Programmable Logic Controllers

Prof. John McLeod

ECE9047/9407, Winter 2021

This lesson briefly introduces programmable logic controllers (PLCs). PLCs are essentially a category of embedded systems, widely used in industry. PLCs are usually programmed using relay ladder logic, also introduced here. This content is heavily based on Chapters 1, 2, and 5 of F. Petruzella's text, *Programmable Logic Controllers*.

Programmable Logic Controllers

Programmable logic controllers (PLCs) are widely used in industrial process control technology. They are essentially a category of embedded systems designed to seamlessly replace outdated wired relay control circuits. PLCs are used in many different situations, but all PLCs are designed to meet some rigid design requirements.

- PLCs are for **control systems**. Consequently, they must accept multiple input data lines and produce multiple output control signals. Intensive computing power is typically not necessary to process the input to generate output.
- PLCs are deployed in the field — typically in a factory — and must be **rugged**. PLCs must continue to operate reliably over a wide range of temperature and humidity, and after exposure to electromagnetic interference and mechanical vibrations.
- PLCs should act as a **real-time system**, where a constant flow of input data is converted to output data almost instantly. This requires efficient programming and possibly some degree of I/O parallelization.
- PLCs should be very **reliable**, both in terms of hardware and software.

Because of their importance to industry, most PLCs systems are proprietary. Compared to other embedded systems we have studied, PLCs are quite expensive (hundreds to thousands of dollars per device). However a failing control system in a factory setting can be very expensive, as it will influence the production line and possibly cause damage to other equipment. Given the above design aspects of a PLC, over their lifetime they are often considerably cheaper for industrial processes than other options that use cheaper hardware.

A major part of the reliability of PLCs comes from their programming language. Unlike other embedded systems whose functionality is programmed in Assembly language (or in C, then converted to Assembly by the compiler), PLCs typically have a pre-installed and proprietary minimalist operating system. The desired functionality of the PLC is implemented on top of this operating system using a simple programming language called **relay ladder logic**. The simple and visual nature of this programming language helps make it easy to spot bugs.

PLC Hardware

All PLCs have the same fundamental components. A PLC is an embedded system, but given their rather specialized use the components of a PLC are considerably more limited than those we've seen previously in other embedded systems.

- All PLCs have a **microprocessor** for controlling input/output and processing data.
- All PLCs have some on-board **memory**, at minimum enough for storing the operating program and temporary storage for local variables used for data processing. Some PLCs may have additional ROM used to record operations data.
- All PLCs have a **power supply**. Virtually every PLC is deployed in an environment with plenty of power infrastructure available, so the power supply is almost always connected to a wall outlet or the power grid. Power efficiency is not a primary design characteristic of a PLC. As keeping a PLC operating is often critical, many PLCs have battery backups (or even backup generators) in case of power failures. Again, virtually every PLC is deployed to a fixed position so small size/weight is also not a primary design characteristic.
- All PLCs have some **input/output ports**. Generally, this I/O is electrical (often digital) and is a collection of single bits. Many PLCs are **modular** and rack-mounted — extra blocks

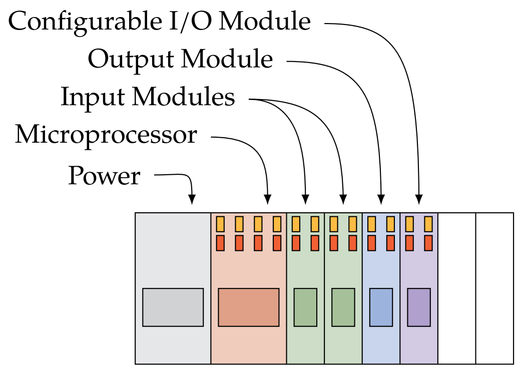

Figure 1: Super-realistic image of a rack-mounted PLC, with two empty slots. Image adapted from Figure 1-10 in Petruzella.

of I/O pins can be slotted into the rack as needed, sometimes even **dynamically** during PLC operation.

- Some PLCs may also have **communications** ports. Typically this is to keep the factory control room appraised of the real-time PLC operating conditions, and/or send the operations data to external storage. Sometimes this may also be used to control PLC operation or reprogram the PLC.

PLCs need some peripheral, either communications or I/O, to program them. It is probably most common to program a PLC using a normal personal computer, and install that program on the PLC over ethernet. Hand-held programmers also exist as intermediates for PLCs that do not have communications peripherals or are not connected to the network. The hand-held device is plugged into a personal computer to receive the program, then taken over to the PLC to install the program. [1]

[1] Security was not traditionally an important consideration for PLCs, but the Stuxnet virus changed that.

Overview of Relay Ladder Logic

PLCs were developed in the 1960s for the automotive industry as a replacement for relay control logic circuits. A relay is an electrically operated switch — originally an electromagnetic coil and a magnetic lever, later a transistor. These switches were hard-wired into a Boolean logic circuit to form the control logic necessary for running industrial processes. Hard-wired circuits are robust and can operate at very high speeds, but are expensive to modify. Complicated circuits are also difficult to trace to check for problems.
PLCs addressed several of these issues, but for backwards compatibility they retained the same "programming language" originally used to design the hard-wired circuits: **relay ladder logic**. [2]

- Relay ladder logic was recently ranked as the 50th most popular programming language. [3]
- Those of you who enjoy programming and are familiar with multiple programming languages will be excited to learn about relay ladder logic. After learning about it, you may say "what..? that isn't a programming language."

Relay ladder logic is a **rule-based langauge** rather than a **procedural language**, like most of the other programming languages you are probably familiar with. The entire point of relay ladder logic is to be a simple, intuitive, and reliable method of programming a microcontroller to act like the old wired relay circuits. The language is very visual, and is really a representation of the diagrams that used to be used to describe the wired circuits used in control logic.

[2] This was also to avoid trying to teach programming to the old-fashioned electromechanical engineers.

[3] N. Diakopoulous, M. Bagavandas, and G. Singh, "Interactive: The Top Programming Languages", *IEEE Spectrum* (2019), click here: Link to article. Lots of fun comments there too about how the list is trash, but that's the internet for you!

Relay Ladder Logic Syntax

Relay ladder logic is designed to mimic a wired relay circuit. In an actual circuit the logic is evaluated essentially simultaneously. In a PLC the logic is evaluated sequentially, but typically fast enough that appears simultaneous.

Definition: A **ladder** is a relay ladder logic program. It is executed sequentially top-to-bottom.

Definition: A **rung** is part of a relay ladder logic program. It is sort of equivalent to a line of code in a conventional programming language. **Rungs** are executed left-to-right.

There are only four basic elements on a **rung**. Because of the century-long progression from electromagnetic relay circuits to PLCs, there are a lot of different names for these elements. Because engineerings (especially old industrial engineers) are resistant to change, these names are all still in use.

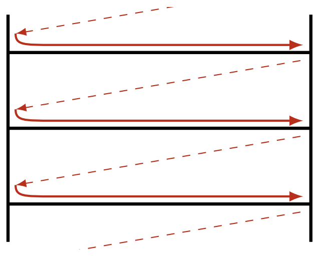
Figure 2: The name "relay ladder logic" is because visually, the code resembles a ladder! At least sort of. Engineers sometimes aren't very good with descriptive names. Also, unlike a ladder, the rungs are read top-to-bottom, and each rung is read left-to-right. Do not use a real ladder this way! You may fall.

Definition: An **input**, also referred to as a **checker**, a **contact**, or a **switch**, is a one-bit source of input data for the program.

Definition: An **output**, also referred to as an **actuator** or a **coil**, is a one-bit output data source.

Pretty simple, eh? The four basic elements in relay ladder logic are these single-bit inputs/outputs, classified by whether they are active low- or active-high.

Syntax: A **normally-open input** returns a logical zero unless that input is active. In conventional electronics, this is described as **active-high**. In an electromechanical circuit, this might be an push-button that only allows electricity to flow when it is pressed.

Syntax: A **normally-closed input** returns a logical one unless that input is active. In conventional electronics, this is described as **active-low**. In an electromechanical circuit, this might be an push-button that breaks the flow of electricity when it is pressed. [4]

[4] Circuit breakers are often undesired in digital logic, because they lead to floating voltages which have an undefined logic value. But this is because digital logic uses *voltage* as the control. The old electromechanical logic circuits often used *current* as the control, so a open circuit (with zero current) has an unambiguous logical value.

Syntax: A **normally-inactive output** returns a logical zero unless that output is activated. In conventional electronics, this is again described as **active-high**. In an electromechanical circuit, this might be motor directly wired to the circuit — it only turns on when current flows.

Syntax: A **normally-active output** returns a logical one unless that output is activated. In conventional electronics, this is described as **active-low**. In an electromechanical circuit, this might be a fly-wheel with an electromagnet as a brake — the fly-wheel stops spinning when current flows through the electromagnet.

There are additional block elements in relay ladder logic (timers, accumulators, comparators, etc.) that take advantage of the full functionality of an embedded system compared to a wired relay circuit, but we will not worry about those in this course.

Symbol: A **normally-open input** is typeset as `-[ ]-`, `-| |-`, or `-] [-`,[5] I will draw it as ⊣ ⊢.

Symbol: A **normally-closed input** is typeset as `-[\]-`, `-[/]-`, `-|\|-`, `-|/|-`, `-]\[-`, or `-]/[-`. I will draw it as ⊣/⊢.

Symbol: A **normally-inactive output** is typeset as `-( )-`. I will draw it as ⊣( )⊢, although some standards draw it as a full circle.

Symbol: A **normally-active output** is typeset as `-(\)-` or `-(/)-`. I will draw it as ⊣(/)⊢, although again some standards draw it as a full circle.

5 Petruzella's textbook uses `-] [-`, but most other sources I have found seem to prefer `-[ ]-`.

A relay ladder logic program is written as a sequence of **rungs** between the side **rails** of the **ladder**, as shown in Figure 3.

- Conceptually, the left **rail** represents high voltage and the low **rail** represents low voltage.
- Depending on the state of the **input** elements on each **rung**, electrical power may or may not be supplied to the **output** elements on that **rung**.
- This is just a conceptual diagram, however — it doesn't represent a real circuit. Remember that each **rung** is evaluated sequentially, top-to-bottom.[6]
- Often the **output** elements on one **rung** are re-purposed as **input** elements on subsequent rungs, making it simpler to construct large conditional blocks

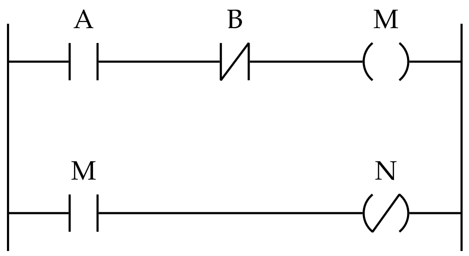

Figure 3: Simple relay ladder logic program. Note that M is used as both the output of the first rung and as an input on the second rung.

6 Hopefully you all have enough circuit background to recognize that this would *not* happen in a real circuit with multiple parallel channels wired between a high and low voltage rail.

The arrangement of elements on each **rung** implements basic Boolean logic operation. For example, in the relay ladder logic program shown in Figure 3, the output element M (presumably a motor) will be activated (note that it is normally inactive) only if input A is active (A is normally open), **and** input B is not active (B is normally closed). The Boolean logic operation is: $M = A \cdot \overline{B}$. In the second rung, the element M is now used as an input: If M is active, then element N will be *deactivated* (N is a normally active output).

- Using an **output** as an **input** on subsequent **rungs** is usually equivalent to writing a single very long rung. The element is written as both input and output to keep the rungs short and more readable.

Each **rung** can have elements in parallel (sort of a "sub-rung", if you like). As mentioned above, two inputs in series acts as a logical **and**. Putting two elements in parallel is a logical **or**. Some examples are shown in Figure 4.

- In Figure 4(a), the output M is activated if A **and** B are activated.

- In Figure 4(b), the output M is activated if A **or** B are activated.

- Figure 4(c) shows a more complicated logic operation. Using $\cdot$ for **and** and $+$ for **or**, the operation is:

$$M = A \cdot (B + C \cdot D) + E \cdot D$$

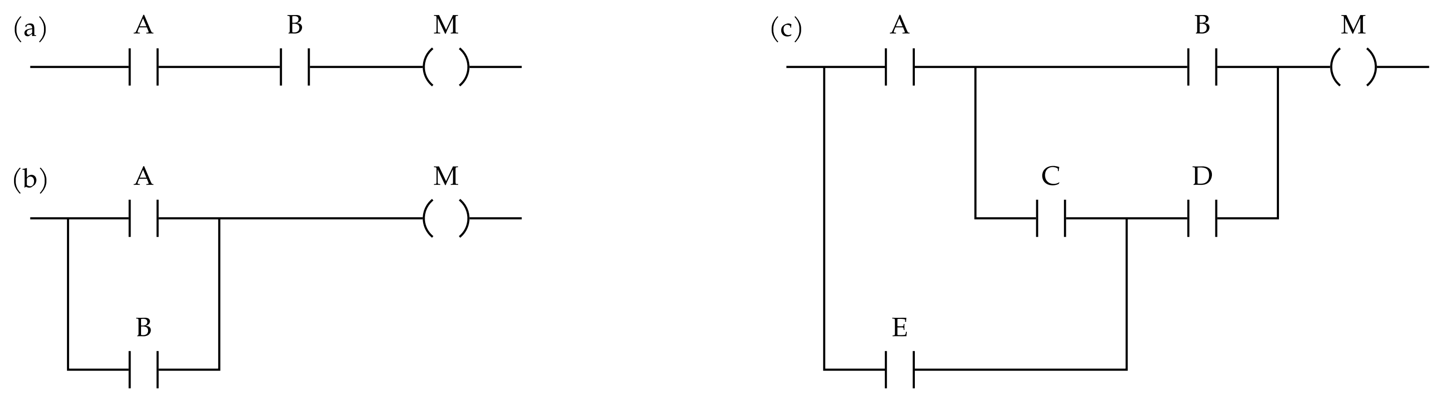

Figure 4: Relay ladder logic examples.

Note that although the depiction of relay ladder logic is similar to an electrical circuit, unlike an electrical circuit the operation always proceeds left-to-right. If you really wired a circuit as shown in Figure 4(c), then an electrical connection to M would be realized if switches E **and** C **and** B were closed. However, that will not happen if a PLC is programmed with the relay ladder logic shown in Figure 4(c).

Some PLCs allow vertical placement of inputs or outputs in the ladder diagram, but most do not. This practice is generally discouraged, because the whole point of relay ladder logic is to make a simple, easy-to-read diagram of the control logic to help error-checking.

An example is shown in Figure 5. Placing C as a vertical input makes the relay ladder logic program in Figure 5(a) is more compact, but harder to read visually. The more explicit implementation of the same logic operation in Figure 5(b) is easier to check for errors.

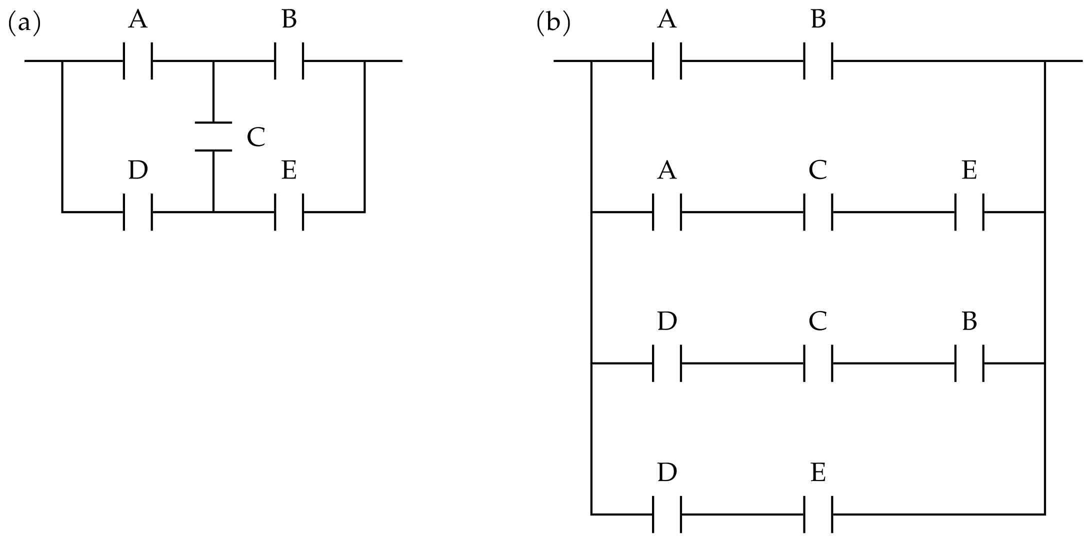

Figure 5: Placing elements vertically is usually not allowed in most relay ladder logic implementations, and is typically discouraged even when it is possible.

Relay ladder logic programs are executed in a continuous loop: starting at the top left corner, the rungs are evaluated left-to-right and top-to-bottom (recursively, if there are sub-rungs for logical **or** constructions), then the program repeats.

- We already mentioned that an output on one rung can be used as an input on a second rung.
- **Feedback** can be introduced into the relay ladder logic program by using an element as an output *and* input on the same rung.

Feedback is important for control logic, because often we want an input to trigger the start of an output, and have that output continue even when the input is removed.
An example of **feedback** is shown in Figure 6. Here we want to implement some simple control logic for a motor.

- When a start button is pressed, the motor should start running.
- The motor should continue to run after the start button is released.
- The motor should be stopped by pressing a stop button.

This is implemented in Figure 6 using a “dummy input/output” called Run. This represents an internal local variable in the PLC’s microprocessor. There is no way to visually recognize that Run is an internal variable rather than a real input/output without inspecting the PLC to see what wires are actually connected to it — and it doesn’t matter. The compiler will figure out which inputs/outputs in the relay ladder logic diagram correspond to real outputs, and which are internal.

- From an implementation perspective, using these “dummy input/outputs” is a good idea since now the actual microcontroller pin that is wired to the motor only needs to be configured as an output, and the PLC does not need to waste time reading the status of the motor when running the program.

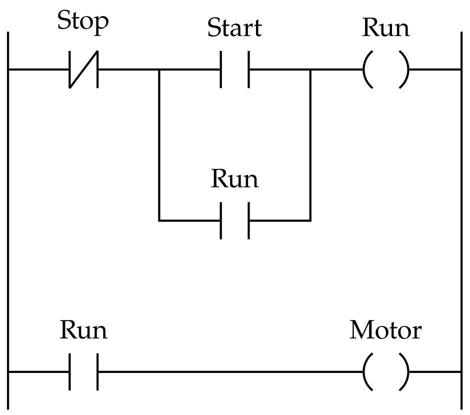

Figure 6: Simple relay ladder logic program for controlling a motor M. A dummy input/output “Run” is used to help facilitate the control logic. “Run” would be implemented as a local variable inside the PLC’s microprocessor — it does not correspond to a physical input/output pin.

In this program, the output Run on the first line is activated if the stop button is not pressed, **and** either the start button is pressed **or** Run was previously activated during the last pass through the program. The actual motor is then activated on the second line by Run.

Relay Ladder Logic Example

A water pump is used to fill two storage tanks. The pump is manually started, and fills the first tank. Then the first tank is full, the pump automatically switches to the second tank. When the second tank is full, the pump automatically switches off. Each tank also has an indicator light that should be lit up when the tank is full. Implement this design in relay ladder logic.

- I will label the two tanks as 1 and 2. Each tank must have a fill sensor (F1, F2) and an indicator light (L1, L2).
- This design must also have a start button (Start), and a pump (Pump).
- Some feedback is needed: the pump should continue operating after the start button is released until both tanks are full. We could use the pump output (Pump) for this, but instead I will use R (for "run") as a dummy input/output variable.
- I will consider each tank to have a valve (V1, V2) on the pump's flow pipe, so to fill a given tank that valve should be activated.
- Finally, how to switch the pump between tanks? If the fill sensor for F1 is inactive (tank is not full), the valve for tank 1 should be opened. If the fill sensor for F1 is active (tank is full), the valve for tank 2 should be opened. This guarantees that tank 1 will always be filled first. In this case, since the pump is turned off when both tanks are full, I consider it acceptable for the valve for tank 2 to be left open.

There is more than one way of solving this problem, and the original problem statement is vague what hardware is actually available, so you may have some different ideas on how to approach this problem. Anyway, here is my solution:

1. The pump should be active if the start button is pressed or the pump was previously active, and if at least one tank is not full.
2. If tank 1 is not full, the valve for tank 1 should be opened.
3. If tank 1 is full, the valve for tank 1 should be closed and the valve for tank 2 should be opened.
4. Indicator lights for both tanks should be active if the fill sensors are active.

This solution is implemented in Figure 7. Note that the fill sensors F1 and F2 are used as "normally-open" and "normally-closed" inputs as is necessary for the control logic.

- Really, these sensors are hardware that is either providing an input of high-voltage or low-voltage: whether I consider these to be "normally-opened" or "normally-closed" is entirely subject to how I interpret "normal" operation.
- As "normally-closed" is the logical **not** of "normally-opened", it is often necessary to alternatively consider an input in both contexts in a reasonably complicated logic circuit.
- The same may be true for outputs, which can be considered as "normally-inactive" or "normally-active" as necessary.

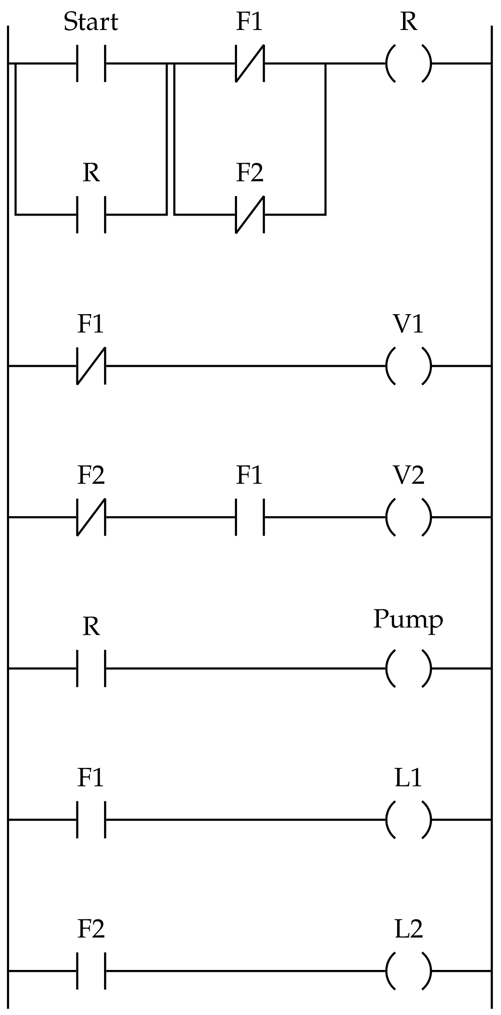

Figure 7: My relay ladder logic implementation of using a pump to sequentially fill two tanks, and automatically stop.