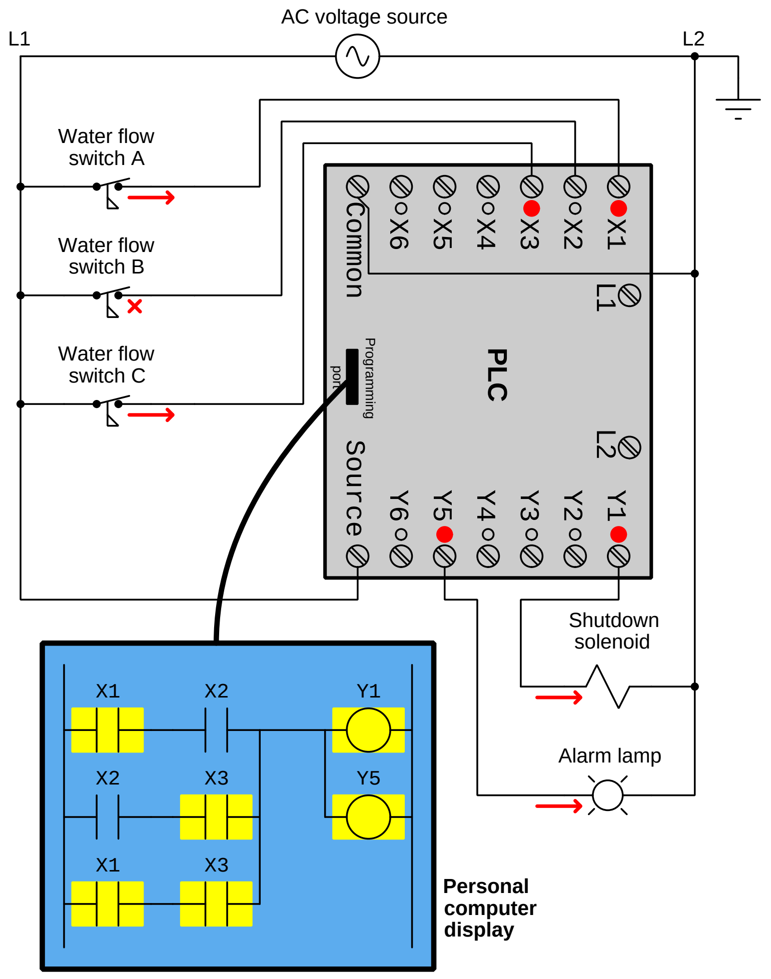
Although this is a practical solution, it involves a fair amount of wiring between relay coils and switch contacts, and the addition of those relays constitute more points of failure because electromechanical relays have moving parts and therefore wear out over time. Furthermore, if we wanted to add more features such as making the alarm lamp energize on a one-out-of-three basis while keeping the shutdown solenoid on two-out-of-three logic, we would likely have to install relays containing more switch contacts and of course spend significant time re-wiring everything. The circuit as shown does indeed work, but it offers no flexibility for future changes.

A *Programmable Logic Controller* is a solid-state digital computer designed to replicate the functionality of those control relays without the inherent limitations. Each of the three water flow switches energizes its own *input terminal* on the PLC (labeled with X designators). The solenoid coil and alarm lamp connect to individual *output terminals* on the PLC (labeled with Y designators). Inside the PLC is a microprocessor executing a program written in a graphical programming language resembling relay contacts and coils, instructing it as to how and when it should activate each Y output based on the states of the X inputs:



Each "relay contact" inside the PLC program is actually a *read instruction* examining the electrical state of its respective X input (as controlled by each flow switch), and based on that state will either "open" or "close" its portion of the virtual circuit. Each "coil" in the PLC program is a *write instruction* commanding its respective Y output to either turn on or turn off real electrical power to the external load. In essence, the PLC program acts like a virtual circuit passing or blocking virtual electricity through virtual contacts to virtual coils.

A helpful feature of modern PLCs is *color highlighting* to show the status of these virtual switches and virtual coils as they block or conduct imaginary electricity. For example, in a case where water flow switches A and C are closed but switch B is open, the PLC would respond as shown below:

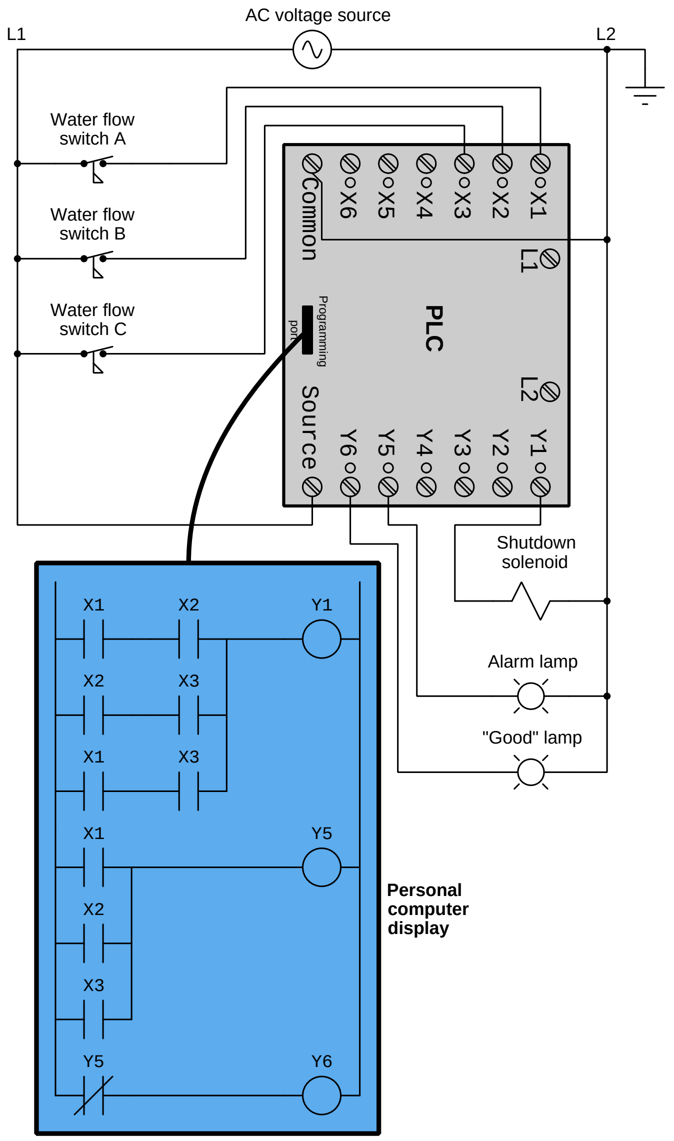


In this example only the bottom "rung" of this "ladder" program has virtual continuity all the way through to provide virtual electricity to virtual coils `Y1` and `Y5`. On the other two rungs we see both `X1` and `X3` "contacts" showing virtual continuity, but because of the non-colored `X2` "contact" in each of those two paths no virtual electricity will "flow" there.

Not only is color highlighting useful for experientially learning how PLCs work, but it is extremely useful when troubleshooting PLC-controlled systems to determine the internal state of the PLC's program at any time[2].

[2]Readers familiar with text-based computer programming languages know this as the *debugging* feature of their development environment, where the software will show you bit states, register values, program step, and many other important parameters to show you what state the program is in at any given step of its execution. This is a standard feature for PLC programming due to its diagnostic value.

Unlike hard-wired relay circuits where additional functionality usually requires additional components and additional wires, with a PLC all we need to do is add and/or modify *virtual wiring* in the program. Take for example this modification where we maintain two-out-of-three logic for the shutdown function but change the alarm function to one-out-of-three and add another lamp to indicate a "good" status in addition to the existing "alarm" status:

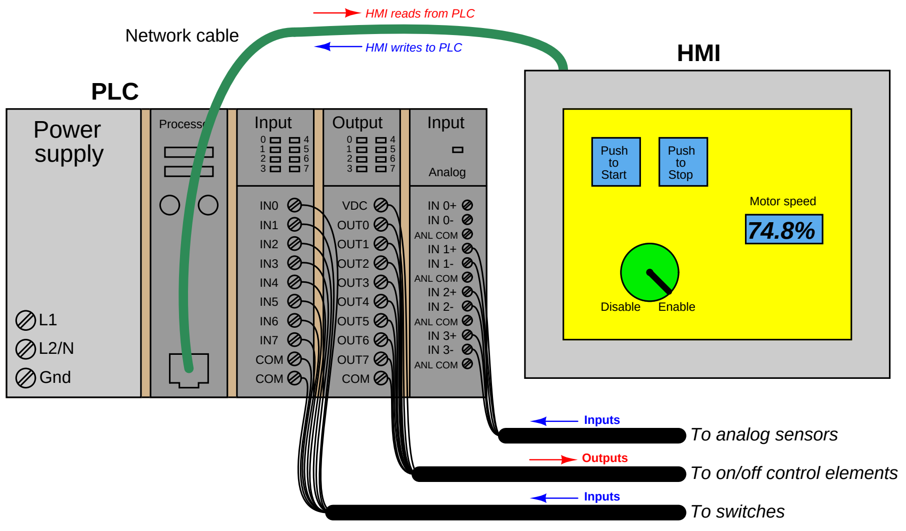


Note the use of a *normally-closed* (NC) virtual contact `Y5`, which will be "closed" (colored) whenever virtual coil `Y5` is "de-energized" (un-colored) and vice-versa, its purpose being to ensure the "good" lamp's status is always opposite that of the "alarm" lamp status.

This exploration of how a PLC may be used to replace electromechanical relays in a shutdown/alarm control circuit describes just a small portion of a modern PLC's capabilities. In addition to virtual contacts and virtual coils, PLCs offer virtual timers, virtual counters, digital communication capability, and a host of other features.

A modern trend in PLC technology is the ability to program in languages other than "ladder diagram". The ladder diagram language was invented for the sake of making PLC programming easy to understand for technicians familiar with electrical wiring, but compared to many other programming languages ladder diagram is extremely limited. So, some PLCs may be programmed in text-based languages and/or in function blocks for increased versatility.

One of the more impressive features of a PLC is its ability to communicate data over digital networks to other PLCs, other computers, and/or to display screens so that operations personnel can visually perceive the states of digital bits and words within the PLC. The common term to describe visual display screens intended for use with PLCs is *Human-Machine Interface* or *HMI*. HMIs are really nothing more than "hardened" personal computers built ruggedly and in a compact format to facilitate their use in industrial environments. Most industrial HMI panels come equipped with touch-sensitive screens, allowing operators to press their fingertips on displayed objects to change screens, view details on portions of the process, etc. An example illustration of an HMI working in conjunction with a PLC is shown here:



Chapter 4

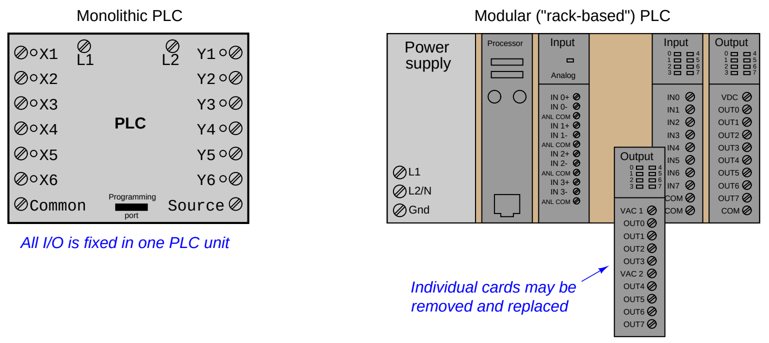
Full Tutorial

Programmable logic controllers are essentially nothing more than special-purpose, industrial computers. As such, they are built far more ruggedly than an ordinary personal computer (PC), and designed to run extremely reliable operating system software[1]. PLCs as a rule do not contain disk drives, cooling fans, or any other moving parts. This is an intentional design decision, intended to maximize the reliability of the hardware in harsh industrial environments where the PLC chassis may be subjected to temperature extremes, vibration, humidity, and airborne particulates (dust, fibers, and/or fumes).

[1]There are such things as *soft PLCs*, which consist of special-purpose software running on an ordinary personal computer (PC) with some common operating system. Soft PLCs enjoy the high speed and immense memory capacity of modern personal computers, but do not possess the same ruggedness either in hardware construction or in operating system design. Their applications should be limited to non-critical controls where neither main process production nor safety would be jeopardized by a control system failure.

4.1 What does a PLC do?

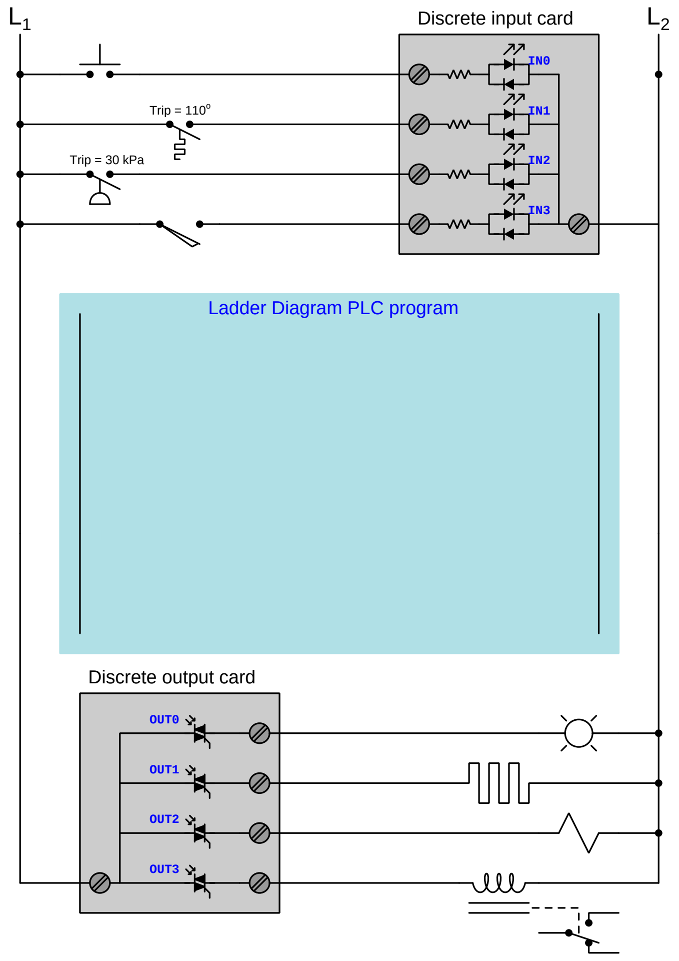
A *Programmable Logic Controller*, or *PLC*, is a general-purpose industrial computer designed to be easily programmed by end-user maintenance and engineering personnel for specific control functions. PLCs have input and output channels (often hosted on removable "I/O cards") intended to connect to field 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.

This Simplified Tutorial will explore some of the basic functionality common to all PLCs programmed in a Ladder Diagram language, and will do so *by example*. Although details of programming convention, syntax, and symbology vary somewhat from one model of PLC to another, the basic concepts explored in this tutorial are largely the same.

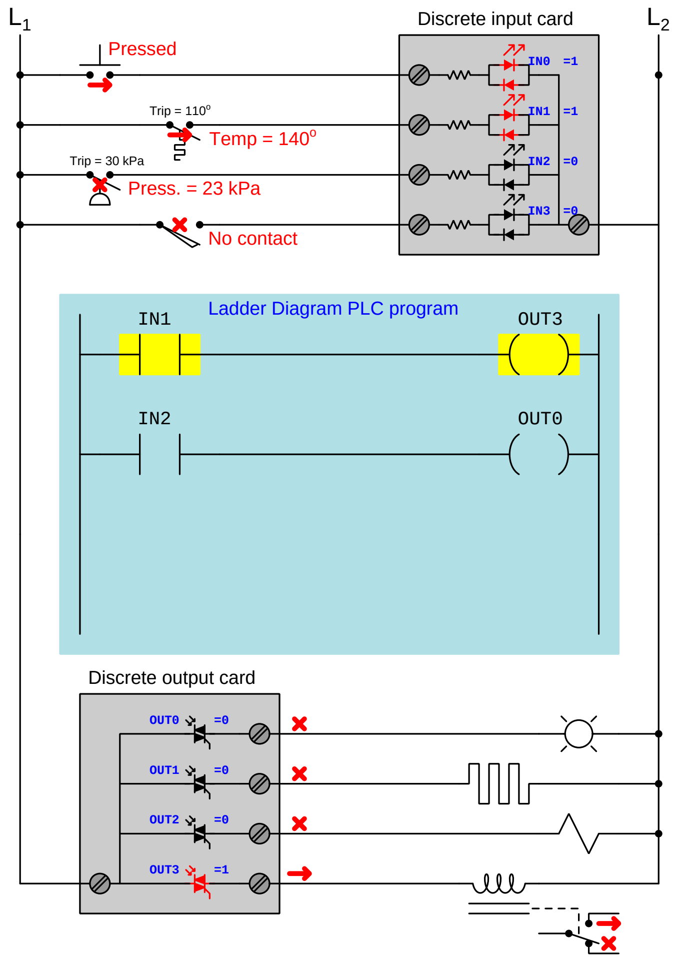
The following diagram shows a PLC separated into three sections: (1) a *discrete input card*, (2) the *program* space in the processor's memory, and (3) a *discrete output card*:



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[2] 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).

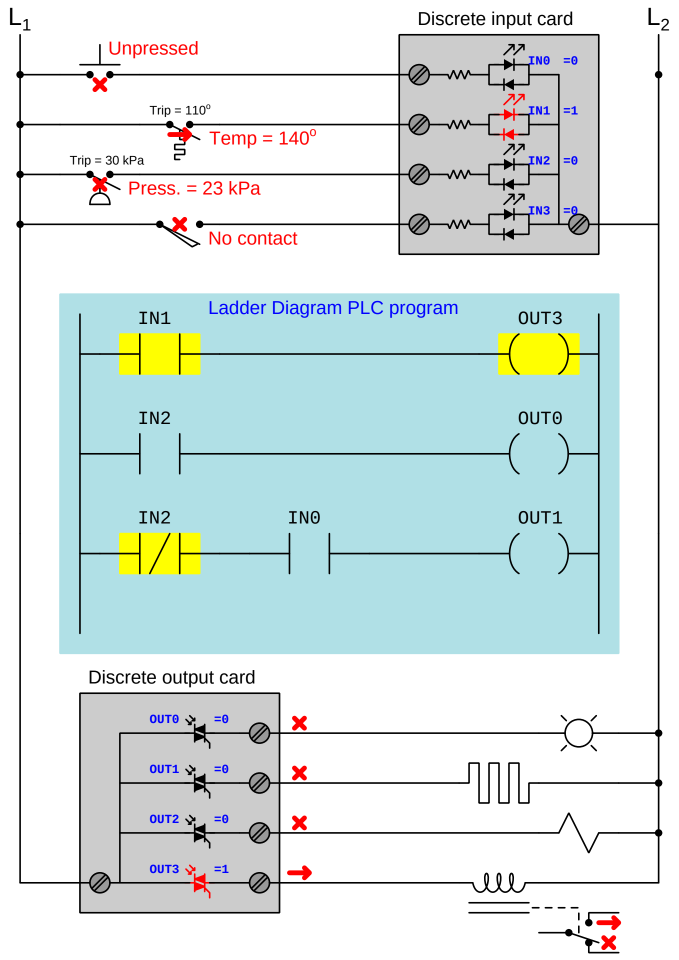
[2]Not shown in this simplified diagram are the optotransistors coupled to the LEDs inside the input card, translating each LED's state to a discrete logic level at the transistor to be interpreted by the PLC's digital processor. Likewise, another set of LEDs driven by the processor's outputs couple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

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



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

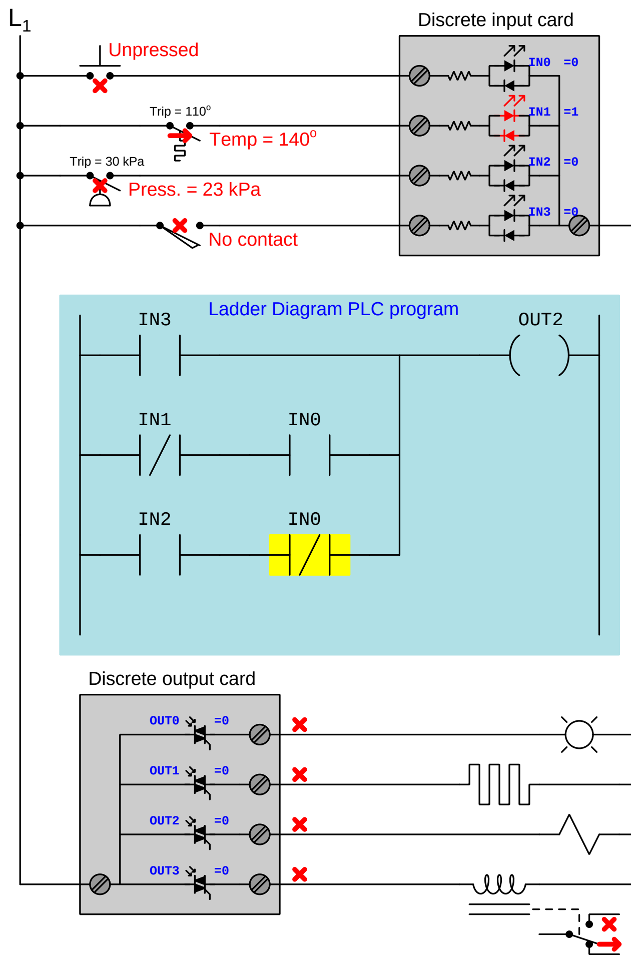
Things get more complex when we begin adding *normally-closed* contact instructions to the program. Consider this next diagram, with updated stimuli and an expanded Ladder Diagram program for the PLC to follow:



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

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

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



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

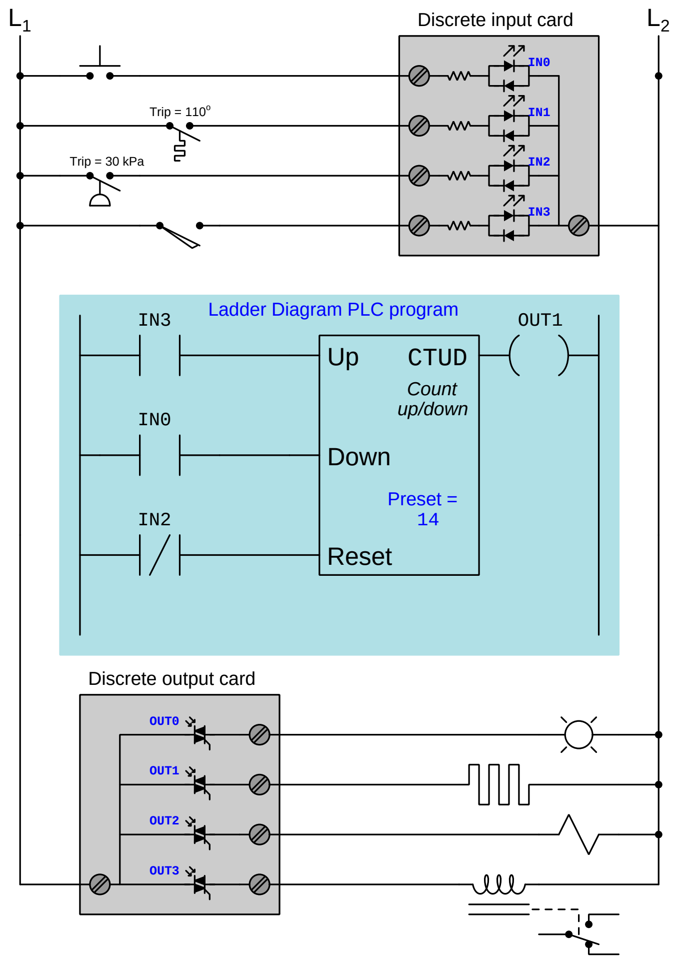
The logical chain of causality from input to output on a PLC is very important to understand, and will be represented here by a sequence of numbered statements:

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

All PLCs follow this chain of logic precisely, and this same causality *must* be mentally tracked in order to successfully analyze a Ladder Diagram program in a PLC. The most confusing part of this for new students seems to be the relationship of contact instructions to real-world switch inputs. Many students have an unfortunate tendency to want to directly[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 are rooted in the same error, namely not deliberately following the chain of causation from beginning to end (i.e. input energization → input bit state → contact instruction actuation → color based on normal type *and* bit state → coil color → output bit state → output energization).

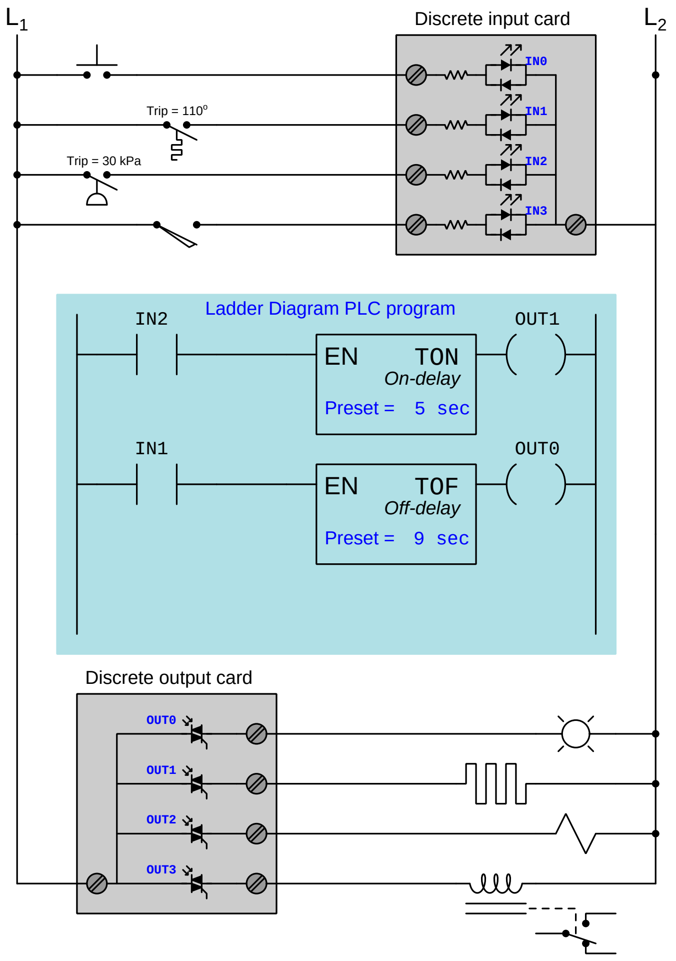
[3]For a normally-open contact instruction, this association is direct. However, for a normally-closed contact instruction it is inverted!

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.

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.

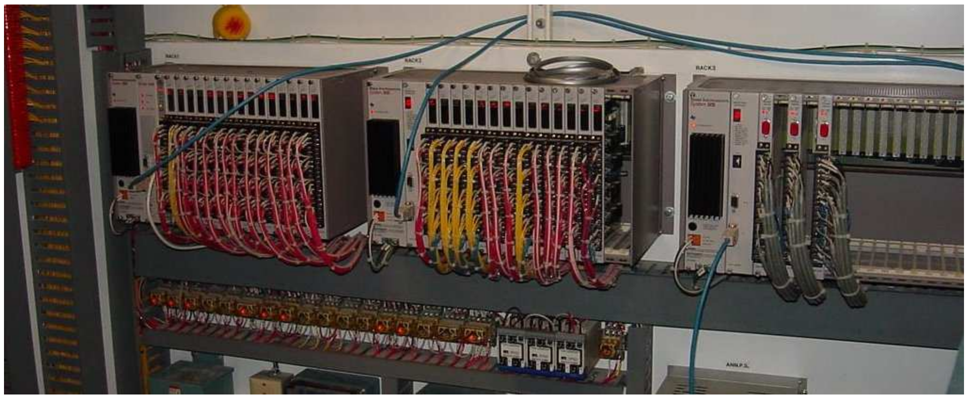
4.2 Types and sizes of PLCs

Large PLC systems consist of a rack into which circuit "cards" are plugged. These cards include processors, input and output (I/O) points, communications ports, and other functions necessary to the operation of a complete PLC system. Such "modular" PLCs may be configured differently according to the specific needs of the application. Individual card failures are also easier to repair in a modular system, since only the failed card need be replaced, not all the cards or the whole card rack.

Small PLC systems consist of a monolithic "brick" containing all processor, I/O, and communication functions. These PLCs are typically far less expensive than their modular cousins, but are also more limited in I/O capability and must be replaced as a whole in the event of failure.

The following photographs show several examples of real PLC systems, some modular and some monolithic. These selections are not comprehensive by any means, as there are many more manufacturers and models of PLC than those I have photographed. They do, however, represent some of the more common brands and models in current (2019) industrial use.

The first photograph is of a Siemens (Texas Instruments) 505 series PLC, installed in a control panel of a municipal wastewater treatment plant. This is an example of a modular PLC, with individual processor, I/O, and communication cards plugged into a rack. Three racks appear in this photograph (two completely filled with cards, and the third only partially filled):

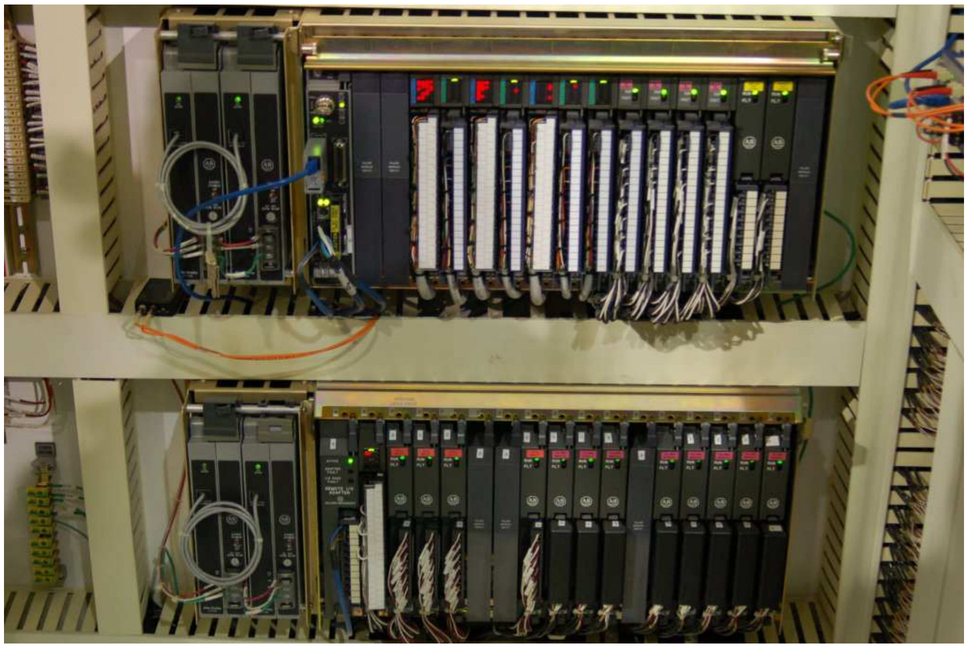


The power supply and processor card for each rack is located on the left-hand end, with I/O cards plugged into slots in the rest of the rack. Input devices such as switches and sensors connect by wire to terminals on *input* cards, while output devices such as lamps, solenoids, and motor contactor coils connect by wire to terminals on *output* cards.

One of the benefits of modular PLC construction is that I/O cards may be changed out as desired, altering the I/O configuration of the PLC as needed. If, for example, the PLC needs to be configured to monitor a greater number of sensors, more input cards may be plugged into the rack and subsequently wired to those sensors. Or, if the *type* of sensor needs to be changed – perhaps from a 24 volt DC sensor to one operating on 120 Volts AC – a different type of input card may be substituted to match the new sensor(s).

In this particular application, the PLC is used to sequence the operation of self-cleaning "trash racks" used to screen large debris such as rags, sticks, and other non-degradable items from municipal wastewater prior to treatment. These trash racks are actuated by electric motors, the captured debris scraped off and transported to a solid waste handling system. The motion of the trash racks, the sensing of wastewater levels and pressures, and the monitoring of any human-operated override controls are all managed by these PLCs. The programming of these PLCs involves timers, counters, sequencers, and other functions to properly manage the continuous operation of the trash racks.

The next photograph shows an Allen-Bradley (Rockwell) PLC-5 system, used to monitor and control the operation of a large natural gas compressor. Two racks appear in this first photograph, with different types of I/O cards plugged into each rack:

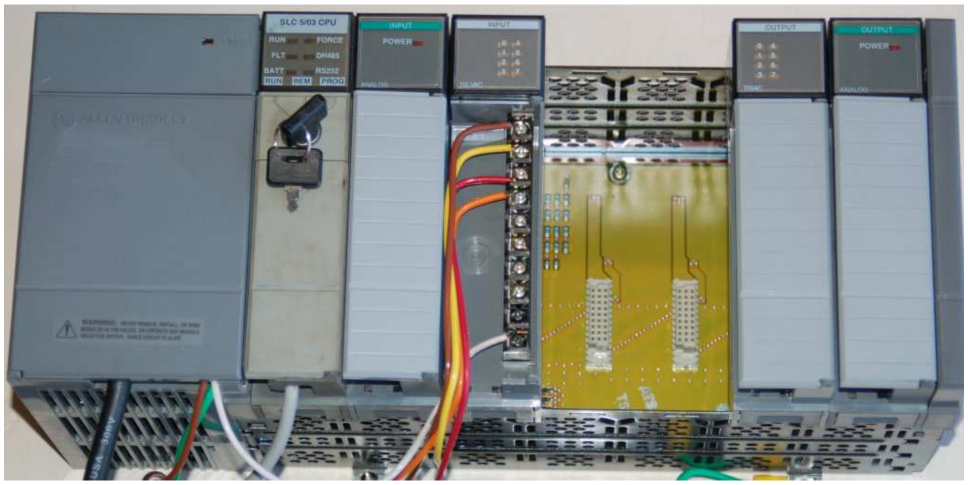


Like the Siemens 505 PLC seen previously, this Allen-Bradley PLC-5 system is fully modular and configurable. The types and locations of the I/O cards inserted into the rack may be altered by appropriately skilled technicians to suit any desired application. The programming of the PLC's processor card may also be altered if a change in the control strategy is desired for any reason.

In this particular application, the PLC is tasked with monitoring certain variables on the gas compressor unit, and taking corrective action if needed to keep the machine productive and safe. The automatic control afforded by the PLC ensures safe and efficient start-ups, shut-downs, and handling of emergency events. The networking and data-logging capability of the PLC ensures that critical data on the compressor unit may be viewed by the appropriate personnel. For this particular compressor station, the data gets communicated from Washington state where the compressor is located all the way to Utah state where the main operations center is located. Human operators in Utah are able to monitor the compressor's operating conditions and issue commands to the compressor over digital networks.

Both the Siemens (formerly Texas Instruments) 505 and Allen-Bradley (Rockwell) PLC-5 systems are considered "legacy" PLC systems by modern standards, the two systems in the previous photographs being over 20 years old each. It is not uncommon to find "obsolete" PLCs still in operation, though. Given their extremely rugged construction and reliable design, these control systems may continue to operate without significant trouble for decades.

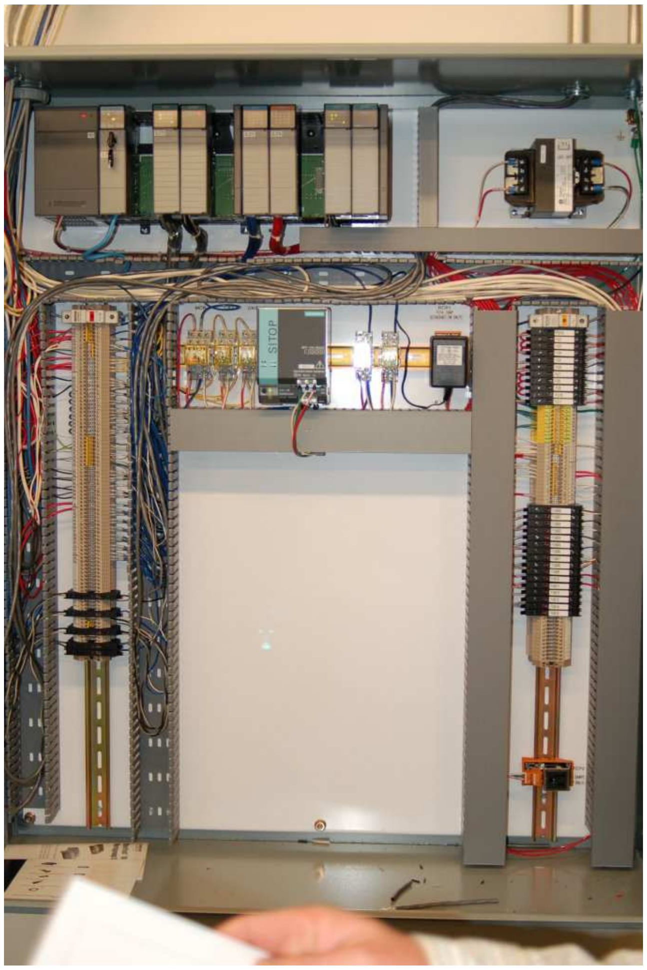
A later model of PLC manufactured by Allen-Bradley was the SLC 500 series (often verbally referred to as the "Slick 500"), also modular in design like the older PLC-5 system, although the racks and modules of the SLC 500 design were more compact than the PLC-5. The SLC 500 rack shown in the next photograph has 7 "slots" for processor and I/O cards to plug in to, numbered 0 through 6 (left to right):



The first three slots of this particular SLC 500 rack (0, 1, and 2) are occupied by the processor card, an analog input card, and a discrete input card, respectively. The slots 3 and 4 are empty (revealing the backplane circuit board and connectors for accepting new cards). The slots 5 and 6 hold discrete output and analog output cards, respectively.

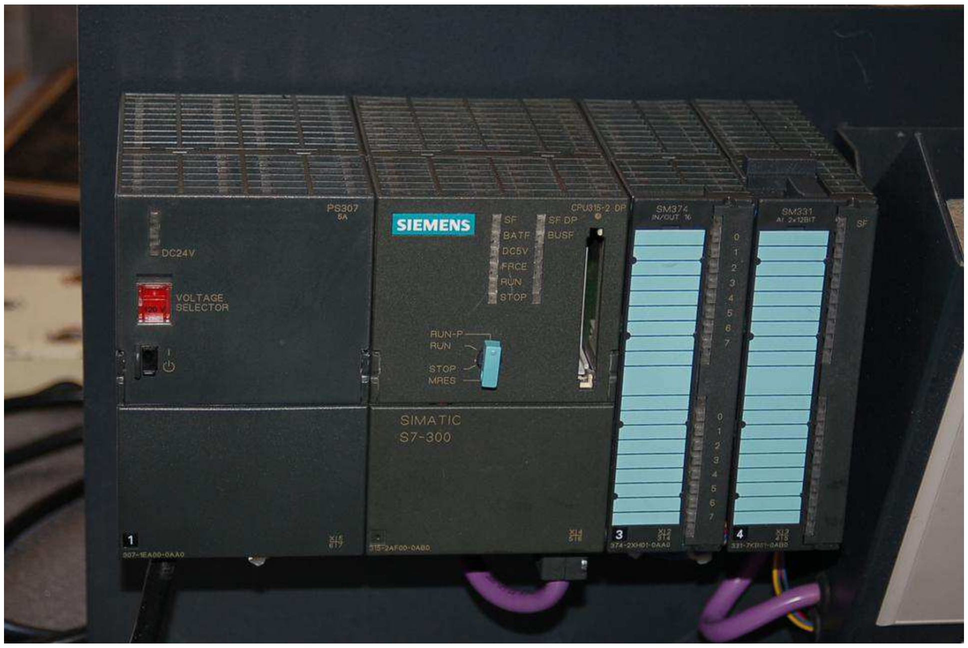
A feature visible on all cards in this system are numerous LED indicators, designed to show the status of each card. The processor card has LED indicators for "Run" mode, "Fault" conditions, "Force" conditions (when either input or output bits have been forced into certain states by the human programmer for testing purposes), and communication network indicators. Each discrete I/O card has indicator LEDs showing the on/off status of each I/O bit, and the analog card has a single LED showing that the card is powered.

A nine-slot SLC 500 system is shown in the next photograph, controlling a high-purity water treatment system for a biopharmaceuticals manufacturing facility. As you can see in this photograph, not all slots in this particular rack are occupied by I/O cards either:

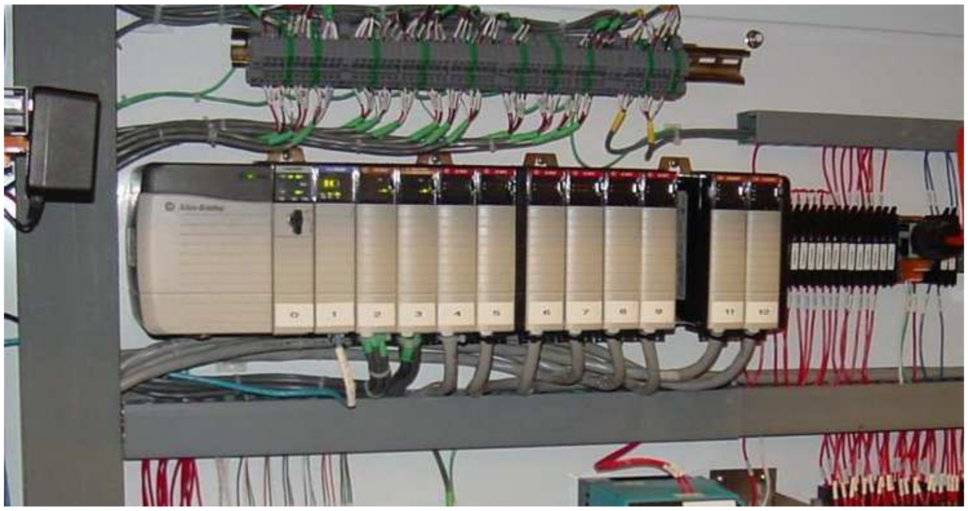


Some of the inputs to this PLC include water level switches, pressure switches, water flow meters, and conductivity meters (to measure the purity of the water, greater electrical conductivity indicating the presence of more dissolved minerals, which is undesirable in this particular process application). In turn, the PLC controls the starting and stopping of water pumps and the switching of water valves to manage the water purification and storage processes.

A modern PLC manufactured by Siemens appears in this next photograph, an S7-300, which is a different design of modular PLC. Instead of individual cards plugging into a rack, this modular PLC design uses individual modules plugging into each other on their sides to form a wider unit:



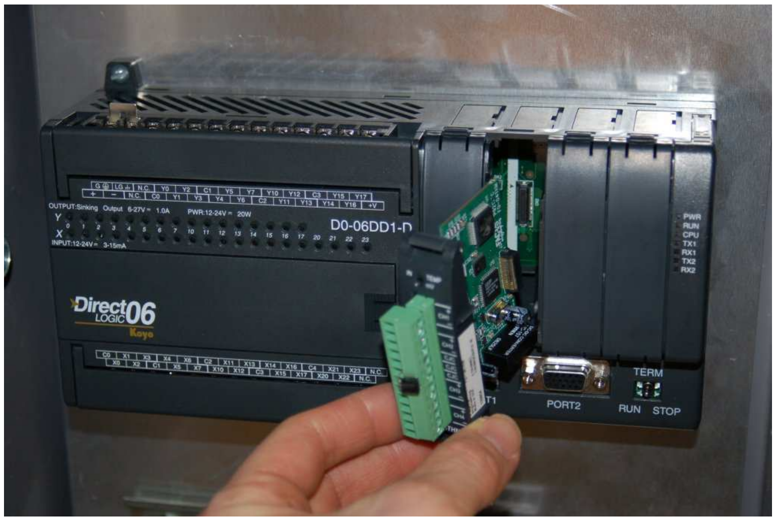
A modern PLC manufactured by Allen-Bradley (Rockwell) is this ControlLogix 5000 system, shown in this photograph used to control a cereal manufacturing process. The modular design of the ControlLogix 5000 system follows the more traditional scheme of individual cards plugged into a rack of fixed size:



While the Siemens S7 and Rockwell ControlLogix PLC platforms represent large-scale, modular PLC systems, there exist much smaller PLCs available for a fraction of the cost. Perhaps the least expensive PLC on the market at this time of writing is the Koyo "CLICK" PLC series, the processor module (with eight discrete input and six discrete output channels built in) shown in my hand (sold for 69 US dollars in the year 2010, and with free programming software!):

This is a semi-modular PLC design, with a minimum of input/output (I/O) channels built into the processor module, but having the capacity to accept multiple I/O modules plugged in to the side, much like the Siemens S7-300 PLC.

Other semi-modular PLCs expand using I/O cards that plug in to the base unit not unlike traditional rack-based PLC systems. The Koyo DirectLogic DL06 is a good example of this type of semi-modular PLC, the following photograph showing a model DL06 accepting a thermocouple input card in one of its four available card slots:



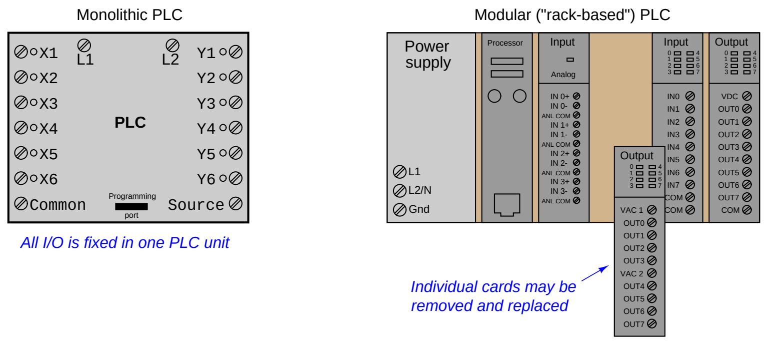
This photograph shows the PLC base unit with 20 discrete input channels and 16 discrete output channels, accepting an analog input card (this particular card is designed to input signals from thermocouples to measure up to four channels of temperature).

Some low-end PLCs are strictly monolithic, with no ability to accept additional I/O modules. This General Electric Series One PLC (used to monitor a small-scale hydroelectric power generating station) is an example of a purely monolithic design, having no "expansion" slots to accept I/O cards:

A disadvantage of monolithic PLC construction is that damaged I/O cannot be independently replaced. If an I/O channel on one of these PLCs becomes damaged, the entire PLC must be replaced to fix the problem. In a modular system, the damaged I/O card may simply be unplugged from the rack and replaced with a new I/O card. Another disadvantage of monolithic PLCs is the inherently fixed nature of the I/O: the end-user cannot customize the I/O configuration to match the application. For these reasons, monolithic PLCs are usually found on small-scale processes with few I/O channels and limited potential for expansion.

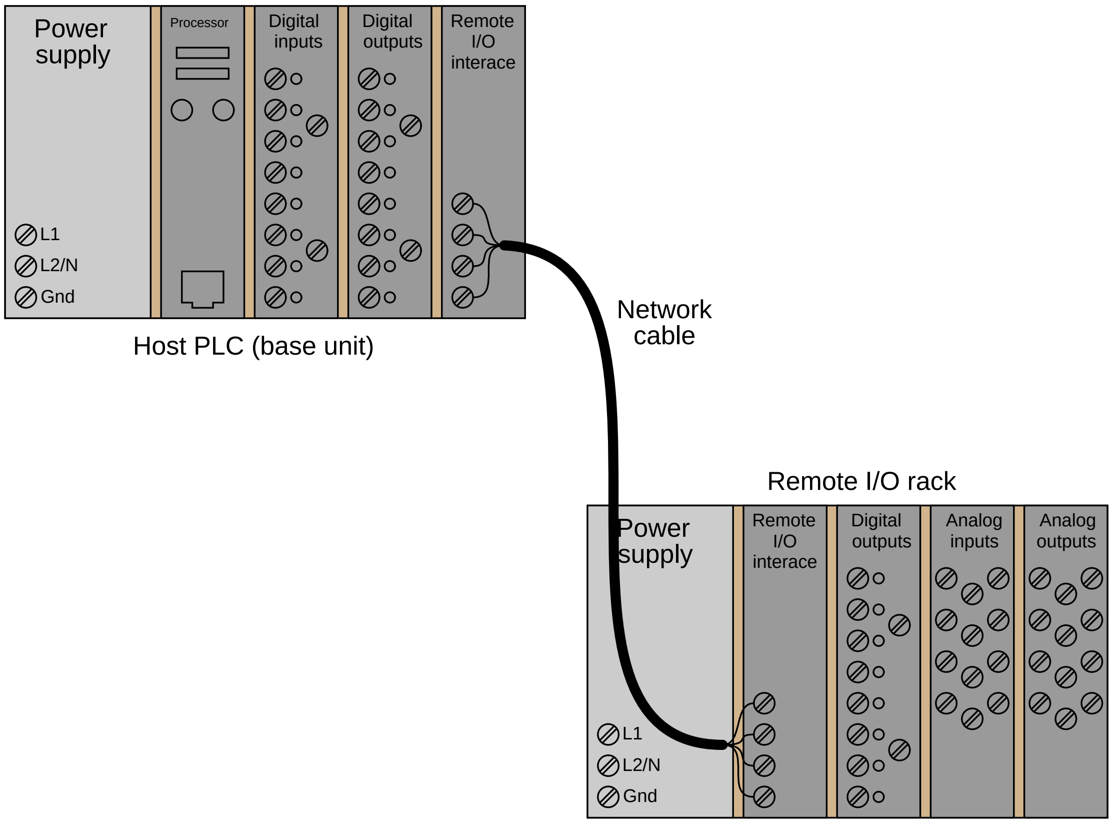
4.3 PLC hardware inputs and outputs (I/O)

Every programmable logic controller must have some means of receiving and interpreting signals from real-world sensors such as switches, and encoders, and also be able to effect control over real-world control elements such as solenoids, valves, and motors. This is generally known as *input/output*, or *I/O*, capability. Monolithic ("brick") PLCs have a fixed amount of I/O capability built into the unit, while modular ("rack") PLCs use individual circuit board "cards" to provide customized I/O capability.



The advantages of using replaceable I/O cards instead of a monolithic PLC design are numerous. First, and most obvious, is the fact that individual I/O cards may be easily replaced in the event of failure without having to replace the entire PLC. Specific I/O cards may be chosen for custom applications, biasing toward discrete cards for applications using many on/off inputs and outputs, or biasing toward analog cards for applications using many 4-20 mA and similar signals. Some PLCs even offer the feature of *hot-swappable* cards, meaning each card may be removed and a new one inserted without de-energizing power to the PLC processor and rack. Please note that one should not assume any system has hot-swappable cards, because if you attempt to change out a card "live" in a system without this feature, you run the risk of damaging the card and/or the rest of the unit it is plugged in to!

Some PLCs have the ability to connect to processor-less remote racks filled with additional I/O cards or modules, thus providing a way to increase the number of I/O channels beyond the capacity of the base unit. The connection from host PLC to remote I/O racks usually takes the form of a special digital network, which may span a great physical distance:

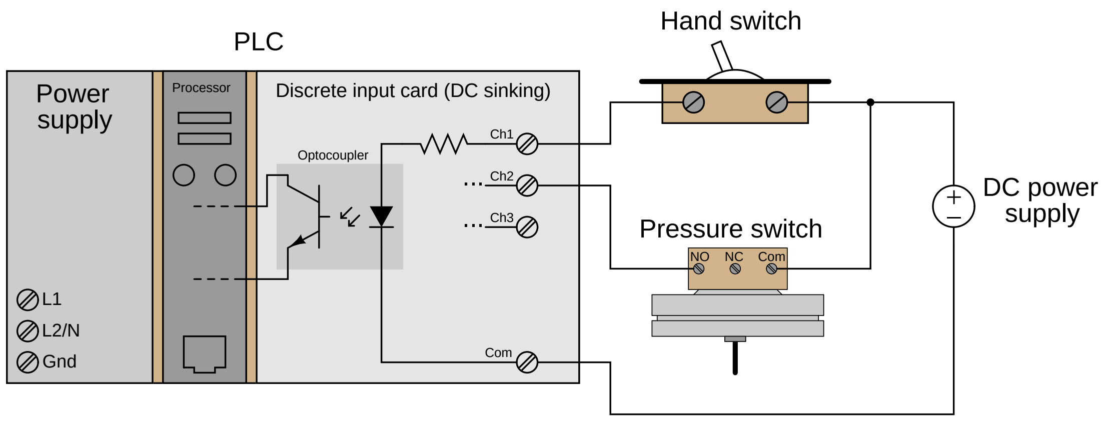


An alternative scheme for system expansion is to network multiple PLCs together, where each PLC has its own dedicated rack and processor. Through the use of communication instructions, one PLC may be programmed to read data from and/or write data to another PLC, effectively using the other PLC as an extension of its own I/O. Although this method is more expensive than remote I/O (where the remote racks lack their own dedicated processors), it provides the capability of stand-alone control in the event the network connection between PLC processors becomes severed.

Input/output capability for programmable logic controllers comes in three basic varieties: *discrete*, *analog*, and *network*; each type discussed in a following subsection.

4.3.1 Discrete I/O

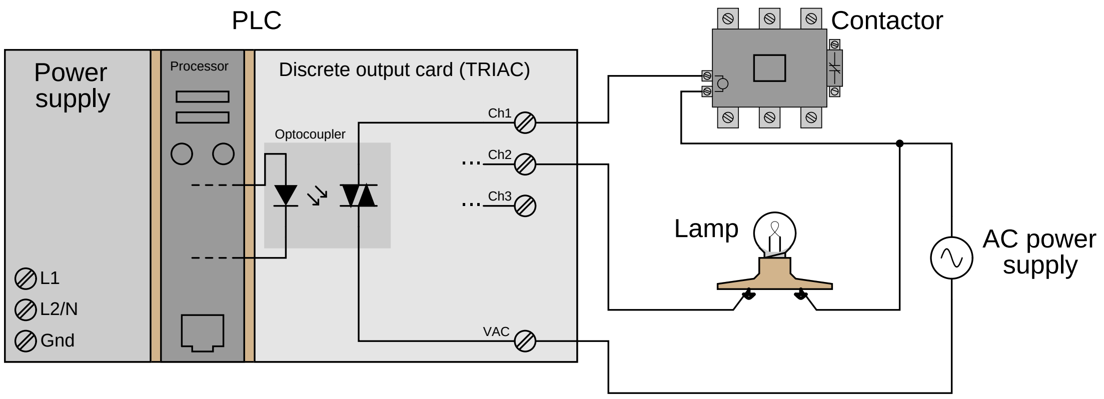
A "discrete" data point is one with only two states *on* and *off*. Process switches, pushbutton switches, limit switches, and proximity switches are all examples of discrete sensing devices. In order for a PLC to be aware of a discrete sensor's state, it must receive a signal from the sensor through a *discrete input* channel. Inside each discrete input module is (typically) a set of light-emitting diodes (LEDs) which will be energized when the corresponding sensing device turns on. Light from each LED shines on a photo-sensitive device such as a phototransistor inside the module, which in turn activates a *bit* (a single element of digital data) inside the PLC's memory. This opto-coupled arrangement makes each input channel of a PLC rather rugged, capable of isolating the sensitive computer circuitry of the PLC from transient voltage "spikes" and other electrical phenomena capable of causing damage:



Energizing an input channel lights the LED inside the optocoupler, turning on the phototransistor, sending a "high" signal to the PLC's microprocessor, setting (1) that bit in the PLC's input register.

The internal schematic diagram for a discrete input module ("card") shown above reveals the componentry typical for a single input channel on that card. Each input channel has its own optocoupler, writing to its own unique memory register bit inside the PLC's memory. Discrete input cards for PLCs typically have 4, 8, 16, or 32 channels.

Indicator lamps, solenoid valves, and motor starters (assemblies consisting of contactors and overload protection devices) are all examples of discrete control devices. In a manner similar to discrete inputs, a PLC connects to any number of different discrete final control devices through a *discrete output channel*[4]. Discrete output modules typically use the same form of opto-isolation to allow the PLC's computer circuitry to send electrical power to loads: the internal PLC circuitry driving an LED which then activates some form of photosensitive switching device.

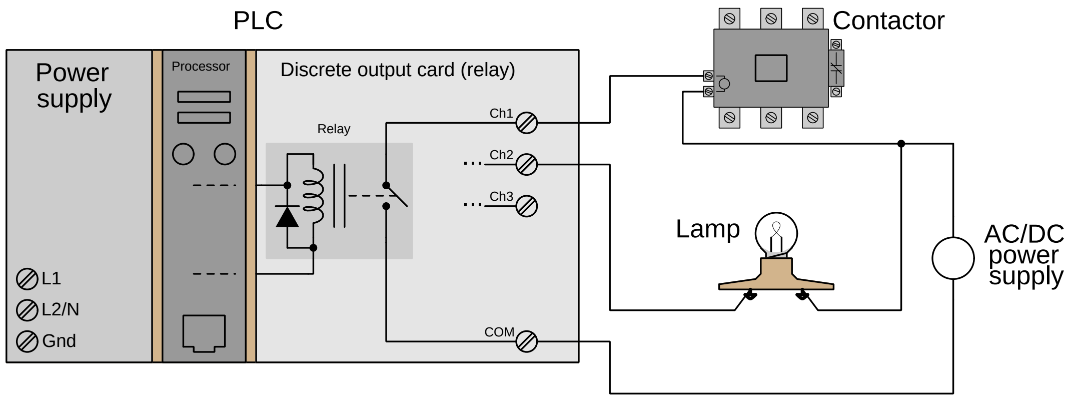


Setting a bit (1) in the PLC's output register sends a "high" signal to the LED inside the optocoupler, turning on the photo-TRIAC, sending AC power to the output channel to energize the load.

As with the schematic diagram for a discrete input module shown previously, the schematic diagram shown here for a discrete output module reveals the componentry typical for a single channel on that card. Each output channel has its own optocoupler, driven by its own unique memory register bit inside the PLC's memory. Discrete output cards for PLCs also typically have 4, 8, 16, or 32 channels.

[4]I/O "channels" are often referred to as "points" in industry lingo. Thus, a "32-point input card" refers to an input circuit with 32 separate channels through which to receive signals from on/off switches and sensors.

A common alternative to opto-isolated semiconductor switching elements such as transistors (DC) or TRIACs (AC) are miniature electromechanical relays. Mechanical relay contacts are capable of switching DC or AC, and provide similar levels of electrical isolation between the PLC's internal logic circuitry and external control circuits compared with opto-isolated transistors and TRIACs:



Setting a bit (1) in the PLC's output register sends a "high" signal to the relay coil, closing the relay contact, completing the circuit for power to energize the load.

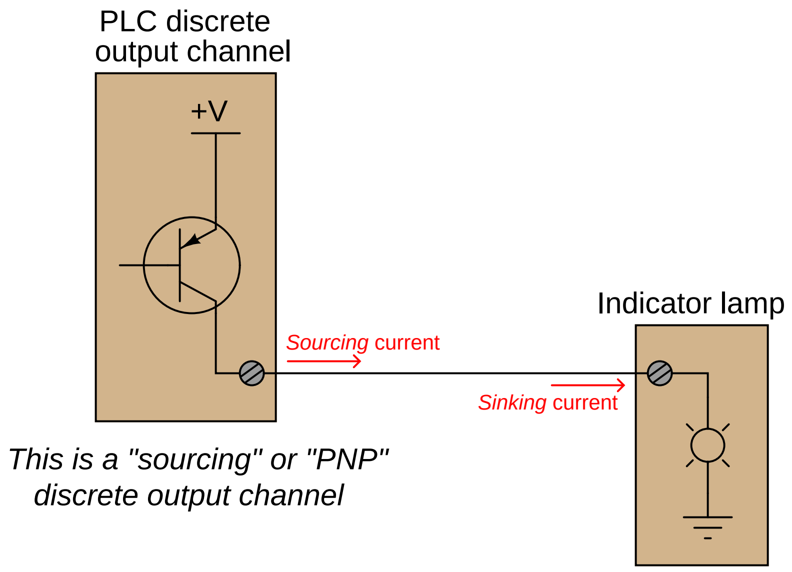
The relay-based discrete PLC output shown here happens to share a "Common" terminal with the other discrete output channels of the PLC. Isolated relay outputs, however, are a popular option. In an isolated relay output array, each relay contact has its own pair of dedicated screw terminals. This is useful when multiple loads requiring separate power supplies must be controlled by the same PLC.

Just like transistor and TRIAC discrete outputs, relay outputs typically allow the end-user to connect their power supply of choice to the switching element. Given that relay contacts are capable of DC and AC current switching, this provides a greater level of versatility than semiconductor elements such as transistors and TRIACs. Relays, of course, have their disadvantages as well. Relay contacts are much slower-responding than semiconductors which have no moving parts. The moving components of relays are subject to wear and failure in ways that semiconductor devices cannot fail. Relay contacts also "bounce" as they move, which may be problematic if the PLC output needs to drive the input of a high-speed counting device because each "bounce" will count as a separate contact closure event.

A term you will frequently encounter regarding PLC output relay contacts is *dry contact*. This simply means the relay contact inside the PLC's output channel has no pre-connected power source, and that the end-user must wire in their own (external) power source in order for the contact to send electrical power to any load. Most semiconductor-based discrete I/O channels are the same way, but interestingly the term "dry" is rarely used to describe semiconductor switching elements, only electromechanical relay contacts. By contrast, a *wetted contact* is internally connected to an electrical source, with terminals provided to directly connect to a load with no external power supply required.

An important concept to master when working with DC discrete I/O is the distinction between *current-sourcing* and *current-sinking* devices. The terms "sourcing" and "sinking" refer to the direction of current (as denoted by conventional flow notation) into or out of a device's control wire[5]. A device sending (conventional flow) current out of its control terminal to some other device(s) is said to be *sourcing* current, while a device accepting (conventional flow) current into its control terminal is said to be *sinking* current.

To illustrate, the following illustration shows a PLC output channel is *sourcing* current to an indicator lamp, which is *sinking* current to ground:

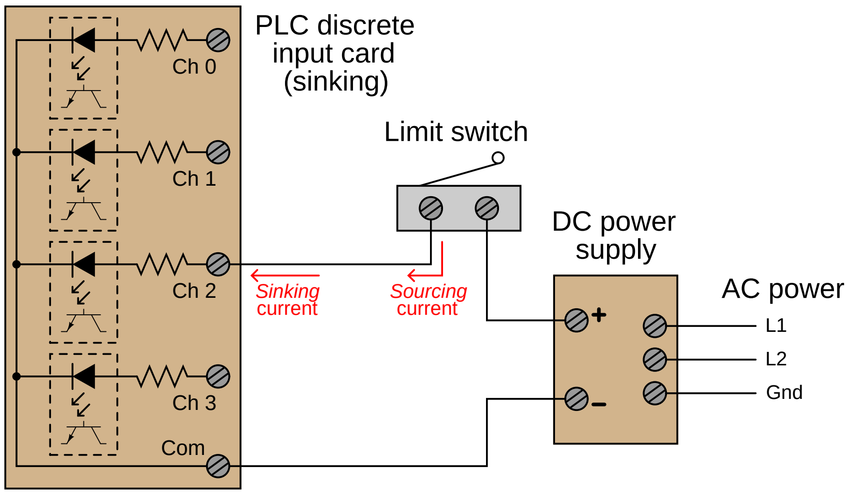

PLC discrete output channel
+V
Sourcing current
Indicator lamp
Sinking current
This is a "sourcing" or "PNP" discrete output channel

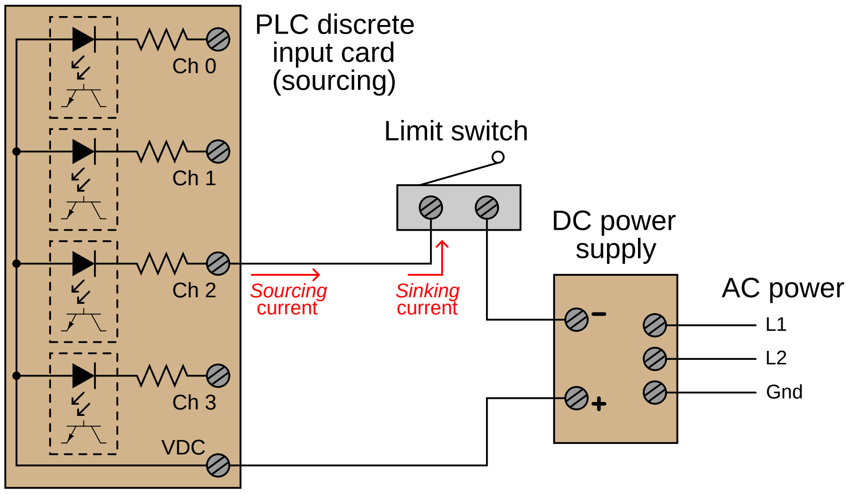
These terms really only make sense when electric current is viewed from the perspective of conventional flow, where the positive terminal of the DC power supply is envisioned to be the "source" of the current, with current finding its way "down" to ground (the negative terminal of the DC power supply). In every circuit formed by the output channel of a PLC driving a discrete control device, or by a discrete sensing device driving an input channel on a PLC, one element in the circuit must be sourcing current while the other is sinking current.

An engineering colleague of mine has a charming way to describe sourcing and sinking: *blowing* and *sucking*. A device that sources current to another "blows" current toward the other device. A device that sinks current "sucks" current from the other device. Many students seem to find these terms helpful in first mastering the distinction between sourcing and sinking despite (or perhaps because of!) their informal nature.

[5]By "control wire," I mean the single conductor connecting the I/O card channel to the field device, as opposed to conductors directly common with either the positive or negative lead of the voltage source. If you focus your attention on this one wire, noting the direction of conventional-flow current through it, the task of determining whether a device is sourcing or sinking current becomes much simpler.

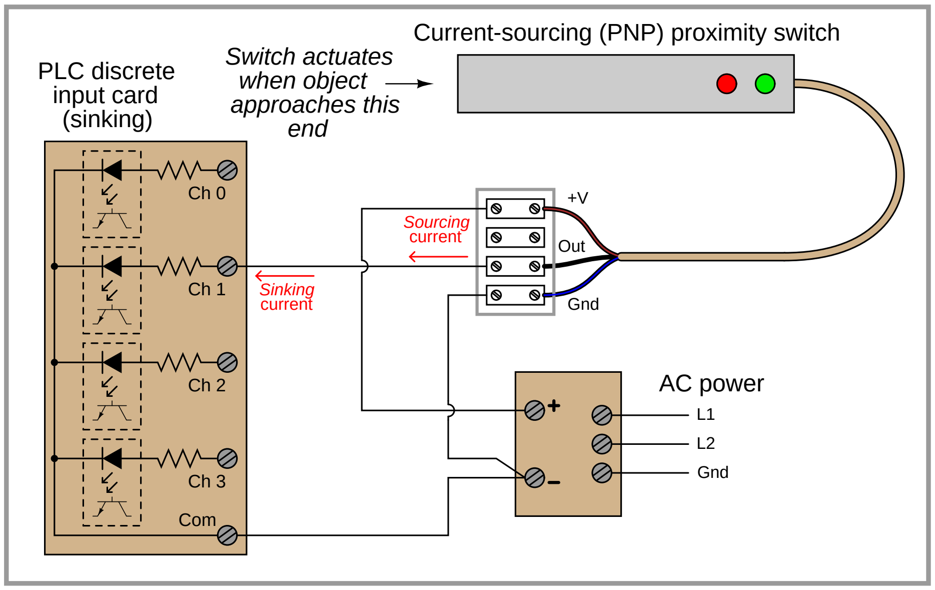
If the discrete device connecting to the PLC is not polarity-sensitive, either type of PLC I/O module will suffice. For example, the following diagrams show a mechanical limit switch connecting to a sinking PLC input and to a sourcing PLC input:

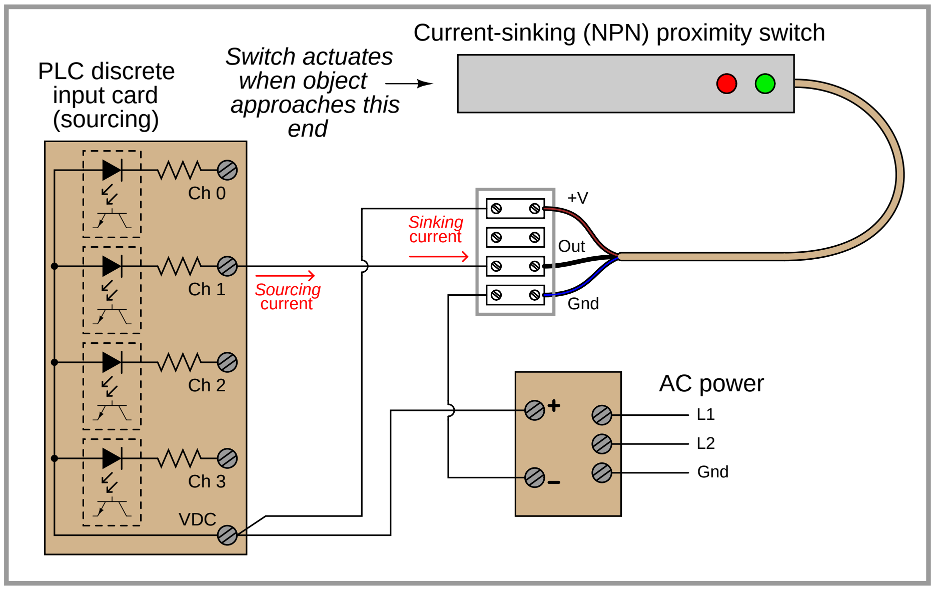




Note the differences in polarity and labeling between the sinking card's common terminal and the sourcing card's common terminal. On the "sinking" card, the input channel terminal is positive while the common ("Com") terminal is negative. On the "sourcing" card, the input channel terminal is negative while the common ("VDC") terminal is positive.

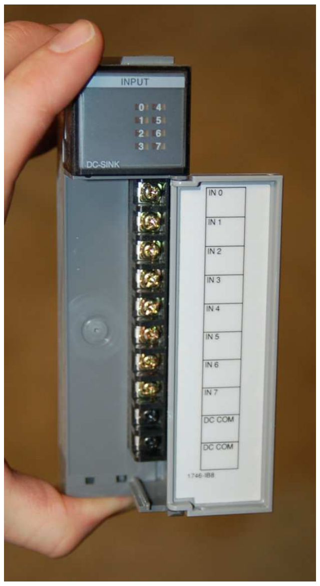
Some discrete sensing devices *are* polarity-sensitive, such as electronic proximity sensors containing transistor outputs. A "sourcing" proximity switch can only interface with a "sinking" PLC input channel, and vice-versa:


Current-sourcing (PNP) proximity switch
PLC discrete input card (sinking)
Switch actuates when object approaches this end
Ch 0
Ch 1
Ch 2
Ch 3
Com
+V
Out
Gnd
Sourcing current
Sinking current
AC power
L1
L2
Gnd


Current-sinking (NPN) proximity switch
PLC discrete input card (sourcing)
Switch actuates when object approaches this end
Ch 0
Ch 1
Ch 2
Ch 3
VDC
+V
Out
Gnd
Sinking current
Sourcing current
AC power
L1
L2
Gnd

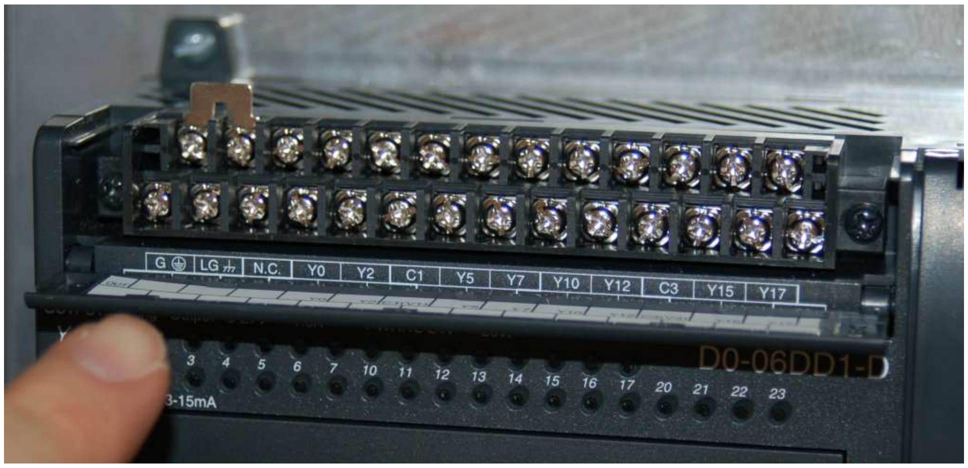
In all cases, the "sourcing" device sends current *out of* its signal terminal while the "sinking" device takes current *into* its signal terminal.

Two photographs of a DC (sinking) discrete input module for an Allen-Bradley model SLC 500 PLC are shown here: one with the plastic cover closed over the connection terminals, and the other with the plastic cover opened up for viewing the terminals. A legend on the inside of the cover shows the purpose of each screw terminal: eight input channels (numbered 0 through 7) and two redundant "DC Com" terminals for the negative pole of the DC power supply to connect:



A standard feature found on practically every PLC discrete I/O module is a set of LED indicators visually indicating the status of each bit (discrete channel). On the SLC 500 module, the LEDs appear as a cluster of eight numbered squares near the top of the module.

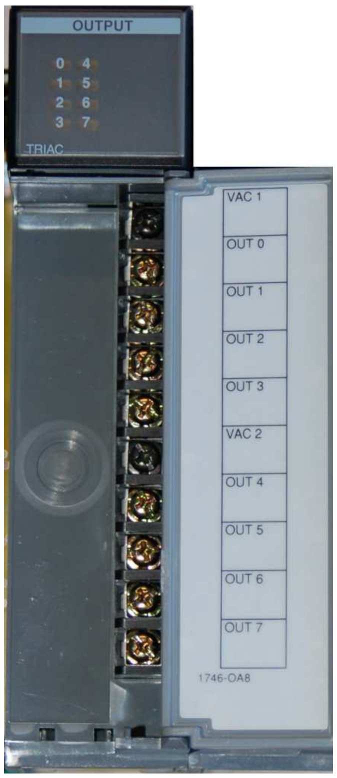
A photograph of discrete output terminals on another brand of PLC (a Koyo model DL06) shows somewhat different labeling:



Here, each output channel terminal is designated with a letter/number code beginning with the letter "Y". Several "common" terminals labeled with "C" codes service clusters of output channels. In this particular case, each "common" terminal is common only to four output channels. With sixteen total output channels on this PLC, this means there are four different "common" terminals. While this may seem somewhat strange (why not just have one "common" terminal for all sixteen output channels?), it more readily permits different DC power supplies to service different sets of output channels.

Electrical polarity is not an issue with AC discrete I/O, since the polarity of AC reverses periodically anyway. However, there is still the matter of whether the "common" terminal on a discrete PLC module will connect to the *neutral* (grounded) or *hot* (ungrounded) AC power conductor.

The next photograph shows a discrete AC output module for an Allen-Bradley SLC 500 PLC, using TRIACs as power switching devices rather than transistors as is customary with DC discrete output modules:



This particular eight-channel module provides two sets of TRIACs for switching power to AC loads, each set of four TRIACs receiving AC power from a "hot" terminal (VAC 1 or VAC 2), the other side of the load device being connected to the "neutral" (grounded) conductor of the AC power source.

Fortunately, the hardware reference manual supplied by the manufacturer of every PLC shows diagrams illustrating how to connect discrete input and output channels to field devices. One should always consult these diagrams before connecting devices to the I/O points of a PLC!

4.3.2 Analog I/O

In the early days of programmable logic controllers, processor speed and memory were too limited to support anything but discrete (on/off) control functions. Consequently, the only I/O capability found on early PLCs were discrete in nature[6]. Modern PLC technology, though, is powerful enough to support the measurement, processing, and output of analog (continuously variable) signals.

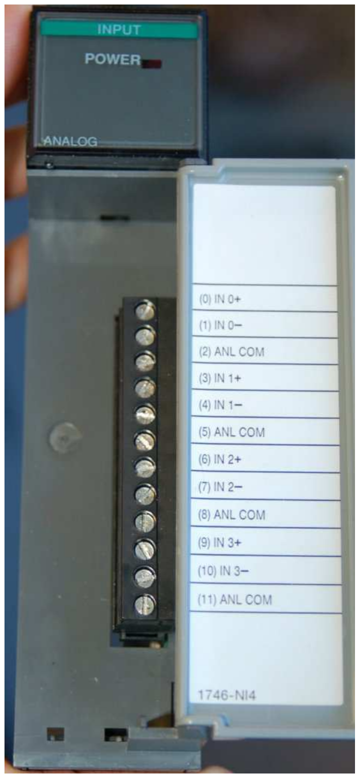
All PLCs are digital devices at heart. Thus, in order to interface with an analog sensor or control device, some "translation" is necessary between the analog and digital worlds. Inside every analog input module is an *ADC*, or *Analog-to-Digital Converter*, circuit designed to convert an analog electrical signal into a multi-bit binary word. Conversely, every analog output module contains a *DAC*, or *Digital-to-Analog Converter*, circuit to convert the PLC's digital command words into analog electrical quantities.

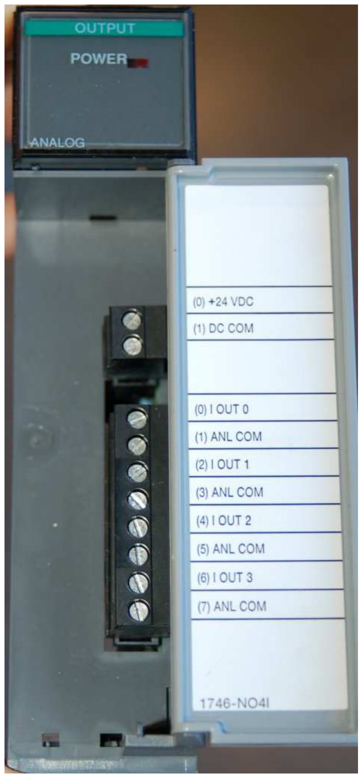
Analog I/O is commonly available for modular PLCs for many different analog signal types, including:

- Voltage (0 to 10 volt, 0 to 5 volt)
- Current (0 to 20 mA, 4 to 20 mA)
- Thermocouple (millivoltage)
- RTD (millivoltage)
- Strain gauge (millivoltage)

[6]Some modern PLCs such as the Koyo "CLICK" are also discrete-only. Analog I/O and processing is significantly more complex to engineer and more expensive to manufacture than discrete control, and so low-end PLCs are more likely to lack analog capability.

The following photographs show two analog I/O cards for an Allen-Bradley SLC 500 modular PLC system, an analog input card and an analog output card. Labels on the terminal cover doors indicate screw terminal assignments:





4.3.3 Network I/O

Many different digital network standards exist for PLCs to communicate with, from PLC to PLC and between PLCs and field devices. One of the earliest digital protocols developed for PLC communication was *Modbus*, originally for the Modicon brand of PLC. Modbus was adopted by other PLC and industrial device manufacturers as a *de facto* standard[7], and remains perhaps the most universal digital protocol available for industrial digital devices today.

Another digital network standard developed by a particular manufacturer and later adopted as a *de facto* standard is *Profibus*, originally developed by Siemens.

[7]A "de facto" standard is one arising naturally out of legacy rather than by an premeditated agreement between parties. Modbus and Profibus networks are considered "de facto" standards because those networks were designed, built, and marketed by pioneering firms prior to their acceptance as standards for others to conform to. In Latin, *de facto* means "from the fact," which in this case refers to the fact of pre-existence: a standard agreed upon to conform to something already in popular use. By contrast, a standard intentionally agreed upon before its physical realization is a *de jure* standard (Latin for "from the law"). FOUNDATION Fieldbus is an example of a *de jure* standard, where a committee arrives at a consensus for a network design and specifications prior to that network being built and marketed by any firm.

4.4 IEC 61131-3 programming languages

Although it seems each model of PLC has its own idiosyncratic standard for programming, there does exist an international standard for controller programming that most PLC manufacturers at least attempt to conform to. This is the IEC 61131-3 standard, which will be the standard presented in this module.

One should take solace in the fact that despite differences in the details of PLC programming from one manufacturer to another and from one model to another, the basic principles are largely the same. There exist much greater disparities between different general-purpose programming languages (e.g. C/C++, BASIC, FORTRAN, Pascal, Java, Ada, etc.) than between the programming languages supported by different PLCs, and this fact does not prevent computer programmers from being "multilingual." I have personally written and/or analyzed programs for over a half-dozen different manufacturers of PLCs (Allen-Bradley, Siemens, Square D, Koyo, Fanuc, Moore Products APACS and QUADLOG, and Modicon), with multiple PLC models within most of those brands, and I can tell you the differences in programming conventions are largely insignificant. After learning how to program one model of PLC, it is quite easy to adapt to programming other makes and models of PLC. If you are learning to program a particular PLC that does not exactly conform to the IEC 61131-3 standard, you will still be able to apply every single principle discussed in this tutorial – the fundamental concepts are truly that universal.

The IEC 61131-3 standard specifies five distinct forms of programming language for industrial controllers:

- Ladder Diagram (LD)
- Structured Text (ST)
- Instruction List (IL)
- Function Block Diagram (FBD)
- Sequential Function Chart (SFC)

Not all programmable logic controllers support all five language types, but nearly all of them support Ladder Diagram (LD), which will be the primary focus of this book.

Programming languages for many industrial devices are limited by design. One reason for this is *simplicity*: any programming language simple enough in structure for someone with no formal computer programming knowledge to understand is necessarily going to be limited in its capabilities. Another reason for programming limitations is *safety*: the more flexible and unbounded a programming language is, the more potential there will be to unintentionally create complicated "run-time" errors when programming. The ISA[8] safety standard number 84 classifies industrial programming languages as either *Fixed Programming Languages* (FPL), *Limited Variability Languages* (LVL), or *Full Variability Languages* (FVL). Ladder Diagram and Function Block Diagram programming are both considered to be "limited variability" languages, whereas Instruction List (and traditional computer programming languages such as C/C++, FORTRAN, BASIC, etc.) are considered "full variability" languages with all the attendant potential for complex errors.

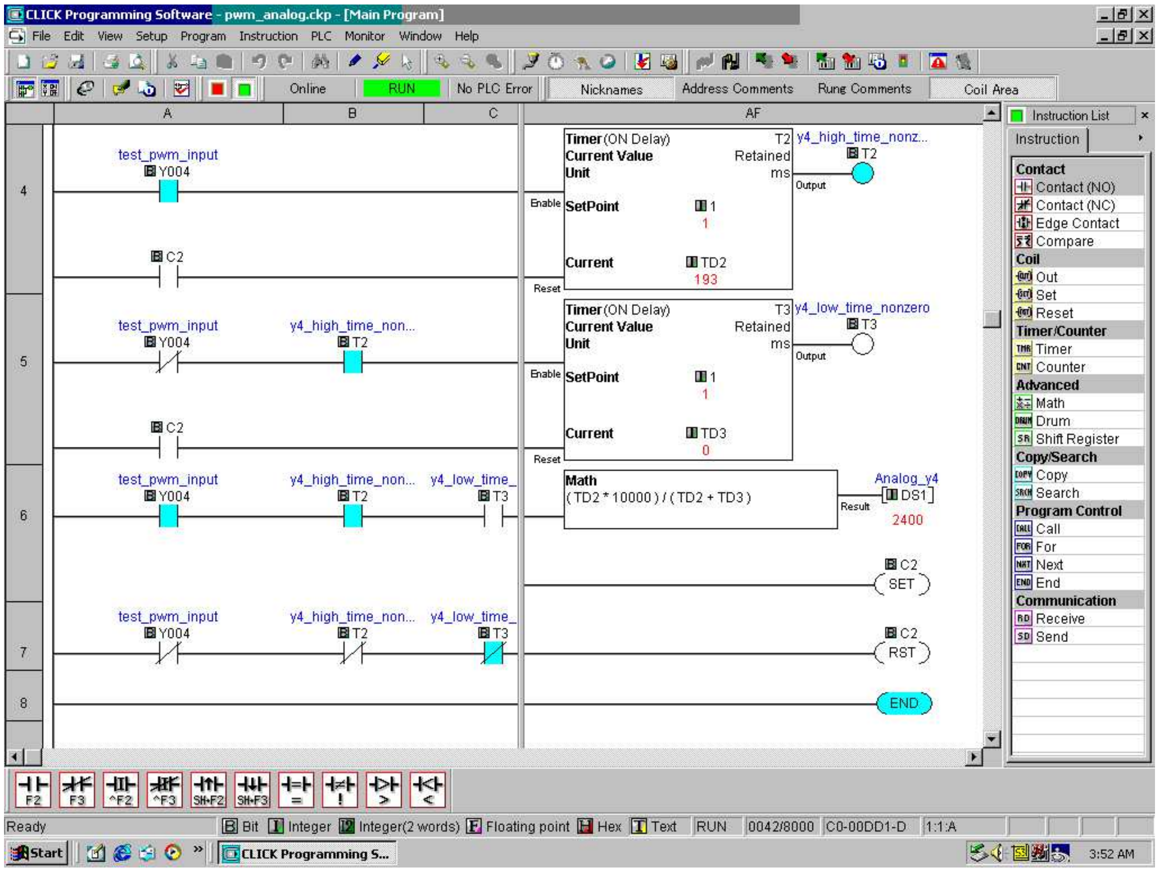
[8]The ISA is the *International Society of Automation, concerned mostly with establishing standards for industrial automation systems.*

4.5 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 inter-relationships between those bits expressed in the form of a diagram *resembling* a circuit.

The following computer screenshot shows a typical Ladder Diagram program[9] 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

[9]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!

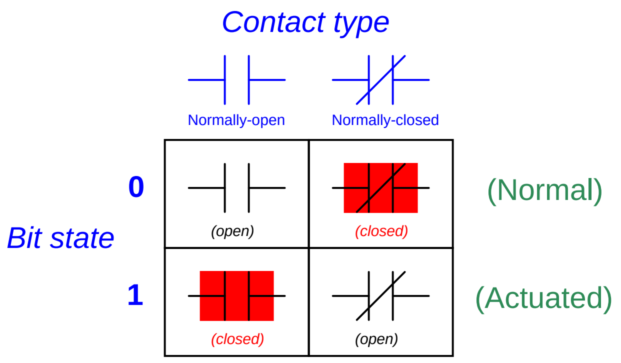
4.5.1 Relating I/O status to virtual elements

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*[10] 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*.

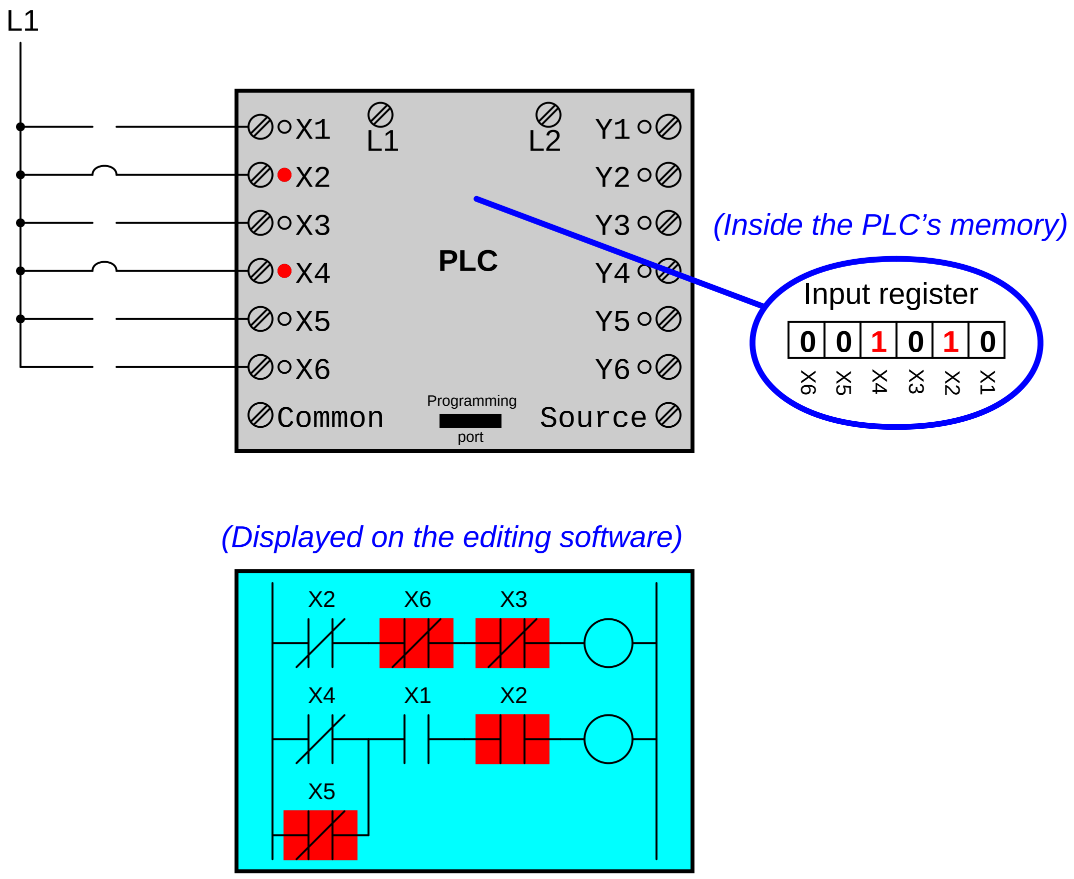
The following table shows how the two types of contacts in a PLC's Ladder Diagram program respond to bit states, using red coloring to signify each contact's virtual conductivity:



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.

[10]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[11] 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:



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.

[11]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:

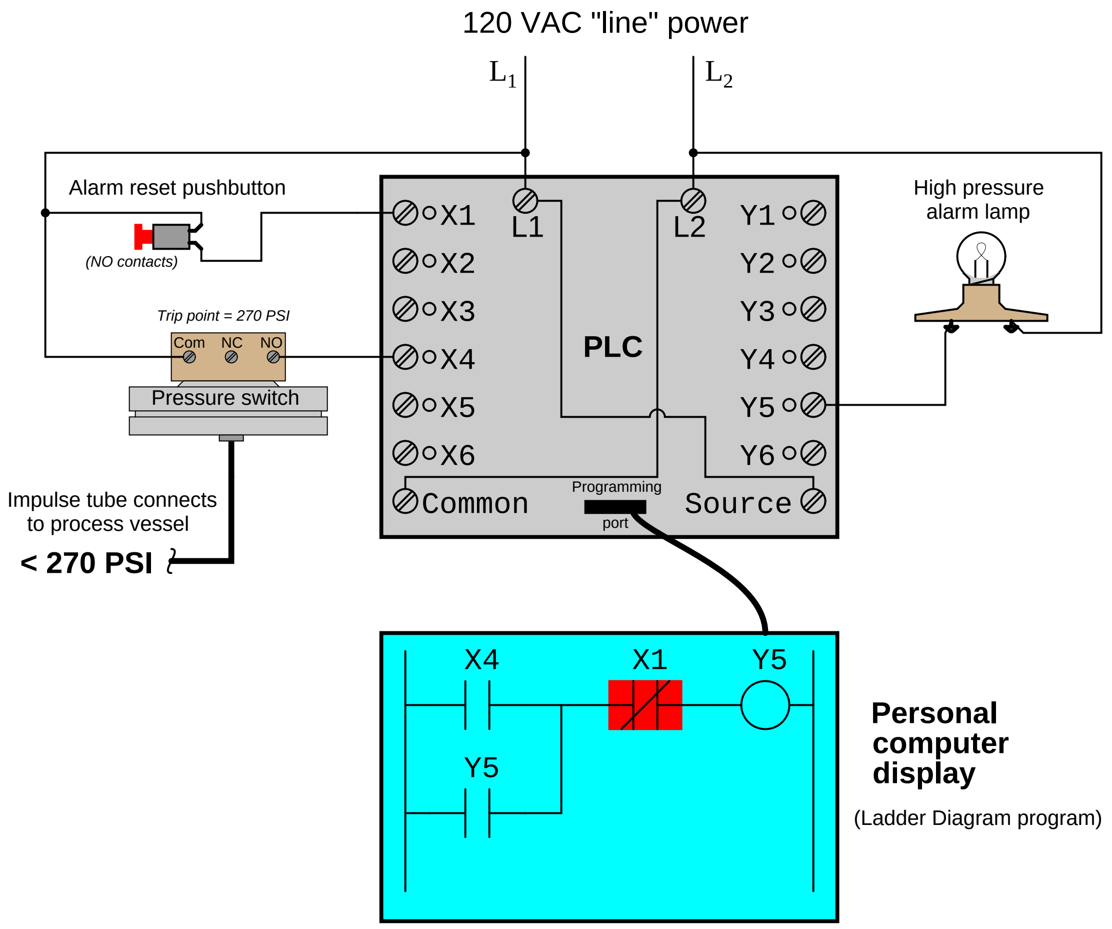
- **Each input bit in the PLC's memory will be a "1" when its input channel is powered, and will be a "0" when its input channel is unpowered**
- **Each virtual contact shown in the program *actuates* whenever it reads a "1" state in its respective bit, and will be *at rest* whenever it reads a "0" state 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.

To further illuminate these fundamental concepts, we will examine the operation of a simple 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, operators will be alerted to both *past* and *present* process vessel overpressure events.

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:

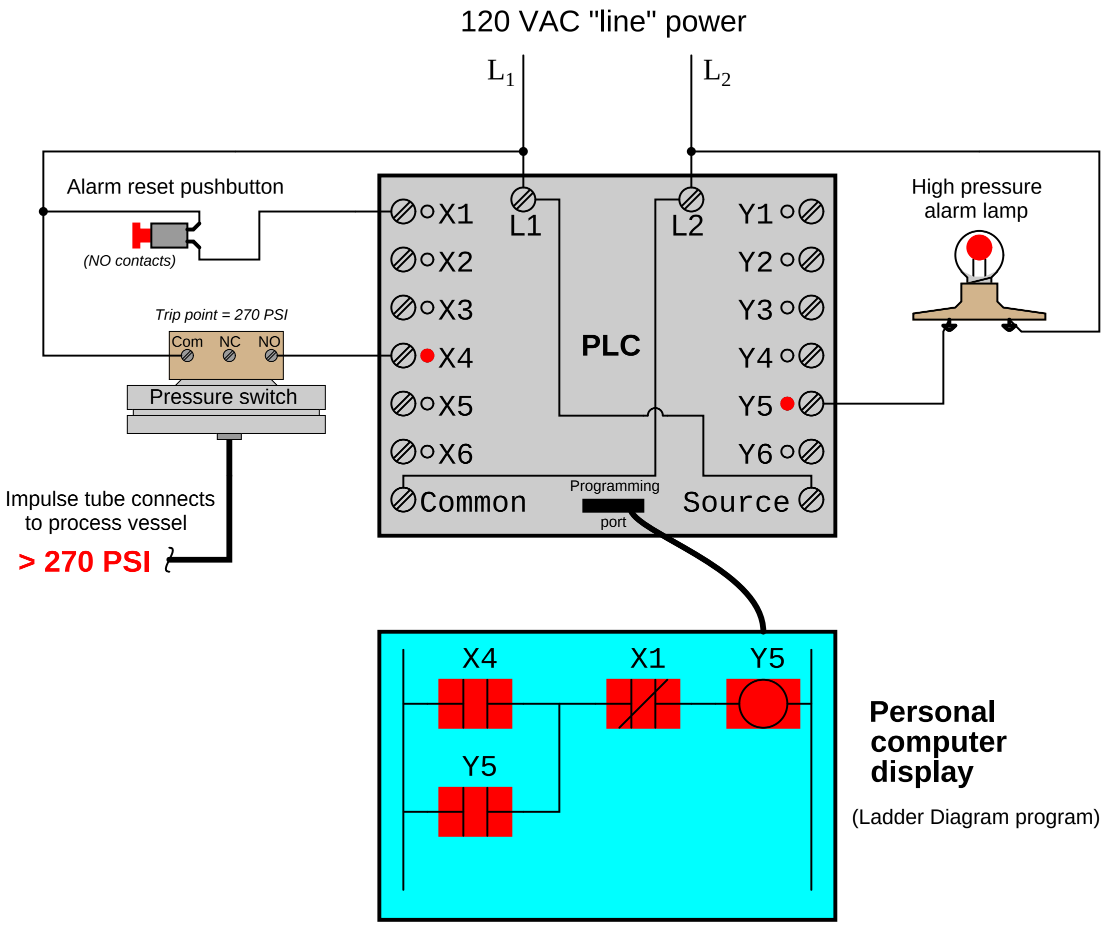


The reset pushbutton connects to discrete input X1 of the PLC, while the pressure switch connects to discrete input X4. The warning lamp connects to discrete output Y5. Red indicator LEDs next to each I/O terminal visually indicate the electrical status of the I/O points, while red-shaded highlighting shows the *virtual power*[12] 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.

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.

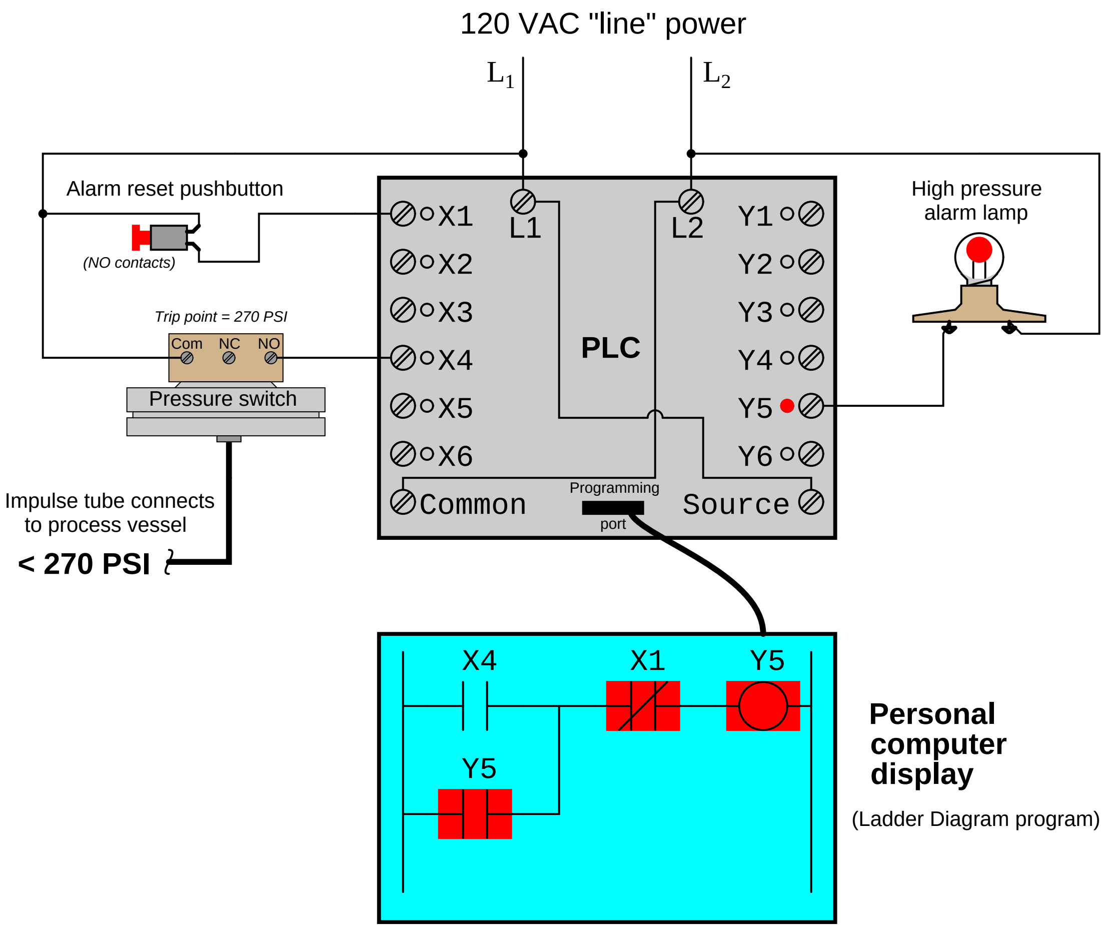
[12]For a PLC program contact, the shading represents virtual "conductivity." For a PLC program coil, the shading represents a set (1) bit.

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`[13] and also energizes the real discrete output `Y5` to energize the warning lamp:



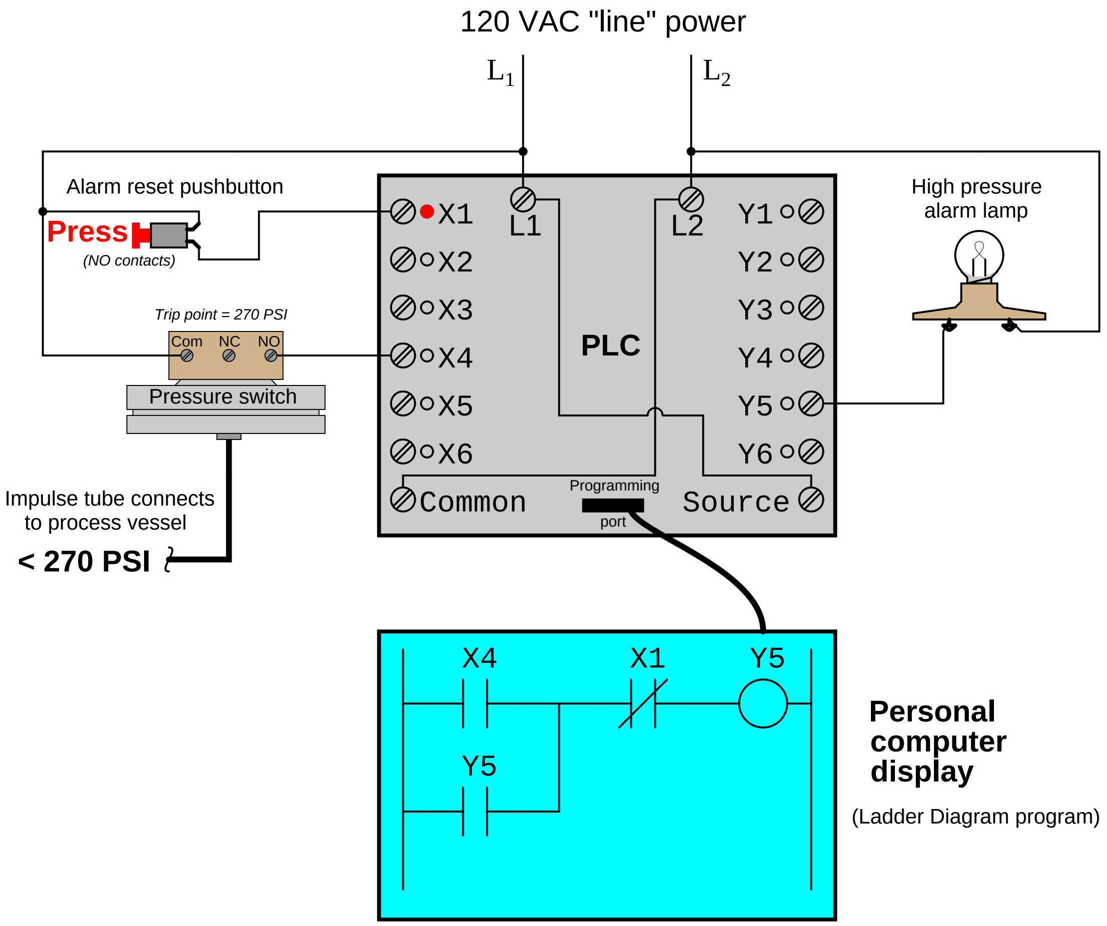
[13]It is worth noting the legitimacy of referencing virtual contacts to output bits (e.g. contact `Y5`), and not just to input bits. A "virtual contact" inside a PLC program is nothing more than an instruction to the PLC's processor to *read* the status of a bit in memory. It matters not whether that 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:



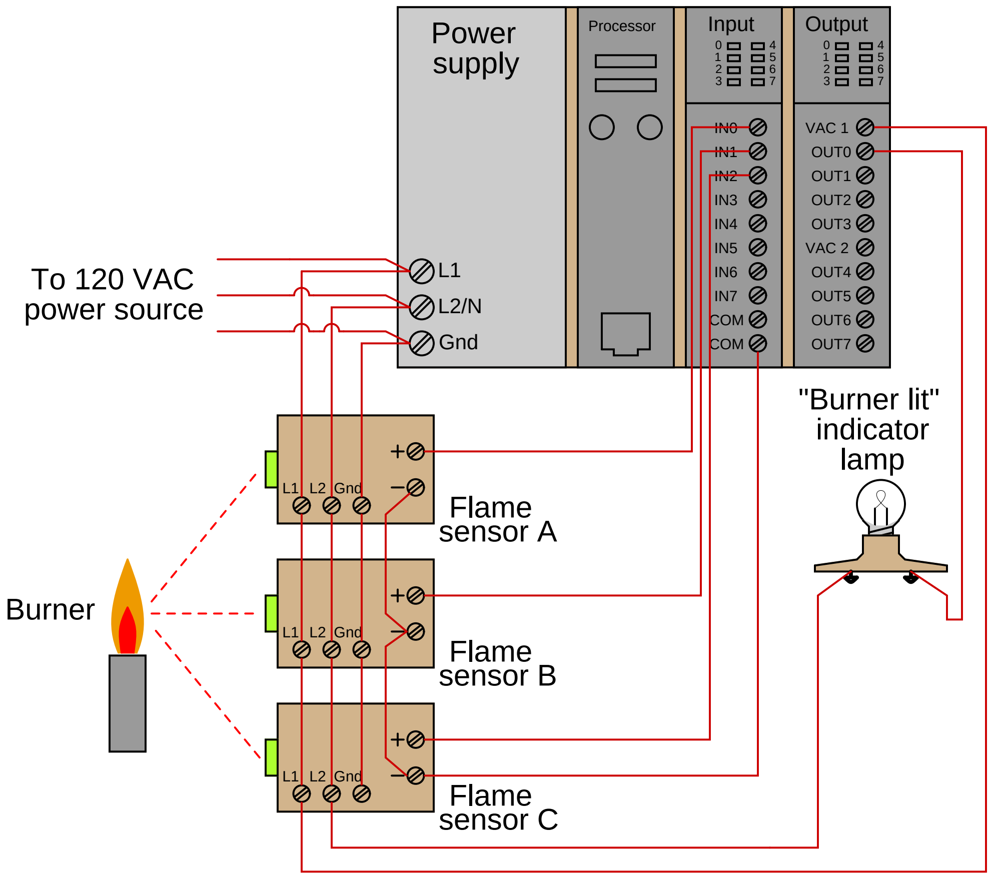
4.5.2 Ladder diagram contacts and coils

The most elementary objects in Ladder Diagram programming are *contacts* and *coils*, intended to mimic the contacts and coils of electromechanical relays. Contacts and coils are *discrete* programming elements, dealing with Boolean (1 and 0; on and off; true and false) variable states. Each contact in a Ladder Diagram PLC program represents the *reading* of a single bit in memory, while each coil represents the *writing* of a single bit in memory.

Discrete input signals to the PLC from real-world switches are read by a Ladder Diagram program by contacts referenced to those input channels. In legacy PLC systems, each discrete input channel has a specific address which must be applied to the contact(s) within that program. In modern PLC systems, each discrete input channel has a tag name created by the programmer which is applied to the contact(s) within the program. Similarly, discrete output channels – referenced by coil symbols in the Ladder Diagram – must also bear some form of address or tag name label.

To illustrate, we will imagine the construction and programming of a redundant flame-sensing system to monitor the status of a burner flame using three sensors. 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.

Our system's wiring is shown in the following diagram:

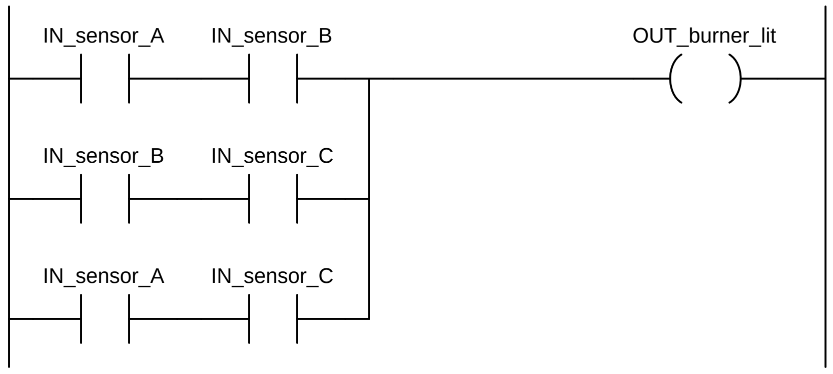


Each flame sensor outputs a DC voltage signal indicating the detection of flame at the burner, either on (24 Volts DC) or off (0 Volts DC). These three discrete DC voltage signals are sensed by the first three channels of the PLC's discrete input card. The indicator lamp is a 120 volt light bulb, and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

To make the ladder 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[14]. 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`.

[14]If this were a legacy Allen-Bradley PLC system using absolute addressing, we would be forced to address the three sensor inputs as `I:1/0`, `I:1/1`, and `I:1/2` (slot 1, channels 0 through 2), and the indicator lamp output as `O:2/0` (slot 2, channel 0). If this were a newer Logix5000 Allen-Bradley PLC, the default tag names would be `Local:1:I.Data.0`, `Local:1:I.Data.1`, and `Local:1:I.Data.2` for the three inputs, and `Local:2:O.Data.0` for the output. However, in either system we have the ability to assign symbolic addresses so we have a way to reference the I/O channels without having to rely on these cumbersome labels. The programs showing in this book exclusively use tag names rather than absolute addresses, since this is the more modern programming convention.

A ladder program to determine if at least two out of the three sensors detect flame is shown here, with the tag names referencing each contact and coil:



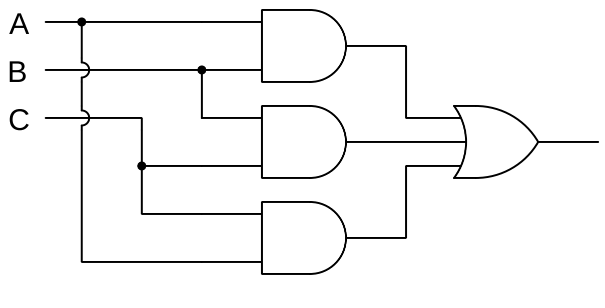
Series-connected contacts in a Ladder Diagram perform the logical `AND` function, while parallel contacts perform the logical `OR` function. Thus, this two-out-of-three flame-sensing program could be verbally described as:

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

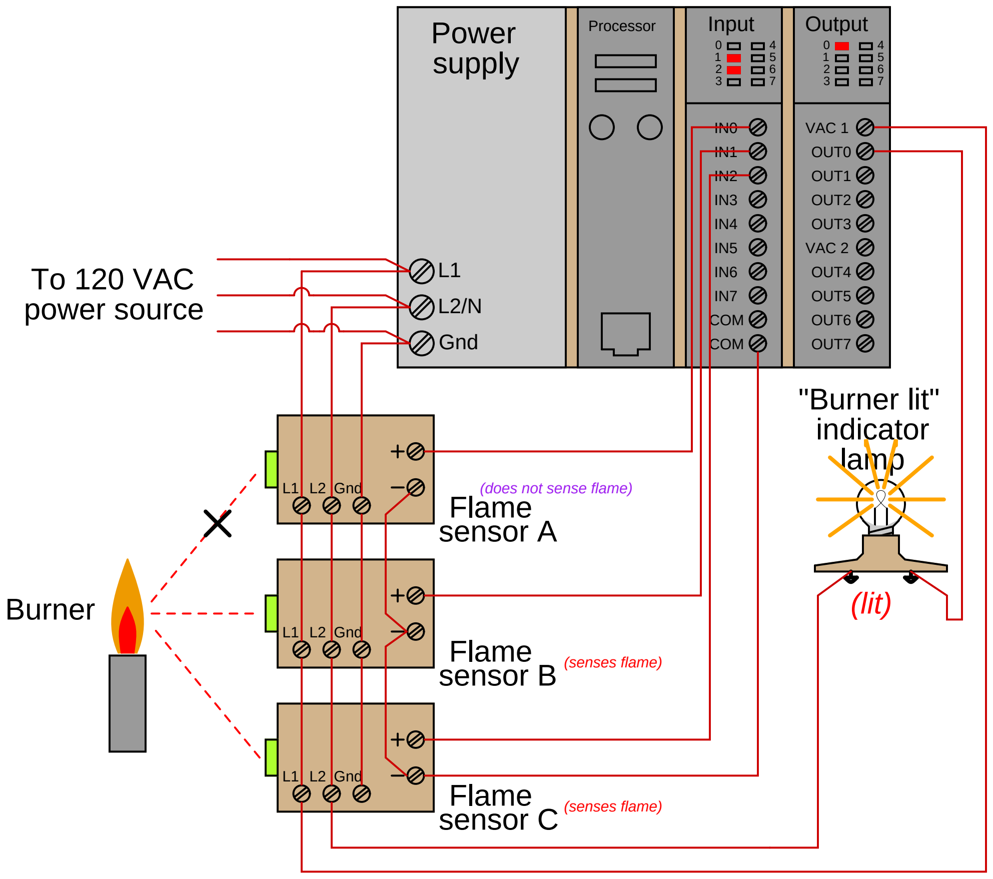
An alternate way to express this is to use the notation of *Boolean algebra*, where multiplication represents the `AND` function and addition represents the `OR` function:

$$\text{Burner\_lit} = AB + BC + AC$$

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



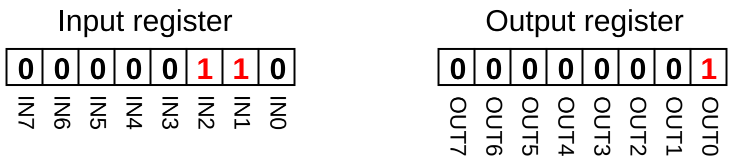
To illustrate how this program would work, we will consider a case where flame sensors B and C detect flame, but sensor A does not[15]. This represents a two-out-of-three-good condition, and so we would expect the PLC to turn on the "Burner lit" indicator light as programmed. From the perspective of 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:



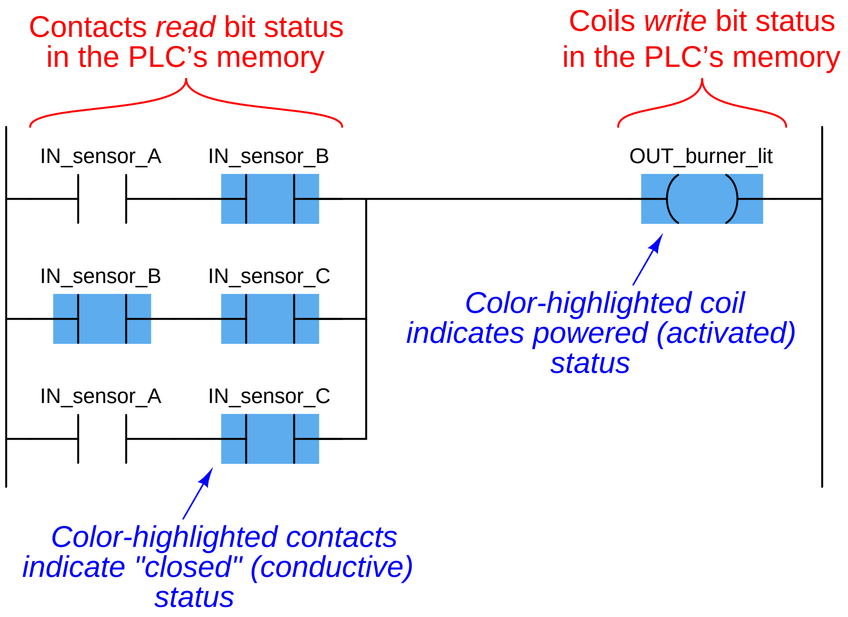
Those two energized input channels "set" bits (1 status) in the PLC's memory representing the status of flame sensors B and C. Flame sensor A's bit will be "clear" (0 status) because its corresponding input channel is de-energized. The fact that the output channel LED is energized (and the "Burner lit"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

[15]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:


Input register
0 0 0 0 0 1 1 0
IN7 IN6 IN5 IN4 IN3 IN2 IN1 IN0
Output register
0 0 0 0 0 0 0 1
OUT7 OUT6 OUT5 OUT4 OUT3 OUT2 OUT1 OUT0

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:

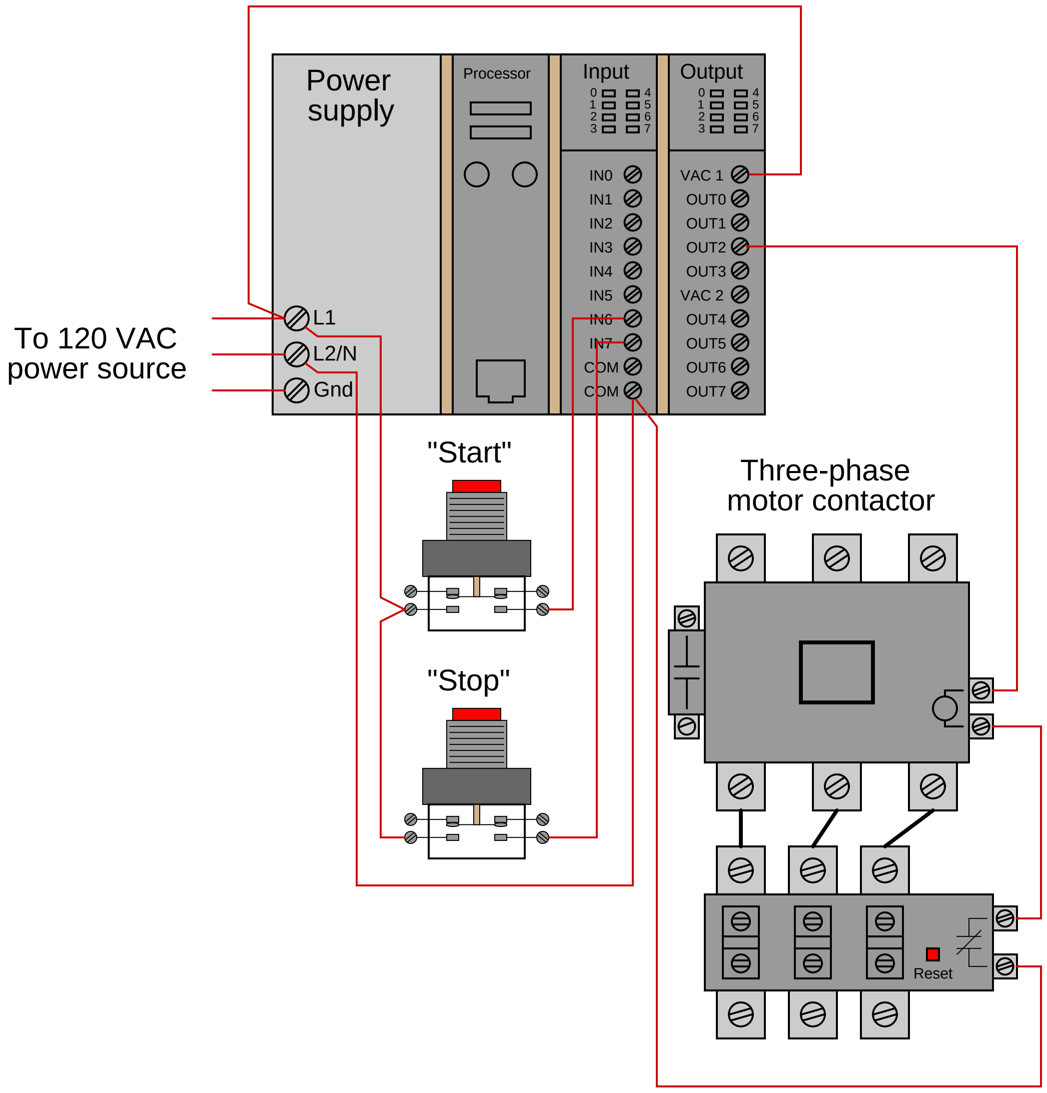

Contacts *read* bit status in the PLC's memory
Coils *write* bit status in the PLC's memory
IN_sensor_A
IN_sensor_B
OUT_burner_lit
IN_sensor_B
IN_sensor_C
Color-highlighted coil indicates powered (activated) status
IN_sensor_A
IN_sensor_C
Color-highlighted contacts indicate "closed" (conductive) status

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.

Contacts and relays are not just useful for implementing simple logic functions, but they may also perform *latching* functions as well. A very common application of this in industrial PLC systems is a latching start/stop program for controlling electric motors by means of momentary-contact pushbutton switches. As before, this functionality will be illustrated by means of an hypothetical example circuit and program:

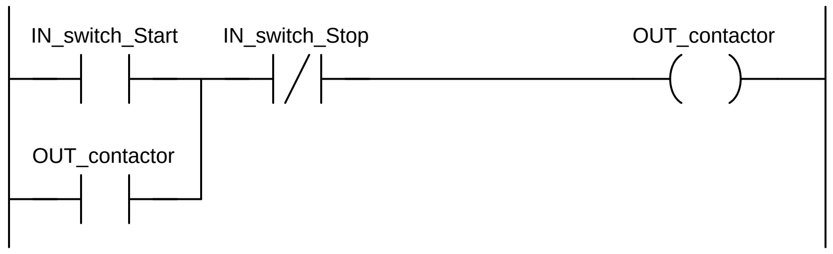


In this system, two pushbutton switches are connected to discrete inputs on a PLC, and the PLC in turn energizes the coil of a motor contactor relay by means of one of its discrete outputs[16].

[16]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[17].

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[18], 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.

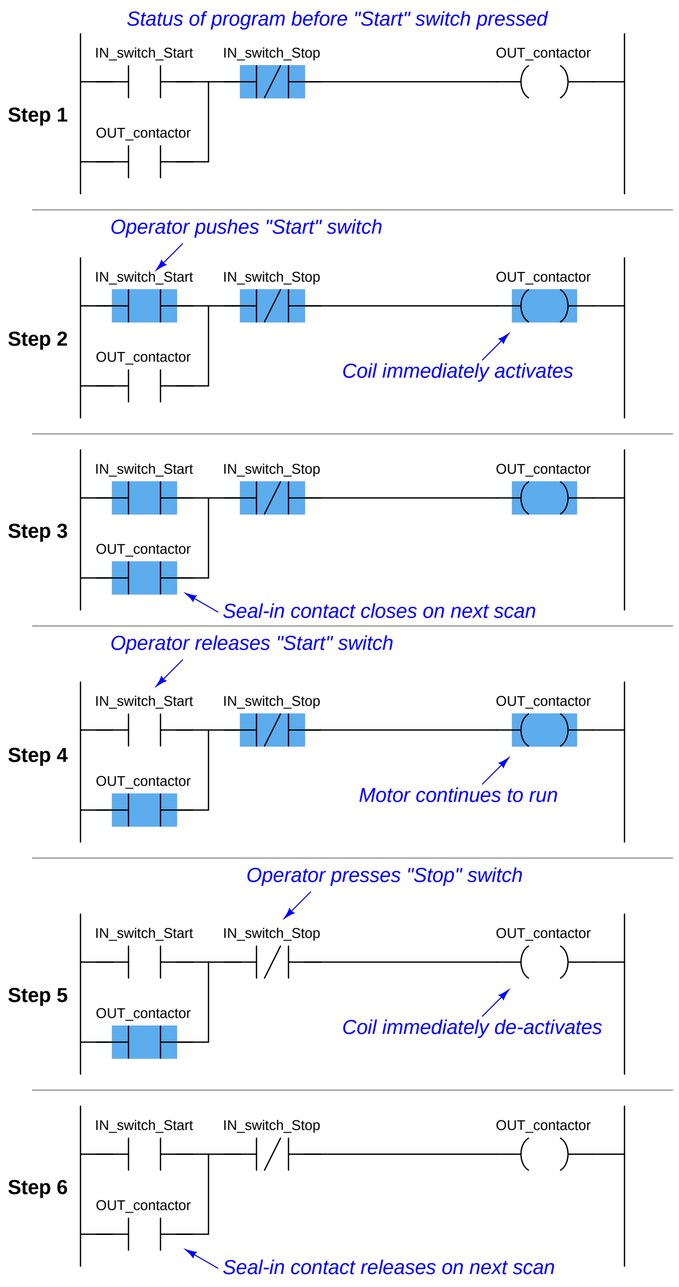
This programming technique is known as *feedback*, where an output variable of a function (in this case, the feedback variable is `OUT_contactor`) is also an input to that same function. The path of feedback is *implicit* rather than *explicit* in Ladder Diagram programming, with the only indication 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.

[17]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.

[18]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:

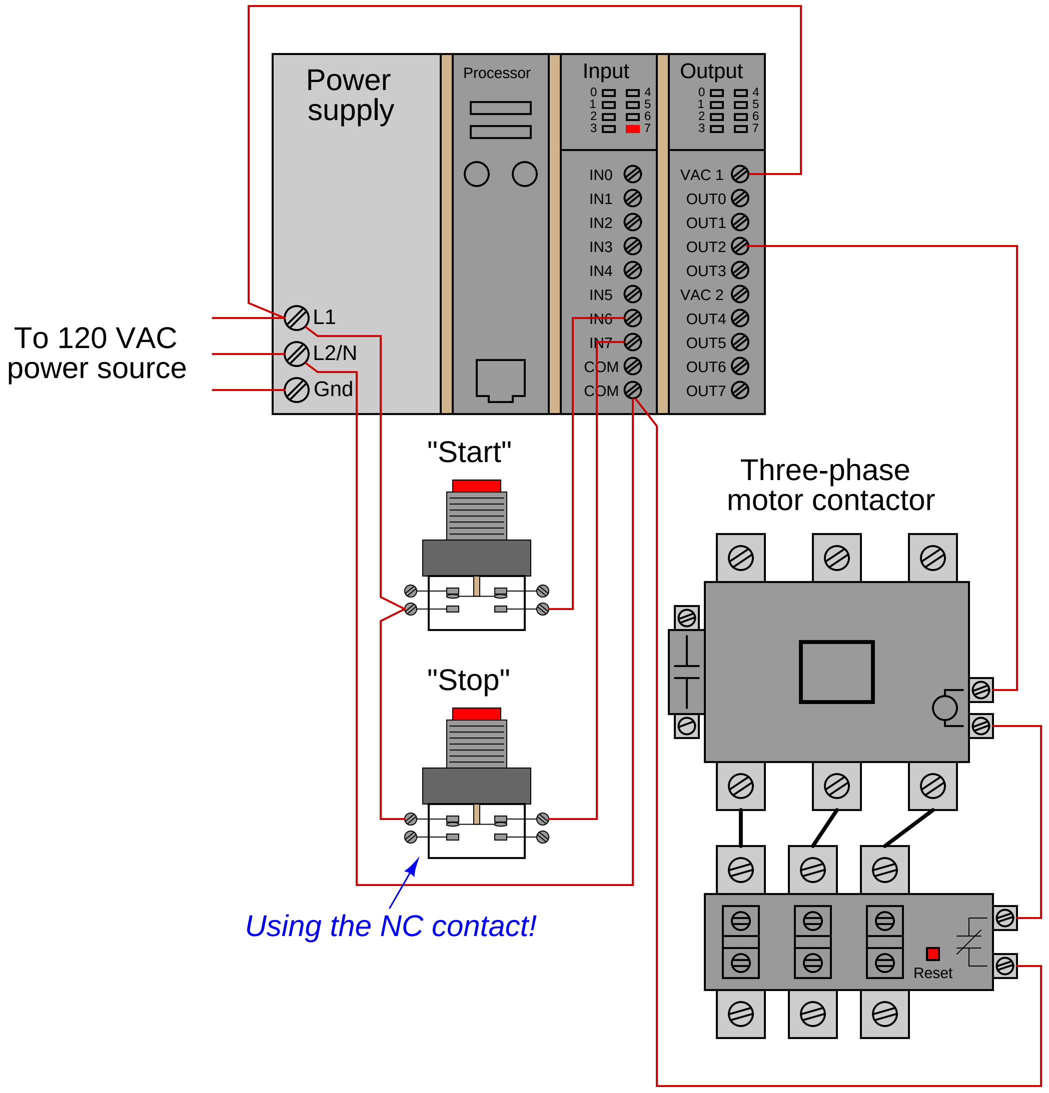

Status of program before "Start" switch pressed
Step 1
IN_switch_Start
IN_switch_Stop
OUT_contactor
OUT_contactor
Operator pushes "Start" switch
Step 2
Coil immediately activates
Step 3
Seal-in contact closes on next scan
Operator releases "Start" switch
Step 4
Motor continues to run
Operator presses "Stop" switch
Step 5
Coil immediately de-activates
Step 6
Seal-in contact releases on next scan

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.

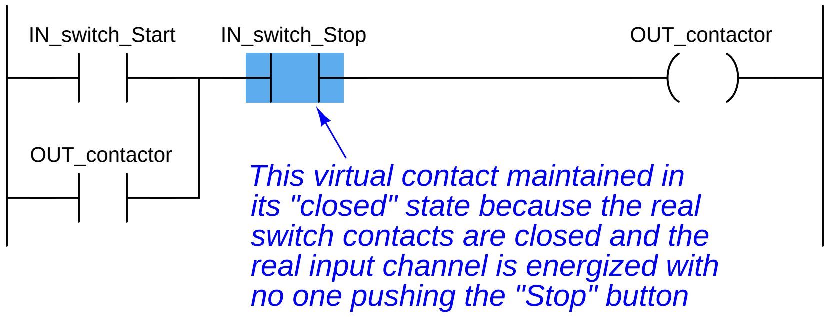
A potential problem with this system as it is designed is that the human operator loses control of the motor in the event of an "open" wiring failure in either pushbutton switch circuit. For instance, if a wire fell off a screw contact for the "Start" pushbutton switch circuit, the motor could not be 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):


IN_switch_Start
IN_switch_Stop
OUT_contactor
OUT_contactor
This virtual contact maintained in its "closed" state because the real switch contacts are closed and the real input channel is energized with no one pushing the "Stop" button

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[19]. 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.

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

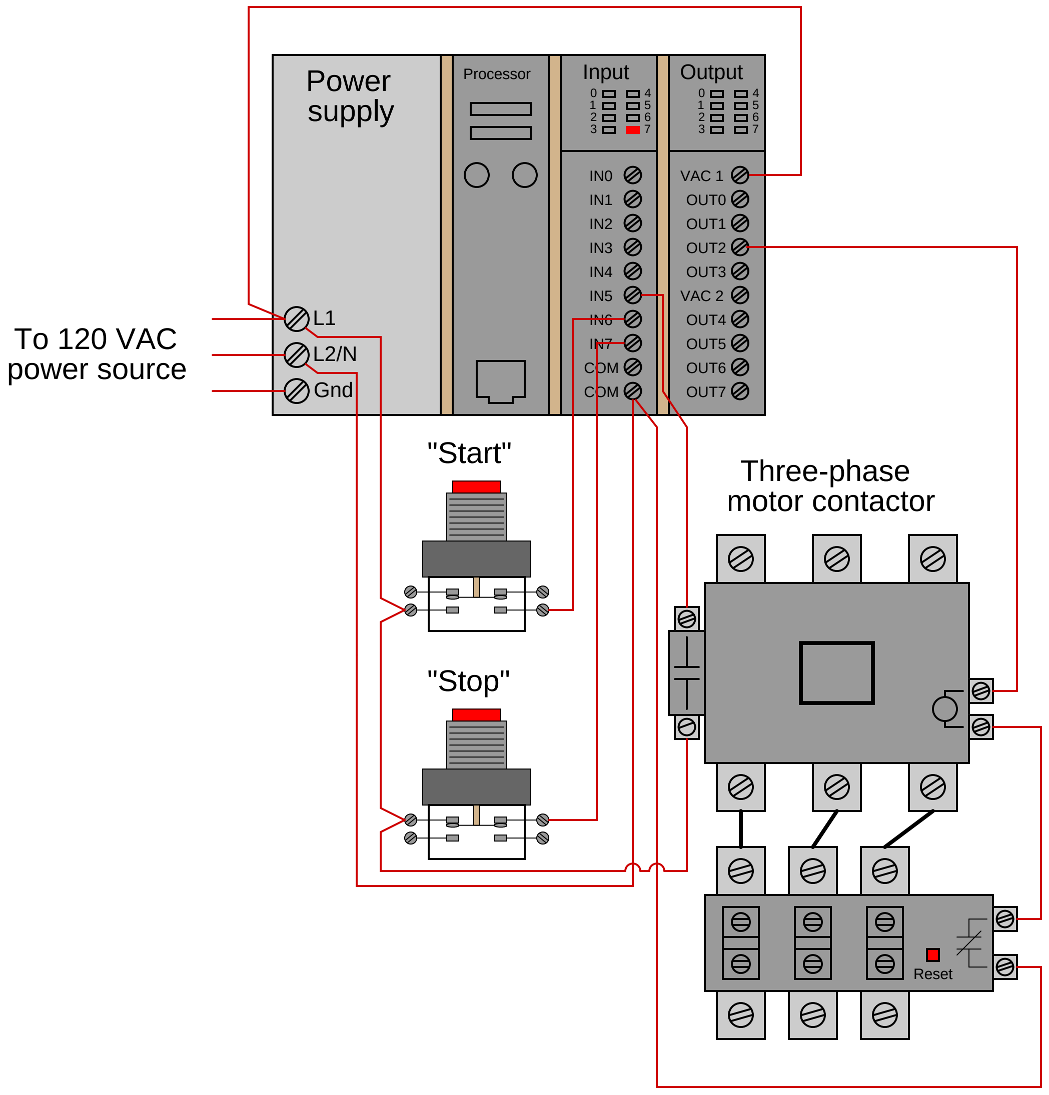
[19]In an effort to alleviate this confusion, the Allen-Bradley corporation (Rockwell) uses the terms *examine if closed* (XIC) and *examine if open* (XIO) to describe "normally open" and "normally closed" virtual contacts, respectively, in their Ladder Diagram programming. The idea here is that a virtual 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 <u>read instruction</u> for the bit set by power coming from the real-life electrical switch contact.*

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.

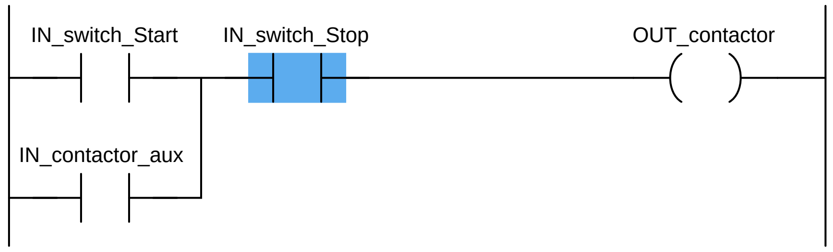
What would be safer is a motor control system that refuses to "latch" on unless the contactor actually energizes when the "Start" switch is pressed. For this to be possible, the PLC must have some way of sensing the contactor's status.

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.

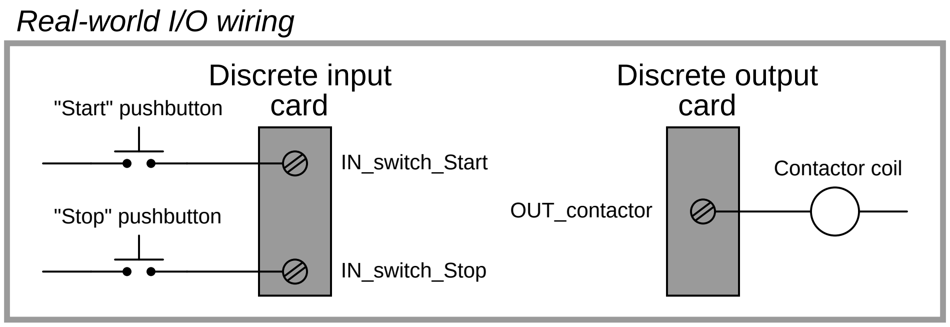
We may modify the PLC program to recognize this status by assigning a new tag name to this input (`IN_contactor_aux`) and using a normally-open virtual contact of this name as the seal-in contact instead of the `OUT_contactor` bit:

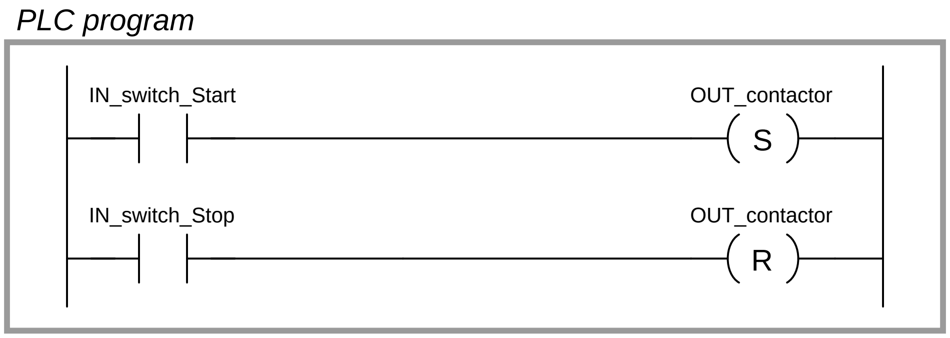


Now, if the contactor fails to energize for any reason when the operator presses the "Start" switch, the PLC's output will fail to latch when the "Start" switch is released. When the open fault in the contactor's coil circuit is cleared, the motor will *not* immediately start up, but rather wait until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

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):





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[20] are examples of what is 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*.

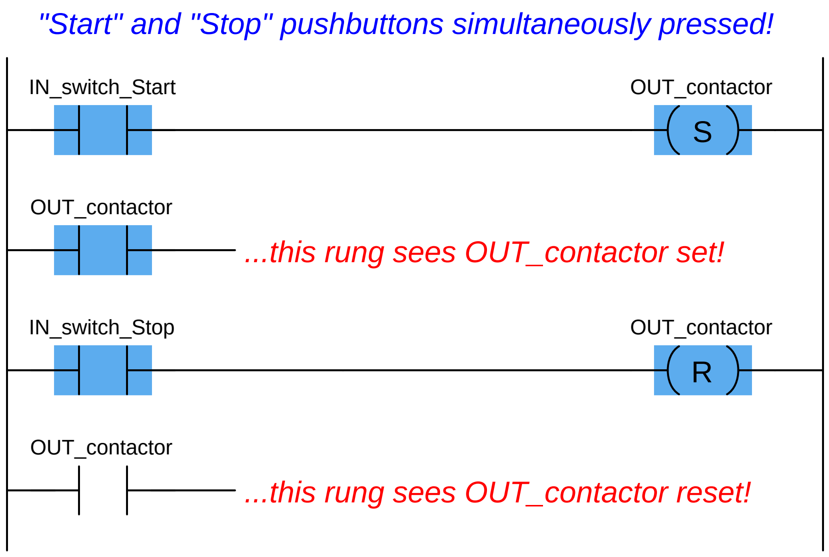
Ordinarily, we try to avoid multiple coils bearing the same label in a PLC Ladder Diagram program. With each coil representing a "write" instruction, multiple coils bearing the same name represents multiple "write" operations to the same bit in the PLC's memory. Here, with latching coils, there is no conflict because each of the coils only writes to the `OUT_contactor` bit when its

[20]Referred to as "Latch" and "Unlatch" coils by Allen-Bradley.

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.

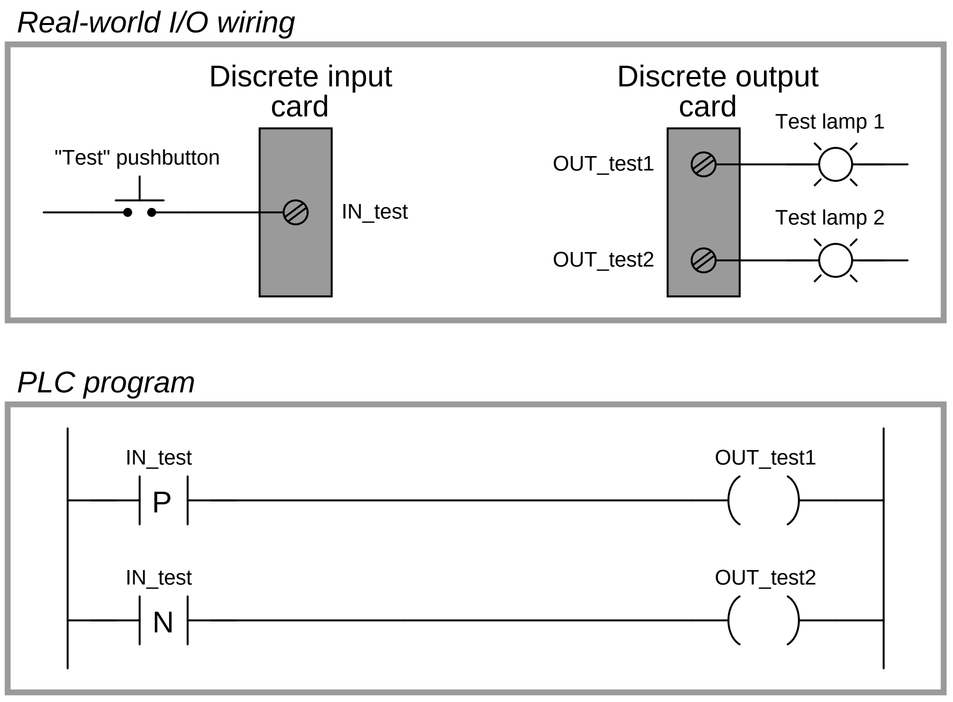
This raises the question: what would happen if *both* pushbutton switches were simultaneously pressed? What would happen if *both* "Set" and "Reset" coils were "energized" at the same time? The result is that the `OUT_contactor` bit would first be "set" (written to a value of 1) then "reset" (written 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).

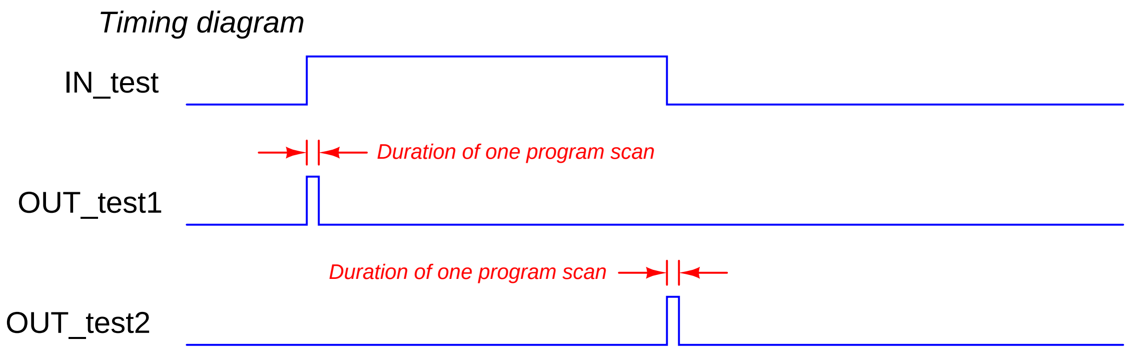
Even if the discrete output is not "confused" due to the conflicting write operations of the "Set" and "Reset" coils, other rungs of the program written between the "Set" and "Reset" rungs might be. Consider for example a case where there were other program rungs following the "Set" and "Reset" rungs 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:


"Start" and "Stop" pushbuttons simultaneously pressed!
IN_switch_Start
OUT_contactor
S
OUT_contactor
...this rung sees OUT_contactor set!
IN_switch_Stop
OUT_contactor
R
OUT_contactor
...this rung sees OUT_contactor reset!

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.

The IEC 61131-3 PLC programming standard specifies *transition-sensing* contacts as well as the more customary "static" contacts. A transition-sensing contact will "actuate" only for a duration of one program scan, even if its corresponding bit remains active. Two types of transition-sensing Ladder Diagram contacts are defined in the IEC standard: one for *positive* transitions and another for *negative* transitions. The following example shows a wiring diagram, Ladder Diagram program, and a timing diagram 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:

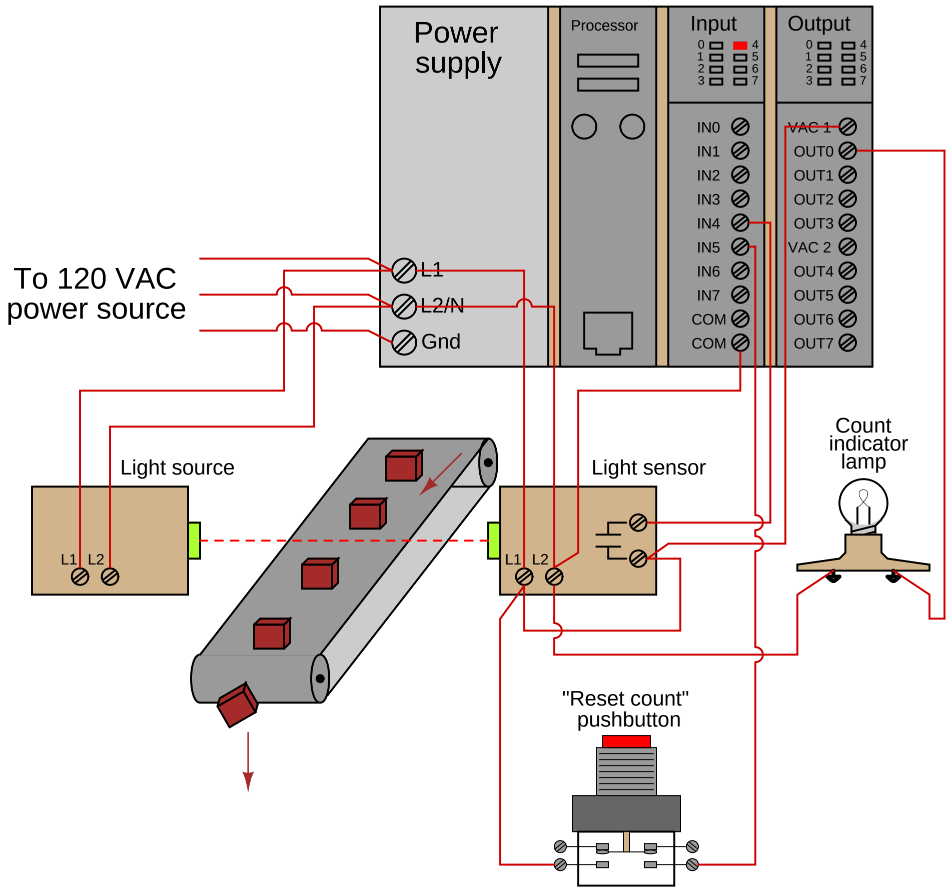
It should be noted that the duration of a single PLC program scan is typically very short: measured in milliseconds. If this program were actually tested in a real PLC, you would probably not be able to see either test lamp light up, since each pulse is so short-lived. Transitional contacts are typically used any time it is desired 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 maintained "true."

Contacts and coils represent only the most basic of instructions in the Ladder Diagram PLC programming language. Many other instructions exist, which will be discussed in the following sections.

4.5.3 Ladder diagram counters

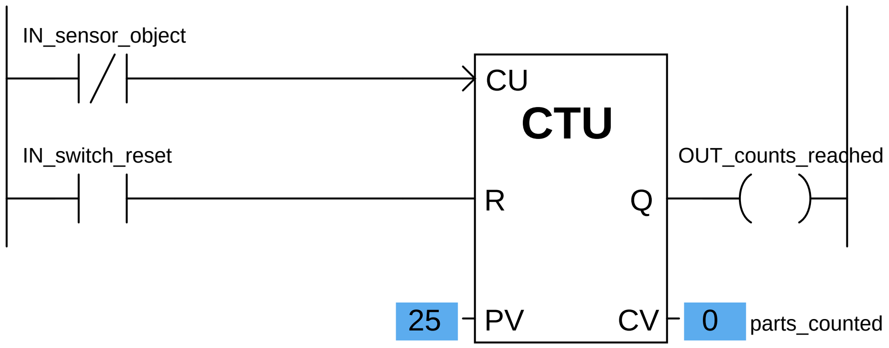
A *counter* is a PLC instruction that either increments (counts up) or decrements (counts down) an integer number value when prompted by the transition of a bit from 0 to 1 ("false" to "true"). Counter instructions come in three basic types: *up* counters, *down* counters, and *up/down* counters. Both "up" and "down" counter instructions have single inputs for triggering counts, whereas "up/down" counters have two trigger inputs: one to make the counter increment and one to make the counter decrement.

To illustrate the use of a counter instruction, we will analyze a PLC-based system designed to count objects as they pass down a conveyor belt:



In this system, a continuous (unbroken) light beam causes the light sensor to close its output contact, energizing discrete channel IN4. When an object on the conveyor belt interrupts the light beam from source to sensor, the sensor's contact opens, interrupting power to input IN4. A pushbutton switch connected to activate discrete input IN5 when pressed will serve as a manual "reset" of the count value. An indicator lamp connected to one of the discrete output channels will serve as an indicator of when the object count value has exceeded some pre-set limit.

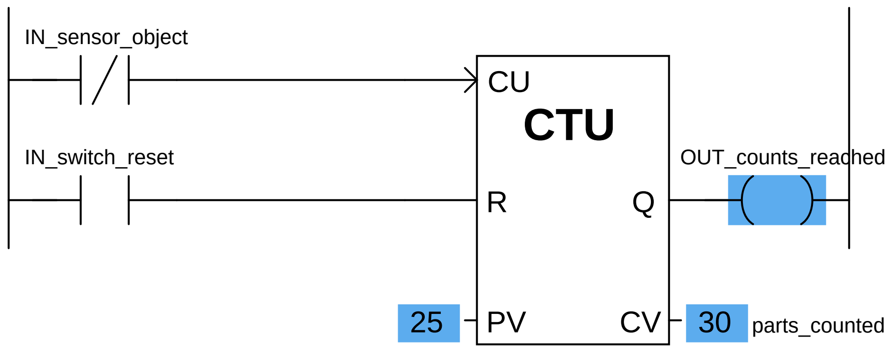
We will now analyze a simple Ladder Diagram program designed to increment a counter instruction each time the light beam breaks:



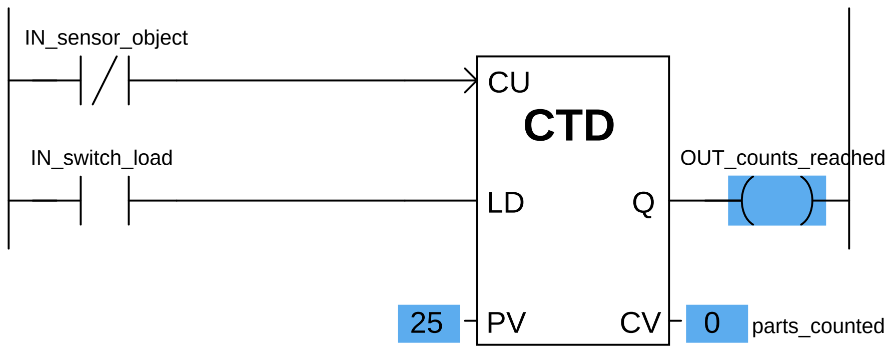
This particular counter instruction (CTU) is an incrementing counter, which means it counts "up" with each off-to-on transition input to its "CU" input. The normally-closed virtual contact (`IN_sensor_object`) is typically held in the "open" state when the light beam is continuous, by virtue of the fact the sensor holds that discrete input channel energized while the beam is continuous. When the beam is broken by a passing object on the conveyor belt, the input channel de-energizes, causing the virtual contact `IN_sensor_object` to "close" and send virtual power to the "CU" input of the counter instruction. This increments the counter just as the leading edge of the object breaks the beam. The second input of the counter instruction box ("R") is the *reset* input, receiving virtual power from the contact `IN_switch_reset` whenever the reset pushbutton is pressed. If this input is activated, the counter immediately resets its current value (CV) to zero.

Status indication is shown in this Ladder Diagram program, with the counter's preset value (PV) of 25 and the counter's current value (CV) of 0 shown highlighted in blue. The preset value is something programmed into the counter instruction before the system put into service, and it serves as a threshold for activating the counter's output (Q), which in this case turns on the count indicator lamp (the `OUT_counts_reached` coil). According to the IEC 61131-3 programming standard, this counter output should activate whenever the current value is equal to or greater than the preset value (Q is active if CV $\geq$ PV).

This is the status of the same program after thirty objects have passed by the sensor on the conveyor belt. As you can see, the current value of the counter has increased to 30, exceeding the preset value and activating the discrete output:



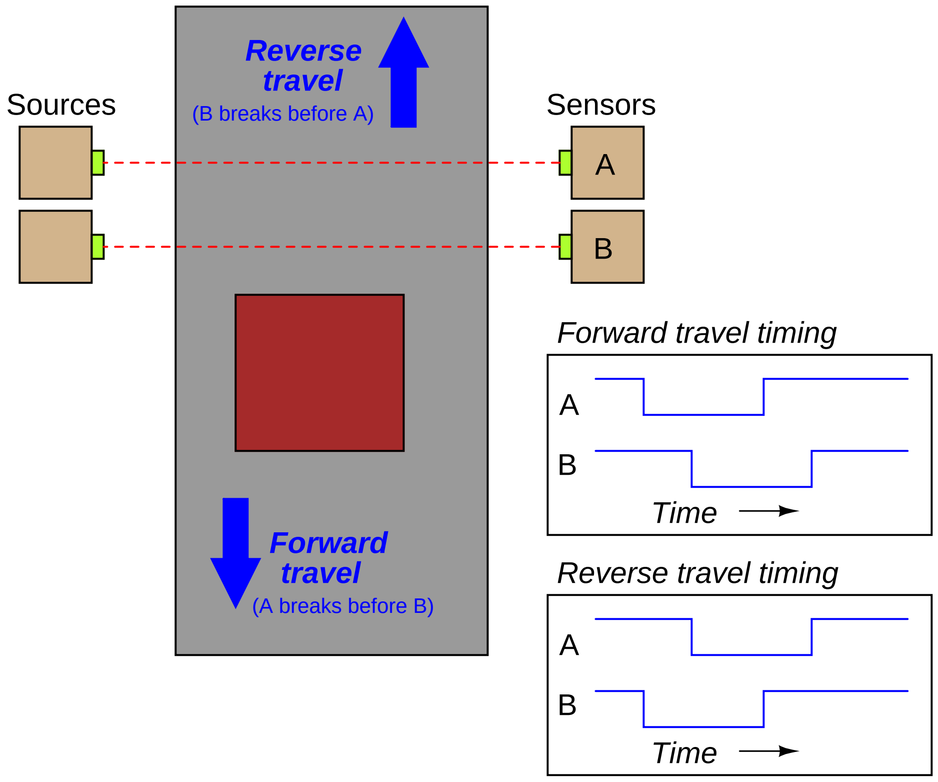
If all we did not care about maintaining an accurate total count of objects past 25 – but merely wished the program to indicate when 25 objects had passed by – we could also use a *down* counter instruction preset to a value of 25, which turns on an output coil when the count reaches zero:



Here, a "load" input causes the counter's current value to equal the preset value (25) when activated. With each sensor pulse received, the counter instruction decrements. When it reaches zero, the Q output activates.

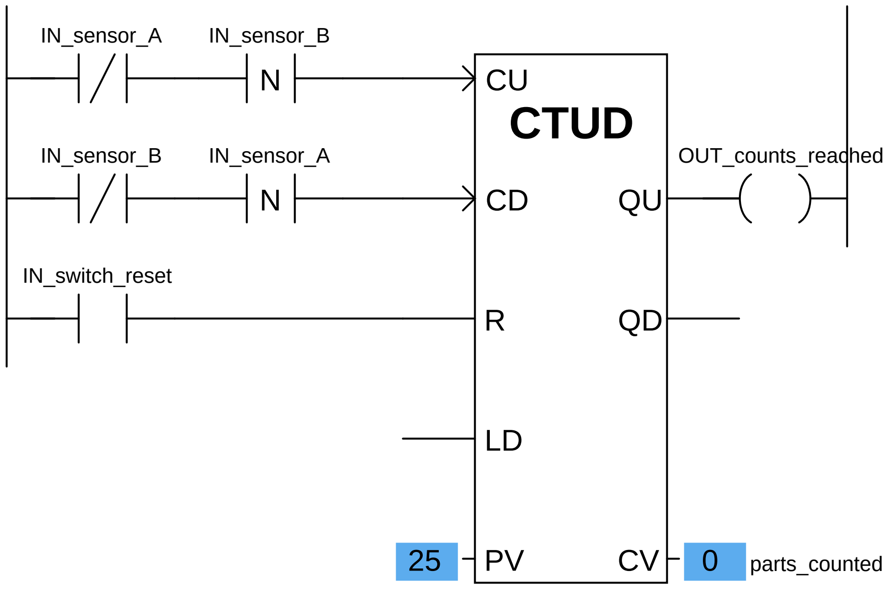
A potential problem in either version of this object-counting system is that the PLC cannot discriminate between forward and reverse motion on the conveyor belt. If, for instance, the conveyor belt were ever reversed in direction, the sensor would continue to count objects that had already passed by before (in the forward direction) as those objects retreated on the belt. This would be a problem because the system would "think" more objects had passed along the belt (indicating greater production) than actually did.

One solution to this problem is to use an up/down counter, capable of both incrementing (counting up) and decrementing (counting down), and equip this counter with two light-beam sensors capable of determining direction of travel. If two light beams are oriented parallel to each other, closer than the width of the narrowest object passing along the conveyor belt, we will have enough information to determine direction of object travel:



This is called *quadrature* signal timing, because the two pulse waveforms are approximately 90^o (one-*quarter* of a period) apart in phase. We can use these two phase-shifted signals to increment or decrement an up/down counter instruction, depending on which pulse leads and which pulse lags.

A Ladder Diagram PLC program designed to interpret the quadrature pulse signals is shown here, making use of negative-transition contacts as well as standard contacts:



The counter will increment (count up) when sensor B de-energizes only if sensor A is already in the de-energized state (i.e. light beam A breaks before B). The counter will decrement (count down) when sensor A de-energizes only if sensor B is already in the de-energized state (i.e. light beam B breaks before A).

Note that the up/down counter has both a "reset" (R) input and a "load" input ("LD") to force the current value. Activating the reset input forces the counter's current value (CV) to zero, just as we saw with the "up" counter instruction. Activating the load input forces the counter's current value to the preset value (PV), just as we saw with the "down" counter instruction. In the case of an up/down counter, there are two Q outputs: a QU (output up) to indicate when the current value is equal to or greater than the preset value, and a QD (output down) to indicate when the current value is equal to or less than zero.

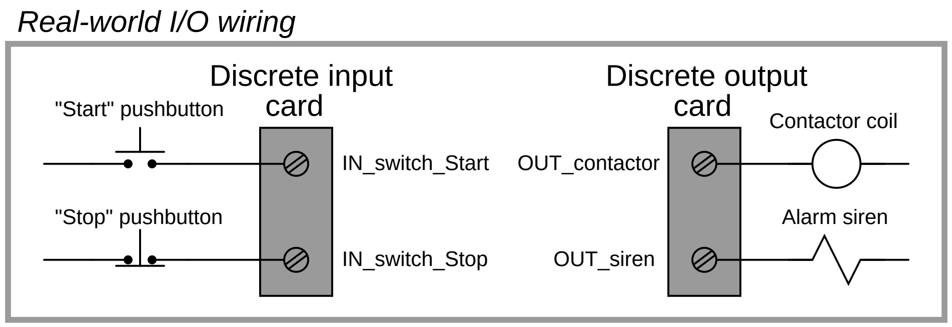
Note how the current value (CV) of each counter shown is associated with a tag name of its own, in this case `parts_counted`. The integer number of a counter's current value (CV) is a variable in the PLC's memory just like boolean values such as `IN_sensor_A` and `IN_switch_reset`, and may be associated with a tag name or symbolic address just the same[21]. This allows other instructions in a PLC program to read (and sometimes write!) values from and to that memory location.

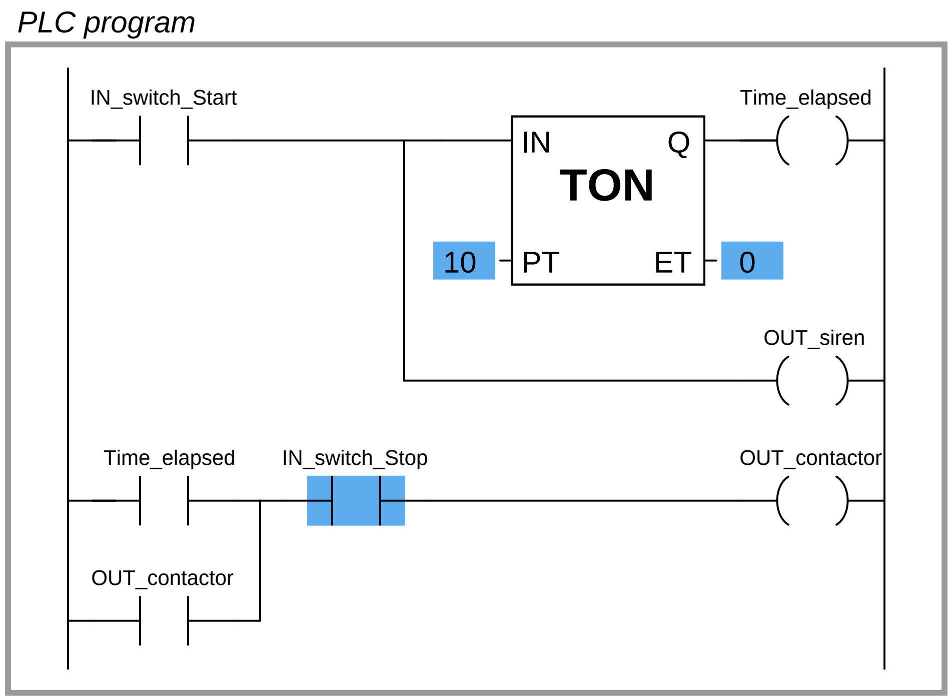
[21]This represents the IEC 61131-3 standard, where each variable within an instruction may be "connected" to its own arbitrary tag name. Other programming conventions may differ somewhat. The Allen-Bradley Logix5000 series of controllers is one of those that differs, following a convention reminiscent of structure element addressing in the C programming language: each counter is given a tag name, and variables in each counter are addressed as elements within that structure. For example, a Logix5000 counter instruction might be named `parts_count`, with the accumulated count value (equivalent to the IEC's "current value") addressed as `parts_count.ACC` (each element within the counter specified as a suffix to the counter's tag name).

4.5.4 Ladder diagram timers

A *timer* is a PLC instruction measuring the amount of time elapsed following an event. Timer instructions come in two basic types: *on-delay* timers and *off-delay* timers. Both "on-delay" and "off-delay" timer instructions have single inputs triggering the timed function.

An "on-delay" timer activates an output only when the input has been active for a minimum amount of time. Take for instance this PLC program, designed to sound an audio alarm siren prior to starting a conveyor belt. To start the conveyor belt motor, the operator must press and hold the "Start" pushbutton for 10 seconds, during which time the siren sounds, warning people to clear away from the conveyor belt that is about to start. Only after this 10-second start delay does the motor actually start (and latch "on"):



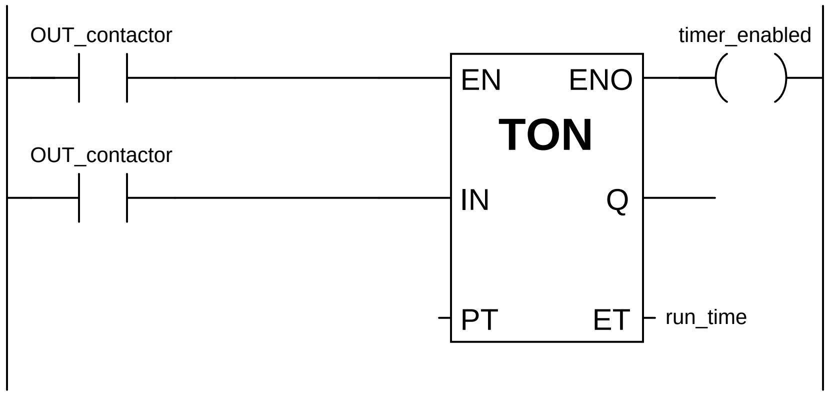


Similar to an "up" counter, the on-delay timer's elapsed time (ET) value increments once per second until the preset time (PT) is reached, at which time its output (Q) activates. In this program, the preset time value is 10 seconds, which means the Q output will not activate until the "Start" switch has been depressed for 10 seconds. The alarm siren output, which is not activated by the timer, energizes immediately when the "Start" pushbutton is pressed.

An important detail regarding this particular timer's operation is that it be *non-retentive*. This means the timer instruction should *not* retain its elapsed time value when the input is de-activated. Instead, the elapsed time value should reset back to zero every time the input de-activates. This ensures the timer resets itself when the operator releases the "Start" pushbutton. A *retentive* on-delay timer, by contrast, maintains its elapsed time value even when the input is de-activated. This makes it useful for keeping "running total" times for some event.

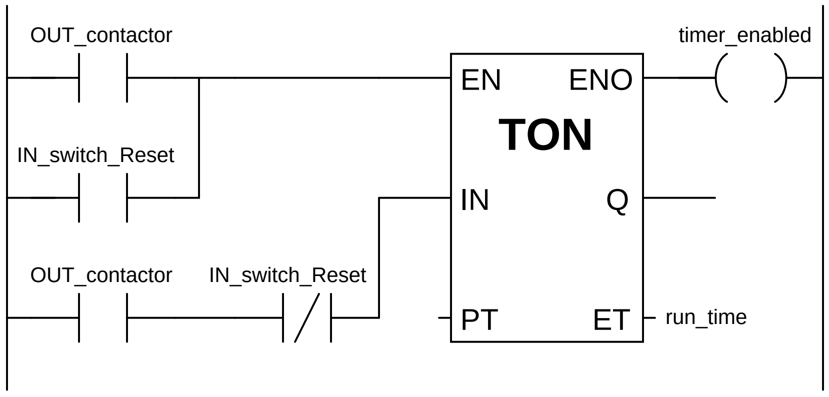
Most PLCs provide retentive and non-retentive versions of on-delay timer instructions, such that the programmer may choose the proper form of on-delay timer for any particular application. The IEC 61131-3 programming standard, however, addresses the issue of retentive versus non-retentive timers a bit differently. According to the IEC 61131-3 standard, a timer instruction may be specified with an additional *enable* input (EN) that causes the timer instruction to behave non-retentively when activated, and retentively when de-activated. The general concept of the enable (EN) input is that the instruction behaves "normally" so long as the enable input is active (in this case, non-retentive timing action is considered "normal" according to the IEC 61131-3 standard), but the instruction "freezes" all execution whenever the enable input de-activates. This "freezing" of operation has the effect of retaining the current time (CT) value even if the input signal de-activates.

For example, if we wished to add a retentive timer to our conveyor control system to record total run time for the conveyor motor, we could do so using an "enabled" IEC 61131-3 timer instruction like this:



When the motor's contactor bit (`OUT_contactor`) is active, the timer is enabled and allowed to time. However, when that bit de-activates (becomes "false"), the timer instruction as a whole is disabled, causing it to "freeze" and retain its current time (CT) value[22]. This allows the motor to be started and stopped, with the timer maintaining a tally of total motor run time.

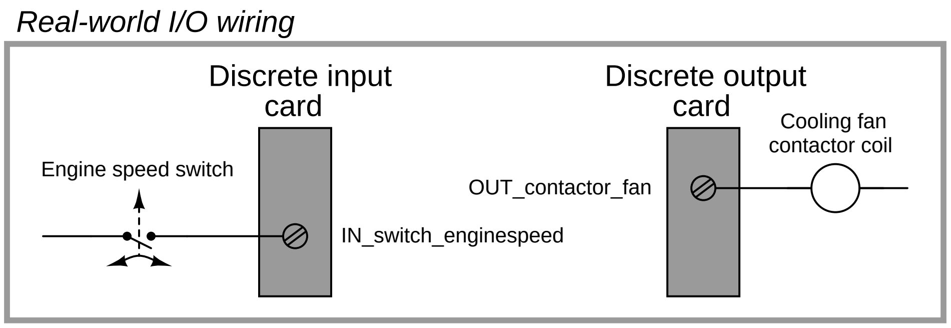
If we wished to give the operator the ability to manually reset the total run time value to zero, we could hard-wire an additional switch to the PLC's discrete input card and add "reset" contacts to the program like this:

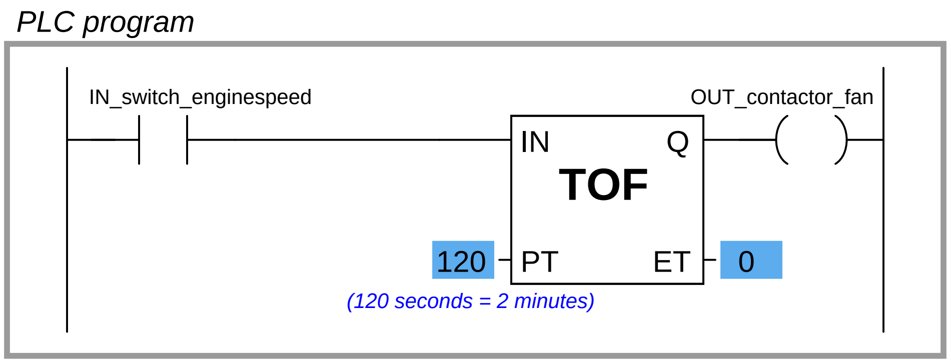


Whenever the "Reset" switch is pressed, the timer is enabled (EN) but the timing input (IN) is disabled, forcing the timer to (non-retentively) reset its current time (CT) value to zero.

[22]The "enable out" (ENO) signal on the timer instruction serves to indicate the instruction's status: it activates when the enable input (EN) activates and de-activates when either the enable input de-activates or the instruction generates an error condition (as determined by the PLC manufacturer's internal programming). The ENO output signal serves no useful purpose in this particular program, but it is available if there were any need for other rungs of the program to be "aware" of the run-time timer's status.

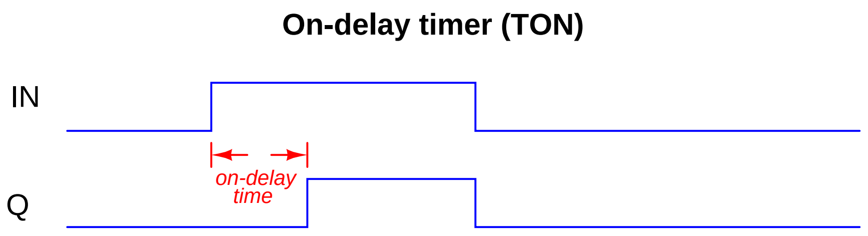
The other major type of PLC timer instruction is the *off-delay* timer. This timer instruction differs from the on-delay type in that the timing function begins as soon as the instruction is de-activated, not when it is activated. An application for an off-delay timer is a cooling fan motor control for a large industrial engine. In this system, the PLC starts an electric cooling fan as soon as the engine is detected as rotating, and keeps that fan running for two minutes following the engine's shut-down to dissipate residual heat:

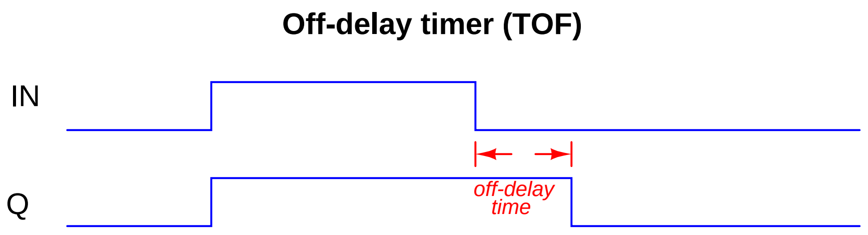




When the input (IN) to this timer instruction is activated, the output (Q) immediately activates (with no time delay at all) to turn on the cooling fan motor contactor. This provides the engine with cooling as soon as it begins to rotate (as detected by the speed switch connected to the PLC's discrete input). When the engine stops rotating, the speed switch returns to its normally-open position, de-activating the timer's input signal which starts the timing sequence. The Q output remains active while the timer counts from 0 seconds to 120 seconds. As soon as it reaches 120 seconds, the output de-activates (shutting off the cooling fan motor) and the elapsed time value remains at 120 seconds until the input re-activates, at which time it resets back to zero.

The following timing diagrams compare and contrast on-delay with off-delay timers:





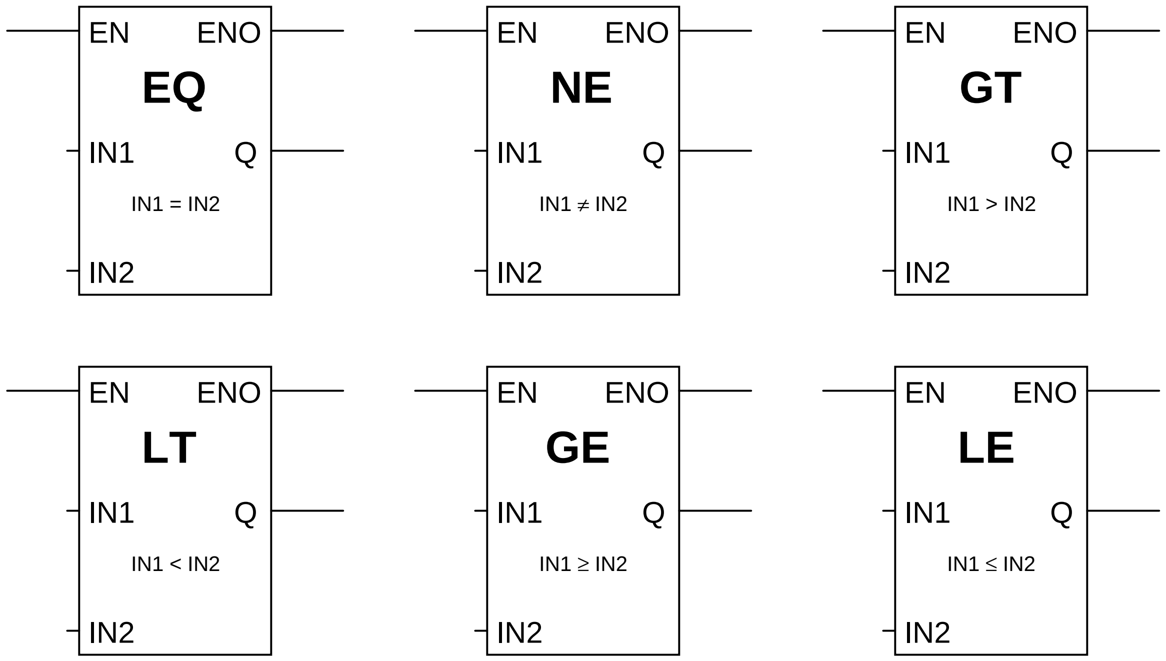
While it is common to find on-delay PLC instructions offered in both retentive and non-retentive forms within the instruction sets of nearly every PLC manufacturer and model, it is almost unheard of to find retentive off-delay timer instructions. Typically, off-delay timers are non-retentive only[23].

[23]The enable (EN) input signals specified in the IEC 61131-3 programming standard make retentive off-delay timers possible (by de-activating the enable input while maintaining the "IN" input in an inactive state), but bear in mind that most PLC implementations of timers do not have separate EN and IN inputs. This means (for most PLC timer instructions) the only input available to activate the timer is the "IN" input, in which case it is *impossible* to create a retentive off-delay timer (since such a timer's elapsed time value would be immediately re-set to zero each time the input re-activates).

4.5.5 Ladder diagram comparison instructions

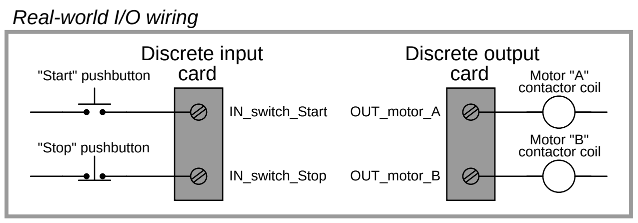
As we have seen with counter and timers, some PLC instructions generate digital values other than simple Boolean (on/off) signals. Counters have current value (CV) registers and timers have elapsed time (ET) registers, both of which are typically integer number values. Many other PLC instructions are designed to receive and manipulate non-Boolean values such as these to perform useful control functions.

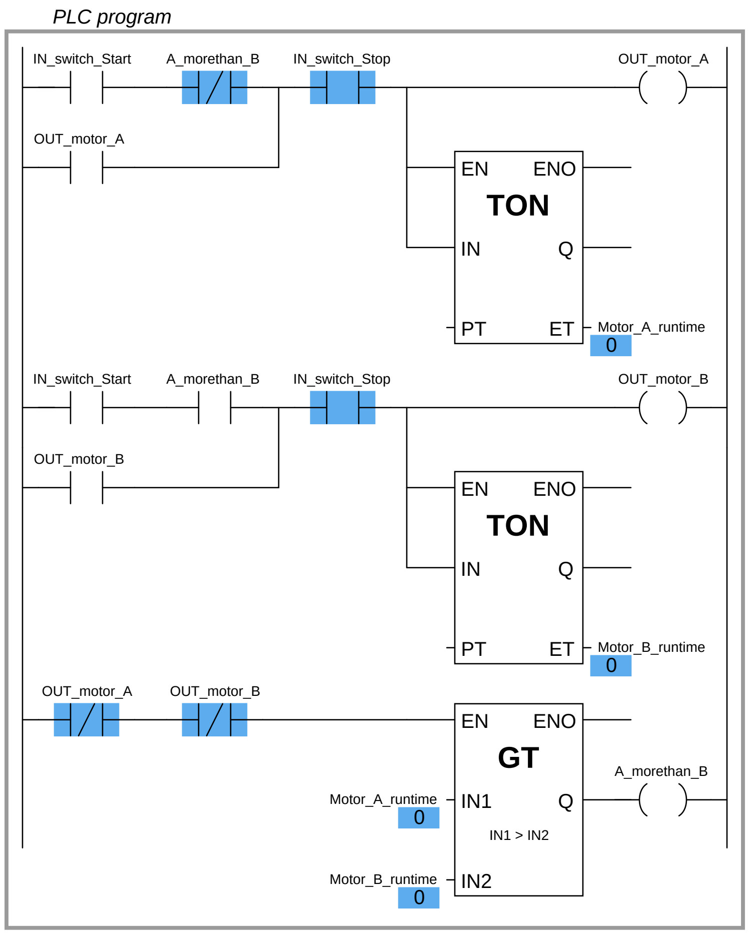
The IEC 61131-3 standard specifies a variety of *data comparison* instructions for comparing two non-Boolean values, and generating Boolean outputs. The basic comparative operations of "less than" ($<$), "greater than" ($>$), "less than or equal to" ($\leq$), "greater than or equal to" ($\geq$), "equal to" ($=$), and "not equal to" ($\neq$) may be found as a series of "box" instructions in the IEC standard:



The Q output for each instruction "box" activates whenever the evaluated comparison function is "true" and the enable input (EN) is active. If the enable input remains active but the comparison function is false, the Q output de-activates. If the enable input de-de-activates, the Q output retains its last state.

A practical application for a comparative function is something called *alternating motor control*, where the run-times of two redundant electric motors[24] are monitored, with the PLC determining which motor to turn on next based on which motor has run the least:





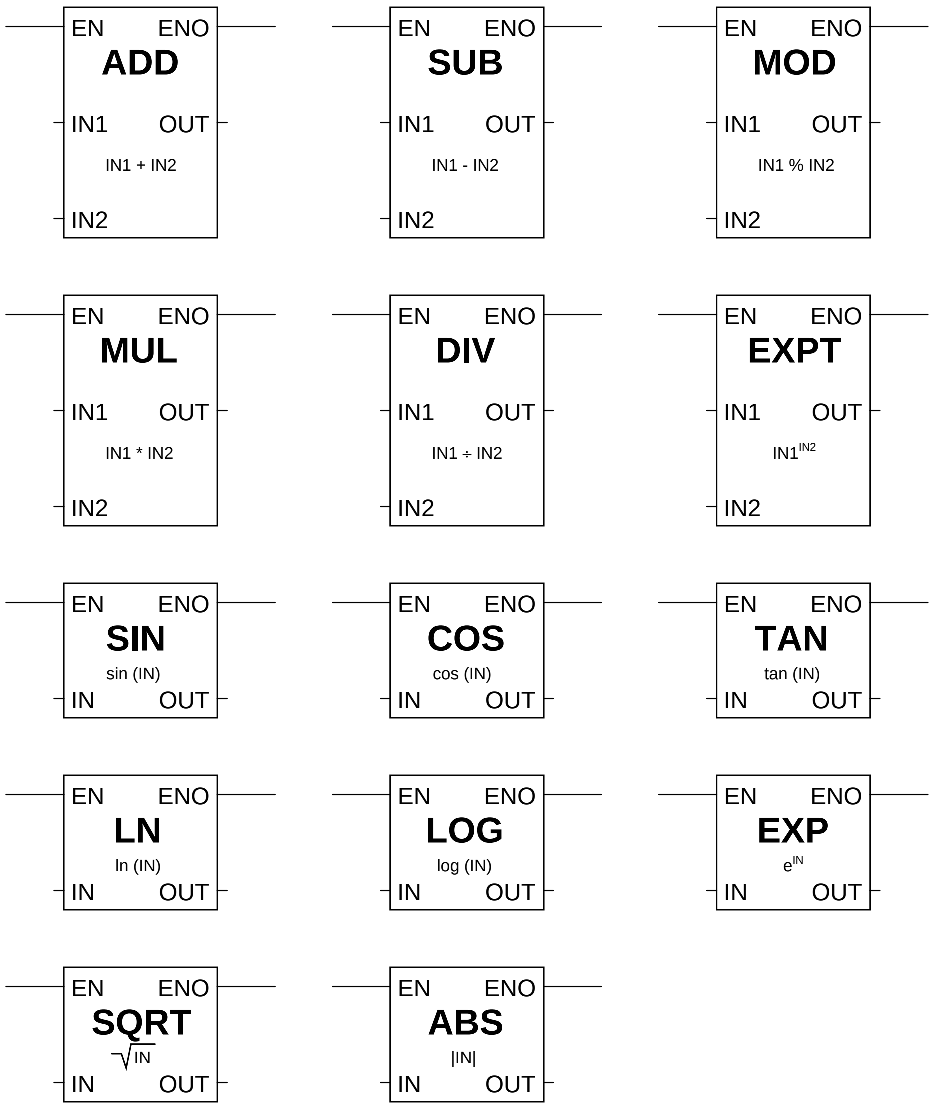
In this program, two retentive on-delay timers keep track of each electric motor's total run time, storing the run time values in two registers in the PLC's memory: `Motor_A_runtime` and

[24]Perhaps two pumps performing the same pumping function, one serving as a backup to the other. Alternating motor control ensures the two motors' run times are matched as closely as possible.

`Motor_B_runtime`. These two integer values are input to the "greater than" instruction box for comparison. If motor A has run longer than motor B, motor B will be the one enabled to start up next time the "start" switch is pressed. If motor A has run less time or the same amount of time as motor B (the scenario shown by the blue-highlighted status indications), motor A will be the one enabled to start. The two series-connected virtual contacts `OUT_motor_A` and `OUT_motor_B` ensure the comparison between motor run times is not made until both motors are stopped. If the comparison were continually made, a situation might arise where *both* motors would start if someone happened to press the Start pushbutton with one motor is already running.

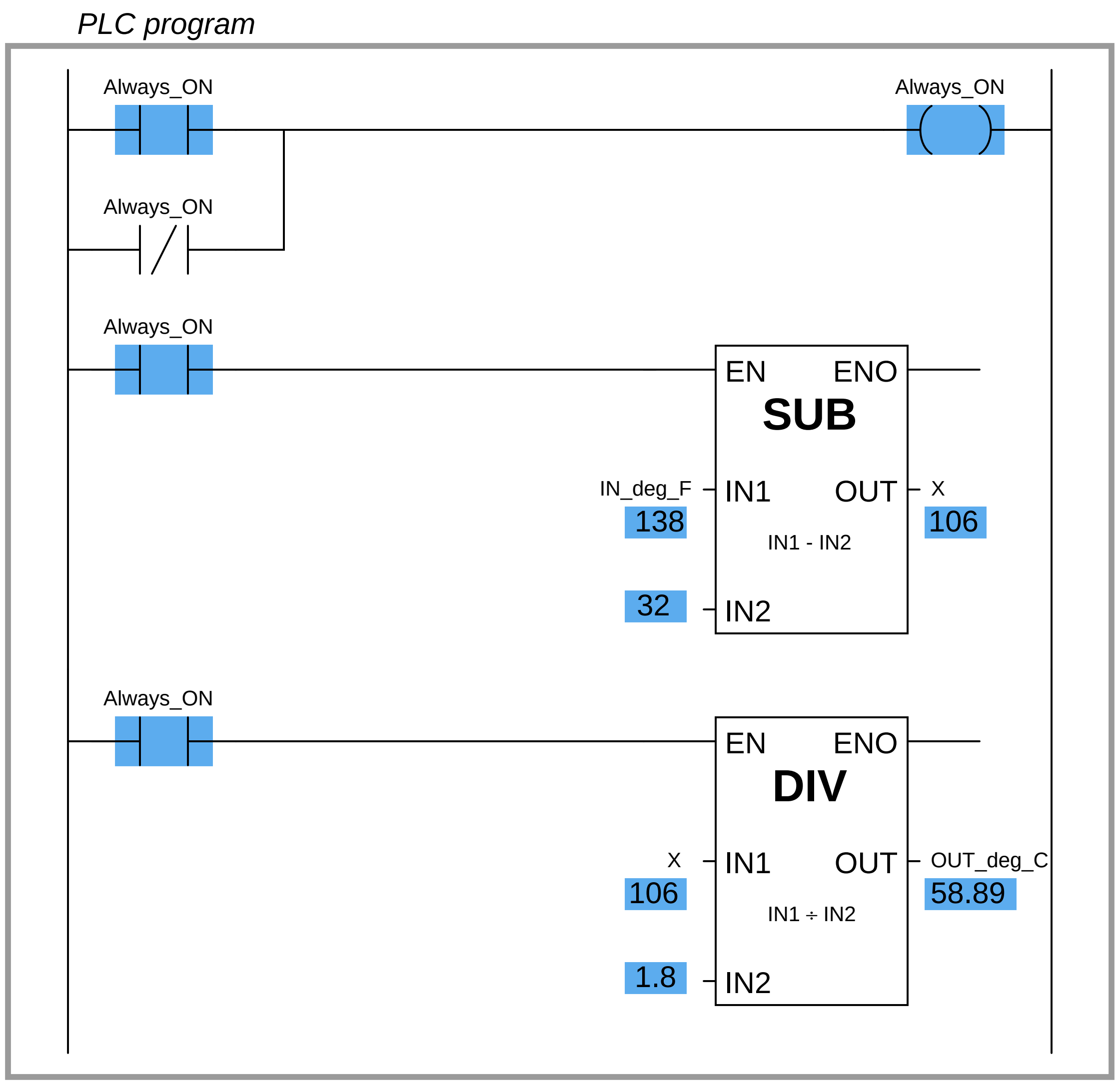
4.5.6 Ladder diagram arithmetic instructions

The IEC 61131-3 standard specifies several dedicated ladder instructions for performing arithmetic calculations. Some of them are shown here:


EN ENO ADD IN1 OUT IN1 + IN2 IN2
EN ENO SUB IN1 OUT IN1 - IN2 IN2
EN ENO MOD IN1 OUT IN1 % IN2 IN2
EN ENO MUL IN1 OUT IN1 * IN2 IN2
EN ENO DIV IN1 OUT IN1 ÷ IN2 IN2
EN ENO EXPT IN1 OUT IN1^IN2 IN2
EN ENO SIN sin (IN) IN OUT
EN ENO COS cos (IN) IN OUT
EN ENO TAN tan (IN) IN OUT
EN ENO LN ln (IN) IN OUT
EN ENO LOG log (IN) IN OUT
EN ENO EXP e^IN IN OUT
EN ENO SQRT √IN IN OUT
EN ENO ABS |IN| IN OUT

As with the data comparison instructions, each of these math instructions must be enabled by an "energized" signal to the enable (EN) input. Input and output values are linked to each math instruction by tag name.

An example showing the use of such instructions is shown here, converting a temperature measurement in units of degrees Fahrenheit to units of degrees Celsius. In this particular case, the program inputs a temperature measurement of 138 ^{o}F and calculates the equivalent temperature of 58.89 ^{o}C:

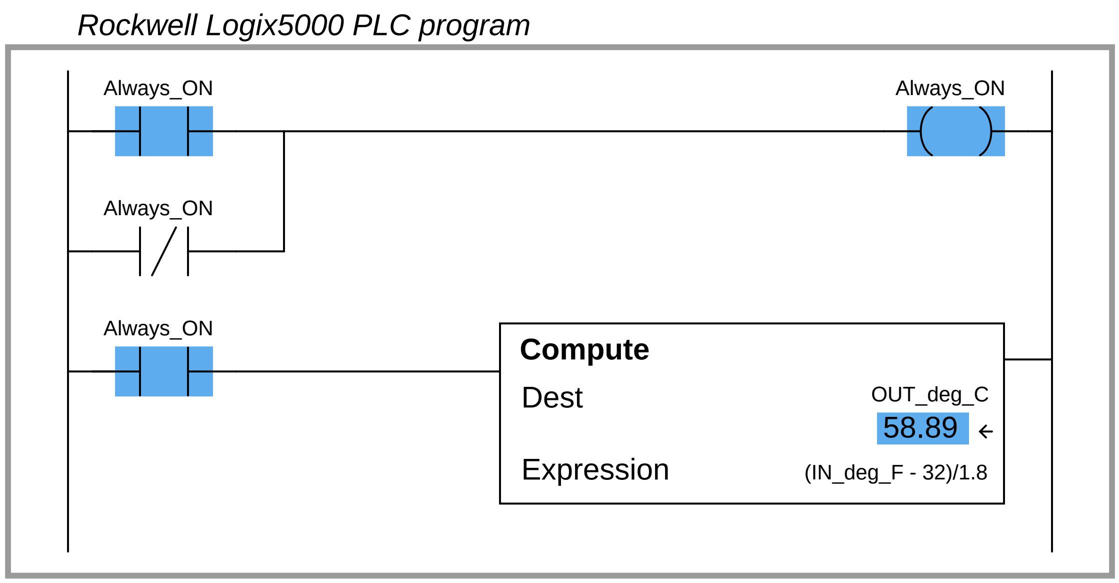


Note how two separate math instructions were required to perform this simple calculation, as well as a dedicated variable (`X`) used to store the intermediate calculation between the subtraction and the division "boxes."

Although not specified in the IEC 61131-3 standard, many programmable logic controllers support Ladder Diagram math instructions allowing the direct entry of arbitrary equations. Rockwell

(Allen-Bradley) Logix5000 programming, for example, has the "Compute" (`CPT`) function, which allows any typed expression to be computed in a single instruction as opposed to using several dedicated math instructions such as "Add," "Subtract," etc. General-purpose math instructions dramatically shorten the length of a ladder program compared to the use of dedicated math instructions for any applications requiring non-trivial calculations.

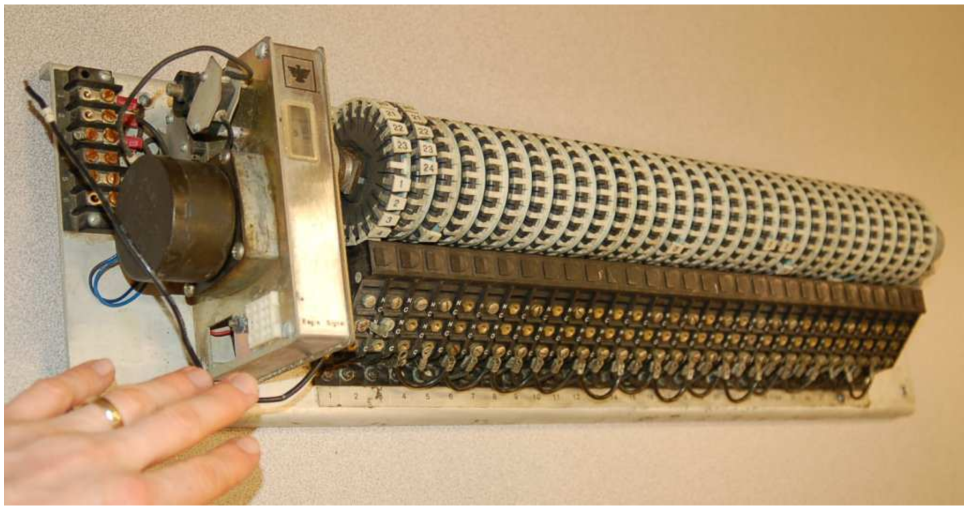
For example, the same Fahrenheit-to-Celsius temperature conversion program implemented in Logix5000 programming only requires a single math instruction and no declarations of intermediate variables:



4.5.7 Ladder diagram sequencer instructions

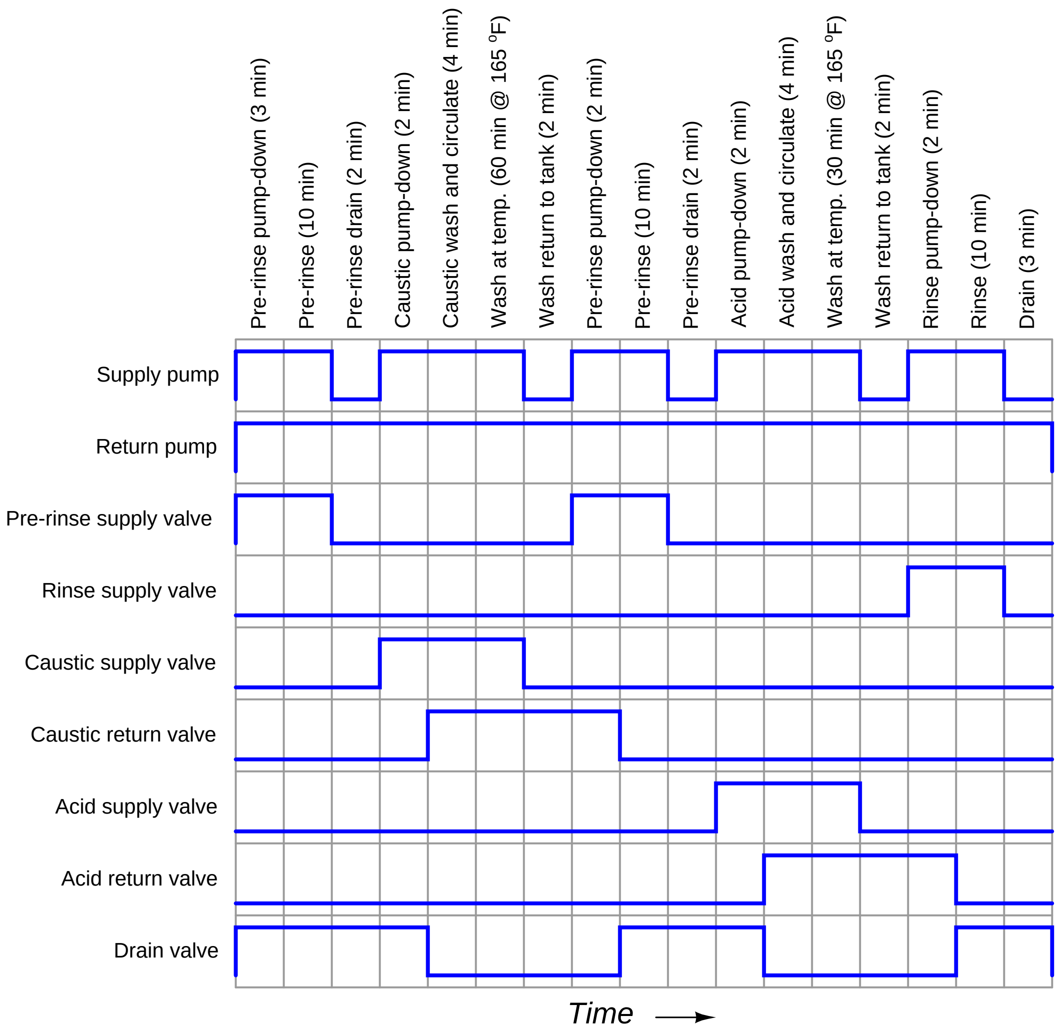
Many industrial processes require control actions to take place in certain, predefined sequences. Batch processes are perhaps the most striking example of this, where materials for making a batch must be loaded into the process vessels, parameters such as temperature and pressure controlled during the batch processing, and then discharge of the product monitored and controlled. Before the advent of reliable programmable logic devices, this form of sequenced control was usually managed by an electromechanical device known as a *drum sequencer*. This device worked on the principle of a rotating cylinder (drum) equipped with tabs to actuate switches as the drum rotated into certain positions. If the drum rotated at a constant speed (turned by a clock motor), those switches would actuate according to a timed schedule[25].

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



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

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



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

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

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

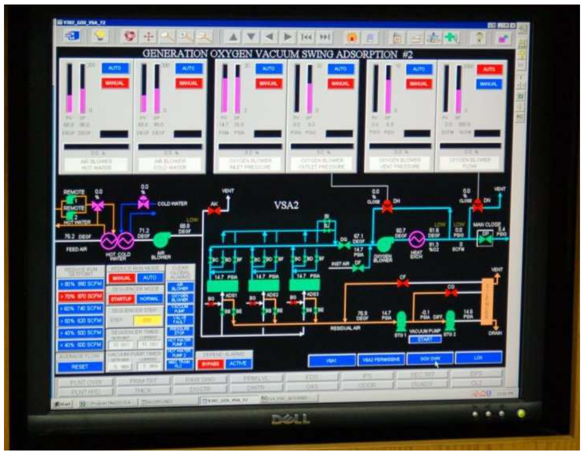
Unfortunately, despite the utility of drum sequence functions and their ease of implementation in digital form, there seems to be very little standardization between PLC manufacturers regarding sequencing instructions. Sadly, the IEC 61131-3 standard (at least at the time of this writing, in 2009) does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

4.6 Human-Machine Interfaces (HMIs)

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

One solution to this problem is a dedicated computer display programmed to provide selective access to certain variable's in the PLC's memory, generally referred to as *Human*[26]*-Machine Interface*, or *HMI*.

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

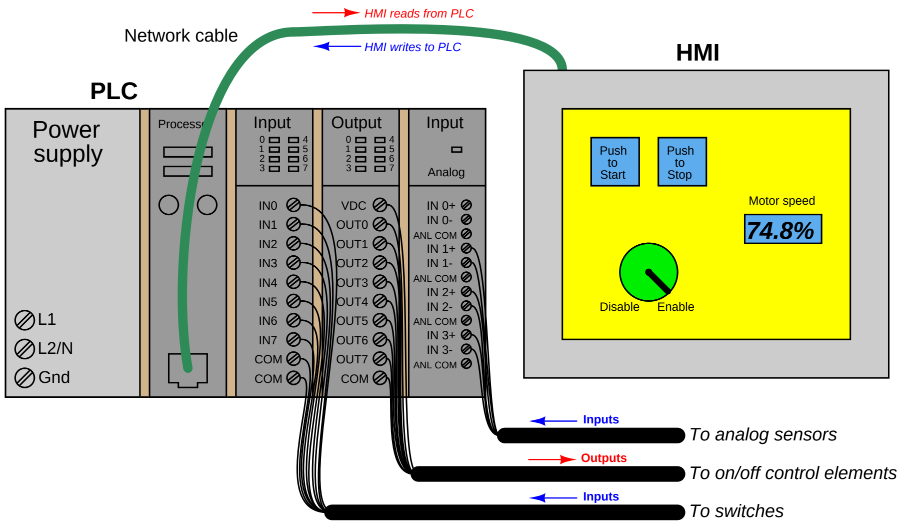


The display shown here happens to be for monitoring a vacuum swing adsorption (VSA) process for purifying oxygen extracted from ambient air. Somewhere, a PLC (or collection of PLCs) is monitoring and controlling this VSA process, with the HMI software acting as a "window" into the PLC's memory to display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

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

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

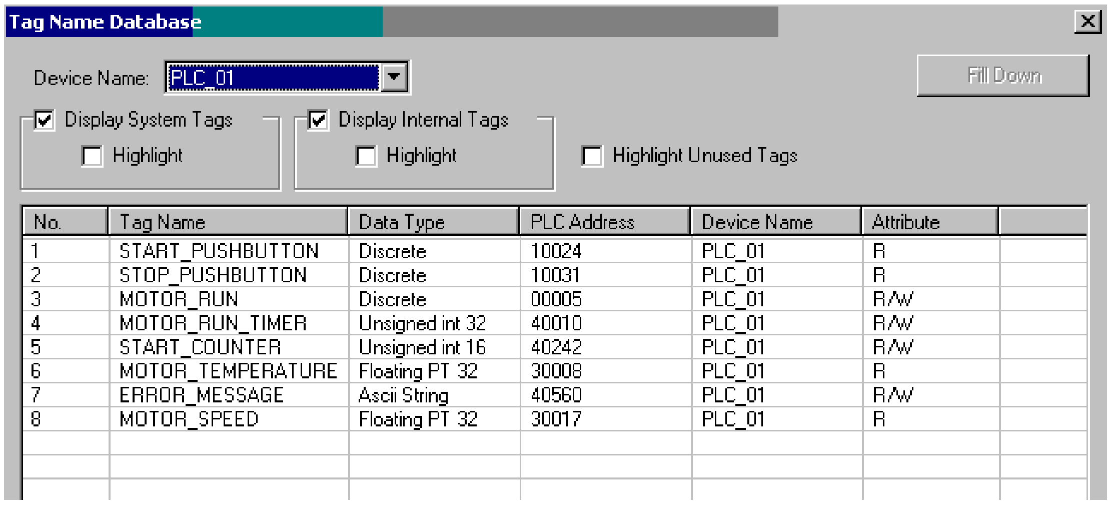
These HMI panels are really nothing more than "hardened" personal computers built ruggedly and in a compact format to facilitate their use in industrial environments. Most industrial HMI panels come equipped with touch-sensitive screens, allowing operators to press their fingertips on displayed objects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



Technicians and/or engineers program HMI displays to read and write data via a digital network to one or more PLCs. Graphical objects arrayed on the display screen of an HMI often mimic real-world indicators and switches, in order to provide a familiar interface for operations personnel. A "pushbutton" object on the face of an HMI panel, for example, would be configured to *write* one bit of data to the PLC, in a manner similar to a real-world switch writing one bit of data to the PLC's input register.

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

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



Device Name: PLC_01

[✓] Display System Tags

[✓] Display Internal Tags

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

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

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

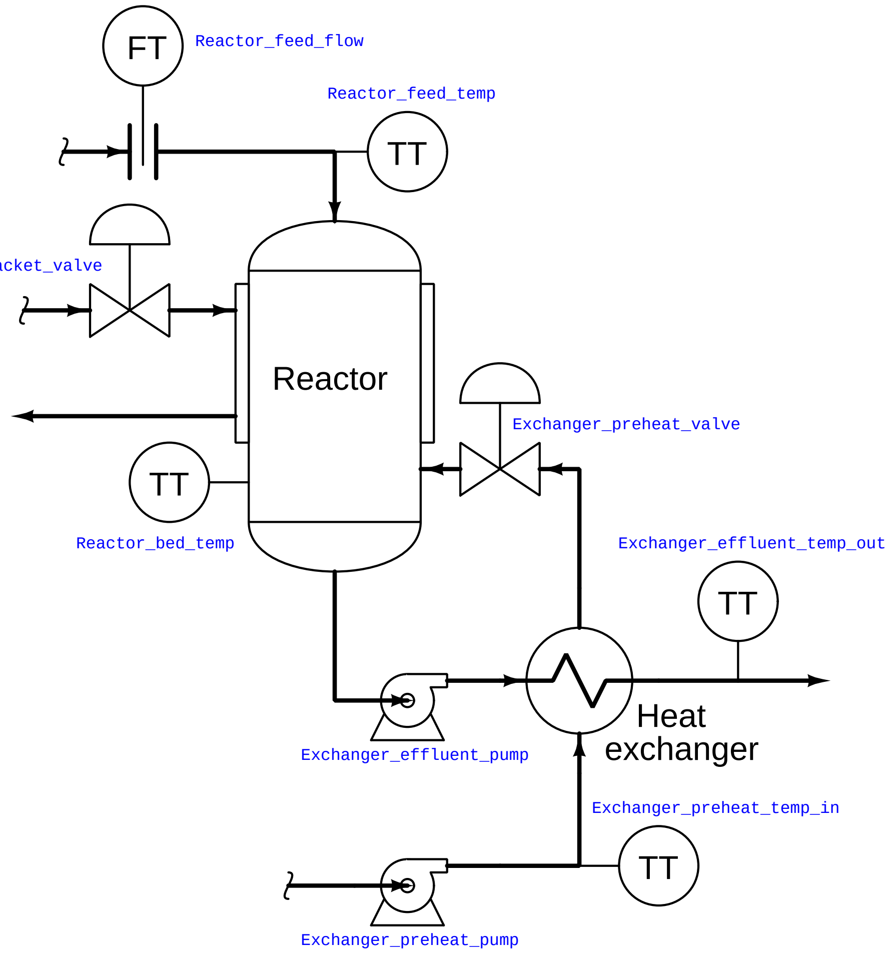
The potential for data conflict also exists for some of the other points in the database, however. A good example of this is the `MOTOR_RUN` bit, which is the bit within the PLC program telling the real-world motor to run. Presumably, this bit gets its data from a coil in the PLC's Ladder Diagram program.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

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

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

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

Another way to leverage the alphabetical listing of tag names to your advantage is to begin each tag name with a word describing its association to a major piece of equipment. Take for instance this example of a chemical fluid processing system with several data points[27] defined in a PLC control system and displayed in an HMI:



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

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

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

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

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

Like programmable logic controllers themselves, the capabilities of HMIs have been steadily increasing while their price decreases. Modern HMIs support graphic trending, data archival, advanced alarming, and even web server ability allowing other computers to easily access certain data over wide-area networks. The ability of HMIs to log data over long periods of time relieves the PLC of having to do this task, which is very memory-intensive. This way, the PLC merely "serves" current data to the HMI, and the HMI is able to keep a record of current and past data using its vastly larger memory reserves[28].

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

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

Chapter 5

Derivations and Technical References

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

5.1 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.

5.1.1 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.).

5.1.2 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.).

5.1.3 Special "system" values

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

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

5.1.4 Free-running clock pulses

- Allen-Bradley SLC 500: status bit `S:4/0` is a free-running clock pulse with a period of 20 milliseconds, which may be used to clock a counter instruction up to 50 to make a 1-second pulse (because 50 times 20 ms = 1000 ms = 1 second).

- Siemens S7-200: Special Memory bit `SM0.5` is a free-running clock pulse with a period of 1 second.

- Koyo (Automation Direct) DirectLogic: Special relay `SP4` is a free-running clock pulse with a period of 1 second.

5.1.5 Standard counter instructions

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

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

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

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

5.1.6 High-speed counter instructions

- Allen-Bradley SLC 500: `HSU` high-speed count-up instruction.

- Siemens S7-200: `HSC` high-speed counter instruction, used in conjunction with the `HDEF` high-speed counter definition instruction.

5.1.7 Timer instructions

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

- Allen-Bradley SLC 500: `TOF` off-delay timer, `TON` on-delay timer, and `RTO` retentive on-delay timer instructions.

- Siemens S7-200: `TOF` off-delay timer, `TON` on-delay timer, and `TONR` retentive on-delay timer instructions.

5.1.8 ASCII text message instructions

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

5.1.9 Analog signal scaling

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

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

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

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

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

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

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

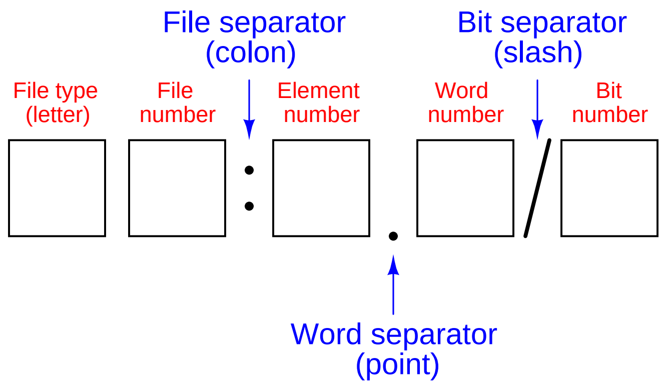
File number	**File type**	**Logical address range**
0	Output image	`O:0` to `O:30`
1	Input image	`I:0` to `I:30`
2	Status	`S:0` to `S:`n
3	Binary	`B3:0` to `B3:255`
4	Timers	`T4:0` to `T4:255`
5	Counters	`C5:0` to `C5:255`
6	Control	`R6:0` to `R6:255`
7	Integer	`N7:0` to `N7:255`
8	Floating-point	`F8:0`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

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

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

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

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



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

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

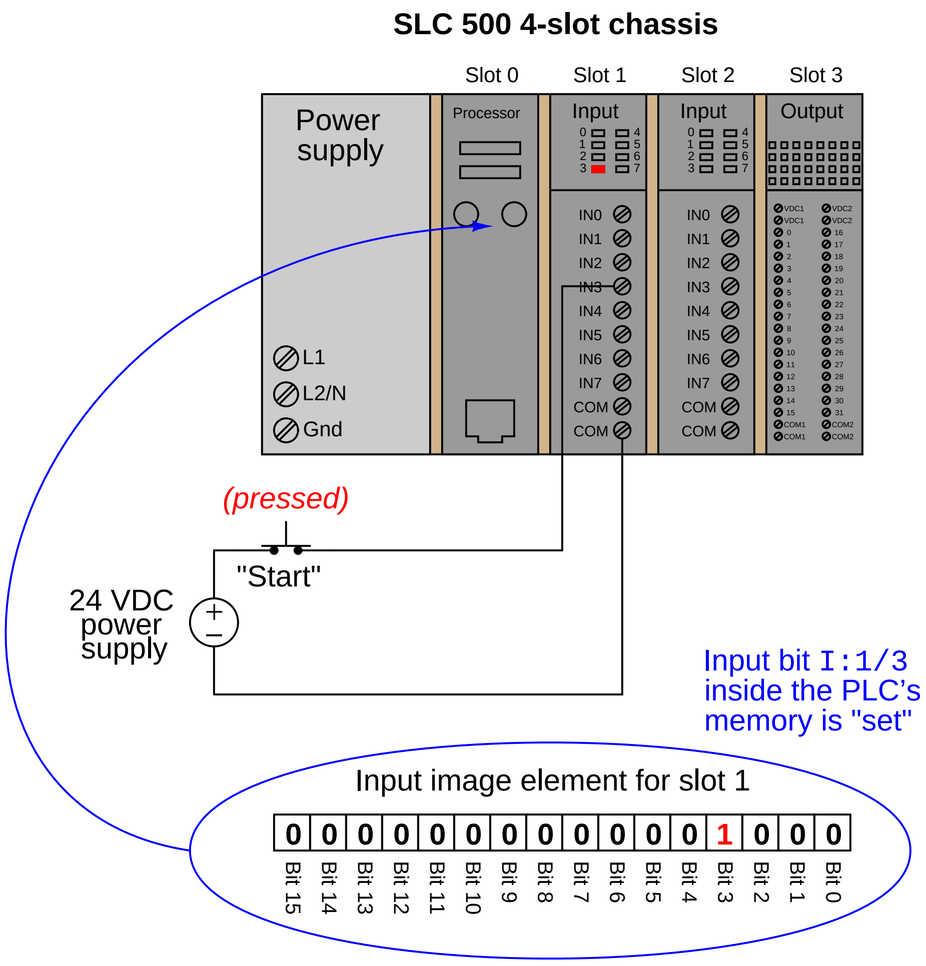
[3]This is not to say one *cannot* specify a particular bit in an otherwise whole word. In fact, this is one of the powerful advantages of Allen-Bradley's addressing scheme: it gives you the ability to precisely specify portions of data, even if that data is not generally intended to be portioned into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

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

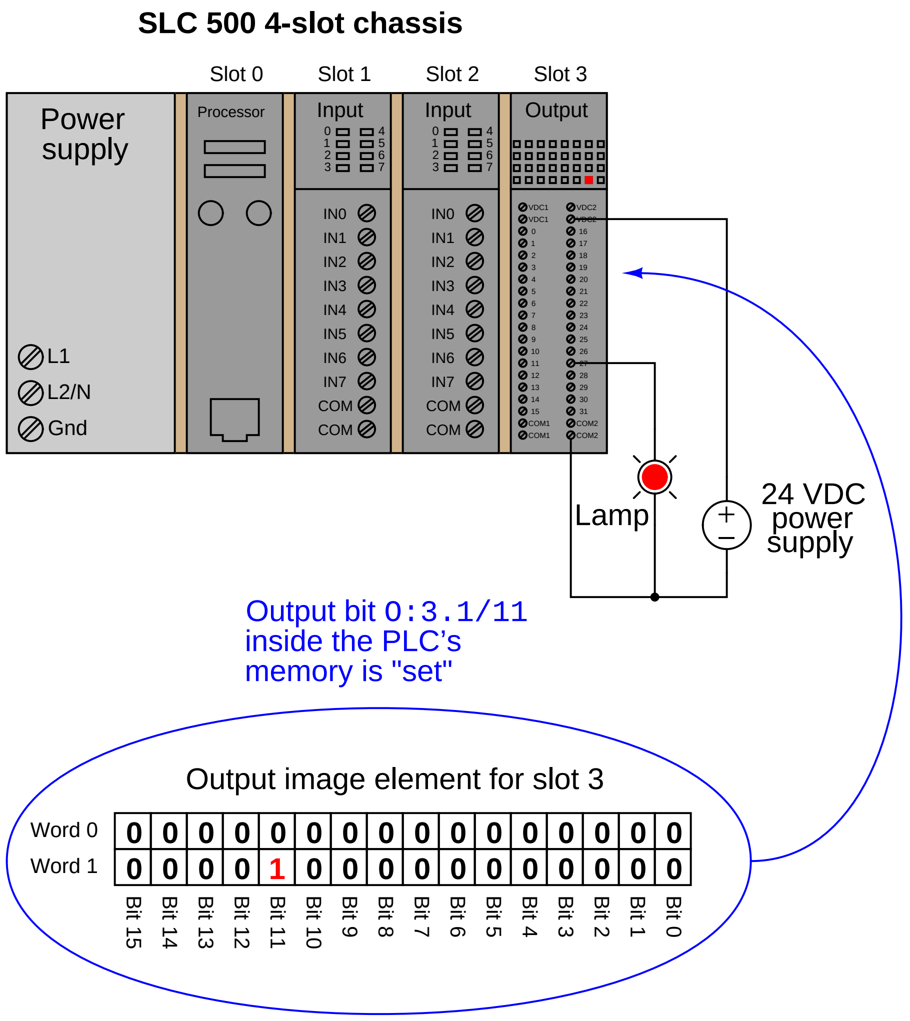
[5]Referencing the Allen-Bradley engineering literature, we see that the accumulator word may alternatively be addressed by number rather than by mnemonic, `T4:2.2` (word 2 being the accumulator word in the timer data structure), and that the "done" bit may be alternatively addressed as `T4:2.0/13` (bit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

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

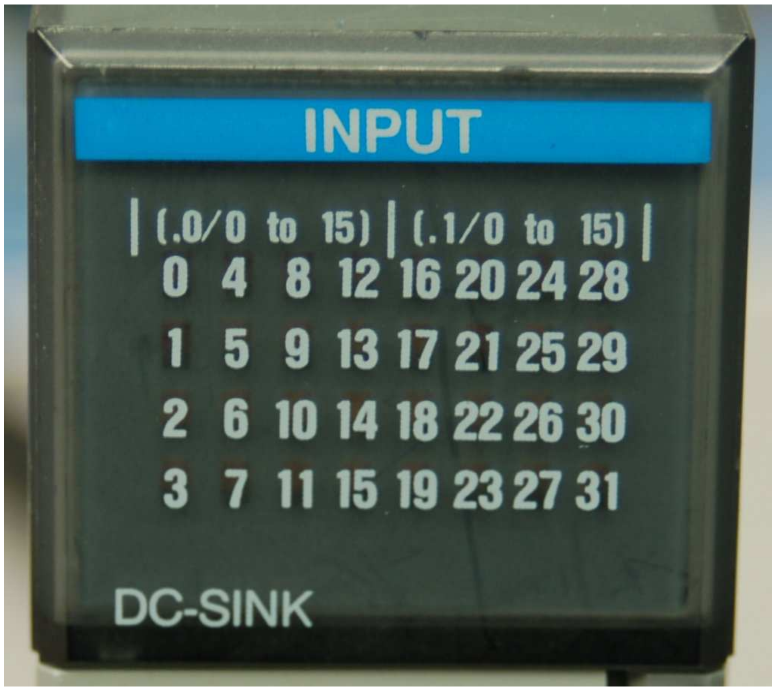
To illustrate the relationship between physical I/O and bits in the PLC's memory, consider this example of an Allen-Bradley SLC 500 PLC, showing one of its discrete input channels energized (the switch being used as a "Start" switch for an electric motor):



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



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



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

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

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

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

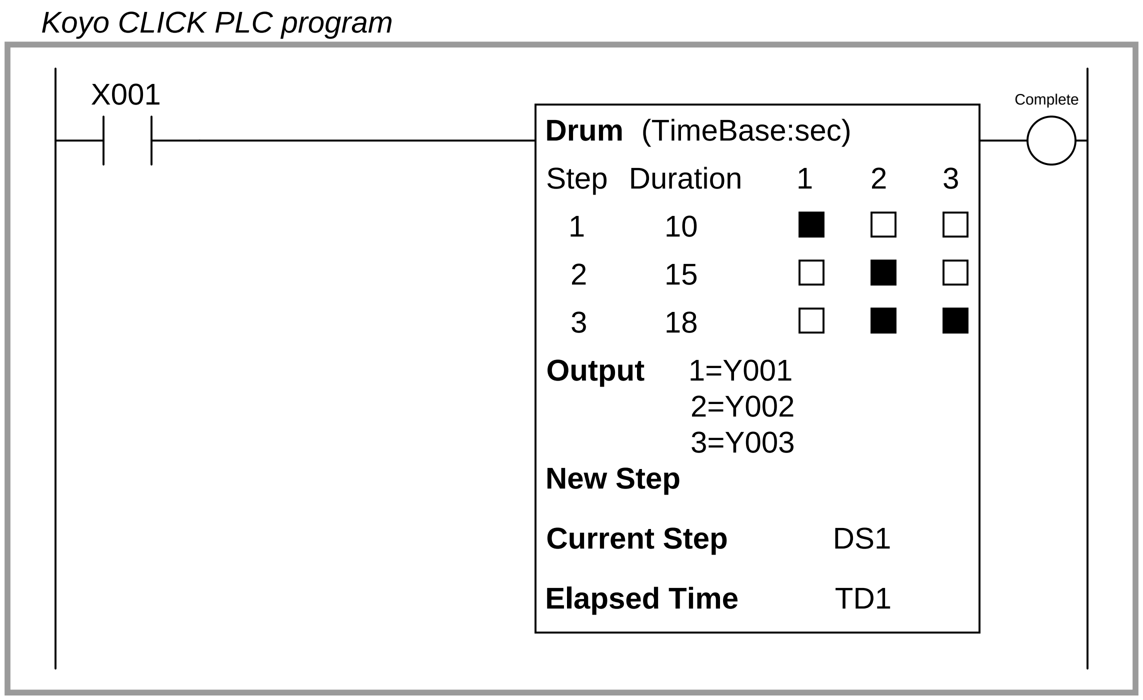
name I/O channels in ways that are meaningful to the system as a whole. To use our simple motor "Start" switch example, it is now possible for the programmer to designate input `I:1/3` (an example of a *direct address*) as "`Motor_start_switch`" (an example of a *symbolic address*) within the program,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

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

Having introduced Allen-Bradley's addressing notation for SLC 500 model PLCs, I will now abandon it in favor of the modern convention of symbolic addressing throughout the rest of this chapter, so as to avoid making the programming examples brand- or model-specific. Each data point within my PLC programs will bear its own tag name rather than a direct (channel-based) address label.

5.3 Koyo "drum" sequencer instructions

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

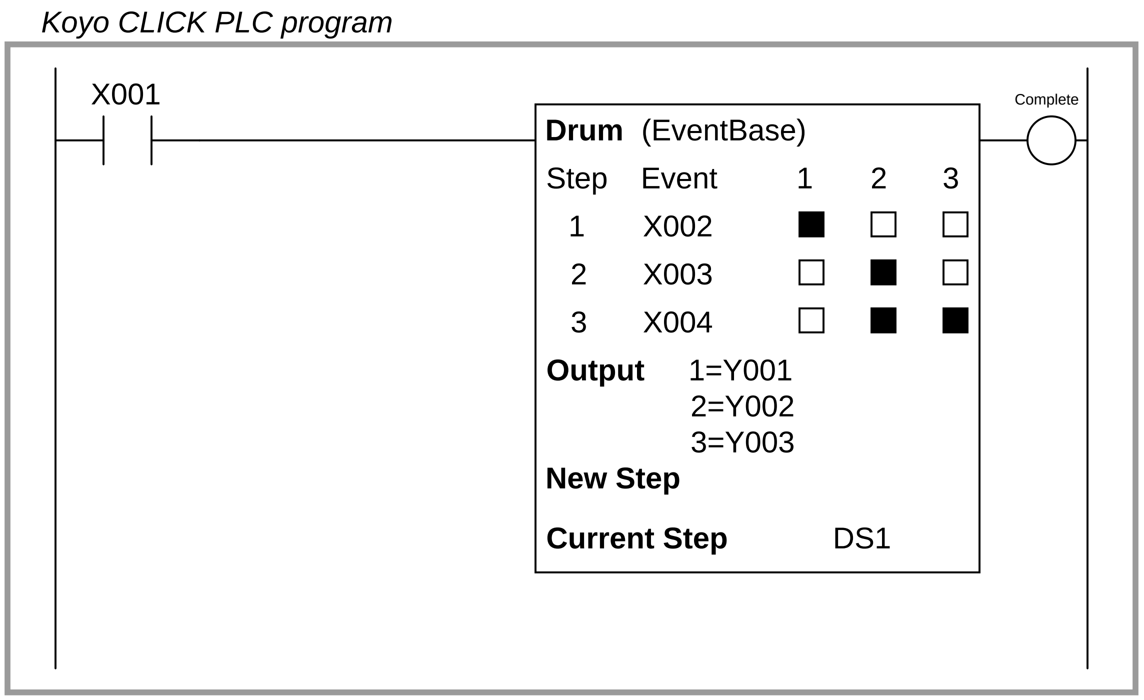


The three-by-three grid of squares represent steps in the sequence and bit states for each step. Rows represent steps, while columns represent output bits written by the drum instruction. In this particular example, a three-step sequence proceeds at the command of a single input (`X001`), and the drum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

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

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

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

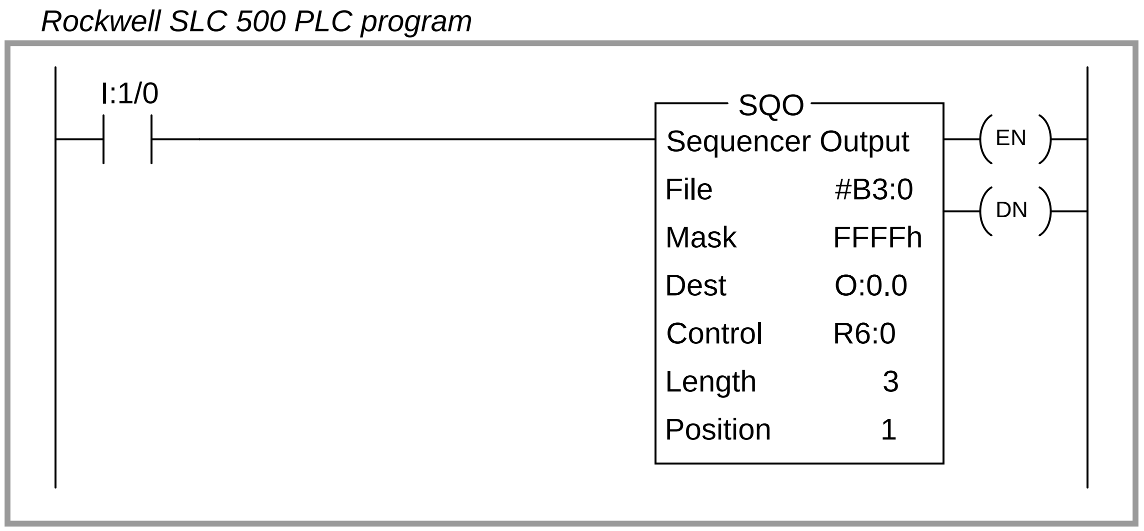


Here, a three-step sequence proceeds when enabled by a single input (`X001`), with the drum instruction's advance from one step to the next proceeding only as the different event condition bits become set. When the input is active, the drum proceeds through its sequence when each event condition is met.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

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

5.4 Allen-Bradley sequencer instructions

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



You will notice there are no colored squares inside the SQO instruction box to specify when certain bits are set or reset throughout the sequence, in contrast to the simplicity of the Koyo PLC's drum instruction. Instead, the Allen-Bradley SQO instruction is told to read a set of 16-bit words beginning at a location in the PLC's memory arbitrarily specified by the programmer, one word at a time. It steps to the next word in that set of words with each new position (step) value. This means Allen-Bradley sequencer instructions rely on the programmer already having pre-loaded an area of the PLC's memory with the necessary 1's and 0's defining the sequence. This makes the Allen-Bradley sequencer instruction more challenging for a human programmer to interpret because the bit states are not explicitly shown inside the SQO instruction box, but it also makes the sequencer far more flexible in that these bits are not fixed parameters of the SQO instruction and therefore may be dynamically altered as the PLC runs. With the Koyo drum instruction, the assigned output states are part of the instruction itself, and are therefore fixed once the program is downloaded to the PLC (i.e. they cannot be altered without editing and re-loading the PLC's program). With the Allen-Bradley, the on-or-off bit states for the sequence may be freely altered[7] during run-time. This is a very useful feature in recipe-control applications, where the recipe is subject to change at the whim of production personnel, and they would rather not have to rely on a technician or an engineer to re-program the PLC for each new recipe.

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

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

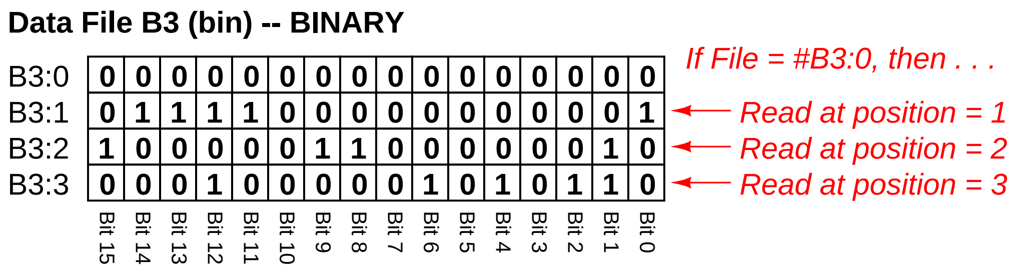
Sequencer instructions in Allen-Bradley PLCs use a notation called *indexed addressing* to specify the locations in memory for the set of 16-bit words it will read. In the example shown above, we see the "File" parameter specified as `#B3:0`. The "#" symbol tells the instruction that this is a *starting*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

The bits read from each indexed word in the sequence are compared against a static mask[8] specifying which bits in the indexed word are relevant. At each position, only these bits are written to the destination address.

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

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

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

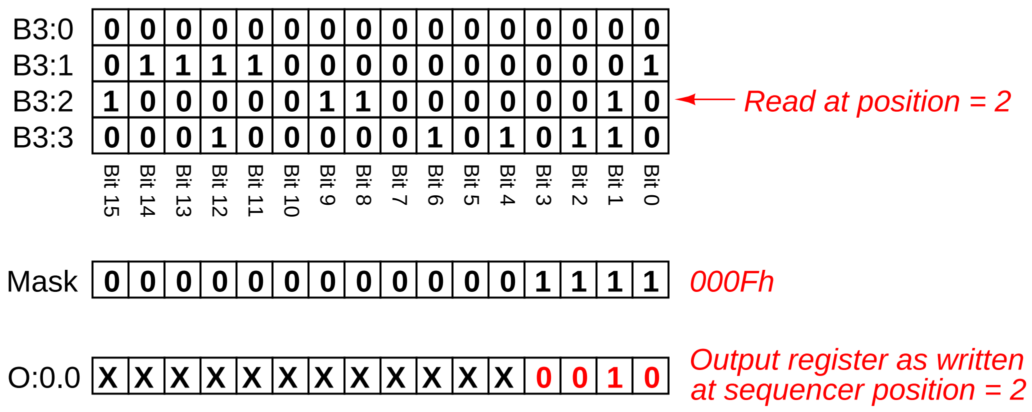


	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
B3:0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B3:1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1
B3:2	1	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0
B3:3	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0

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

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

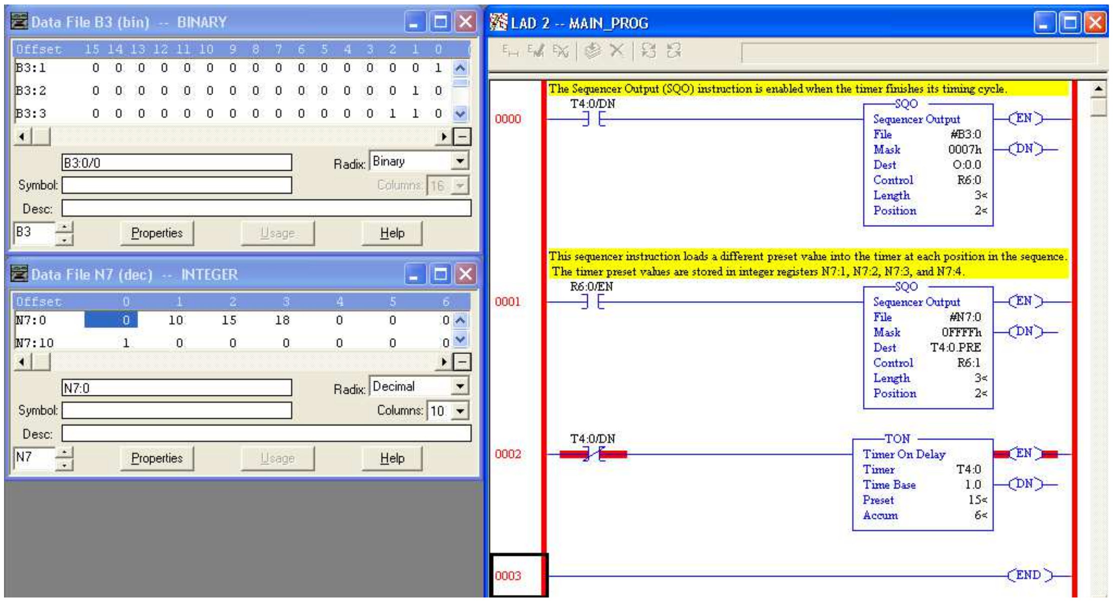
Let's see what would happen with an SQO instruction having a mask value of 000Fh, starting from file index `#B3:0`, and writing to a destination that is output register `O:0.0`, given the bit array values in file `B3` shown above:



	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
B3:0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B3:1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1
B3:2	1	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0
B3:3	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	0
Mask	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
O:0.0	X	X	X	X	X	X	X	X	X	X	X	X	0	0	1	0

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

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



Data File B3 (bin) -- BINARY

Offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B3:1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
B3:2	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
B3:3	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0

B3:0/0 Radix: Binary
Symbol:
Columns: 16
Desc:
B3

Data File N7 (dec) -- INTEGER

Offset	0	1	2	3	4	5	6
N7:0	0	10	15	18	0	0	0
N7:10	1	0	0	0	0	0	0

N7:0 Radix: Decimal
Symbol:
Columns: 10
Desc:
N7

LAD 2 -- MAIN_PROG

0000 The Sequencer Output (SQO) instruction is enabled when the timer finishes its timing cycle.
T4:0/DN — SQO Sequencer Output; File #B3:0; Mask 0007h; Dest O:0.0; Control R6:0; Length 3<; Position 2< — (EN) (DN)

0001 This sequencer instruction loads a different preset value into the timer at each position in the sequence. The timer preset values are stored in integer registers N7:1, N7:2, N7:3, and N7:4.
R6:0/EN — SQO Sequencer Output; File #N7:0; Mask 0FFFFh; Dest T4:0.PRE; Control R6:1; Length 3<; Position 2< — (EN) (DN)

0002 T4:0/DN — TON Timer On Delay; Timer T4:0; Time Base 1.0; Preset 15<; Accum 6< — (EN) (DN)

0003 (END)

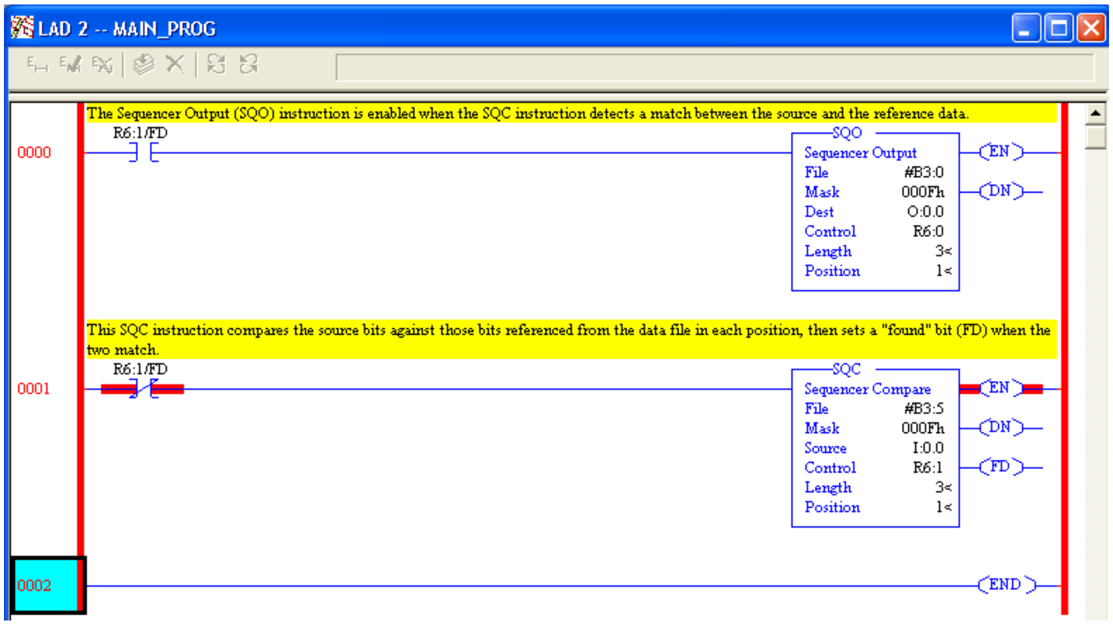
The first SQO instruction reads bits in the B3 file array, sending only the three least-significant of them to the output register `O:0.0` (as specified by the 0007h mask value). The second SQO instruction reads integer number values from elements of the `N7` integer file and places them into the "preset"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

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

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

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

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



The Sequencer Output (SQO) instruction is enabled when the SQC instruction detects a match between the source and the reference data.

```
0000   R6:1/FD          SQO
                        Sequencer Output        EN
                        File        #B3:0
                        Mask        000Fh       DN
                        Dest        O:0.0
                        Control     R6:0
                        Length      3<
                        Position    1<
```

This SQC instruction compares the source bits against those bits referenced from the data file in each position, then sets a "found" bit (FD) when the two match.

```
0001   R6:1/FD          SQC
                        Sequencer Compare       EN
                        File        #B3:5
                        Mask        000Fh       DN
                        Source      I:0.0
                        Control     R6:1        FD
                        Length      3<
                        Position    1<

0002                                            END
```

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

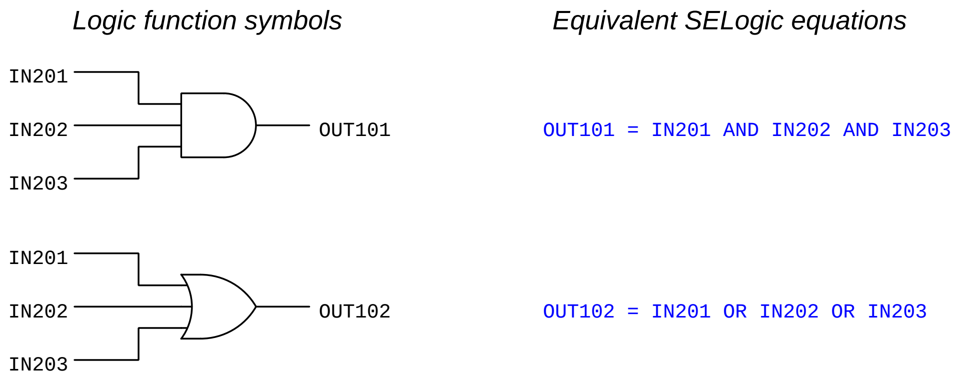
Lastly, Allen-Bradley PLCs offer a third sequencing instruction called *Sequencer Load* (SQL), which performs the opposite function as the Sequencer Output (SQO). An SQL instruction takes data from a designated source and writes it into an indexed register according to a position count value, rather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

5.5 SELogic control equations

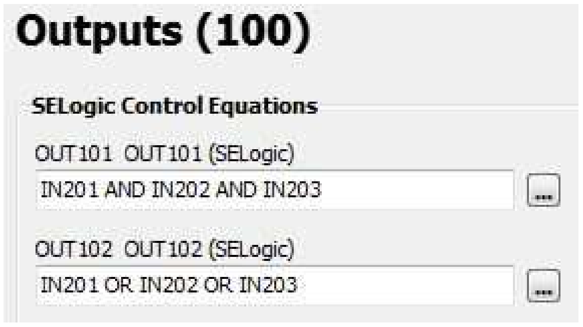
Some programmable logic controllers use Boolean-style equations to describe logical functions. A company called Schweitzer Engineering Laboratories (SEL) manufactures control equipment for the electric power industry, including a wide range of *protective relays* and *programmable automation controllers*,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

5.5.1 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:


Logic function symbols
Equivalent SELogic equations
IN201
IN202
IN203
OUT101
OUT101 = IN201 AND IN202 AND IN203
IN201
IN202
IN203
OUT102
OUT102 = IN201 OR IN202 OR IN203

SEL programming software[9] 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:



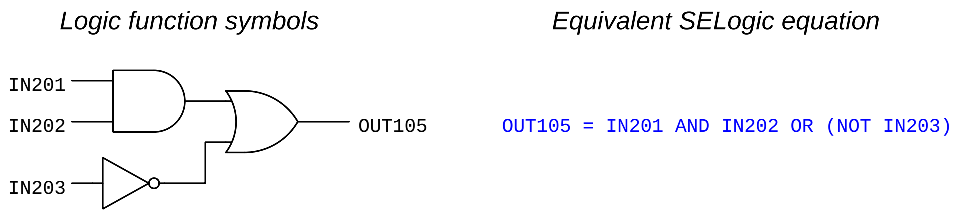
OUT101 OUT101 (SELogic)
IN201 AND IN202 AND IN203

OUT102 OUT102 (SELogic)
IN201 OR IN202 OR IN203

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.

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

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


Logic function symbols
Equivalent SELogic equation
IN201
IN202
IN203
OUT105
OUT105 = IN201 AND IN202 OR (NOT IN203)

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:

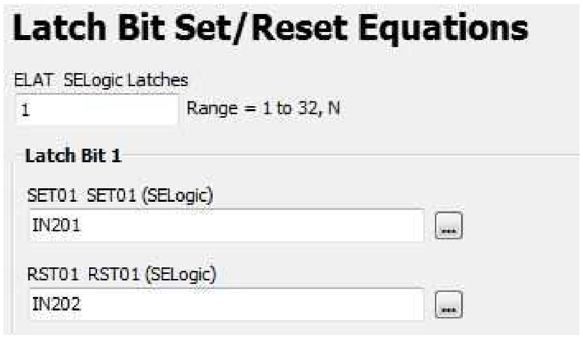
```
OUT102 = IN205 AND IN208 # THIS IS AN AND FUNCTION
```

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

5.5.2 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:


Latch Bit Set/Reset Equations

ELAT SELogic Latches
1 Range = 1 to 32, N

Latch Bit 1

SET01 SET01 (SELogic)
IN201

RST01 RST01 (SELogic)
IN202

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:

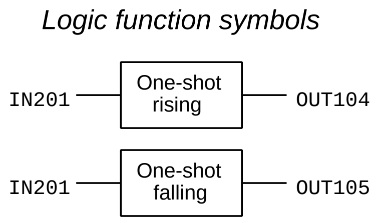
```
ELAT = 1
SET01 = IN201
RST01 = IN202
```

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
```

5.5.3 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:

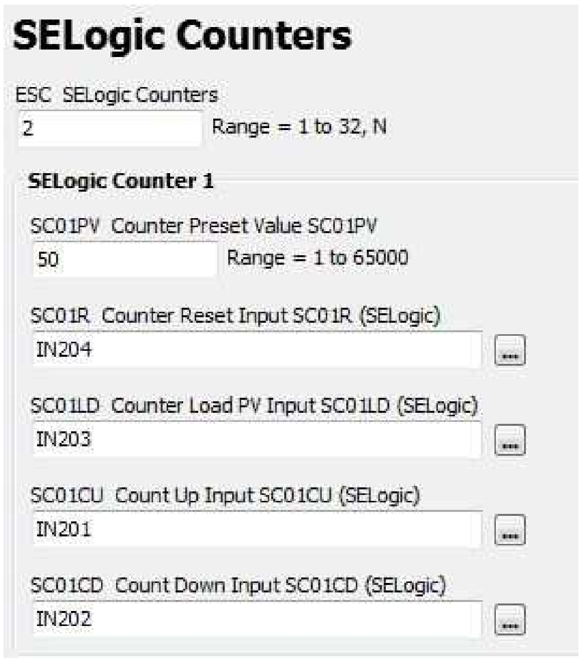


Equivalent SELogic equations

```
OUT104 = R_TRIG IN201

OUT105 = F_TRIG IN201
```

5.5.4 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:



SELogic Counters

ESC SELogic Counters
2 Range = 1 to 32, N

SELogic Counter 1

SC01PV Counter Preset Value SC01PV
50 Range = 1 to 65000

SC01R Counter Reset Input SC01R (SELogic)
IN204

SC01LD Counter Load PV Input SC01LD (SELogic)
IN203

SC01CU Count Up Input SC01CU (SELogic)
IN201

SC01CD Count Down Input SC01CD (SELogic)
IN202

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
```

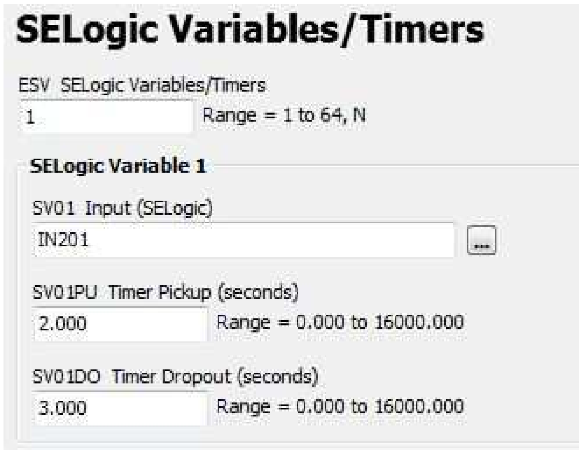
Once configured, the accumulated value of the first counter instruction is addressed simply as `SC01`. Discrete outputs based on the attainment of certain count values may be driven by comparison statements such as `=`, `>`, and `<`. For example, to activate output `OUT107` whenever the first counter's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

5.5.5 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:


SELogic Variables/Timers

ESV SELogic Variables/Timers
1 Range = 1 to 64, N

SELogic Variable 1

SV01 Input (SELogic)
IN201

SV01PU Timer Pickup (seconds)
2.000 Range = 0.000 to 16000.000

SV01DO Timer Dropout (seconds)
3.000 Range = 0.000 to 16000.000

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

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

```
OUT105 = SV01T
```

Chapter 6

Questions

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

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

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

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

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

General challenges following tutorial reading

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

General follow-up challenges for assigned problems

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

General follow-up challenges for experiments or projects

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

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

6.1 Conceptual reasoning

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

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

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

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

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

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

6.1.1 Reading outline and reflections

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

Francis Bacon's advice is a blueprint for effective education: reading provides the learner with knowledge, writing focuses the learner's thoughts, and critical dialogue equips the learner to confidently communicate and apply their learning. Independent acquisition and application of knowledge is a powerful skill, well worth the effort to cultivate. To this end, students should read these educational resources closely, write their own outline and 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:

$\boxed{\surd}$ Briefly OUTLINE THE TEXT, as though you were writing a detailed Table of Contents. Feel free to rearrange the order if it makes more sense that way. Prepare to articulate these points in detail and to answer questions from your classmates and instructor. Outlining is a good self-test of thorough reading because you cannot outline what you have not read or do not comprehend.

$\boxed{\surd}$ 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.

$\boxed{\surd}$ 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.

$\boxed{\surd}$ 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.

$\boxed{\surd}$ 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.

$\boxed{\surd}$ Specifically identify any points you found CONFUSING. The reason for doing this is to help diagnose misconceptions and overcome barriers to learning.

6.1.2 Foundational concepts

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

$\boxed{???}$

$\boxed{???}$

$\boxed{???}$

$\boxed{???}$

$\boxed{???}$

$\boxed{???}$

$\boxed{???}$

$\boxed{???}$

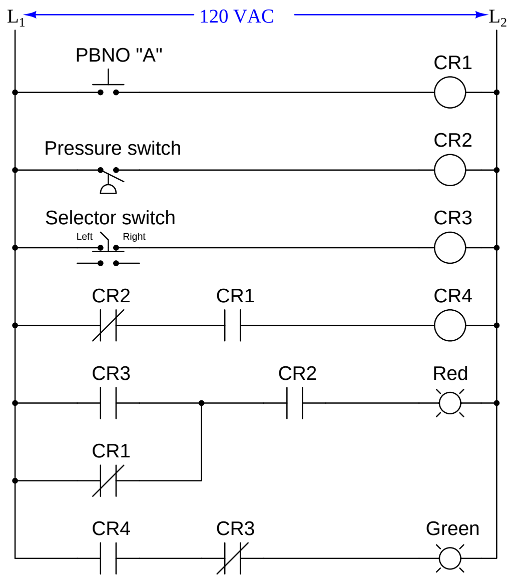
$\boxed{???}$

$\boxed{???}$

$\boxed{???}$

6.1.3 Relay ladder logic analogy for a PLC

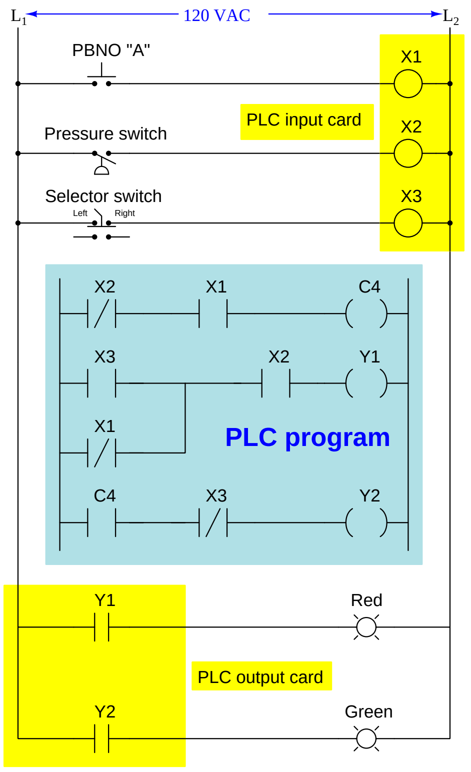
Analyze the status of all relay contacts and lamps in this hard-wired relay "ladder logic" control circuit:



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:



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?

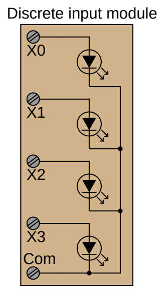
Challenges

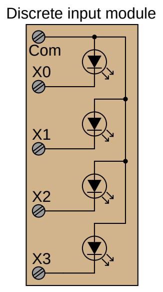
- A good problem-solving technique to apply in both diagrams is *annotation*, where you indicate the presence of continuity and power versus non-continuity/unpowered. In PLC programs this usually appears in the form of color-highlighting surrounding each instruction symbol (virtual contact or virtual coil). Annotate these diagrams to show discrete logic states for each of their elements.

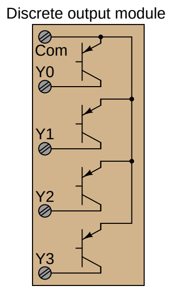
6.1.4 Sourcing versus sinking PLC I/O

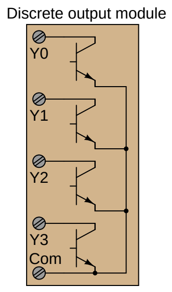
Discrete (on/off) I/O for PLCs often works on AC (alternating current) power. AC input circuitry usually consists of an optocoupler (LED) with rectification and a large dropping resistor to allow 120 volt AC operation. AC output circuitry usually consists of TRIACs. Explain how both of these technologies work.

DC I/O for a PLC generally consists of optocoupled LEDs for inputs and bipolar transistors for outputs. Some examples are shown in the following schematics. Note carefully the different variations:









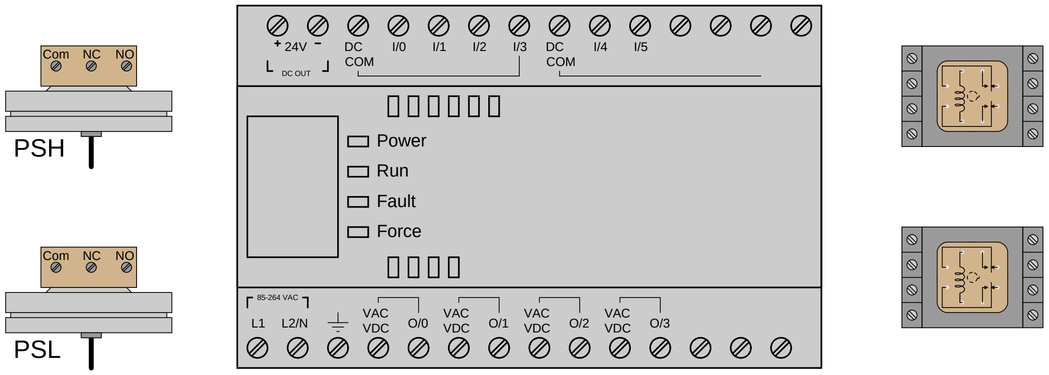
Determine for each of these input and output module types, whether they would be properly designated *sourcing* or *sinking*.

Challenges

- Determine how real input and output devices (e.g. switches, solenoid coils) would need to be connected to the I/O terminals of these modules.

6.1.5 Sketching wires to PLC discrete I/O

Sketch the wires necessary to connect two pressure switches and two relay coils to the following Allen-Bradley MicroLogix 1000 PLC (model 1761-L10BWA, with 6 discrete DC inputs either sourcing or sinking, and 4 discrete relay contact outputs). Be sure to wire the two switches so they *source* current to the PLC's inputs (the low-pressure switch to `I/2` and the high-pressure switch to `I/5`, normally-open contacts on both) and wire the relay coils so the PLC *sources* current to them (`O/0` and `O/1`):

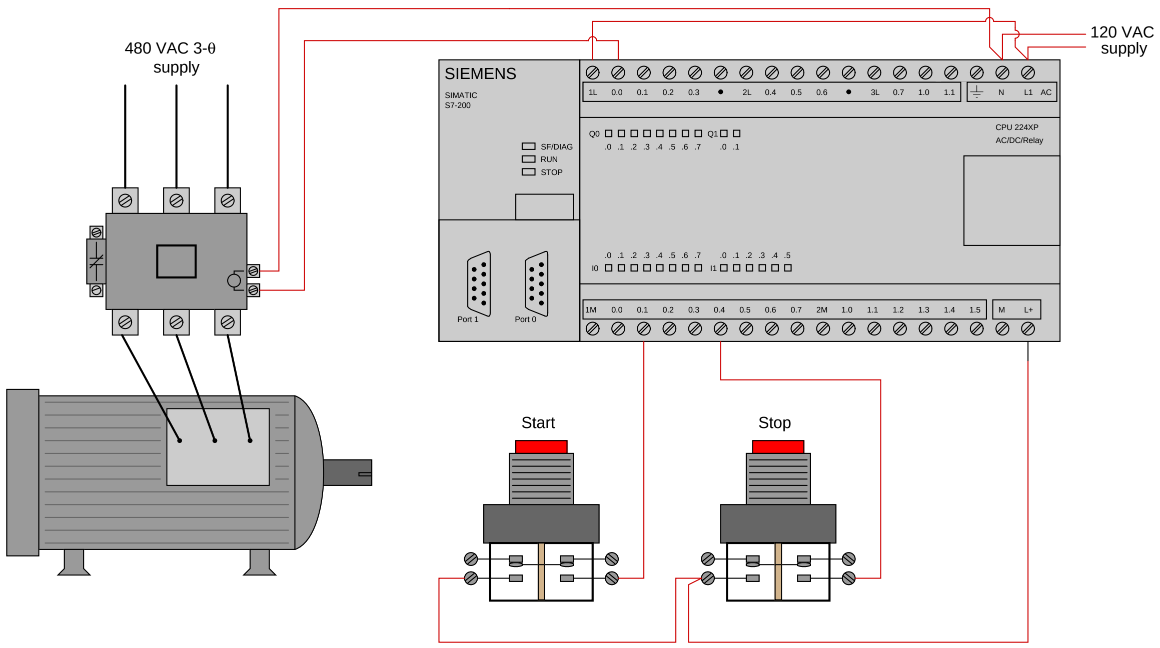


Challenges

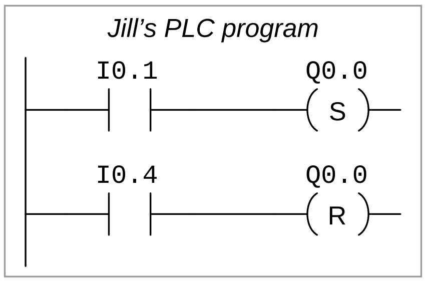
- Define "sinking" and "sourcing" as these terms apply to PLC I/O terminals.

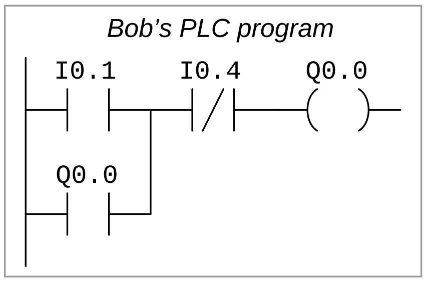
6.1.6 Two different motor control programs

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:





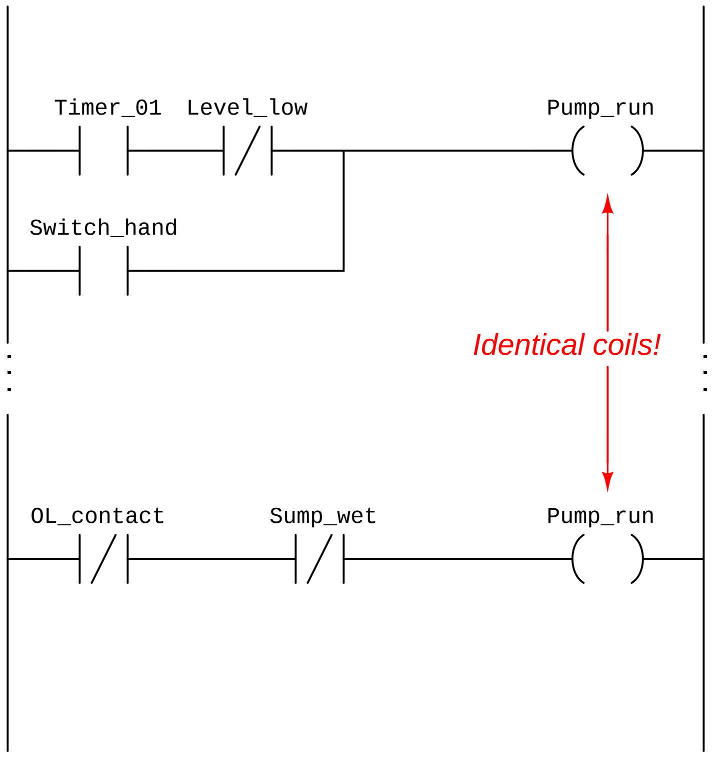
Explain how both of these PLC programs function properly to control the starting and stopping of the electric motor.

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.

6.1.7 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

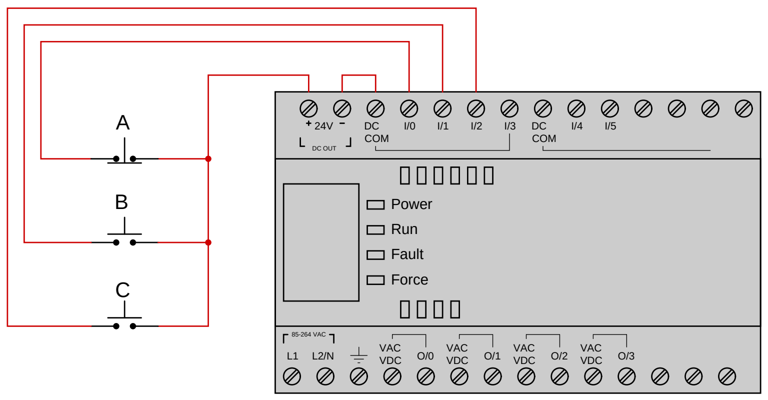
Challenges

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

6.1.8 Determining bit statuses from switch conditions

PLC #1

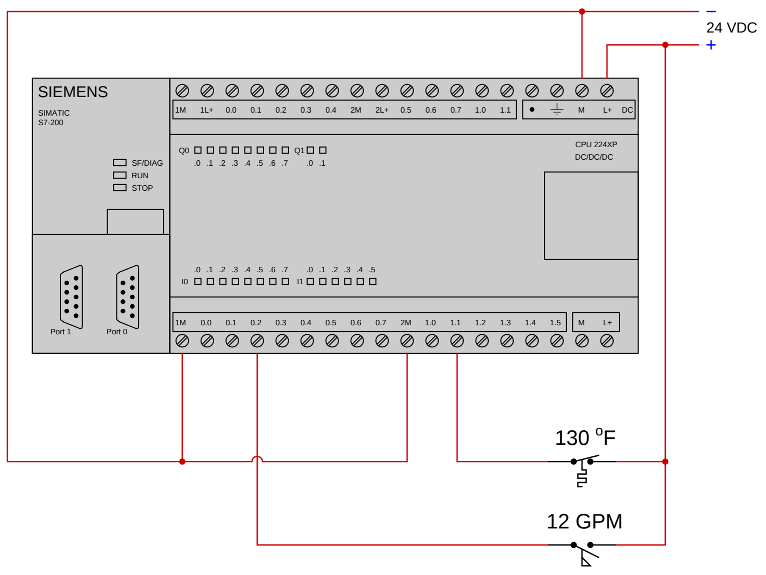
Suppose we have an Allen-Bradley MicroLogix 1000 PLC connected to three momentary-contact pushbutton switches as shown in this illustration:



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

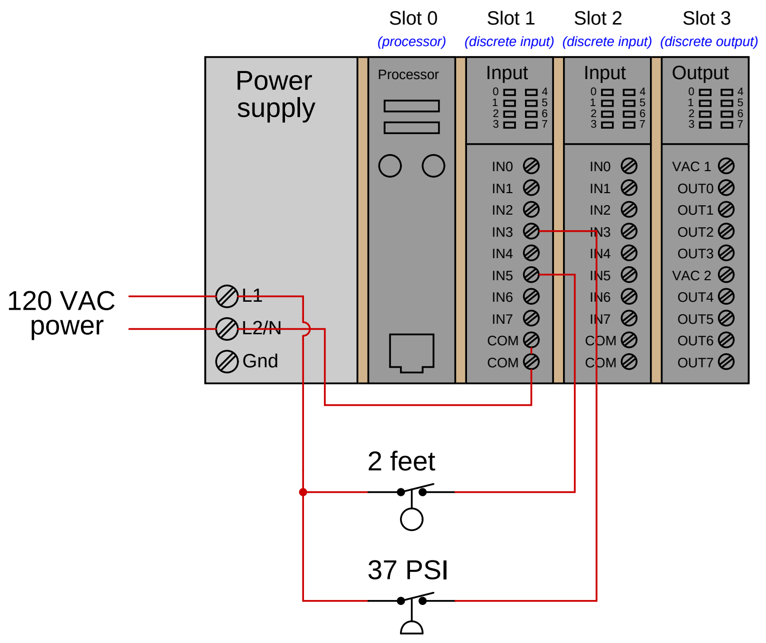
Suppose we have a Siemens S7-200 PLC connected to two process switches as shown in this illustration:



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

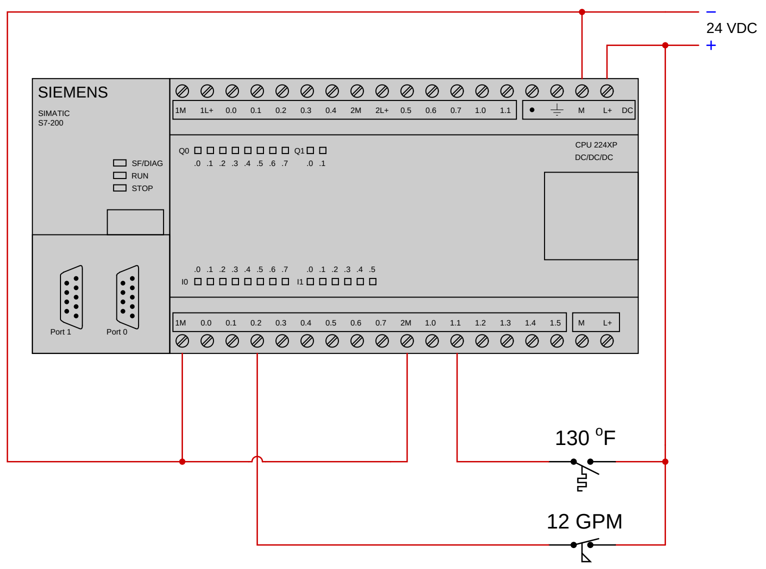
Suppose we have an Allen-Bradley SLC 500 PLC connected to two process switches as shown in this illustration:



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

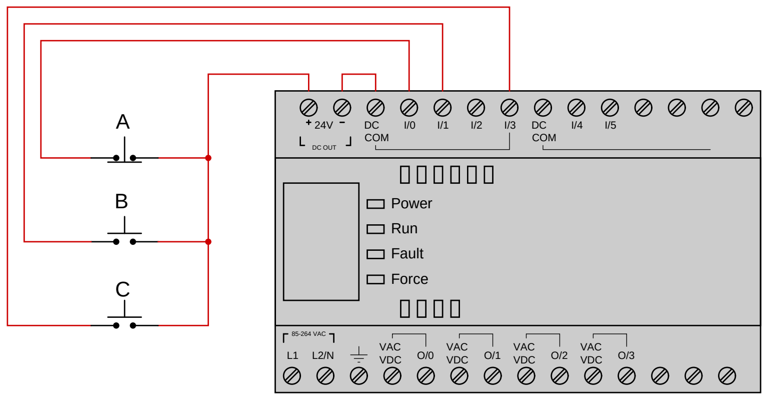
Suppose we have a Siemens S7-200 PLC connected to two process switches as shown in this illustration:



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:



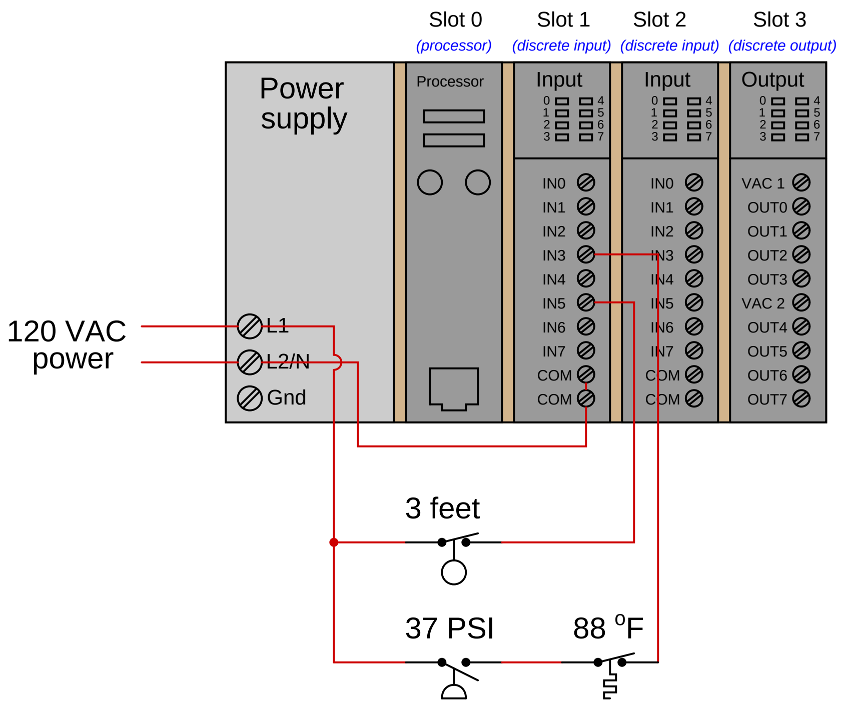
Determine the bit statuses of `I:0/0`, `I:0/1`, and `I:0/3` when switch A is pressed, switch B is unpressed (released), and switch C is pressed.

Challenges

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

6.1.9 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:

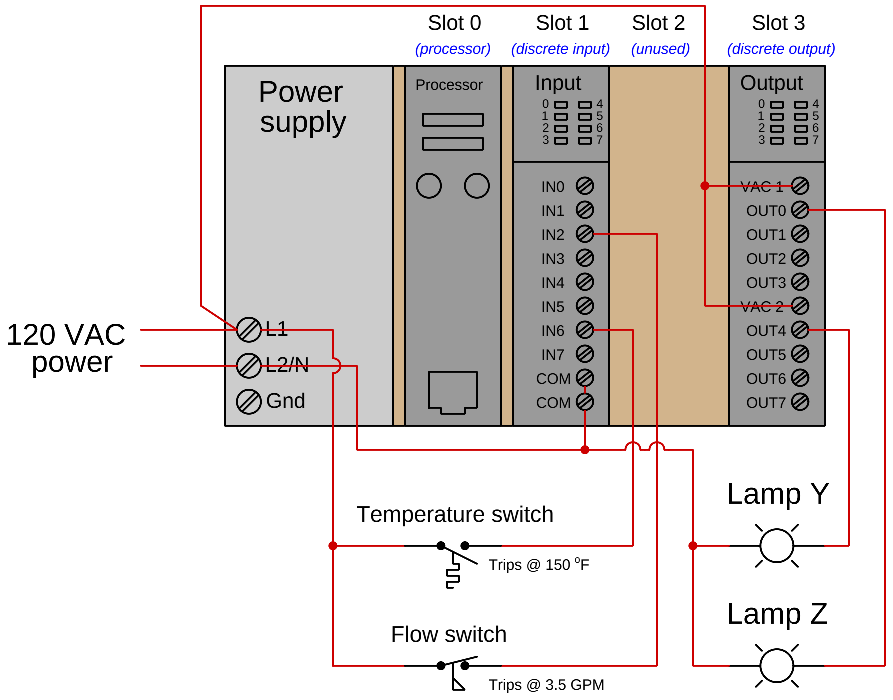
- `I:1/3` = 1
- `I:1/5` = 0

Challenges

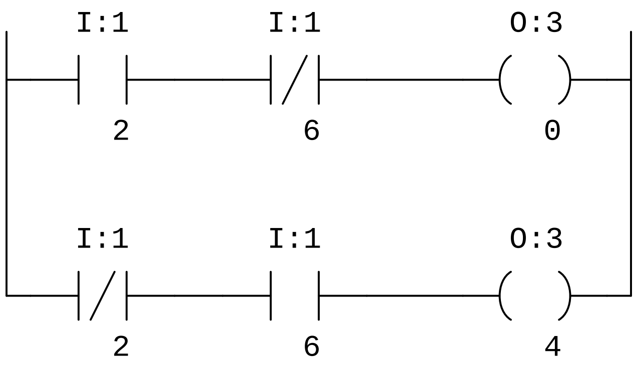
- What do the numbers "1" and "3" mean for input bit `I:1/3` in this Allen-Bradley PLC?
- What do the numbers "1" and "5" mean for input bit `I:1/5` in this Allen-Bradley PLC?
- How would you specify the bit address for the fourth channel on the other input card of this PLC?

6.1.10 Determining color highlighting from switch conditions

Suppose we have an Allen-Bradley model "SLC 500" PLC connected to a pair of momentary-contact pushbutton switches and light bulbs 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.

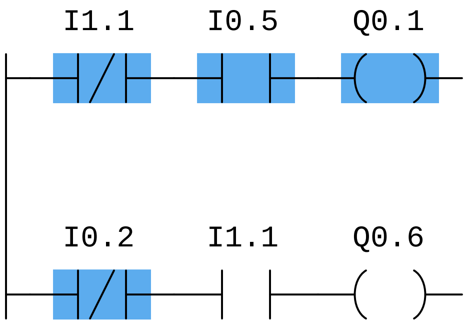
Challenges

- 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?

6.1.11 Determining bit statuses from color highlighting

PLC #1

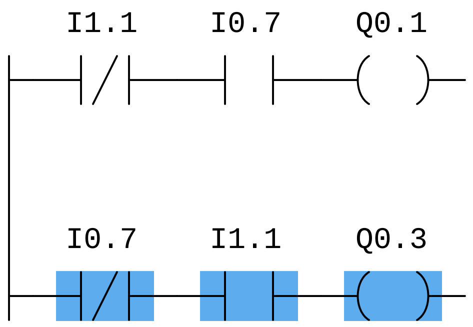
Examine this "live" display of a Siemens S7-300 PLC's program, and from this determine all bit statuses represented by the color highlighting in this ladder logic program:



- `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:



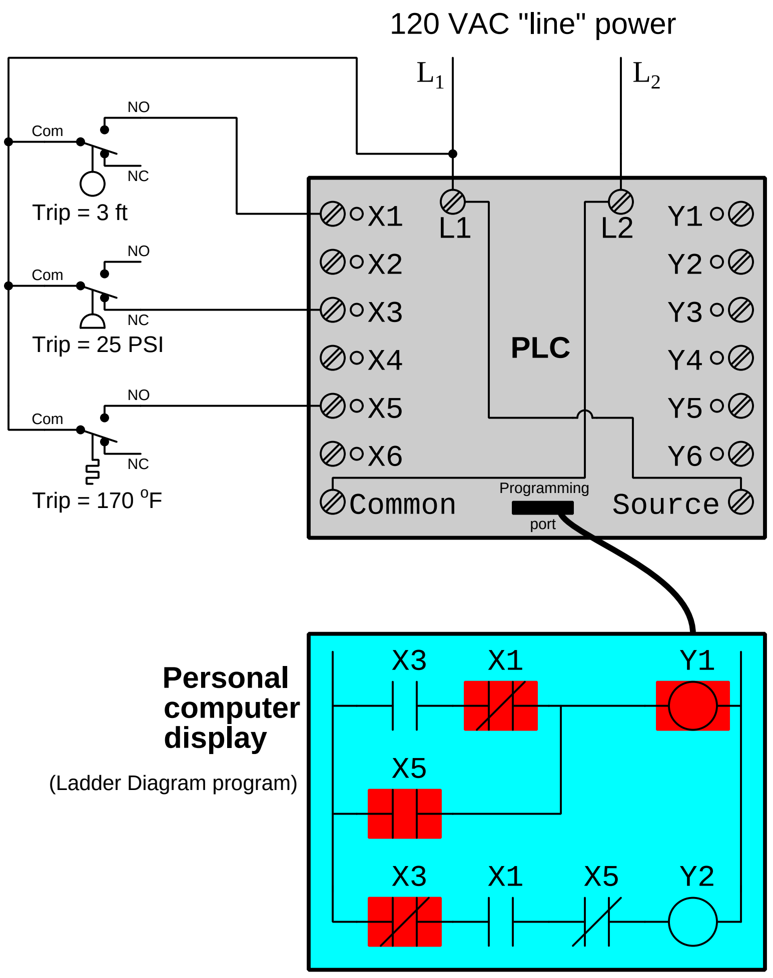
- `I0.7` =
- `I1.1` =
- `Q0.1` =
- `Q0.3` =

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?

6.1.12 Determining process switch stimuli from color highlighting

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



Based on the highlighting you see in the "live" PLC program display, determine as best you can the pressure and temperature stimulating each switch.

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

- X1 =
- X3 =
- X5 =
- Y1 =

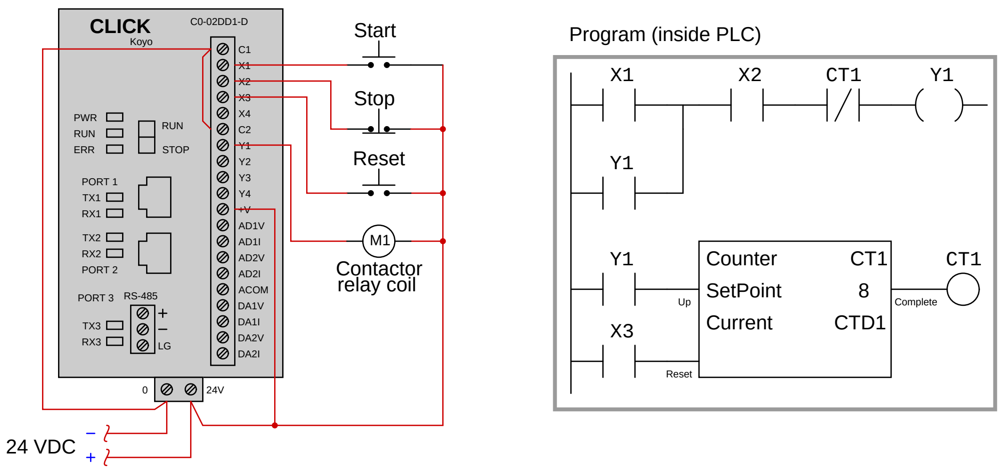
- `Y2` =

$\boxed{\text{Challenges}}$

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

6.1.13 Motor start-up limit counter

This Koyo "CLICK" PLC has been programmed to control the starting and stopping of an electric motor, including a *counter* instruction to prevent the motor from being started up more than a specified number of times:



Identify the counter instruction in the program shown, its input "connections", and also how the result of the counter reaching its pre-set limit forces the motor to stop. Also, determine the maximum number of times the motor may be started up, assuming the counter's current value goes to zero when the Reset button is pressed.

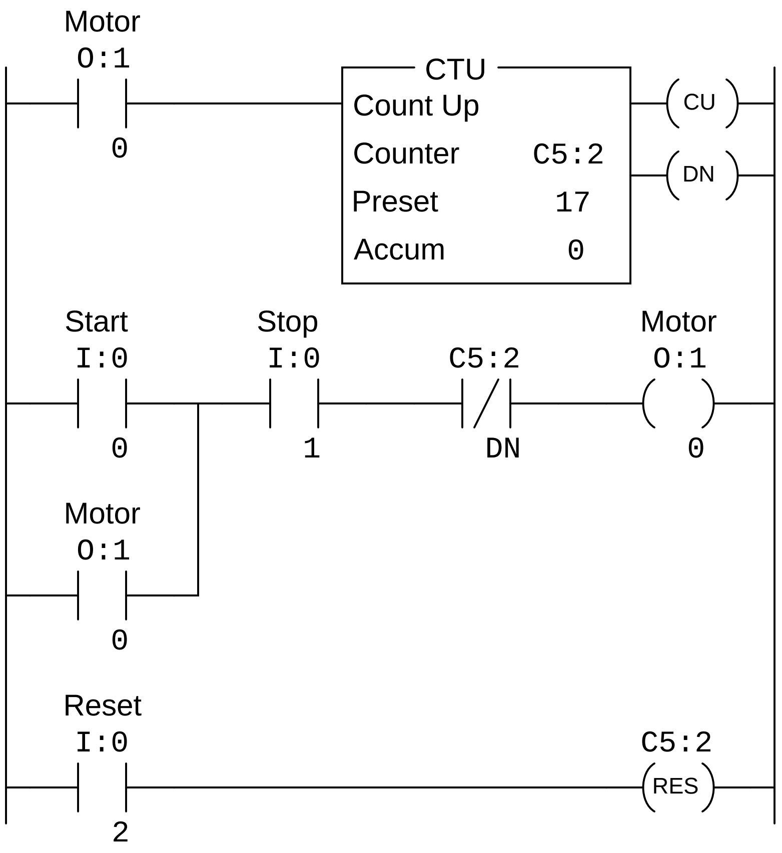
Finally, determine how to modify this PLC program so that the counter may be manually reset by the operator without requiring a separate pushbutton labeled "Reset".

Challenges

- If an operator presses the "Start" button multiple times while the motor is already running, do these button-presses get counted by the counter instruction, or do only the real motor start-up events get counted?
- What do you suppose the label "CTD1" represents inside the counter instruction?
- Note the number of times the bit `Y1` is referenced inside this PLC program: once in a coil instruction and twice in contact instructions. Is there any limit to how many times a bit address may be used in a PLC program?
- Describe the purpose of the first contact instruction labeled `Y1` in this program, explaining why it is often referred to as a *seal-in* contact.

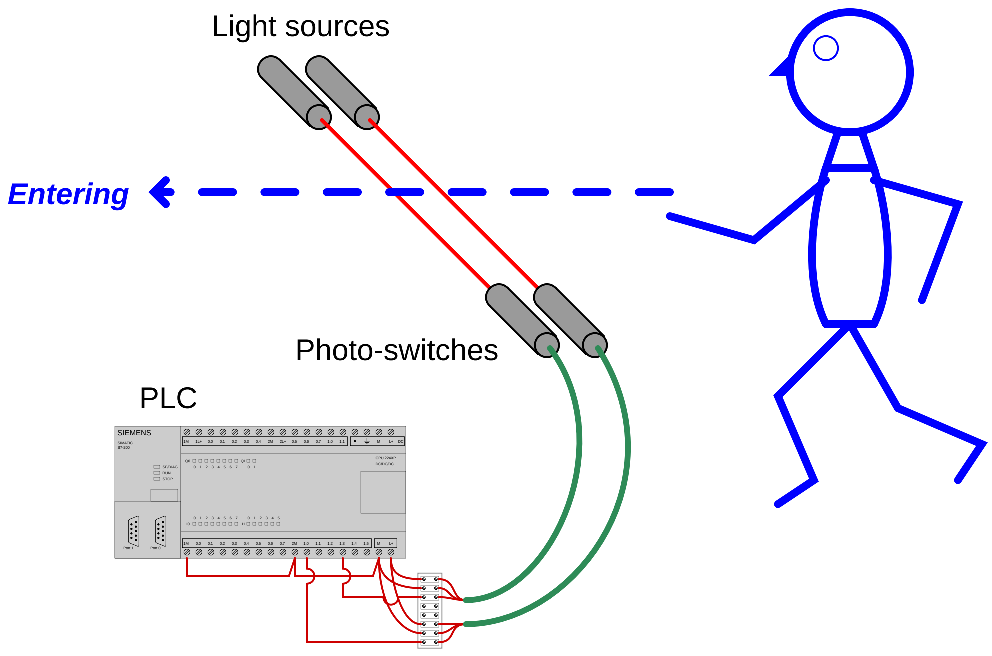
6.1.14 Allen-Bradley counter program

Analyze this Allen-Bradley PLC program and explain what it is supposed to do:

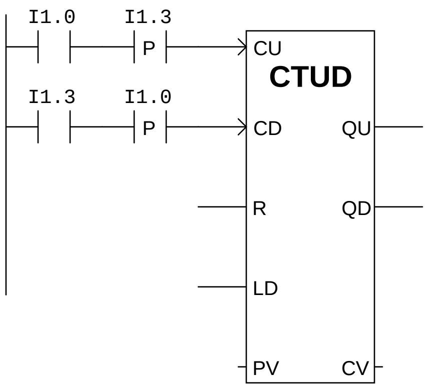


6.1.15 Room occupancy counter

This Siemens S7-200 PLC has been programmed to count the number of people in a room, by incrementing a counter every time a person enters through the doorway, and decrementing that same counter whenever someone exits through the same doorway. The two optical switches activate whenever their respective light beams are broken by someone passing through. Their horizontal separation is just a couple of inches – much less than the girth of a person's torso. The operating status of each switch is that it energizes the PLC input when the light beam is broken:



Examine the program in this PLC for counting people, and determine how it is able to differentiate between a person entering the room and a person leaving the room:

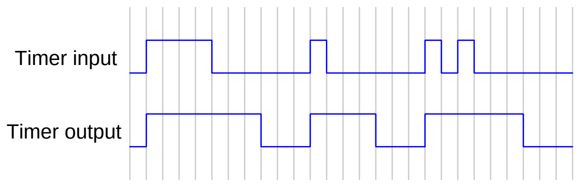


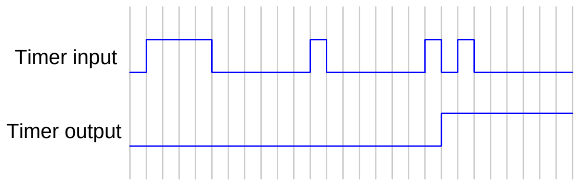
Challenges

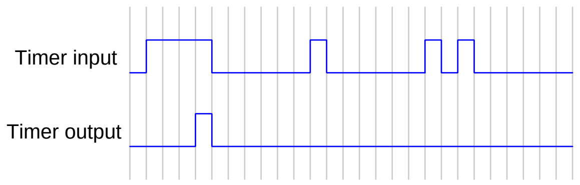
- Explain how a *timing diagram* of the switch states would be helpful in analyzing the operation of this PLC program.
- *Transition* (edge-detecting) functions are implemented in Allen-Bradley PLCs using the *one-shot rising* (OSR) instruction. Research how the OSR instruction is used, and how it differs from the "P" and "N" contacts shown in this Siemens PLC program.
- Will this system still function properly if the optical sensors are spaced farther apart than the width of a human body? Explain why or why not.

6.1.16 PLC timing diagrams

Identify the type of PLC timer instruction (i.e. *on-delay*, *off-delay*, or *retentive*) capable of producing the following timing diagrams, as well as each timer instruction's preset value in *seconds*. Assume each horizontal-axis division of this timing diagram represents one second:





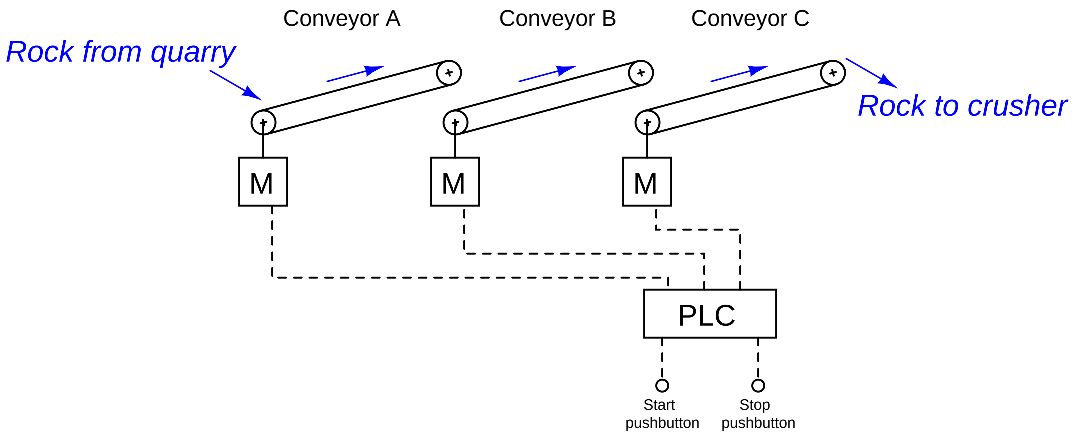


Challenges

- Modify the amount of time delay for any of these timer functions, and re-sketch the output.

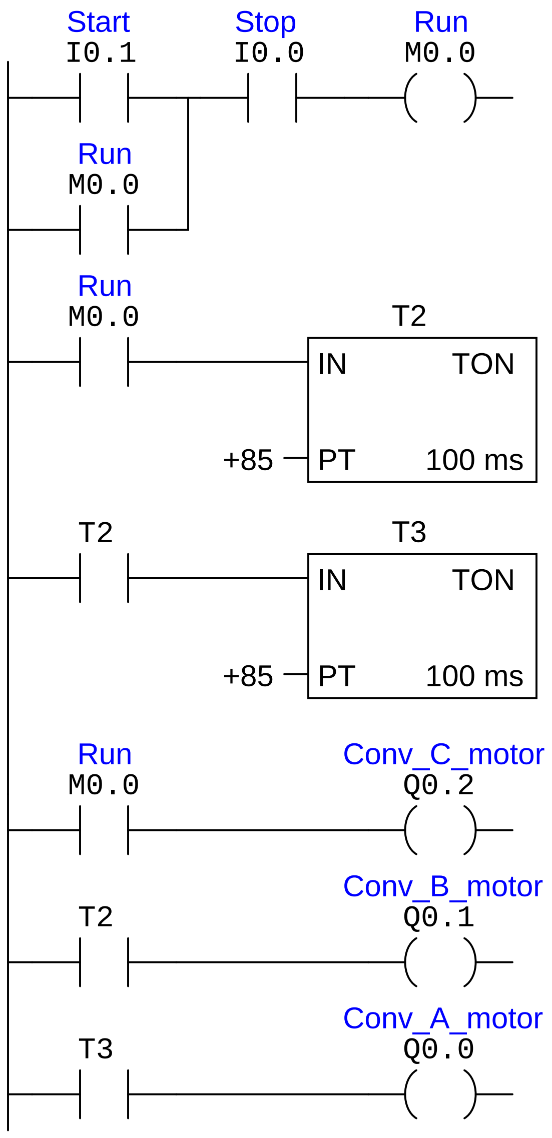
6.1.17 Sequenced-start conveyor belts

A gravel-crushing operation uses three long conveyor belts to move rock from the quarry to the crusher. The belts must be started up in a particular sequence to avoid overloading the electric motors driving them:



First, determine a start-up sequence that makes sense: which conveyor belt should start first, next, and last? What might happen if the sequence were reversed? Why not simply start all conveyor motors simultaneously?

This operation uses a Siemens S7 series PLC to control the three conveyor belts. Analyze this program and explain how it accomplishes the task of starting up the three conveyors in sequence:



Lastly, determine where you might add a contact instruction for an *emergency shutoff* safety switch, so that all three conveyors stop simultaneously if ever the safety switch is actuated.

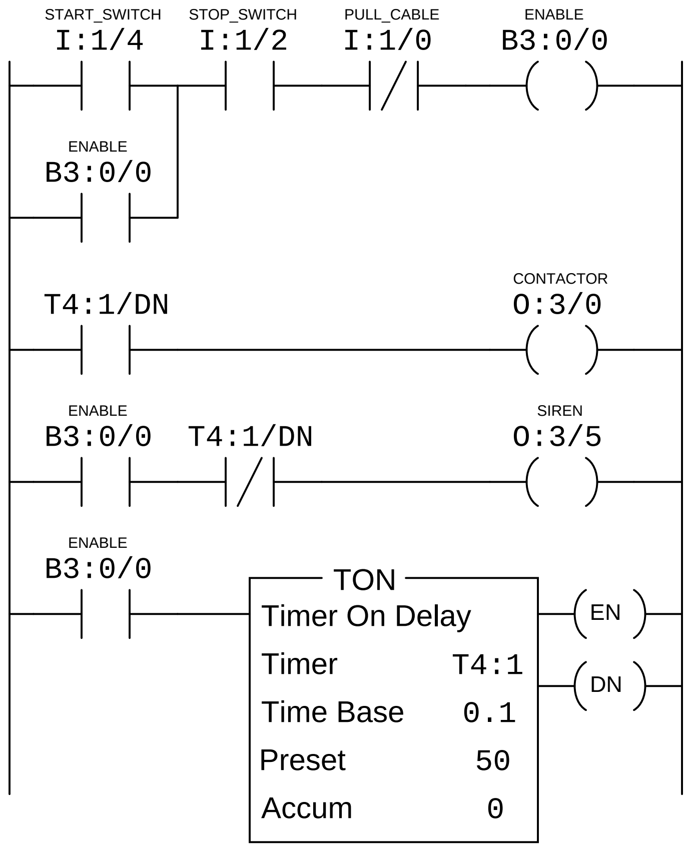
Challenges

- How long is the time delay between conveyor start-ups? How might this time delay be altered if needed?

- Suppose a warning siren were added to the system, sounding for a full 15 seconds before the first conveyor belt starts. How would you modify the PLC program to include this additional functionality?
- Suppose a technician uses the PLC's *force* utility to force bit `T2` to a "0" state. How will this affect the operation of the system? Could the consequences of this force be dangerous in any way?
- Suppose a technician uses the PLC's *force* utility to force bit `Q0.1` to a "0" state. How will this affect the operation of the system? Could the consequences of this force be dangerous in any way?

6.1.18 Switch contact types for a timed conveyor control

Suppose an Allen-Bradley PLC controls the starting and stopping of a conveyor belt, using a timer to sound an audible warning siren for 5 seconds before the conveyor belt starts up (to warn people before the belt begins to move):



Determine the necessary contact connections (form-A or form-B) on the real-life Start, Stop, and emergency Pull-Cable switches to complement the virtual contact types in the PLC program.

Start switch = form-A *or* form-B?

Stop switch = form-A *or* form-B?

Pull-Cable switch = form-A *or* form-B?

Challenges

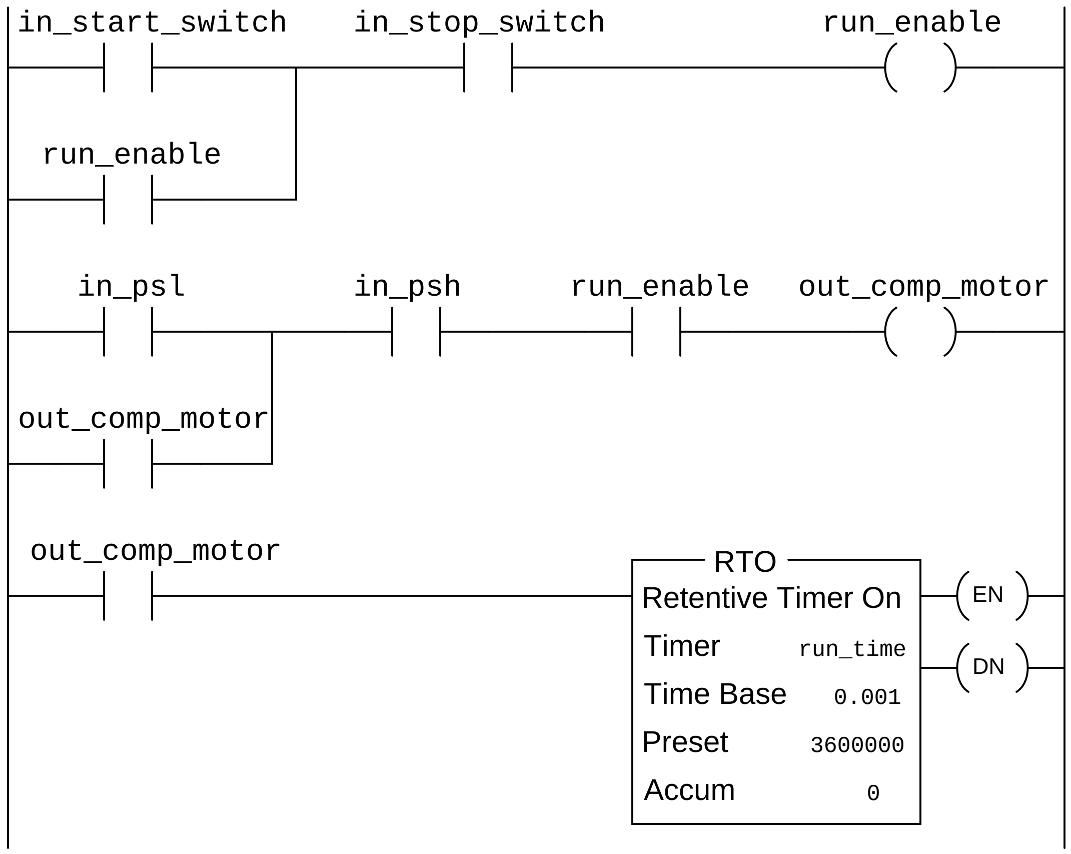
- How could you modify this program so that the operator has to hold the "Start" pushbutton switch actuated for the duration of the warning siren before the motor would start (i.e.

everything would simply stop if the operator only *momentarily* pressed the "Start" button)?

- Suppose a technician decides to use the *force* utility in the PLC to force bit `B3:0/0` to a "0" state in order to test the warning siren's operation without actually starting up the conveyor belt. Explain what is flawed with this testing strategy, and identify a better approach.
- How will this system behave if the pull-cable switch fails open?
- How will this system behave if the stop switch fails shorted?

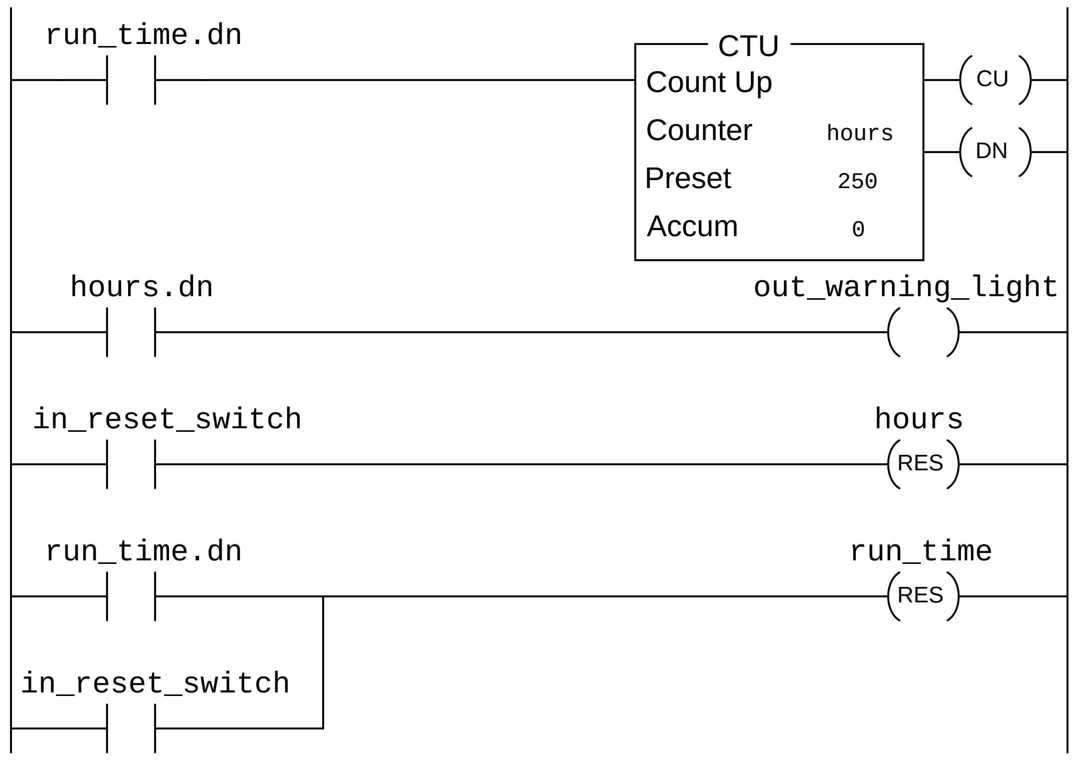
6.1.19 Air compressor control program

An Allen-Bradley Logix5000 PLC is used to control the starting and stopping of an air compressor based on momentary-contact pushbutton switch inputs as well as high and low pressure switches (PSH and PSL, respectively). Analyze this program and explain how it is supposed to work:



(continued on next page)

(continued from previous page)



In particular, answer these following questions:

- Determine the "normal" electrical statuses of all switches (e.g. NO or NC) connected to the inputs of this PLC, based on an examination of the respective contact instructions within the PLC program.
- Why is is important that a *retentive* timer instruction be used for the calculation of total run-time?
- What is the significance of the maintenance warning light controlled by this PLC?

Challenges

- Note how all instructions in this Logix5000 PLC program are addressed by *tagname* rather than by hardware addresses (e.g. `I:2/6`, `O:3/1`). How do you suppose the PLC "knows" which real I/O points to associate with which instructions in the program?
- How will this system behave if the reset switch fails shorted?

- How will this system behave if the high-pressure switch fails open?
- How will this system behave if the high-pressure switch fails shorted?
- How will this system behave if the low-pressure switch fails open?
- How will this system behave if the low-pressure switch fails shorted?

6.2 Quantitative reasoning

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

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

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

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

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

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

6.2.1 Miscellaneous physical constants

Note: constants shown in **bold** type are *exact*, not approximations. Values inside of parentheses show one standard deviation (σ) of uncertainty in the final digits: for example, Avogadro's number given as $6.02214179(30) \times 10^{23}$ means the center value ($6.02214179 \times 10^{23}$) plus or minus $0.00000030 \times 10^{23}$.

Avogadro's number (N_A) = $6.02214179(30) \times 10^{23}$ per mole (mol^{-1})

Boltzmann's constant (k) = $1.3806504(24) \times 10^{-23}$ Joules per Kelvin (J/K)

Electronic charge (e) = $1.602176487(40) \times 10^{-19}$ Coulomb (C)

Faraday constant (F) = $9.64853399(24) \times 10^{4}$ Coulombs per mole (C/mol)

Magnetic permeability of free space (μ_0) = $1.25663706212(19) \times 10^{-6}$ Henrys per meter (H/m)

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

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

Gravitational constant (G) = $6.67428(67) \times 10^{-11}$ cubic meters per kilogram-seconds squared (m^3/kg-s^2)

Molar gas constant (R) = 8.314472(15) Joules per mole-Kelvin (J/mol-K) = 0.08205746(14) liters-atmospheres per mole-Kelvin

Planck constant (h) = $6.62606896(33) \times 10^{-34}$ joule-seconds (J-s)

Stefan-Boltzmann constant (σ) = $5.670400(40) \times 10^{-8}$ Watts per square meter-Kelvin^4 (W/m^2·K^4)

Speed of light in a vacuum (c) = **299792458 meters per second** (m/s) = 186282.4 miles per second (mi/s)

Note: All constants taken from NIST data "Fundamental Physical Constants – Extensive Listing", from `http://physics.nist.gov/constants`, National Institute of Standards and Technology (NIST), 2006; with the exception of the permeability of free space which was taken from NIST's *2018 CODATA recommended values* database.

6.2.2 Introduction to spreadsheets

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

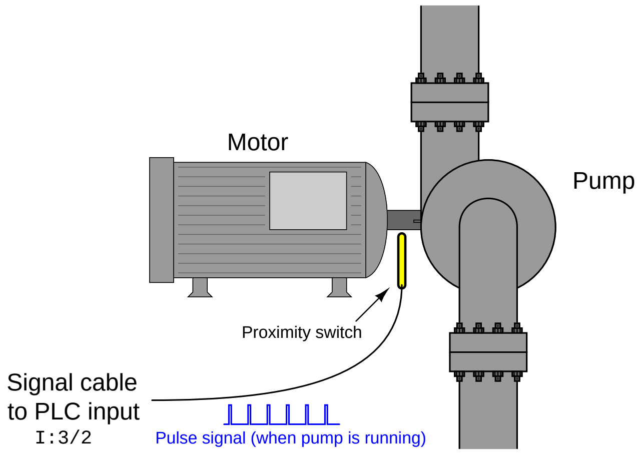
Note how the square-root term (y) is calculated in cell C1, and the denominator term (z) in cell C2. This makes the two final formulae (in cells B1 and B2) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary[10] – all that matters is that they properly reference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

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

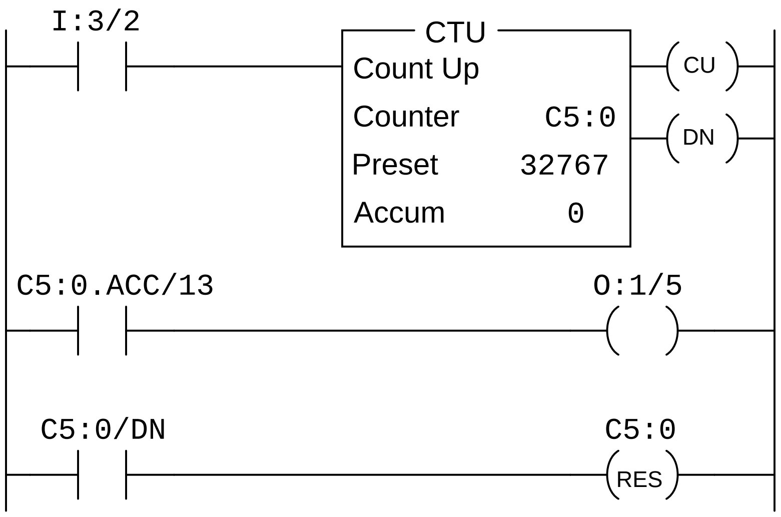
[10]My personal preference is to locate all the "given" data in the upper-left cells of the spreadsheet grid (each data point flanked by a sensible name in the cell to the left and units of measurement in the cell to the right as illustrated in the first distance/time spreadsheet example), sometimes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

6.2.3 Frequency divider program

An important pump in a chemical process is turned by an electric motor, and operators want to have visual indication in the control room that the pump is indeed turning. There is no way to attach a speed switch to the pump shaft (that would be too easy!). Instead, someone has installed a proximity switch near the pump shaft, situated to pick up the passing of a keyway in the shaft with each rotation. Thus, the proximity switch will output a "pulse" signal when the pump shaft is spinning:


Motor
Pump
Proximity switch
Signal cable to PLC input I:3/2
Pulse signal (when pump is running)

Operations personnel wanted the indicator light in the control room to blink when the pump is running, for an indication of shaft motion. The problem is, the shaft turns much too fast (approximately 1750 RPM) to directly drive the indicator with the proximity switch signal, and so an Allen-Bradley PLC was programmed to produce a slower blink using this program:



Explain how this program works to fulfill the function of a *frequency divider*, converting the high-speed pulse signal of the proximity switch into a low-speed blink for the operator light.

Challenges

- Explain how a *frequency divider* circuit built out of J-K flip-flop integrated circuits functions, and then describe how this PLC program is similar in principle.
- Explain how to speed up the blinking rate of the light for any given motor shaft speed.

6.2.4 Integer format error between PLC and HMI

Suppose a PLC program contains a counter instruction that counts in unsigned 16-bit integer format. An HMI connected to this PLC, however, is configured to read and interpret this counter's accumulated value as a *BCD* number instead of unsigned binary.

Determine how the HMI will interpret and display a PLC counter accumulated value of `38199` (decimal).

Challenges

- If the PLC's counter increments by one, what will the HMI display read?
- If the PLC's counter increments by two, what will the HMI display read?
- If the PLC's counter increments by three, what will the HMI display read?

6.3 Diagnostic reasoning

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

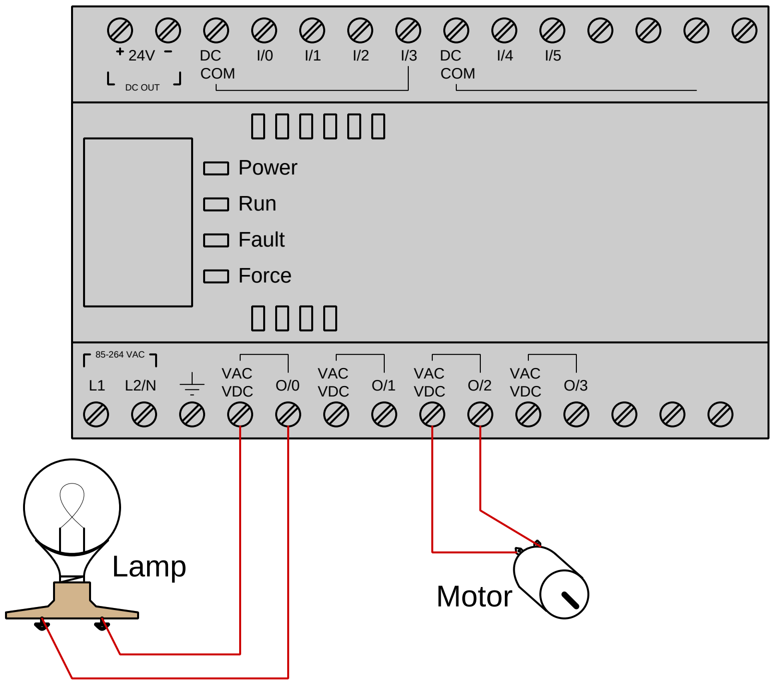
As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of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

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

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

6.3.1 Incorrect PLC output wiring

The PLC shown below is supposed to control the energization of a lamp and a small motor. Someone wired both loads to different PLC output channels as shown in the illustration below, but was surprised to find neither load energized when its respective PLC output channel activated:



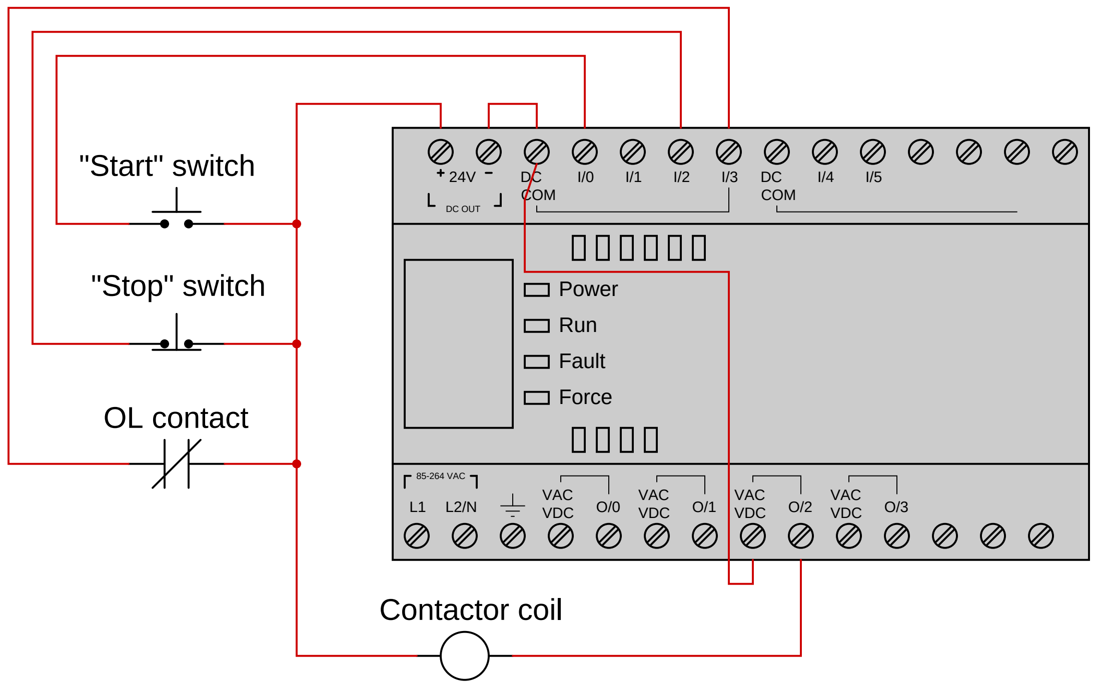
Explain what is wrong in this circuit, and then re-sketch the wiring so that these loads will function properly. Incidentally, this same error happens to be a *very* common misconception among students new to PLCs. Identify what the fundamental misconception is, and how it may be remedied.

Challenges

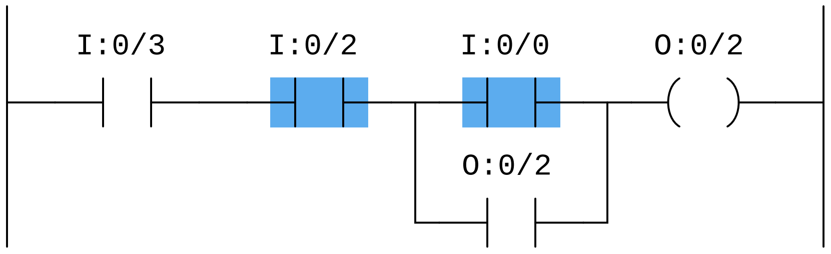
- What information should we ideally have regarding the PLC and the two loads shown in order to design a complete circuit that will function safely and reliably?
- From the information available in the illustration, is it possible to formulate an educated guess about the type of discrete output channels offered by this PLC?

6.3.2 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:



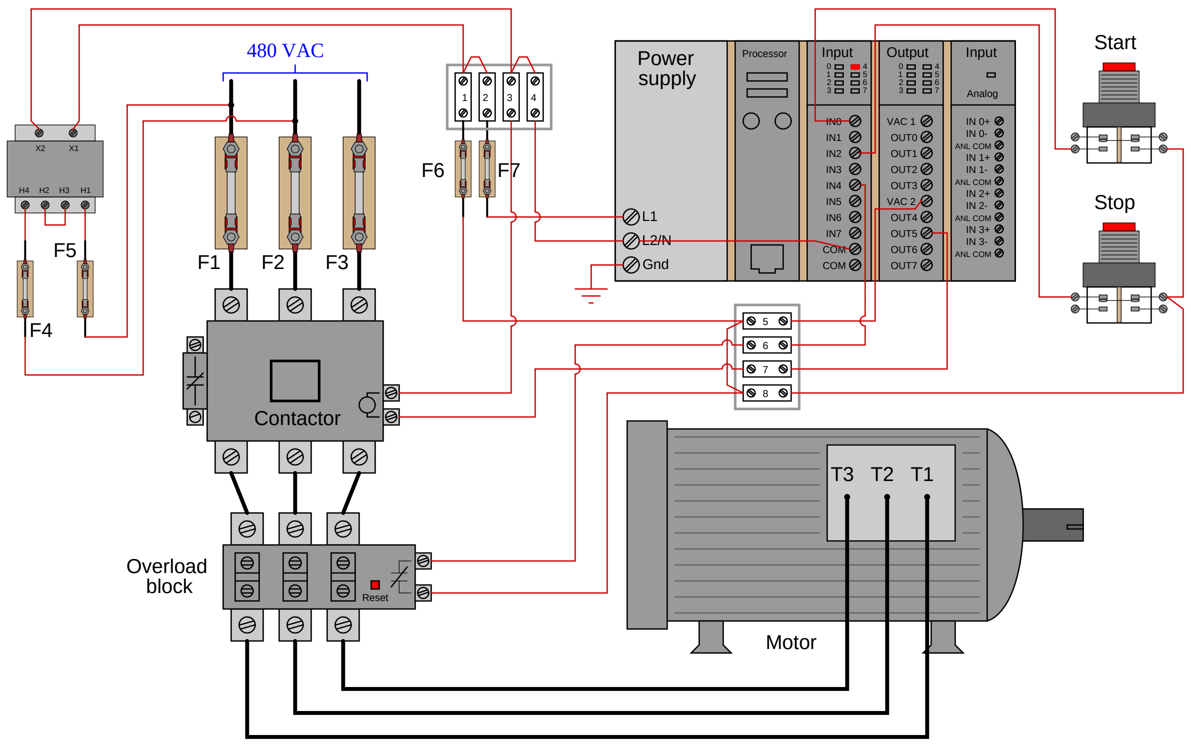
Identify at least two causes that could account for all you see here.

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.

6.3.3 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:



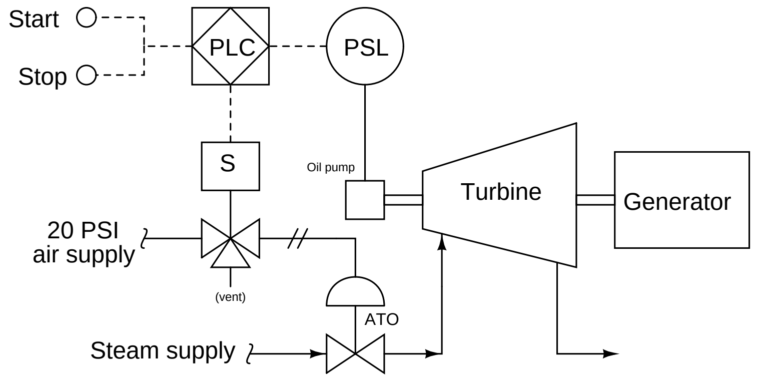
This motor control system has a problem, though: the motor refuses to start when the "Start" pushbutton is pressed. Closely examine the pictorial diagram (including the status LEDs on the PLC's I/O cards), then identify at least two faults that could account for the motor's refusal to start.

Challenges

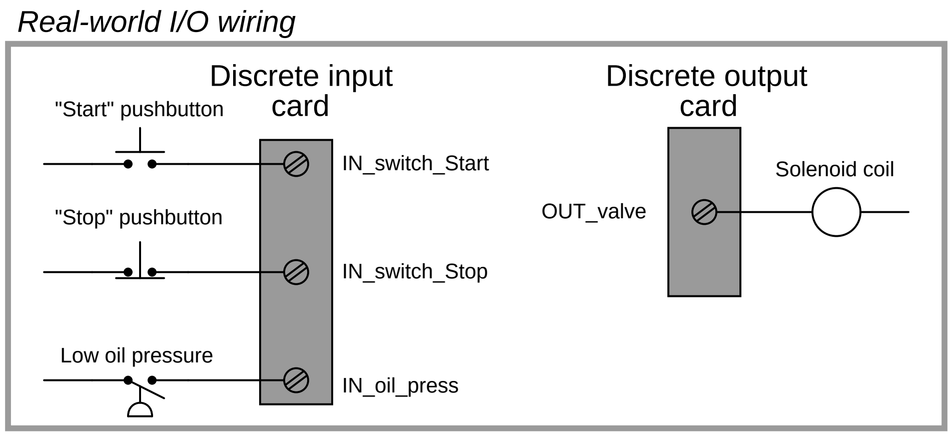
- Explain why knowledge of the PLC program is not necessary to diagnose this fault.

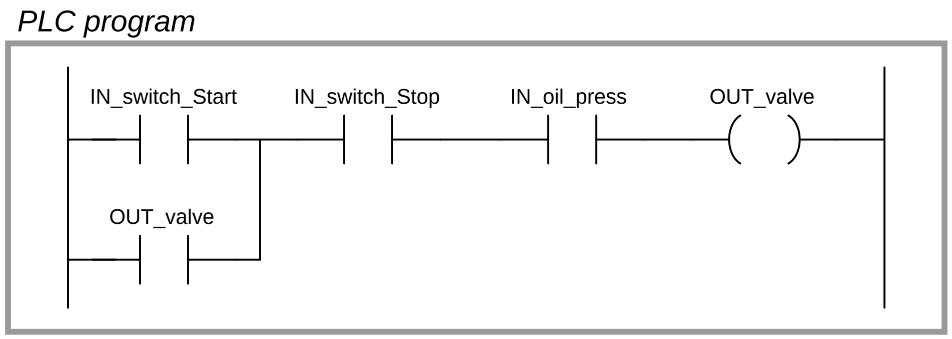
6.3.4 Turbine low-oil trip

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:





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.

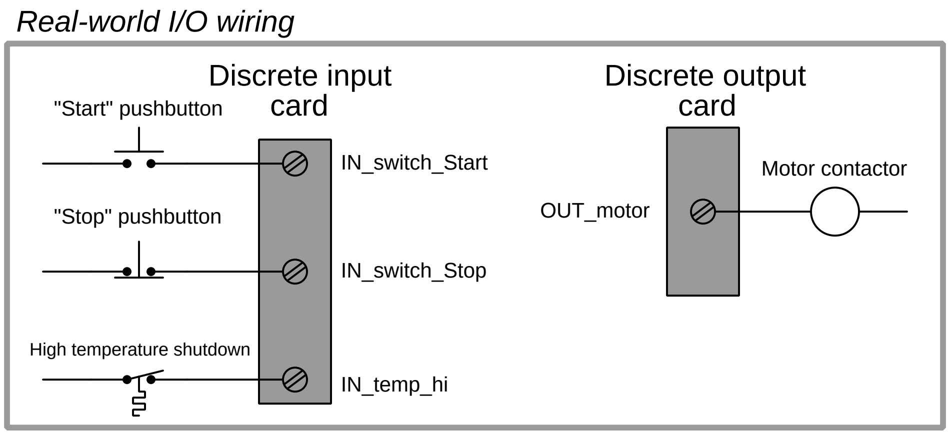
Challenges

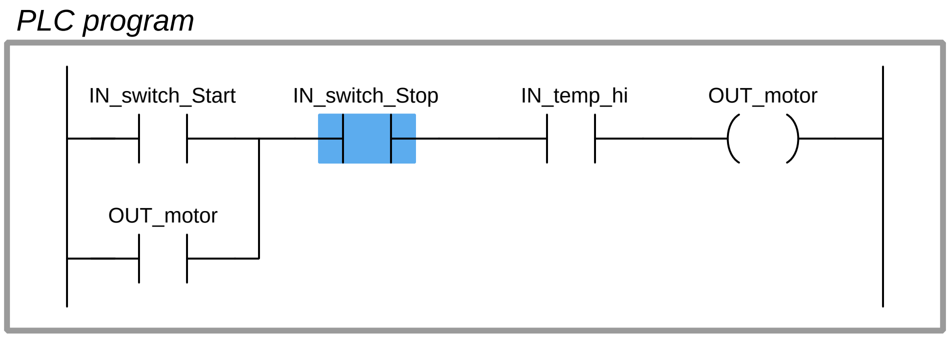
- Explain how this problem could be fixed by the addition of a *timer* instruction to the PLC program.

6.3.5 Motor starter diagnosis from color highlighting

PLC #1

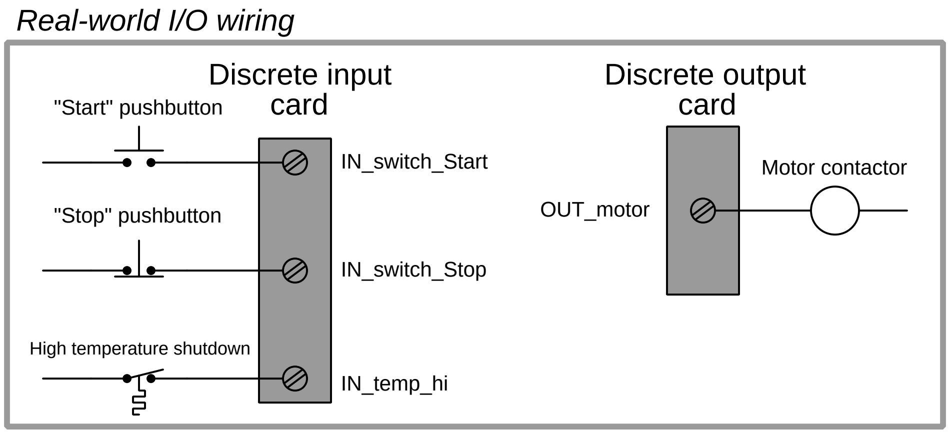
Identify one plausible fault to explain why this PLC-controlled motor refuses to start up when the "Start" button is pressed, given the following wiring diagram and online PLC program display shown below. Note: no one is pushing any buttons at this time.

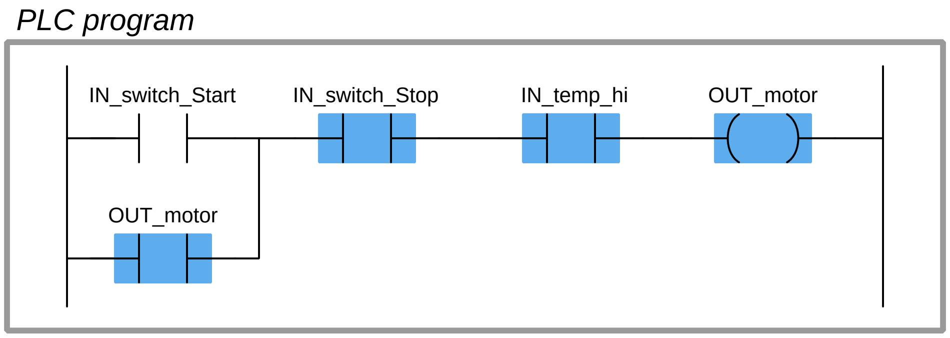




PLC #2

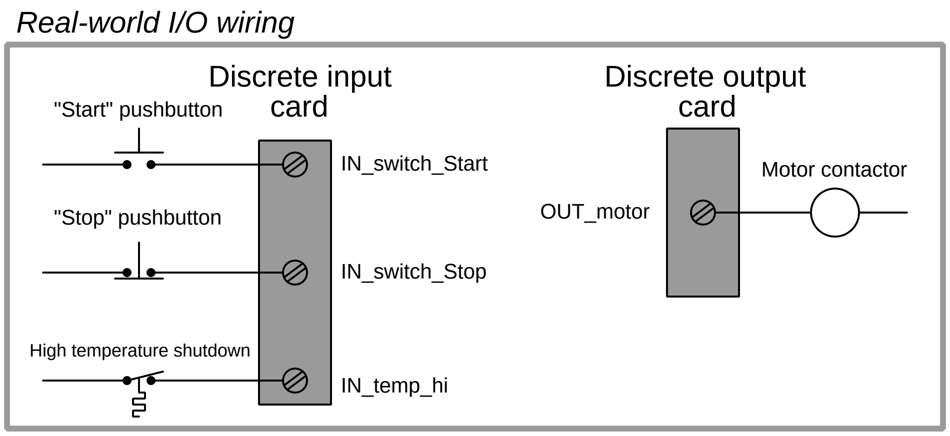
Identify one plausible fault to explain why this PLC-controlled motor refuses to start up when the "Start" button is pressed, given the following wiring diagram and online PLC program display shown below. Note: no one is pushing any buttons at this time.

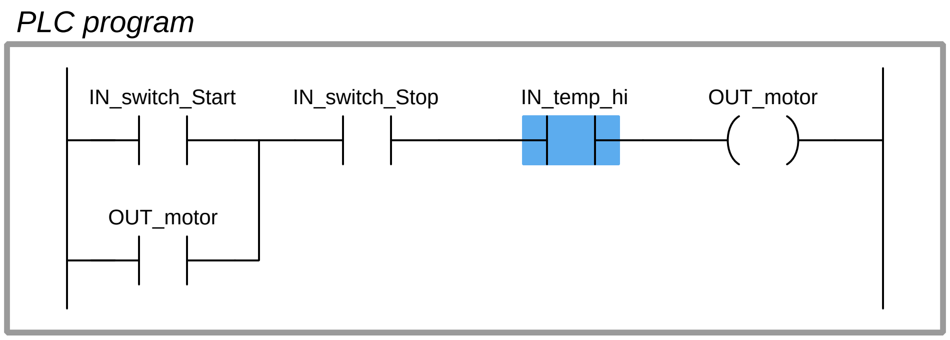




PLC #3

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.



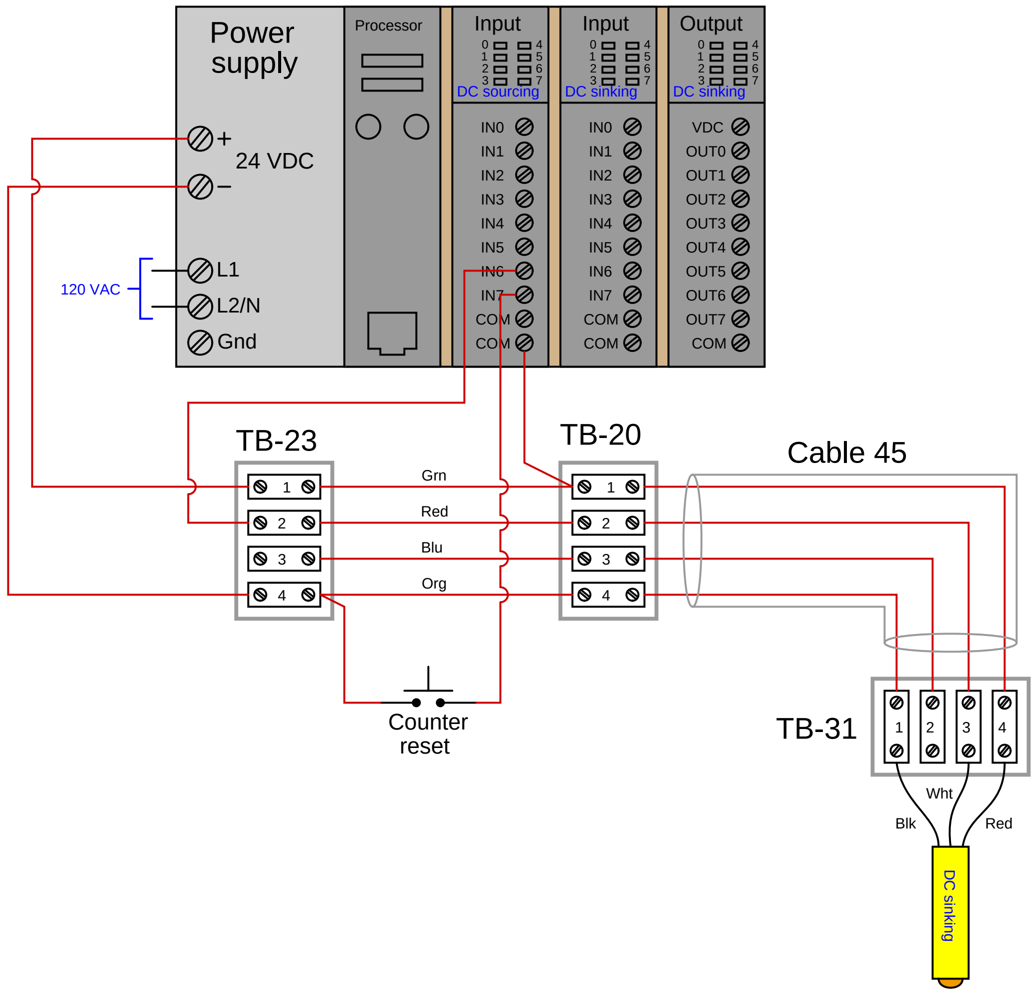


Challenges

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

6.3.6 Cannery counter diagnosis

A PLC is used to count the number of cans traveling by on a conveyor belt in a fish canning factory. An optical proximity switch detects the passage of each can, sending a discrete (on/off) signal to one of the PLC's input channels. The PLC then counts the number of pulses to determine the number of cans that have passed by:



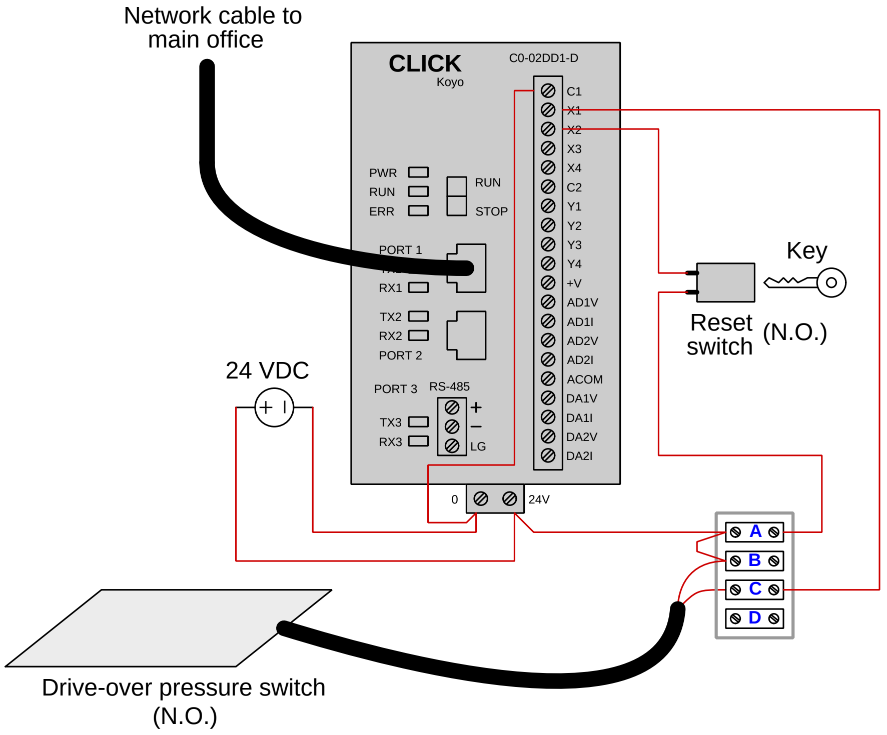
One day the canning line operator tells you the PLC has stopped counting even though cans continue to run past the proximity switch as the conveyor belt moves. Identify what you would do to begin diagnosing this problem, justifying each step you would take.

Challenges

- Identify different areas or components within this system that could possibly be at fault, as a prelude to identifying specific diagnostic steps.
- Are there any ways you could diagnose this problem without the use of test equipment (e.g. multimeter)?
- Explain the significance of the "sourcing" and "sinking" labels on the I/O cards as well as the proximity switch.

6.3.7 Parking garage counter faults

The following Koyo CLICK PLCs are supposed to count the number of cars entering a parking garage, using a pressure-sensitive switch that the cars drive over when entering the garage. The car-count value is sent to a computer in the main office via a network cable plugged into the PLC. The parking attendant is able to reset the count to 0 at the end of his shift, using a key-switch:



PLC #1

Suppose the counter's current value as displayed on the main office computer is stuck at 574 no matter how many more cars drive over the pressure switch and enter the garage. Explain how you would go about diagnosing the problem in this system, justifying each step you would take.

PLC #2

Suppose the counter's current value as displayed on the main office computer is stuck at 1357 no matter how many more cars drive over the pressure switch and enter the garage. A voltmeter connected to the terminals of the reset switch registers 23.9 Volts DC all the time. Does this voltmeter test provide any diagnostically useful information to us?

PLC #3

Suppose the counter's current value as displayed on the main office computer is stuck at 822 no matter how many more cars drive over the pressure switch and enter the garage. A voltmeter connected between terminals `X2` and `C1` registers 0.0 Volts DC all the time. Does this voltmeter test provide any diagnostically useful information to us?

PLC #4

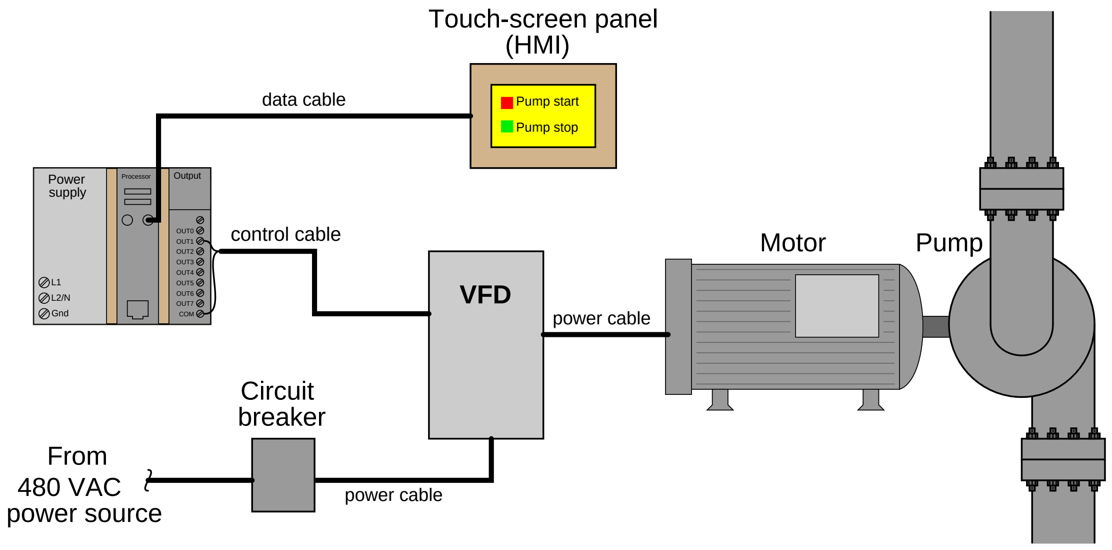
Suppose the counter's current value as displayed on the main office computer is stuck at 0 no matter how many more cars drive over the pressure switch and enter the garage. A voltmeter connected between terminals `X2` and `C1` registers 25.1 Volts DC all the time. Does this voltmeter test provide any diagnostically useful information to us?

Challenges

- For any diagnostic tests deemed useless, identify a better diagnostic test.
- For any diagnostic tests deemed useful, identify the *next* diagnostic test you would take.

6.3.8 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.

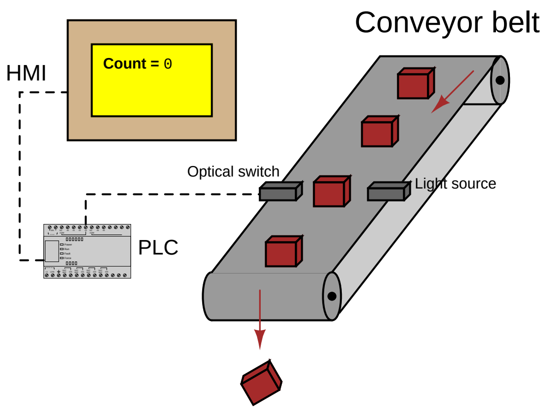
- Circuit breaker off
- Touch-screen panel malfunctioning
- Programming error in PLC
- Faulted power cable between VFD and motor
- Faulted power cable between breaker and VFD
- PLC output card malfunctioning
- Open control cable
- Shorted control cable
- Open data cable

$\boxed{\text{Challenges}}$

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

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

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



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

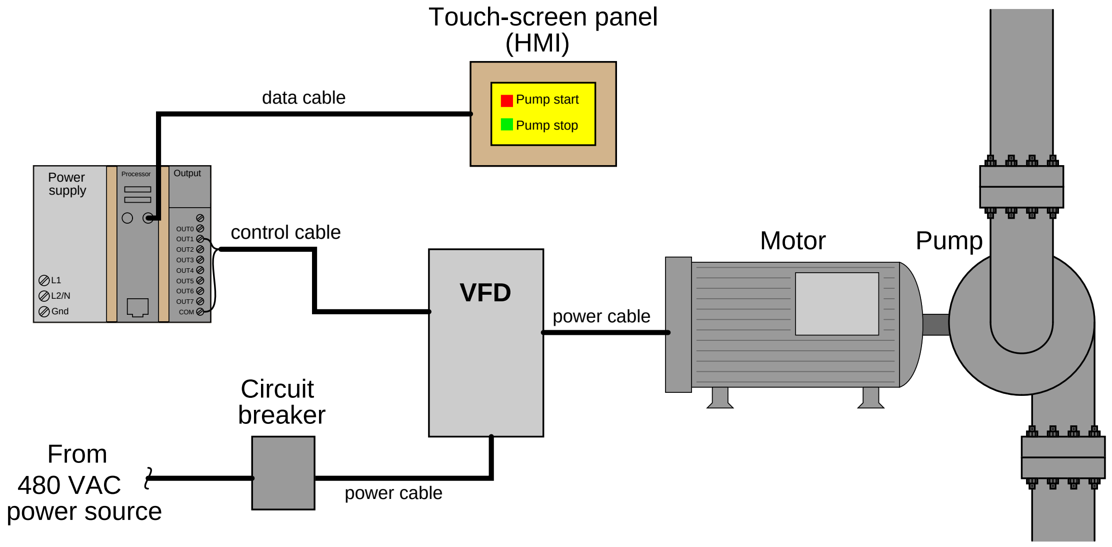
Brainstorm at least five different faults that could account for this problem, and then devise a "next test" you would conduct to narrow the field of potential faults. The simpler this test (i.e. the least amount of time to conduct and the less complicated test equipment required), the better!

Challenges

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

6.3.10 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?

Chapter 7

Projects and Experiments

The following project and experiment descriptions outline things you can build to help you understand circuits. With any real-world project or experiment there exists the potential for physical harm. *Electricity can be very dangerous in certain circumstances, and you should follow proper safety precautions at all times!*

7.1 Recommended practices

This section outlines some recommended practices for all circuits you design and construct.

7.1.1 Safety first!

Electricity, when passed through the human body, causes uncomfortable sensations and in large enough measures[1] will cause muscles to involuntarily contract. The overriding of your nervous system by the passage of electrical current through your body is particularly dangerous in regard to your heart, which is a vital muscle. Very large amounts of current can produce serious internal burns in addition to all the other effects.

Cardio-pulmonary resuscitation (CPR) is the standard first-aid for any victim of electrical shock. This is a very good skill to acquire if you intend to work with others on dangerous electrical circuits. You should never perform tests or work on such circuits unless someone else is present who is proficient in CPR.

As a general rule, any voltage in excess of 30 Volts poses a definitive electric shock hazard, because beyond this level human skin does not have enough resistance to safely limit current through the body. "Live" work of any kind with circuits over 30 volts should be avoided, and if unavoidable should only be done using electrically insulated tools and other protective equipment (e.g. insulating shoes and gloves). If you are unsure of the hazards, or feel unsafe at any time, stop all work and distance yourself from the circuit!

A policy I strongly recommend for students learning about electricity is to *never come into electrical contact*[2] *with an energized conductor, no matter what the circuit's voltage*[3] *level!* Enforcing this policy may seem ridiculous when the circuit in question is powered by a single battery smaller than the palm of your hand, but it is precisely this instilled habit which will save a person from bodily harm when working with more dangerous circuits. Experience has taught me that students who learn early on to be careless with safe circuits have a tendency to be careless later with dangerous circuits!

In addition to the electrical hazards of shock and burns, the construction of projects and running of experiments often poses other hazards such as working with hand and power tools, potential

[1]Professor Charles Dalziel published a research paper in 1961 called "The Deleterious Effects of Electric Shock" detailing the results of electric shock experiments with both human and animal subjects. The threshold of perception for human subjects holding a conductor in their hand was in the range of 1 milliampere of current (less than this for alternating current, and generally less for female subjects than for male). Loss of muscular control was exhibited by half of Dalziel's subjects at less than 10 milliamperes alternating current. Extreme pain, difficulty breathing, and loss of all muscular control occurred for over 99% of his subjects at direct currents less than 100 milliamperes and alternating currents less than 30 milliamperes. In summary, it doesn't require much electric current to induce painful and even life-threatening effects in the human body! Your first and best protection against electric shock is maintaining an insulating barrier between your body and the circuit in question, such that current from that circuit will be unable to flow through your body.

[2]By "electrical contact" I mean either directly touching an energized conductor with any part of your body, or indirectly touching it through a conductive tool. The only physical contact you should ever make with an energized conductor is via an electrically insulated tool, for example a screwdriver with an electrically insulated handle, or an insulated test probe for some instrument.

[3]Another reason for consistently enforcing this policy, even on low-voltage circuits, is due to the dangers that even some low-voltage circuits harbor. A single 12 Volt automobile battery, for example, can cause a surprising amount of damage if short-circuited simply due to the high current levels (i.e. very low internal resistance) it is capable of, even though the voltage level is too low to cause a shock through the skin. Mechanics wearing metal rings, for example, are at risk from severe burns if their rings happen to short-circuit such a battery! Furthermore, even when working on circuits that are simply too low-power (low voltage and low current) to cause any bodily harm, touching them while energized can pose a threat to the circuit components themselves. In summary, it generally wise (and *always* a good habit to build) to "power down" *any* circuit before making contact between it and your body.

contact with high temperatures, potential chemical exposure, etc. You should never proceed with a project or experiment if you are unaware of proper tool use or lack basic protective measures (e.g. personal protective equipment such as safety glasses) against such hazards.

Some other safety-related practices should be followed as well:

- All power conductors extending outward from the project must be *firmly* strain-relieved (e.g. "cord grips" used on line power cords), so that an accidental tug or drop will not compromise circuit integrity.

- All electrical connections must be sound and appropriately made (e.g. soldered wire joints rather than twisted-and-taped; terminal blocks rather than solderless breadboards for high-current or high-voltage circuits). Use "touch-safe" terminal connections with recessed metal parts to minimize risk of accidental contact.

- Always provide overcurrent protection in any circuit you build. *Always.* This may be in the form of a fuse, a circuit breaker, and/or an electronically current-limited power supply.

- Always ensure circuit conductors are rated for more current than the overcurrent protection limit. *Always.* A fuse does no good if the wire or printed circuit board trace will "blow" before it does!

- Always bond metal enclosures to Earth ground for any line-powered circuit. *Always.* Ensuring an equipotential state between the enclosure and Earth by making the enclosure electrically common with Earth ground ensures no electric shock can occur simply by one's body bridging between the Earth and the enclosure.

- Avoid building a high-energy circuit when a low-energy circuit will suffice. For example, I always recommend beginning students power their first DC resistor circuits using small batteries rather than with line-powered DC power supplies. The intrinsic energy limitations of a dry-cell battery make accidents highly unlikely.

- Use line power receptacles that are GFCI (Ground Fault Current Interrupting) to help avoid electric shock from making accidental contact with a "hot" line conductor.

- Always wear eye protection when working with tools or live systems having the potential to eject material into the air. Examples of such activities include soldering, drilling, grinding, cutting, wire stripping, working on or near energized circuits, etc.

- Always use a step-stool or stepladder to reach high places. Never stand on something not designed to support a human load.

- When in doubt, *ask an expert.* If anything even seems remotely unsafe to you, do not proceed without consulting a trusted person fully knowledgeable in electrical safety.

7.1.2 Other helpful tips

Experience has shown the following practices to be very helpful, especially when students make their own component selections, to ensure the circuits will be well-behaved:

- Avoid resistor values less than 1 kΩ or greater than 100 kΩ, unless such values are definitely necessary[4]. Resistances below 1 kΩ may draw excessive current if directly connected to a voltage source of significant magnitude, and may also complicate the task of accurately measuring current since any ammeter's non-zero resistance inserted in series with a low-value circuit resistor will significantly alter the total resistance and thereby skew the measurement. Resistances above 100 kΩ may complicate the task of measuring voltage since any voltmeter's finite resistance connected in parallel with a high-value circuit resistor will significantly alter the total resistance and thereby skew the measurement. Similarly, AC circuit impedance values should be between 1 kΩ and 100 kΩ, and for all the same reasons.

- Ensure all electrical connections are low-resistance and physically rugged. For this reason, one should avoid *compression splices* (e.g. "butt" connectors), solderless breadboards[5], and wires that are simply twisted together.

- Build your circuit with **testing** in mind. For example, provide convenient connection points for test equipment (e.g. multimeters, oscilloscopes, signal generators, logic probes).

- Design permanent projects with **maintenance** in mind. The more convenient you make maintenance tasks, the more likely they will get done.

- **Always document and save your work**. Circuits lacking schematic diagrams are more difficult to troubleshoot than documented circuits. Similarly, circuit construction is simpler when a schematic diagram precedes construction. Experimental results are easier to interpret when comprehensively recorded. Consider modern videorecording technology for this purpose where appropriate.

- **Record your steps** when troubleshooting. **Talk to yourself** when solving problems. These simple steps clarify thought and simplify identification of errors.

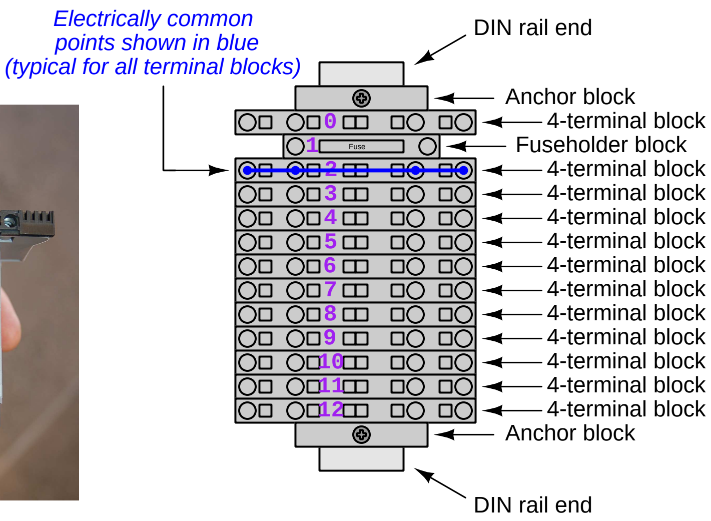
[4]An example of a necessary resistor value much less than 1 kΩ is a *shunt resistor* used to produce a small voltage drop for the purpose of sensing current in a circuit. Such shunt resistors must be low-value in order not to impose an undue load on the rest of the circuit. An example of a necessary resistor value much greater than 100 kΩ is an electrostatic *drain resistor* used to dissipate stored electric charges from body capacitance for the sake of preventing damage to sensitive semiconductor components, while also preventing a path for current that could be dangerous to the person (i.e. shock).

[5]Admittedly, solderless breadboards are very useful for constructing complex electronic circuits with many components, especially DIP-style integrated circuits (ICs), but they tend to give trouble with connection integrity after frequent use. An alternative for projects using low counts of ICs is to solder IC sockets into prototype printed circuit boards (PCBs) and run wires from the soldered pins of the IC sockets to terminal blocks where reliable temporary connections may be made.

7.1.3 Terminal blocks for circuit construction

Terminal blocks are the standard means for making electric circuit connections in industrial systems. They are also quite useful as a learning tool, and so I highly recommend their use in lieu of solderless breadboards[6]. Terminal blocks provide highly reliable connections capable of withstanding significant voltage and current magnitudes, and they force the builder to think very carefully about component layout which is an important mental practice. Terminal blocks that mount on standard 35 mm DIN rail[7] are made in a wide range of types and sizes, some with built-in disconnecting switches, some with built-in components such as rectifying diodes and fuseholders, all of which facilitate practical circuit construction.

I recommend every student of electricity build their own terminal block array for use in constructing experimental circuits, consisting of several terminal blocks where each block has at least 4 connection points all electrically common to each other[8] and at least one terminal block that is a fuse holder for overcurrent protection. A pair of anchoring blocks hold all terminal blocks securely on the DIN rail, preventing them from sliding off the rail. Each of the terminals should bear a number, starting from 0. An example is shown in the following photograph and illustration:



Screwless terminal blocks (using internal spring clips to clamp wire and component lead ends) are preferred over screw-based terminal blocks, as they reduce assembly and disassembly time, and also minimize repetitive wrist stress from twisting screwdrivers. Some screwless terminal blocks require the use of a special tool to release the spring clip, while others provide buttons[9] for this task which may be pressed using the tip of any suitable tool.

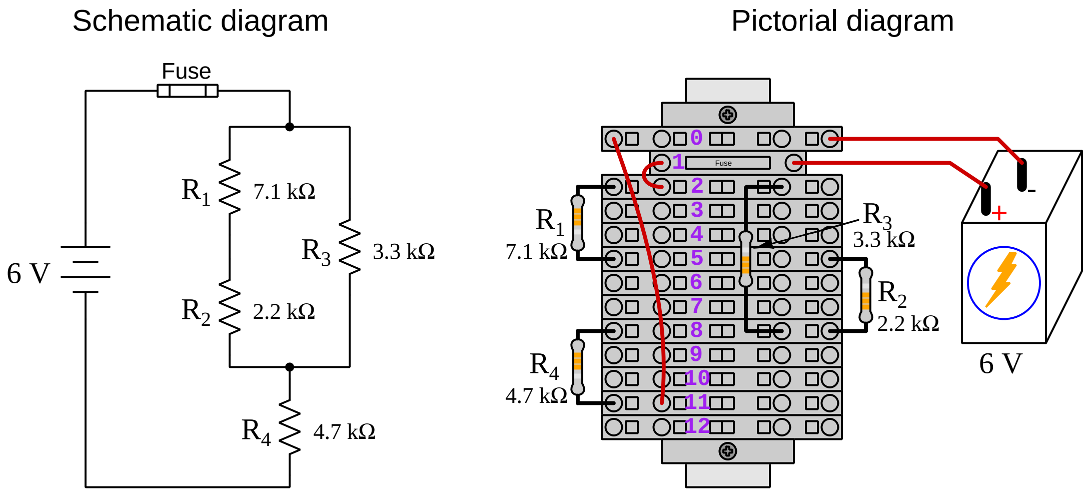
[6]Solderless breadboard are preferable for complicated electronic circuits with multiple integrated "chip" components, but for simpler circuits I find terminal blocks much more practical. An alternative to solderless breadboards for "chip" circuits is to solder chip sockets onto a PCB and then use wires to connect the socket pins to terminal blocks. This also accommodates *surface-mount* components, which solderless breadboards do not.

[7]DIN rail is a metal rail designed to serve as a mounting point for a wide range of electrical and electronic devices such as terminal blocks, fuses, circuit breakers, relay sockets, power supplies, data acquisition hardware, etc.

[8]Sometimes referred to as *equipotential*, *same-potential*, or *potential distribution* terminal blocks.

[9]The small orange-colored squares seen in the above photograph are buttons for this purpose, and may be actuated by pressing with any tool of suitable size.

The following example shows how such a terminal block array might be used to construct a series-parallel resistor circuit consisting of four resistors and a battery:



Numbering on the terminal blocks provides a very natural translation to SPICE[10] netlists, where component connections are identified by terminal number:

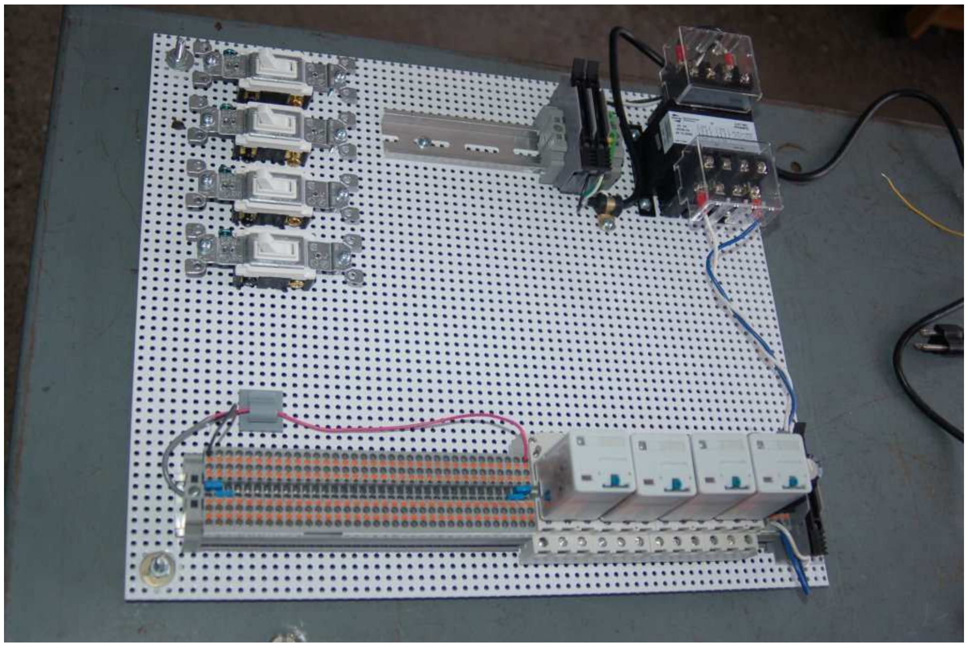
```
* Series-parallel resistor circuit
v1 1 0 dc 6
r1 2 5 7100
r2 5 8 2200
r3 2 8 3300
r4 8 11 4700
rjmp1 1 2 0.01
rjmp2 0 11 0.01
.op
.end
```

Note the use of "jumper" resistances `rjmp1` and `rjmp2` to describe the wire connections between terminals 1 and 2 and between terminals 0 and 11, respectively. Being resistances, SPICE requires a resistance value for each, and here we see they have both been set to an arbitrarily low value of 0.01 Ohm realistic for short pieces of wire.

Listing all components and wires along with their numbered terminals happens to be a useful documentation method for any circuit built on terminal blocks, independent of SPICE. Such a "wiring sequence" may be thought of as a *non-graphical description* of an electric circuit, and is exceptionally easy to follow.

[10]SPICE is computer software designed to analyze electrical and electronic circuits. Circuits are described for the computer in the form of *netlists* which are text files listing each component type, connection node numbers, and component values.

An example of a more elaborate terminal block array is shown in the following photograph, with terminal blocks and "ice-cube" style electromechanical relays mounted to DIN rail, which is turn mounted to a perforated subpanel[11]. This "terminal block board" hosts an array of thirty five undedicated terminal block sections, four SPDT toggle switches, four DPDT "ice-cube" relays, a step-down control power transformer, bridge rectifier and filtering capacitor, and several fuses for overcurrent protection:



Four plastic-bottomed "feet" support the subpanel above the benchtop surface, and an unused section of DIN rail stands ready to accept other components. Safety features include electrical bonding of the AC line power cord's ground to the metal subpanel (and all metal DIN rails), mechanical strain relief for the power cord to isolate any cord tension from wire connections, clear plastic finger guards covering the transformer's screw terminals, as well as fused overcurrent protection for the 120 Volt AC line power and the transformer's 12 Volt AC output. The perforated holes happen to be on $\frac{1}{4}$ inch centers with a diameter suitable for tapping with 6-32 machine screw threads, their presence making it very easy to attach other sections of DIN rail, printed circuit boards, or specialized electrical components directly to the grounded metal subpanel. Such a "terminal block board" is an inexpensive[12] yet highly flexible means to construct physically robust circuits using industrial wiring practices.

[11]An electrical *subpanel* is a thin metal plate intended for mounting inside an electrical enclosure. Components are attached to the subpanel, and the subpanel in turn bolts inside the enclosure. Subpanels allow circuit construction outside the confines of the enclosure, which speeds assembly. In this particular usage there is no enclosure, as the subpanel is intended to be used as an open platform for the convenient construction of circuits on a benchtop by students. In essence, this is a modern version of the traditional *breadboard* which was literally a wooden board such as might be used for cutting loaves of bread, but which early electrical and electronic hobbyists used as platforms for the construction of circuits.

[12]At the time of this writing (2019) the cost to build this board is approximately $250 US dollars.

7.1.4 Conducting experiments

An *experiment* is an exploratory act, a test performed for the purpose of assessing some proposition or principle. Experiments are the foundation of the *scientific method*, a process by which careful observation helps guard against errors of speculation. All good experiments begin with an *hypothesis*, defined by the American Heritage Dictionary of the English Language as:

> An assertion subject to verification or proof, as (a) A proposition stated as a basis for argument or reasoning. (b) A premise from which a conclusion is drawn. (c) A conjecture that accounts, within a theory or ideational framework, for a set of facts and that can be used as a basis for further investigation.

Stated plainly, an hypothesis is an *educated guess* about cause and effect. The correctness of this initial guess matters little, because any well-designed experiment will reveal the truth of the matter. In fact, *incorrect* hypotheses are often the most valuable because the experiments they engender lead us to surprising discoveries. One of the beautiful aspects of science is that it is more focused on the process of *learning* than about the status of *being correct*[13]. In order for an hypothesis to be valid, it must be testable[14], which means it must be a claim possible to refute given the right data. Hypotheses impossible to critique are useless.

Once an hypothesis has been formulated, an experiment must be designed to test that hypothesis. A well-designed experiment requires careful regulation of all relevant variables, both for personal safety and for prompting the hypothesized results. If the effects of one particular variable are to be tested, the experiment must be run multiple times with different values of (only) that particular variable. The experiment set up with the "baseline" variable set is called the *control*, while the experiment set up with different value(s) is called the *test* or *experimental*.

For some hypotheses a viable alternative to a physical experiment is a *computer-simulated experiment* or even a *thought experiment*. Simulations performed on a computer test the hypothesis against the physical laws encoded within the computer simulation software, and are particularly useful for students learning new principles for which simulation software is readily available[15].

[13]Science is more about clarifying our view of the universe through a systematic process of error detection than it is about proving oneself to be right. Some scient*ists* may happen to have large egos – and this may have more to do with the ways in which large-scale scientific research is *funded* than anything else – but scientific *method* itself is devoid of ego, and if embraced as a practical philosophy is quite an effective stimulant for humility. Within the education system, scientific method is particularly valuable for helping students break free of the crippling fear of *being wrong*. So much emphasis is placed in formal education on assessing correct retention of facts that many students are fearful of saying or doing anything that might be perceived as a mistake, and of course making mistakes (i.e. having one's hypotheses disproven by experiment) is an indispensable tool for learning. Introducing science in the classroom – *real* science characterized by individuals forming actual hypotheses and testing those hypotheses by experiment – helps students become self-directed learners.

[14]This is the principle of *falsifiability*: that a scientific statement has value only insofar as it is liable to disproof given the requisite experimental evidence. Any claim that is unfalsifiable – that is, a claim which can *never* be disproven by any evidence whatsoever – could be completely wrong and we could never know it.

[15]A very pertinent example of this is learning how to analyze electric circuits using simulation software such as SPICE. A typical experimental cycle would proceed as follows: (1) Find or invent a circuit to analyze; (2) Apply your analytical knowledge to that circuit, predicting all voltages, currents, powers, etc. relevant to the concepts you are striving to master; (3) Run a simulation on that circuit, collecting "data" from the computer when complete; (4) Evaluate whether or not your hypotheses (i.e. predicted voltages, currents, etc.) agree with the computer-generated results; (5) If so, your analyses are (provisionally) correct – if not, examine your analyses and the computer simulation again to determine the source of error; (6) Repeat this process as many times as necessary until you achieve mastery.

Thought experiments are useful for detecting inconsistencies within your own understanding of some subject, rather than testing your understanding against physical reality.

Here are some general guidelines for conducting experiments:

- The clearer and more specific the hypothesis, the better. Vague or unfalsifiable hypotheses are useless because they will fit *any* experimental results, and therefore the experiment cannot teach you anything about the hypothesis.
- Collect as much data (i.e. information, measurements, sensory experiences) generated by an experiment as is practical. This includes the time and date of the experiment, too!
- *Never* discard or modify data gathered from an experiment. If you have reason to believe the data is unreliable, write notes to that effect, but never throw away data just because you think it is untrustworthy. It is quite possible that even "bad" data holds useful information, and that someone else may be able to uncover its value even if you do not.
- Prioritize *quantitative* data over *qualitative* data wherever practical. Quantitative data is more specific than qualitative, less prone to subjective interpretation on the part of the experimenter, and amenable to an arsenal of analytical methods (e.g. statistics).
- Guard against your own bias(es) by making your experimental results available to others. This allows other people to scrutinize your experimental design and collected data, for the purpose of detecting and correcting errors you may have missed. Document your experiment such that others may independently replicate it.
- Always be looking for sources of error. No physical measurement is perfect, and so it is impossible to achieve *exact* values for any variable. Quantify the amount of uncertainty (i.e. the "tolerance" of errors) whenever possible, and be sure your hypothesis does not depend on precision better than this!
- Always remember that scientific confirmation is provisional – no number of "successful" experiments will prove an hypothesis true for all time, but a single experiment can disprove it. Put into simpler terms, *truth is elusive but error is within reach.*
- Remember that scientific method is about *learning*, first and foremost. An unfortunate consequence of scientific triumph in modern society is that science is often viewed by non-practitioners as an unerring source of truth, when in fact science is an ongoing process of challenging existing ideas to probe for errors and oversights. This is why it is perfectly acceptable to have a failed hypothesis, and why the only truly failed experiment is one where nothing was learned.

The following is an example of a well-planned and executed experiment, in this case a physical experiment demonstrating Ohm's Law.

```
Planning Time/Date = 09:30 on 12 February 2019

HYPOTHESIS: the current through any resistor should be exactly proportional
to the voltage impressed across it.

PROCEDURE: connect a resistor rated 1 k Ohm and 1/4 Watt to a variable-voltage
DC power supply.  Use an ammeter in series to measure resistor current and
a voltmeter in parallel to measure resistor voltage.

RISKS AND MITIGATION: excessive power dissipation may harm the resistor and/
or pose a burn hazard, while excessive voltage poses an electric shock hazard.
30 Volts is a safe maximum voltage for laboratory practices, and according to
Joule's Law a 1000 Ohm resistor will dissipate 0.25 Watts at 15.81 Volts
(P = V^2 / R), so I will remain below 15 Volts just to be safe.

Experiment Time/Date = 10:15 on 12 February 2019

DATA COLLECTED:
  (Voltage)    (Current)         (Voltage)   (Current)
   0.000 V   =  0.000 mA          8.100    =  7.812 mA
   2.700 V   =  2.603 mA          10.00 V  =  9.643 mA
   5.400 V   =  5.206 mA          14.00 V  =  13.49 mA

Analysis Time/Date = 10:57 on 12 February 2019

ANALYSIS: current definitely increases with voltage, and although I expected
exactly one milliAmpere per Volt the actual current was usually less than
that.  The voltage/current ratios ranged from a low of 1036.87 (at 8.1 Volts)
to a high of 1037.81 (at 14 Volts), but this represents a variance of only
-0.0365% to +0.0541% from the average, indicating a very consistent
proportionality -- results consistent with Ohm's Law.

ERROR SOURCES: one major source of error is the resistor's value itself.  I
did not measure it, but simply assumed color bands of brown-black-red meant
exactly 1000 Ohms.  Based on the data I think the true resistance is closer
to 1037 Ohms.  Another possible explanation is multimeter calibration error.
However, neither explains the small positive and negative variances from the
average.  This might be due to electrical noise, a good test being to repeat
the same experiment to see if the variances are the same or different.  Noise
should generate slightly different results every time.
```

The following is an example of a well-planned and executed *virtual* experiment, in this case demonstrating Ohm's Law using a computer (SPICE) simulation.

```
Planning Time/Date = 12:32 on 14 February 2019

HYPOTHESIS: for any given resistor, the current through that resistor should be
exactly proportional to the voltage impressed across it.

PROCEDURE: write a SPICE netlist with a single DC voltage source and single
1000 Ohm resistor, then use NGSPICE version 26 to perform a "sweep" analysis
from 0 Volts to 25 Volts in 5 Volt increments.

    * SPICE circuit
    v1 1 0 dc
    r1 1 0 1000
    .dc v1 0 25 5
    .print dc v(1) i(v1)
    .end

RISKS AND MITIGATION: none.

DATA COLLECTED:
      DC transfer characteristic  Thu Feb 14 13:05:08  2019
    -----------------------------------------------------------
    Index   v-sweep         v(1)            v1#branch
    -----------------------------------------------------------
    0       0.000000e+00    0.000000e+00    0.000000e+00
    1       5.000000e+00    5.000000e+00    -5.00000e-03
    2       1.000000e+01    1.000000e+01    -1.00000e-02
    3       1.500000e+01    1.500000e+01    -1.50000e-02
    4       2.000000e+01    2.000000e+01    -2.00000e-02
    5       2.500000e+01    2.500000e+01    -2.50000e-02

Analysis Time/Date = 13:06 on 14 February 2019

ANALYSIS: perfect agreement between data and hypothesis -- current is precisely
1/1000 of the applied voltage for all values.  Anything other than perfect
agreement would have probably meant my netlist was incorrect.  The negative
current values surprised me, but it seems this is just how SPICE interprets
normal current through a DC voltage source.

ERROR SOURCES: none.
```

As gratuitous as it may seem to perform experiments on a physical law as well-established as Ohm's Law, even the examples listed previously demonstrate opportunity for real learning. In the physical experiment example, the student should identify and explain why their data does not perfectly agree with the hypothesis, and this leads them naturally to consider sources of error. In the computer-simulated experiment, the student is struck by SPICE's convention of denoting regular current through a DC voltage source as being *negative* in sign, and this is also useful knowledge for future simulations. Scientific experiments are most interesting when things *do not* go as planned!

Aside from verifying well-established physical laws, simple experiments are extremely useful as educational tools for a wide range of purposes, including:

- Component familiarization (e.g. *Which terminals of this switch connect to the NO versus NC contacts?*)
- System testing (e.g. *How heavy of a load can my AC-DC power supply source before the semiconductor components reach their thermal limits?*)
- Learning programming languages (e.g. *Let's try to set up an "up" counter function in this PLC!*)

Above all, the priority here is to inculcate the habit of hypothesizing, running experiments, and analyzing the results. This experimental cycle not only serves as an excellent method for self-directed learning, but it also works exceptionally well for troubleshooting faults in complex systems, and for these reasons should be a part of every technician's and every engineer's education.

7.1.5 Constructing projects

Designing, constructing, and testing projects is a very effective means of practical education. Within a formal educational setting, projects are generally chosen (or at least vetted) by an instructor to ensure they may be reasonably completed within the allotted time of a course or program of study, and that they sufficiently challenge the student to learn certain important principles. In a self-directed environment, projects are just as useful as a learning tool but there is some risk of unwittingly choosing a project beyond one's abilities, which can lead to frustration.

Here are some general guidelines for managing projects:

- Define your goal(s) before beginning a project: what do you wish to achieve in building it? What, exactly, should the completed project *do*?
- Analyze your project prior to construction. Document it in appropriate forms (e.g. schematic diagrams), predict its functionality, anticipate all associated risks. In other words, *plan ahead.*
- Set a reasonable budget for your project, and stay within it.
- Identify any deadlines, and set reasonable goals to meet those deadlines.
- Beware of *scope creep*: the tendency to modify the project's goals before it is complete.
- Document your progress! An easy way to do this is to use photography or videography: take photos and/or videos of your project as it progresses. Document failures as well as successes, because both are equally valuable from the perspective of learning.

7.2 Experiment: contact and coil demonstration program

Conduct an experiment to explore the behavior of *contact* and *coil* Ladder Diagram instructions in a functioning PLC. Your experiment should test the behavior of all basic contact and coil instructions offered by the particular model of PLC available to you, and do so as simply as possible. You are strongly recommended *not* to write a PLC program that might serve some practical purpose, but rather to write the simplest possible version[16] of a proof-of-concept program merely showcasing how each instruction works.

An acceptable PLC program for this experiment must meet these three criteria:

- **Simple** – nothing "extra" included in the program to detract from the fundamental behavior of the instruction(s) being explored.
- **Complete** – nothing missing from the program relevant to the fundamental behavior of the instruction(s) being explored.
- **Clearly documented** – every rung clearly commented in your own words.

The practical purpose of this experiment is to serve as a reference for your future self modeling how each of your PLC's instructions work. This is why the experiment should not be a practical program, because programs written to perform some practical purpose become complicated by the details of that purpose.

Your analysis of each instruction's function should be written as *comments* in the Ladder Diagram program.

EXPERIMENT CHECKLIST:

- Prior to experimentation:

  $\boxed{\surd}$ Write an hypothesis (i.e. a detailed description of what you expect will happen) unambiguous enough that it could be disproven given the right data.

  $\boxed{\surd}$ Write a procedure to test the hypothesis, complete with adequate controls and documentation (e.g. schematic diagrams, programming code).

  $\boxed{\surd}$ Identify any risks (e.g. shock hazard, component damage) and write a mitigation plan based on best practices and component ratings.

- During experimentation:

  $\boxed{\surd}$ Safe practices followed at all times (e.g. no contact with energized circuit).

  $\boxed{\surd}$ Correct equipment usage according to manufacturer's recommendations.

  $\boxed{\surd}$ All data collected, ideally quantitative with full precision (i.e. no rounding).

[16]As few rungs of code as possible, with as few instructions on each rung as possible.

- After each experimental run:

  $\boxed{\surd}$ If the results fail to match the hypothesis, identify the error(s), correct the hypothesis and/or revise the procedure, and re-run the experiment.

  $\boxed{\surd}$ Identify any uncontrolled sources of error in the experiment.

- After all experimental re-runs:

  $\boxed{\surd}$ Save all data for future reference.

  $\boxed{\surd}$ Write an analysis of experimental results and lessons learned.

Challenges

- Science is an *iterative* process, and for this reason is never complete. Following the results of your experiment, what would you propose for your *next* hypothesis and *next* experimental procedure? Hint: if your experiment produced any unexpected results, exploring those unexpected results is often a very good basis for the next experiment!
- Where in the PLC's memory are the single-bit registers (e.g. input registers, output registers, and internal bit registers) located? What symbol(s) are used to address each one?
- What happens when two contact instructions are linked to the same bit address in the PLC's memory? Do these contact instructions operated differently, or identically?
- Does your PLC offer a special type of contact or other bit-level instruction to detect the *transition* of a bit from one state to another? If so, how is this instruction used?
- What happens when two coil instructions are linked to the same bit address in the PLC's memory, but driven to different states (e.g. one "energized" and the other "de-energized")?
- Experiment with using the *force* utility in your PLC to force certain bits to fixed values regardless of program operation. How will the operation of your program be affected if a particular input bit is forced? How will the operation of your program be affected if a particular output bit is forced? How can you tell from the live program display that bits have been forced to fixed values?

7.3 Experiment: PLC implementation of basic logic functions

Conduct an experiment demonstrating how a circuit using toggle switches (inputs) and a PLC may be used to implement multiple basic logic functions (e.g. AND, OR, NAND, NOR, NOT, XOR, XNOR) all operating off of the same inputs. Each logic function's output should be represented by its own discrete output on the PLC.

EXPERIMENT CHECKLIST:

- Prior to experimentation:

  $\boxed{\surd}$ Write an hypothesis (i.e. a detailed description of what you expect will happen) unambiguous enough that it could be disproven given the right data.

  $\boxed{\surd}$ Write a procedure to test the hypothesis, complete with adequate controls and documentation (e.g. schematic diagrams, programming code).

  $\boxed{\surd}$ Identify any risks (e.g. shock hazard, component damage) and write a mitigation plan based on best practices and component ratings.

- During experimentation:

  $\boxed{\surd}$ Safe practices followed at all times (e.g. no contact with energized circuit).

  $\boxed{\surd}$ Correct equipment usage according to manufacturer's recommendations.

  $\boxed{\surd}$ All data collected, ideally quantitative with full precision (i.e. no rounding).

- After each experimental run:

  $\boxed{\surd}$ If the results fail to match the hypothesis, identify the error(s), correct the hypothesis and/or revise the procedure, and re-run the experiment.

  $\boxed{\surd}$ Identify any uncontrolled sources of error in the experiment.

- After all experimental re-runs:

  $\boxed{\surd}$ Save all data for future reference.

  $\boxed{\surd}$ Write an analysis of experimental results and lessons learned.

Challenges

- How would you program logic functions having more than two inputs?
- Science is an *iterative* process, and for this reason is never complete. Following the results of your experiment, what would you propose for your *next* hypothesis and *next* experimental procedure? Hint: if your experiment produced any unexpected results, exploring those unexpected results is often a very good basis for the next experiment!
- Write a different PLC program to implement the exact same logic functions.

7.4 Experiment: PLC implementation of an arbitrary truth table

Conduct an experiment demonstrating how a circuit using toggle switches (inputs) and a PLC may be used to implement an arbitrary truth table for a four-input combinational logic function. A truth table template is given here for your use, to arbitrarily write "1" and "0" states in the output column:

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

EXPERIMENT CHECKLIST:

- Prior to experimentation:

  $\boxed{\surd}$ Write an hypothesis (i.e. a detailed description of what you expect will happen) unambiguous enough that it could be disproven given the right data.

  $\boxed{\surd}$ Write a procedure to test the hypothesis, complete with adequate controls and documentation (e.g. schematic diagrams, programming code).

  $\boxed{\surd}$ Identify any risks (e.g. shock hazard, component damage) and write a mitigation plan based on best practices and component ratings.

- <u>During experimentation:</u>

  $\boxed{\surd}$ Safe practices followed at all times (e.g. no contact with energized circuit).

  $\boxed{\surd}$ Correct equipment usage according to manufacturer's recommendations.

  $\boxed{\surd}$ All data collected, ideally quantitative with full precision (i.e. no rounding).

- <u>After each experimental run:</u>

  $\boxed{\surd}$ If the results fail to match the hypothesis, identify the error(s), correct the hypothesis and/or revise the procedure, and re-run the experiment.

  $\boxed{\surd}$ Identify any uncontrolled sources of error in the experiment.

- <u>After all experimental re-runs:</u>

  $\boxed{\surd}$ Save all data for future reference.

  $\boxed{\surd}$ Write an analysis of experimental results and lessons learned.

$\boxed{\text{Challenges}}$

- Identify a truth table function possible to implement with no PLC at all.
- Science is an *iterative* process, and for this reason is never complete. Following the results of your experiment, what would you propose for your *next* hypothesis and *next* experimental procedure? Hint: if your experiment produced any unexpected results, exploring those unexpected results is often a very good basis for the next experiment!
- It is possible to implement your chosen logic function using *only* toggle switches and no PLC?
- Write a different PLC program to implement the exact same logic function.

7.5 Experiment: PLC-controlled motor starter

Devise and execute an experiment to control the starting and stopping of either an AC or a DC motor using a PLC to perform the latching function and two momentary-contact switches to signal "Start" and "Stop". The "Start" switch should be wired normally-open, and the "Stop" switch normally-closed, so that any open switch wiring faults favor a stopped motor rather than a running motor.

EXPERIMENT CHECKLIST:

- Prior to experimentation:

  $\boxed{\surd}$ Write an hypothesis (i.e. a detailed description of what you expect will happen) unambiguous enough that it could be disproven given the right data.

  $\boxed{\surd}$ Write a procedure to test the hypothesis, complete with adequate controls and documentation (e.g. schematic diagrams, programming code).

  $\boxed{\surd}$ Identify any risks (e.g. shock hazard, component damage) and write a mitigation plan based on best practices and component ratings.

- During experimentation:

  $\boxed{\surd}$ Safe practices followed at all times (e.g. no contact with energized circuit).

  $\boxed{\surd}$ Correct equipment usage according to manufacturer's recommendations.

  $\boxed{\surd}$ All data collected, ideally quantitative with full precision (i.e. no rounding).

- After each experimental run:

  $\boxed{\surd}$ If the results fail to match the hypothesis, identify the error(s), correct the hypothesis and/or revise the procedure, and re-run the experiment.

  $\boxed{\surd}$ Identify any uncontrolled sources of error in the experiment.

- After all experimental re-runs:

  $\boxed{\surd}$ Save all data for future reference.

  $\boxed{\surd}$ Write an analysis of experimental results and lessons learned.

$\boxed{\text{Challenges}}$

- Science is an *iterative* process, and for this reason is never complete. Following the results of your experiment, what would you propose for your *next* hypothesis and *next* experimental procedure? Hint: if your experiment produced any unexpected results, exploring those unexpected results is often a very good basis for the next experiment!

- Identify some alternative means for interposing between the PLC and the electric motor, since it is unlikely you will find a PLC with a discrete output channel rated to directly switch motor current.
- Identify some alternative programming strategies for implementing the necessary latching function.

7.6 Experiment: counter demonstration program

Conduct an experiment to explore the behavior of *counter* Ladder Diagram instructions in a functioning PLC. Your experiment should test the behavior of all basic counter instructions offered by the particular model of PLC available to you, and do so as simply as possible. You are strongly recommended *not* to write a PLC program that might serve some practical purpose, but rather to write the simplest possible version[17] of a proof-of-concept program merely showcasing how each instruction works.

An acceptable PLC program for this experiment must meet these three criteria:

- **Simple** – nothing "extra" included in the program to detract from the fundamental behavior of the instruction(s) being explored.
- **Complete** – nothing missing from the program relevant to the fundamental behavior of the instruction(s) being explored.
- **Clearly documented** – every rung clearly commented in your own words.

The practical purpose of this experiment is to serve as a reference for your future self modeling how each of your PLC's instructions work. This is why the experiment should not be a practical program, because programs written to perform some practical purpose become complicated by the details of that purpose.

Your analysis of each instruction's function should be written as *comments* in the Ladder Diagram program.

EXPERIMENT CHECKLIST:

- Prior to experimentation:

  ✓ Write an hypothesis (i.e. a detailed description of what you expect will happen) unambiguous enough that it could be disproven given the right data.

  ✓ Write a procedure to test the hypothesis, complete with adequate controls and documentation (e.g. schematic diagrams, programming code).

  ✓ Identify any risks (e.g. shock hazard, component damage) and write a mitigation plan based on best practices and component ratings.

- During experimentation:

  ✓ Safe practices followed at all times (e.g. no contact with energized circuit).

  ✓ Correct equipment usage according to manufacturer's recommendations.

  ✓ All data collected, ideally quantitative with full precision (i.e. no rounding).

[17] As few rungs of code as possible, with as few instructions on each rung as possible.

- After each experimental run:

  $\boxed{\surd}$ If the results fail to match the hypothesis, identify the error(s), correct the hypothesis and/or revise the procedure, and re-run the experiment.

  $\boxed{\surd}$ Identify any uncontrolled sources of error in the experiment.

- After all experimental re-runs:

  $\boxed{\surd}$ Save all data for future reference.

  $\boxed{\surd}$ Write an analysis of experimental results and lessons learned.

$\boxed{\text{Challenges}}$

- Science is an *iterative* process, and for this reason is never complete. Following the results of your experiment, what would you propose for your *next* hypothesis and *next* experimental procedure? Hint: if your experiment produced any unexpected results, exploring those unexpected results is often a very good basis for the next experiment!
- How can you make a single counter both *increment* (count up) and *decrement* (count down)?
- How many bits are used by your PLC in each counter instruction? How can you tell?
- Can you "force" a counter to some accumulator value in the same way you can force a discrete bit to a certain value?
- Is it possible to "pre-load" a counter to some non-zero value at the command of a single bit, such as a pushbutton switch being pressed?

7.7 Experiment: timer demonstration program

Conduct an experiment to explore the behavior of *timer* Ladder Diagram instructions in a functioning PLC. Your experiment should test the behavior of all basic timer instructions offered by the particular model of PLC available to you, and do so as simply as possible. You are strongly recommended *not* to write a PLC program that might serve some practical purpose, but rather to write the simplest possible version[18] of a proof-of-concept program merely showcasing how each instruction works.

An acceptable PLC program for this experiment must meet these three criteria:

- **Simple** – nothing "extra" included in the program to detract from the fundamental behavior of the instruction(s) being explored.
- **Complete** – nothing missing from the program relevant to the fundamental behavior of the instruction(s) being explored.
- **Clearly documented** – every rung clearly commented in your own words.

The practical purpose of this experiment is to serve as a reference for your future self modeling how each of your PLC's instructions work. This is why the experiment should not be a practical program, because programs written to perform some practical purpose become complicated by the details of that purpose.

Your analysis of each instruction's function should be written as *comments* in the Ladder Diagram program.

EXPERIMENT CHECKLIST:

- Prior to experimentation:

  ☑ Write an hypothesis (i.e. a detailed description of what you expect will happen) unambiguous enough that it could be disproven given the right data.

  ☑ Write a procedure to test the hypothesis, complete with adequate controls and documentation (e.g. schematic diagrams, programming code).

  ☑ Identify any risks (e.g. shock hazard, component damage) and write a mitigation plan based on best practices and component ratings.

- During experimentation:

  ☑ Safe practices followed at all times (e.g. no contact with energized circuit).

  ☑ Correct equipment usage according to manufacturer's recommendations.

  ☑ All data collected, ideally quantitative with full precision (i.e. no rounding).

[18] As few rungs of code as possible, with as few instructions on each rung as possible.

- After each experimental run:

  $\boxed{\surd}$ If the results fail to match the hypothesis, identify the error(s), correct the hypothesis and/or revise the procedure, and re-run the experiment.

  $\boxed{\surd}$ Identify any uncontrolled sources of error in the experiment.

- After all experimental re-runs:

  $\boxed{\surd}$ Save all data for future reference.

  $\boxed{\surd}$ Write an analysis of experimental results and lessons learned.

$\boxed{\text{Challenges}}$

- Science is an *iterative* process, and for this reason is never complete. Following the results of your experiment, what would you propose for your *next* hypothesis and *next* experimental procedure? Hint: if your experiment produced any unexpected results, exploring those unexpected results is often a very good basis for the next experiment!
- Can a timer instruction be made to count backwards?
- How many bits are used by your PLC in each timer instruction? How can you tell?
- Can you "force" a timer to some accumulator value in the same way you can force a discrete bit to a certain value?
- Is it possible to "pre-load" a timer to some non-zero value at the command of a single bit, such as a pushbutton switch being pressed?

7.8 Experiment: PLC-controlled inrush-limiting motor starter

Devise and execute an experiment to control the starting and stopping of either an AC or a DC motor using a PLC to perform the latching function and sequencing of multiple contactors to limit the motor's inrush current. Two momentary-contact switches will signal "Start" and "Stop".

EXPERIMENT CHECKLIST:

- Prior to experimentation:

  $\boxed{\surd}$ Write an hypothesis (i.e. a detailed description of what you expect will happen) unambiguous enough that it could be disproven given the right data.

  $\boxed{\surd}$ Write a procedure to test the hypothesis, complete with adequate controls and documentation (e.g. schematic diagrams, programming code).

  $\boxed{\surd}$ Identify any risks (e.g. shock hazard, component damage) and write a mitigation plan based on best practices and component ratings.

- During experimentation:

  $\boxed{\surd}$ Safe practices followed at all times (e.g. no contact with energized circuit).

  $\boxed{\surd}$ Correct equipment usage according to manufacturer's recommendations.

  $\boxed{\surd}$ All data collected, ideally quantitative with full precision (i.e. no rounding).

- After each experimental run:

  $\boxed{\surd}$ If the results fail to match the hypothesis, identify the error(s), correct the hypothesis and/or revise the procedure, and re-run the experiment.

  $\boxed{\surd}$ Identify any uncontrolled sources of error in the experiment.

- After all experimental re-runs:

  $\boxed{\surd}$ Save all data for future reference.

  $\boxed{\surd}$ Write an analysis of experimental results and lessons learned.

$\boxed{\text{Challenges}}$

- Science is an *iterative* process, and for this reason is never complete. Following the results of your experiment, what would you propose for your *next* hypothesis and *next* experimental procedure? Hint: if your experiment produced any unexpected results, exploring those unexpected results is often a very good basis for the next experiment!
- Identify some alternative means for interposing between the PLC and the electric motor, since it is unlikely you will find a PLC with a discrete output channel rated to directly switch motor current.

- Identify some alternative programming strategies for implementing the necessary latching function.

7.9 Experiment: HMI display of PLC bits and words

Devise and execute an experiment configuring a Human-Machine Interface (HMI) unit to read data bits and data words inside of a PLC. Recommended bits and words to read include those associated with counter and timer instructions, so that the HMI display will show both accumulated values and I/O bits for these PLC instructions while the PLC is running.

EXPERIMENT CHECKLIST:

- Prior to experimentation:
  - ☑ Write an hypothesis (i.e. a detailed description of what you expect will happen) unambiguous enough that it could be disproven given the right data.
  - ☑ Write a procedure to test the hypothesis, complete with adequate controls and documentation (e.g. schematic diagrams, programming code).
  - ☑ Identify any risks (e.g. shock hazard, component damage) and write a mitigation plan based on best practices and component ratings.
- During experimentation:
  - ☑ Safe practices followed at all times (e.g. no contact with energized circuit).
  - ☑ Correct equipment usage according to manufacturer's recommendations.
  - ☑ All data collected, ideally quantitative with full precision (i.e. no rounding).
- After each experimental run:
  - ☑ If the results fail to match the hypothesis, identify the error(s), correct the hypothesis and/or revise the procedure, and re-run the experiment.
  - ☑ Identify any uncontrolled sources of error in the experiment.
- After all experimental re-runs:
  - ☑ Save all data for future reference.
  - ☑ Write an analysis of experimental results and lessons learned.

Challenges

- Science is an *iterative* process, and for this reason is never complete. Following the results of your experiment, what would you propose for your *next* hypothesis and *next* experimental procedure? Hint: if your experiment produced any unexpected results, exploring those unexpected results is often a very good basis for the next experiment!
- Identify some of the different digital data types (e.g. signed integer, unsigned integer, floating-point, Boolean) readable by the HMI from the PLC, and the word size used by each PLC instruction.

7.10 Experiment: arithmetic demonstration program

Conduct an experiment to explore the behavior of *arithmetic* Ladder Diagram instructions in a functioning PLC. Your experiment should test the behavior of all basic arithmetic operations offered by the particular model of PLC available to you, and do so as simply as possible. You are strongly recommended *not* to write a PLC program that might serve some practical purpose, but rather to write the simplest possible version[19] of a proof-of-concept program merely showcasing how each instruction works.

An acceptable PLC program for this experiment must meet these three criteria:

- **Simple** – nothing "extra" included in the program to detract from the fundamental behavior of the instruction(s) being explored.
- **Complete** – nothing missing from the program relevant to the fundamental behavior of the instruction(s) being explored.
- **Clearly documented** – every rung clearly commented in your own words.

The practical purpose of this experiment is to serve as a reference for your future self modeling how each of your PLC's instructions work. This is why the experiment should not be a practical program, because programs written to perform some practical purpose become complicated by the details of that purpose.

Your analysis of each instruction's function should be written as *comments* in the Ladder Diagram program.

EXPERIMENT CHECKLIST:

- Prior to experimentation:

  ✓ Write an hypothesis (i.e. a detailed description of what you expect will happen) unambiguous enough that it could be disproven given the right data.

  ✓ Write a procedure to test the hypothesis, complete with adequate controls and documentation (e.g. schematic diagrams, programming code).

  ✓ Identify any risks (e.g. shock hazard, component damage) and write a mitigation plan based on best practices and component ratings.

- During experimentation:

  ✓ Safe practices followed at all times (e.g. no contact with energized circuit).

  ✓ Correct equipment usage according to manufacturer's recommendations.

  ✓ All data collected, ideally quantitative with full precision (i.e. no rounding).

[19] As few rungs of code as possible, with as few instructions on each rung as possible.

- After each experimental run:

  $\boxed{\surd}$ If the results fail to match the hypothesis, identify the error(s), correct the hypothesis and/or revise the procedure, and re-run the experiment.

  $\boxed{\surd}$ Identify any uncontrolled sources of error in the experiment.

- After all experimental re-runs:

  $\boxed{\surd}$ Save all data for future reference.

  $\boxed{\surd}$ Write an analysis of experimental results and lessons learned.

$\boxed{\text{Challenges}}$

- Science is an *iterative* process, and for this reason is never complete. Following the results of your experiment, what would you propose for your *next* hypothesis and *next* experimental procedure? Hint: if your experiment produced any unexpected results, exploring those unexpected results is often a very good basis for the next experiment!
- What happens when you instruct an arithmetic operation in the PLC to do something mathematically undefined, such as dividing by zero?
- Do the PLC's arithmetic instructions operate on integer values, floating-point values, or both?

7.11 Experiment: data transfer demonstration program

Conduct an experiment to explore the behavior of *data transfer* (i.e. communication/networking) Ladder Diagram instructions in a functioning PLC. Your experiment should test the behavior of all basic data transfer instructions offered by the particular model of PLC available to you, and do so as simply as possible. You are strongly recommended *not* to write a PLC program that might serve some practical purpose, but rather to write the simplest possible version[20] of a proof-of-concept program merely showcasing how each instruction works.

An acceptable PLC program for this experiment must meet these three criteria:

- **Simple** – nothing "extra" included in the program to detract from the fundamental behavior of the instruction(s) being explored.
- **Complete** – nothing missing from the program relevant to the fundamental behavior of the instruction(s) being explored.
- **Clearly documented** – every rung clearly commented in your own words.

The practical purpose of this experiment is to serve as a reference for your future self modeling how each of your PLC's instructions work. This is why the experiment should not be a practical program, because programs written to perform some practical purpose become complicated by the details of that purpose.

Your analysis of each instruction's function should be written as *comments* in the Ladder Diagram program.

EXPERIMENT CHECKLIST:

- Prior to experimentation:

  ✓ Write an hypothesis (i.e. a detailed description of what you expect will happen) unambiguous enough that it could be disproven given the right data.

  ✓ Write a procedure to test the hypothesis, complete with adequate controls and documentation (e.g. schematic diagrams, programming code).

  ✓ Identify any risks (e.g. shock hazard, component damage) and write a mitigation plan based on best practices and component ratings.

- During experimentation:

  ✓ Safe practices followed at all times (e.g. no contact with energized circuit).

  ✓ Correct equipment usage according to manufacturer's recommendations.

  ✓ All data collected, ideally quantitative with full precision (i.e. no rounding).

[20] As few rungs of code as possible, with as few instructions on each rung as possible.

- After each experimental run:

  $\boxed{\surd}$ If the results fail to match the hypothesis, identify the error(s), correct the hypothesis and/or revise the procedure, and re-run the experiment.

  $\boxed{\surd}$ Identify any uncontrolled sources of error in the experiment.

- After all experimental re-runs:

  $\boxed{\surd}$ Save all data for future reference.

  $\boxed{\surd}$ Write an analysis of experimental results and lessons learned.

$\boxed{\text{Challenges}}$

- Science is an *iterative* process, and for this reason is never complete. Following the results of your experiment, what would you propose for your *next* hypothesis and *next* experimental procedure? Hint: if your experiment produced any unexpected results, exploring those unexpected results is often a very good basis for the next experiment!
- How are the instructions configured or selected to either send or receive data?
- How the communicating PLCs *addressed* on the interconnecting network, if at all.
- How is the data communication triggered? In other words, what signals the starting time of each message?
- How are communication errors signaled within the PLC program?
- What happens when you unplug the communication cable(s) during data communication?

7.12 Project: PLC-controlled system

Wire and program a PLC to automatically monitor and/or control some small system. Ideas include, but are not limited to:

- Control the starting and stopping of an electric motor, using hand switches for Start and Stop initiation, with the PLC performing such functions as latching, counting the number of start/stop cycles, calculating total motor run time, flagging maintenance alerts after a certain amount of run time is reached, etc.
- Automatic starting and stopping of an air compressor to maintain relatively constant air pressure in a vessel.
- Retrofit the controls on a consumer-grade appliance, removing the factory-original control system and replacing it with a PLC. Appliance suggestions include a bread-making machine, a clothes-washing machine, a coffee making machine, a food processor, a convection oven, a pressure cooker, etc.
- Build a security alarm system for a home, sensing door and window statuses, and providing a means for only authorized people to enter the room.
- Create an alarm clock complete with audio and visual alert capabilities. Provide a means for any user to calibrate the clock against a known time standard and also be able to set wake-up times without having to access the PLC coding.
- Build a weather station monitoring temperature, wind speed, wind direction, etc. Provide alarms for cold-weather and hot-weather conditions, average daily temperature, etc.
- Build a solar tracker, automatically aiming a solar collector (photovoltaic panel, or concentrating mirror) directly at the sun using servo motors.
- Build a miniature traffic light system for a set of intersections, sequencing green/amber/red traffic lights, detecting cars on the road (using proximity switches), etc.
- Build a robot capable of navigating around a room, sensing and avoiding obstacles.

PROJECT CHECKLIST:

- Prior to construction:
  - $\boxed{\surd}$ Prototype diagram(s) and description of project scope.
  - $\boxed{\surd}$ Risk assessment/mitigation plan.
  - $\boxed{\surd}$ Timeline and action plan.
- During construction:
  - $\boxed{\surd}$ Safe work habits (e.g. no contact made with energized circuit at any time).
  - $\boxed{\surd}$ Correct equipment usage according to manufacturer's recommendations.

- $\boxed{\surd}$ Timeline and action plan amended as necessary.
- $\boxed{\surd}$ Maintain the originally-planned project scope (i.e. avoid adding features!).

- After completion:
  - $\boxed{\surd}$ All functions tested against original plan.
  - $\boxed{\surd}$ Full, accurate, and appropriate documentation of all project details.
  - $\boxed{\surd}$ Complete bill of materials.
  - $\boxed{\surd}$ Written summary of lessons learned.

$\boxed{\text{Challenges}}$

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

Appendix A

Problem-Solving Strategies

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

- Study principles, not procedures. Don't be satisfied with merely knowing how to compute solutions – learn *why* those solutions work.

- Identify what it is you need to solve, identify all relevant data, identify all units of measurement, identify any general principles or formulae linking the given information to the solution, and then identify any "missing pieces" to a solution. Annotate all diagrams with this data.

- Sketch a diagram to help visualize the problem. When building a real system, always devise a plan for that system and analyze its function *before* constructing it.

- Follow the units of measurement and meaning of every calculation. If you are ever performing mathematical calculations as part of a problem-solving procedure, and you find yourself unable to apply each and every intermediate result to some aspect of the problem, it means you don't understand what you are doing. Properly done, every mathematical result should have practical meaning for the problem, and not just be an abstract number. You should be able to identify the proper units of measurement for each and every calculated result, and show where that result fits into the problem.

- Perform "thought experiments" to explore the effects of different conditions for theoretical problems. When troubleshooting real systems, perform *diagnostic tests* rather than visually inspecting for faults, the best diagnostic test being the one giving you the most information about the nature and/or location of the fault with the fewest steps.

- Simplify the problem until the solution becomes obvious, and then use that obvious case as a model to follow in solving the more complex version of the problem.

- Check for exceptions to see if your solution is incorrect or incomplete. A good solution will work for *all* known conditions and criteria. A good example of this is the process of testing scientific hypotheses: the task of a scientist is not to find support for a new idea, but rather to *challenge* that new idea to see if it holds up under a battery of tests. The philosophical principle of *reductio ad absurdum* (i.e. disproving a general idea by finding a specific case where it fails) is useful here.

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

Appendix B

Instructional philosophy

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- No text should be considered fully and adequately read unless and until you can express every idea *in your own words, using your own examples.*

- You should always articulate your thoughts as you read the text, noting points of agreement, confusion, and epiphanies. Feel free to print the text on paper and then write your notes in the margins. Alternatively, keep a journal for your own reflections as you read. This is truly a helpful tool when digesting complicated concepts.

- Never take the easy path of highlighting or underlining important text. Instead, *summarize* and/or *comment* on the text using your own words. This actively engages your mind, allowing you to more clearly perceive points of confusion or misunderstanding on your own.

- A very helpful strategy when learning new concepts is to place yourself in the role of a teacher, if only as a mental exercise. Either explain what you have recently learned to someone else, or at least *imagine* yourself explaining what you have learned to someone else. The simple act of having to articulate new knowledge and skill forces you to take on a different perspective, and will help reveal weaknesses in your understanding.

- Perform each and every mathematical calculation and thought experiment shown in the text on your own, referring back to the text to see that your results agree. This may seem trivial and unnecessary, but it is critically important to ensuring you actually understand what is presented, especially when the concepts at hand are complicated and easy to misunderstand. Apply this same strategy to become proficient in the use of *circuit simulation software*, checking to see if your simulated results agree with the results shown in the text.

- Above all, recognize that learning is hard work, and that a certain level of frustration is unavoidable. There are times when you will struggle to grasp some of these concepts, and that struggle is a natural thing. Take heart that it will yield with persistent and varied[4] effort, and never give up!

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

To high standards of education,

Tony R. Kuphaldt

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

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

Appendix C

Tools used

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

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

The `GNU/Linux` computer operating system

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

Bram Moolenaar's `Vim` text editor

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

Donald Knuth's TeX typesetting system

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

Leslie Lamport's LaTeX extensions to TeX

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

Tim Edwards' `Xcircuit` drafting program

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

Gimp graphic image manipulation program

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

SPICE circuit simulation program

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

Andrew D. Hwang's ePiX mathematical visualization programming library

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

`gnuplot` mathematical visualization software

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

`Python` programming language

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

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

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

Section 1 – Definitions.

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

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

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

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

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

Section 3 – License Conditions.

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,

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.

6 December 2022 – corrected an error in the Simplified Tutorial where I wrote "parallel" instead of "series" (!), and also added questions to the Introduction chapter. Also edited image_6221 to add a symbol showing normally-closed switch B to be in the open state, and added some content to the Simplified Tutorial discussing HMIs.

28 November 2022 – placed questions at the top of the itemized list in the Introduction chapter prompting students to devise experiments related to the tutorial content.

6 May 2022 – re-wrote Simplified Tutorial chapter.

5 May 2022 – added a Case Tutorial chapter.

10 May 2021 – commented out or deleted empty chapters.

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

29 June 2020 – clarified an answer in the "Allen-Bradley counter program" question regarding the number of times the motor will be allowed to start up.

20 May 2019 – added questions related to Human-Machine Interfaces (HMIs).

12 May 2019 – added more SELogic examples (S-R latch) to Technical References section, and made minor edits to other instruction examples in that section. Also added several new questions in Conceptual, Quantitative, and Diagnostic Reasoning sections. Also, re-structured sections and subsections related to Ladder Diagram (LD) programming. Also, added a section on Human-Machine Interfaces, or HMIs. Also, added an Experiment on using an HMI to read PLC bits and words.

9 May 2019 – added some PLC counter-based questions.

9 May 2019 – added more questions as well as a Project for a PLC-controlled system.

1 May 2019 – added more questions in all categories (Conceptual, Quantitative, and Diagnostic), and also included "dry" versus "wet" output contacts for PLC discrete outputs.

23 April 2019 – added more questions in all categories (Conceptual, Quantitative, and Diagnostic), and also included "dry" versus "wet" output contacts for PLC discrete outputs.

20 April 2019 – added some more Experiments.

19 April 2019 – minor edits to the demonstration program experiments.

16 April 2019 – continued adding content to the Technical References section on SELogic control equations for SEL's protective relays and programmable controllers.

15 April 2019 – added a Technical References section on SELogic control equations for SEL's protective relays and programmable controllers.

14 April 2019 – added a lot of content to the Full Tutorial, as well as to the Technical References chapter.

9 April 2019 – continued adding more content to the Simplified Tutorial.

8 April 2019 – added a Technical Reference section comparing basic Ladder Diagram instructions of some common PLC models.

7 April 2019 – added content to Introduction and Tutorials.

30 March 2019 – document first created.

Index