

N9LYA TECHNICAL GUIDES

PLC Industrial Automation

Allen-Bradley and Siemens Systems

Jerome M. Kutche

N9LYA

Second Edition · September 2026

About This Book

Most PLC books teach you one vendor's software. You finish able to write a rung that turns on a lamp in a simulator, and the first time you stand in front of an actual motor control center with a print in one hand and a multimeter in the other, none of it connects to what's in front of you.

That gap is the reason this book exists.

I spent three decades in industrial electrical and controls work — manufacturing, materials handling, heavy industry — after six years building RF test equipment for the telecom and cable industry. In that time I've programmed, wired, commissioned, and troubleshot both of the two platforms that actually dominate the American plant floor: Allen-Bradley/Rockwell and Siemens. I've stood in front of motor control centers that were on fire, in the metaphorical sense, at two in the morning, and I've spent slow Tuesday afternoons doing the unglamorous documentation work that keeps the next guy from standing there too.

A PLC does not exist in isolation. It exists at the center of a system — sensors feeding it information, contactors and drives taking orders from it, a network carrying its data somewhere a person can see it, and a safety system watching all of it for the moment something needs to stop right now. A book that covers ladder logic and stops there has covered maybe a third of the job. This one covers the programming, but it also covers the motor control center on the other end of the output card, the drive that's actually spinning the motor, the sensor whose signal is drifting because nobody cleaned the lens in eight months, and the network that ties the whole plant together badly if you let it.

Both major platforms get real coverage, not a chapter each with the rest of the book quietly assuming you use one of them. Chapter 9 and Chapter 10 go deep on Allen-Bradley and Siemens respectively, and Chapter 11 gives you the actual differences that matter when you're choosing or inheriting a platform, instead of vendor talking points.

Safety gets treated as engineering, not a footnote. Chapter 20 covers safety PLCs and safety instrumented systems properly, because a standard PLC and a safety PLC are not the same category of device, and treating them as interchangeable is how people get hurt.

Troubleshooting gets its own chapter, because that is the actual job, most days. Anyone can wire a new install from a print. Finding out why a system that worked yesterday doesn't work today, under production pressure, with a plant manager standing behind you, is a different skill and it is the one this book spends the most effort on.

Who This Is For

Electricians and maintenance technicians who work around PLC-controlled equipment and want to understand what the panel is actually doing, not just how to swap a fuse.

Engineers new to the plant floor, who learned control theory but not the physical reality of a motor control center or the politics of a plant network.

Hobbyists and makers moving from Arduino-class projects into industrial-grade control, who need to understand why industrial equipment is built the way it is.

Anyone inheriting an existing automation system with incomplete documentation — which is to say, nearly everyone who does this work for a living.

How to Use It

Each chapter ends with a **Bench Check** — practical questions and exercises. If you have access to a PLC (even a small used one on a home bench, which is a genuinely good way to learn this) or a simulator, do them with hardware in front of you. Ladder logic that makes sense on the page sometimes surprises you the first time you watch it scan.

Where I describe ladder logic, I use plain text rung notation rather than graphics, because that is how you will sketch it on a piece of paper standing at a panel, and because it reads the same regardless of what software generated the original.

Where I discuss specific platforms, I describe the concepts and the general instruction/programming environment rather than exact menu paths, because software versions change and the underlying ideas don't.

A Word Before You Start

You are going to make mistakes in a simulator or a bench setup, and that's the entire point of having one. You are also, someday, going to be troubleshooting a live system with production waiting, and the difference between a calm, systematic technician and a panicked one is almost entirely a matter of having done this enough times that the process is boring. This book can't give you the hours. It can give you the process, so the hours count for more.

About the Author

Jerome M. Kutche holds the amateur radio callsign N9LYA, first licensed in 1991 and upgraded to General class in 2003. His interest in electronics was sparked in the early 1970s by a mentor named Frank Kime, and his understanding of interference and grounding owes a great deal to the late Mike Baugh, W8AKF, an RFI mentor of rare patience.

From 1990 to 1996 he worked at Applied Instruments in Indianapolis, building RF test and measurement equipment for the telecommunications and cable television industries, alongside engineers Doyle Haywood, David Polstra, and Jerry Brown.

He then moved into industrial electrical and controls work, accumulating more than three decades across manufacturing, materials, and heavy industry — Allen-Bradley and Siemens PLC systems, motor control centers, plant electrical maintenance and supervision. He holds NFPA 70E, lockout/tagout, arc flash, and thirty-nine other professional certifications.

He has worked amateur radio contacts through the Space Shuttle SAREX program, the Mir space station, and the International Space Station, using a two-watt handheld radio.

Part I — Foundations

Chapter 1: What a PLC Actually Does

A programmable logic controller is, underneath the marketing, a specialized industrial computer that does one thing: it reads inputs, executes a program, and writes outputs, over and over, forever, without getting bored or distracted.

That sounds trivial. It isn't, and the reason it isn't is reliability under conditions that would kill a regular computer. A PLC lives in an enclosure that might see 100°F ambient temperature, vibration from adjacent machinery, electrical noise from variable frequency drives six feet away, and power that sags and spikes with every big motor starting on the same feeder. It is expected to run for years without a reboot, and when it does need attention, the person servicing it is often standing in front of it with the machine still running around them.

Why Not Just Use a Regular Computer

People ask this, reasonably. A desktop PC is cheaper and more powerful. The answer is that a desktop PC's operating system was never designed for **deterministic real-time execution** — the guarantee that a given operation happens within a known, bounded time, every single time. When Windows decides to run a background update, your process scheduling gets delayed by however long that takes. In a PLC, the scan is the scan. It executes the same program, in the same order, in a predictable amount of time, scan after scan, and that predictability is what lets you build safety and process logic on top of it with confidence.

The other piece is **survivability**. Industrial PCs exist, and they're used for HMI and data logging. But the core control function — the part that decides whether a motor runs or a valve opens — nearly always lives on a PLC, because a PLC is built to survive a power hit and come back exactly where it

The Industrial Control Loop

The PLC connects field conditions to controlled action



N9LYA TECHNICAL GUIDES

Figure 1: The industrial control loop

left off, without corruption, without a blue screen, and without someone needing to physically reboot it.

The Basic Loop

At the simplest level, every PLC does this, continuously:

1. Read all inputs into memory (input scan)
2. Execute the user program against that memory (logic scan)
3. Write outputs from memory to the physical world (output scan)
4. Housekeeping (communications, diagnostics)
5. Repeat

That loop is the **scan cycle**, and Chapter 4 covers it in depth, because a huge amount of confusing PLC behavior — a rung that “should” work but doesn’t, a race condition between two parts of a program — traces back to not understanding exactly when in the cycle things happen.

What a PLC Is Not

It is not a general-purpose computer. You don’t browse the web on it, and you shouldn’t be tempted to make it do things outside deterministic control — that’s what the HMI, the SCADA layer, and the plant network are for.

It is not magic. Every ladder rung is doing something a relay panel used to do with actual relays and actual wires. Understanding that history — Chapter 5 opens with it — makes ladder logic make sense in a way that jumping straight into software syntax never does.

It is not, by itself, safe. A standard PLC has no inherent safety rating. If your process needs to guarantee that a guard door stops a machine, that function belongs on safety-rated hardware, covered in Chapter 20 — not on standard logic that “should” work.

PLC, PAC, or Edge Controller

The line between categories has blurred over the last decade, and it’s worth understanding the terms even though they overlap in practice.

A **PAC (Programmable Automation Controller)** is essentially a PLC with expanded capability — more processing power, native support for motion control, more sophisticated data handling, and often the ability to run multiple program languages and tasks concurrently with more PC-like flexibility, while retaining the deterministic scan and industrial hardware ruggedness of a traditional PLC. Rockwell’s ControlLogix and Siemens’s S7-1500 are both frequently described as PACs rather than “plain” PLCs for exactly this reason, though in daily plant-floor conversation, people still just call them PLCs and nobody’s confused by it.

An **edge controller** or **edge device** sits at the boundary between the control network and IT/cloud systems, often running standard PLC-style logic alongside genuinely IT-style capabilities — containerized applications, direct cloud connectivity, more powerful analytics than a traditional PLC would attempt locally. These are increasingly common for condition monitoring and analytics applications layered on top of, rather than replacing, the core control function that still lives on a traditional PLC or PAC.

The practical takeaway: don’t get hung up on the terminology. Ask what a given controller actually needs to do — deterministic control of a process, with what I/O count, what safety requirements, what networking and data needs — and select hardware against those requirements, whatever the vendor’s marketing calls the product category that quarter.

Bench Check

1. Explain, in your own words, why deterministic scan timing matters more on a plant floor than raw processing speed.

2. Name two environmental conditions a PLC in an industrial enclosure routinely survives that would be a problem for a desktop PC.
3. Why shouldn't a safety-critical guard interlock be implemented purely in standard PLC logic?
4. What functionally distinguishes a PAC from a traditional PLC, beyond marketing language?

Chapter 2: PLC Hardware Architecture

Open a PLC cabinet and, regardless of vendor, you'll find the same functional pieces arranged slightly differently: a power supply, a processor (CPU), input modules, output modules, and increasingly, network interfaces tying it all together.

Power Supply

Most industrial PLCs run their logic on 24VDC, fed from a separate power supply that converts incoming 120/240VAC (or sometimes 480VAC three-phase, stepped down) to clean, regulated DC. **This is worth understanding for troubleshooting:** a flickering, intermittent PLC fault is very often not a program bug at all — it's a marginal 24V supply sagging under load, especially when someone has added field devices to a rack that was sized for the original I/O count and nothing more.

The Processor (CPU)

The CPU is where the program lives and executes. Modern processors hold the program in non-volatile memory (so it survives a power loss) and execute it from working memory. Processors are rated by scan time, memory capacity, and I/O capacity — how many local and remote I/O points they can address.

Redundant processors exist for processes where downtime is extremely costly — a hot standby CPU takes over within one scan if the primary fails. Most plant-floor applications don't need this; it adds cost and complexity that isn't justified unless the cost of an unplanned stop genuinely exceeds it.

I/O Modules

Discrete (digital) I/O modules handle on/off signals — a limit switch, a push button, a contactor coil, a pilot light. Chapter 3 covers these in detail.

Analog I/O modules handle continuously variable signals — a 4-20mA transmitter reading a tank level, a 0-10V speed reference to a drive. Also covered in Chapter 3.

Specialty modules handle things that don't fit either category cleanly: high-speed counters for encoder input, thermocouple/RTD inputs for temperature, motion control modules for servo positioning, and communication modules for protocols the CPU doesn't speak natively.

Rack-Based vs. Modular vs. Brick

Larger systems use a **rack** (or "chassis") that the CPU and I/O modules plug into, sharing a backplane that carries power and data between modules. This is expandable — need more I/O, add a module and possibly another rack, connected by a remote I/O link.

Smaller systems use a **modular** or **brick** style, where the CPU and a fixed set of I/O are one physical unit, sometimes with limited expansion via add-on modules that snap onto the side. These are cheaper for small, fixed applications — a single machine that will never grow — but painful if the application does grow beyond what was planned.

A rule I've learned the hard way: when specifying a new system, size the I/O count for what the machine will need in five years, not what it needs on day one. The cost difference between a rack with two spare slots and one with none is trivial next to the cost of a forklift upgrade later.

Remote I/O

Not every input and output lives physically next to the processor. A plant floor might have I/O scattered across a large area — a motor control center here, a set of sensors on a conveyor two hundred feet away, a control station at an operator position. **Remote I/O** racks talk back to the main processor over a network (Chapter 17 covers the protocols), letting you put I/O physically close to the field devices it serves instead of running every individual wire back to a central cabinet.

This matters for both cost (much less copper) and reliability (shorter individual runs, less susceptibility to noise pickup over distance) — but it adds a network dependency: if that network link goes down, that remote I/O rack usually goes to a predefined fault state, and your program needs to handle that gracefully rather than assuming the data is always current.

Bench Check

1. A PLC is faulting intermittently under load, with no pattern in the program logic. What's the first thing worth checking, and why?
2. When would you choose a modular/brick PLC over a rack-based system, and when would that choice bite you later?
3. What happens to a remote I/O rack's outputs, typically, when its communication link to the main processor fails — and why does your program need to account for that explicitly?

Chapter 3: I/O — Discrete and Analog



Figure 2: Discrete and analog I/O

Discrete Inputs

A discrete input is simply: is this circuit open or closed? A limit switch, a push button, a proximity sensor, a relay contact from another system — all of them present the input module with either a voltage (sourcing) or a path to common (sinking), and the module translates that into a single bit in the PLC's memory.

Sourcing vs. sinking trips up more people than it should. A **sourcing** device supplies positive voltage to the input when active (common in North American 24VDC practice, often called PNP). A **sinking** device provides a path to ground/common when active (common in European practice, often called NPN). Mismatch a sourcing input module with a sinking field device and nothing works, with no obvious fault indication beyond “the light doesn't come on.” Check the datasheet, always, before wiring — don't assume based on what the last project used.

Discrete Outputs

Discrete outputs are the inverse: the PLC turns a bit on or off, and the output module translates that into a real-world signal — usually closing a relay, triggering a transistor output, or switching a triac.

Relay outputs can switch AC or DC, handle moderate current directly, but wear out mechanically and switch relatively slowly — fine for a pilot light, marginal for something cycling rapidly.

Transistor outputs (DC only, sourcing or sinking depending on module type) switch fast and have no mechanical wear, ideal for high-speed or high-cycle applications, but typically source lower current directly — often you're switching a small relay or driving directly into a modern contactor coil designed for it, not a large motor starter coil directly.

Triac outputs (AC only) offer similar speed/longevity advantages for AC loads.

A practical habit: never run a motor starter coil, solenoid, or any inductive load directly off a PLC output at its rated maximum without checking the module's actual switching capability under inductive load — inductive loads generate back-EMF on switch-off that can exceed a transistor output's rating and kill the output point. Use an interposing relay for anything beefy, and it will save you from replacing output cards.

Analog Inputs

Analog signals represent a continuously variable quantity — temperature, pressure, level, flow, speed. Industrial analog signaling standardized decades ago on two dominant formats:

4-20mA current loop. The signal is represented as current, not voltage, which is the key advantage: current doesn't change over a wire run the way voltage does, so a 4-20mA signal survives hundreds of feet of cable without the accuracy loss a voltage signal would suffer from wire resistance. Zero (or the low end of the process variable) is represented as 4mA, not 0mA — this is deliberate. **A reading of 0mA means a broken wire or dead transmitter, not “zero,”** and that distinction between “the process is at zero” and “the signal is gone entirely” is exactly why the standard uses 4mA as the floor.

0-10V (or sometimes 0-5V, 1-5V) voltage signaling is common for shorter runs and speed references to drives, but is more susceptible to noise and voltage drop over distance than a current loop.

Analog Outputs

Same two standards in reverse: the PLC writes an analog output (typically 4-20mA or 0-10V) to control something continuously — a valve position, a drive speed reference.

Resolution and Scaling

Analog modules digitize the incoming signal at some resolution — commonly 12 to 16 bits. A 16-bit module gives you 65,536 discrete steps across its range, which sounds like enormous precision, but **the module's resolution is not the same as the loop's actual accuracy** — a cheap transmitter with 1% accuracy doesn't get more accurate because the PLC digitizes its output finely. Match your

expectations to the weakest link in the loop, which is very often the field transmitter or the wiring, not the PLC.

Scaling converts the raw digitized value (say, 0-32767 counts for a unipolar 16-bit input) into engineering units (0-100 PSI, 0-500°F). Get the scaling wrong — inverted, offset, or using the wrong raw range for your module's actual output — and everything downstream that reads that value is subtly wrong in a way that isn't obvious until something behaves strangely at one end of the range.

Bench Check

1. A proximity sensor is wired to an input module and the input never comes on, though the sensor's own LED indicates it's detecting the target. What's the first thing to check?
2. Why is 4-20mA preferred over 0-10V for a signal run of 300 feet?
3. An engineer tells you a 16-bit analog input module gives 0.01°F accuracy on a temperature loop. What's wrong with that claim?
4. Why shouldn't you wire a large motor starter coil directly to a PLC transistor output without an interposing relay?

Chapter 4: The Scan Cycle

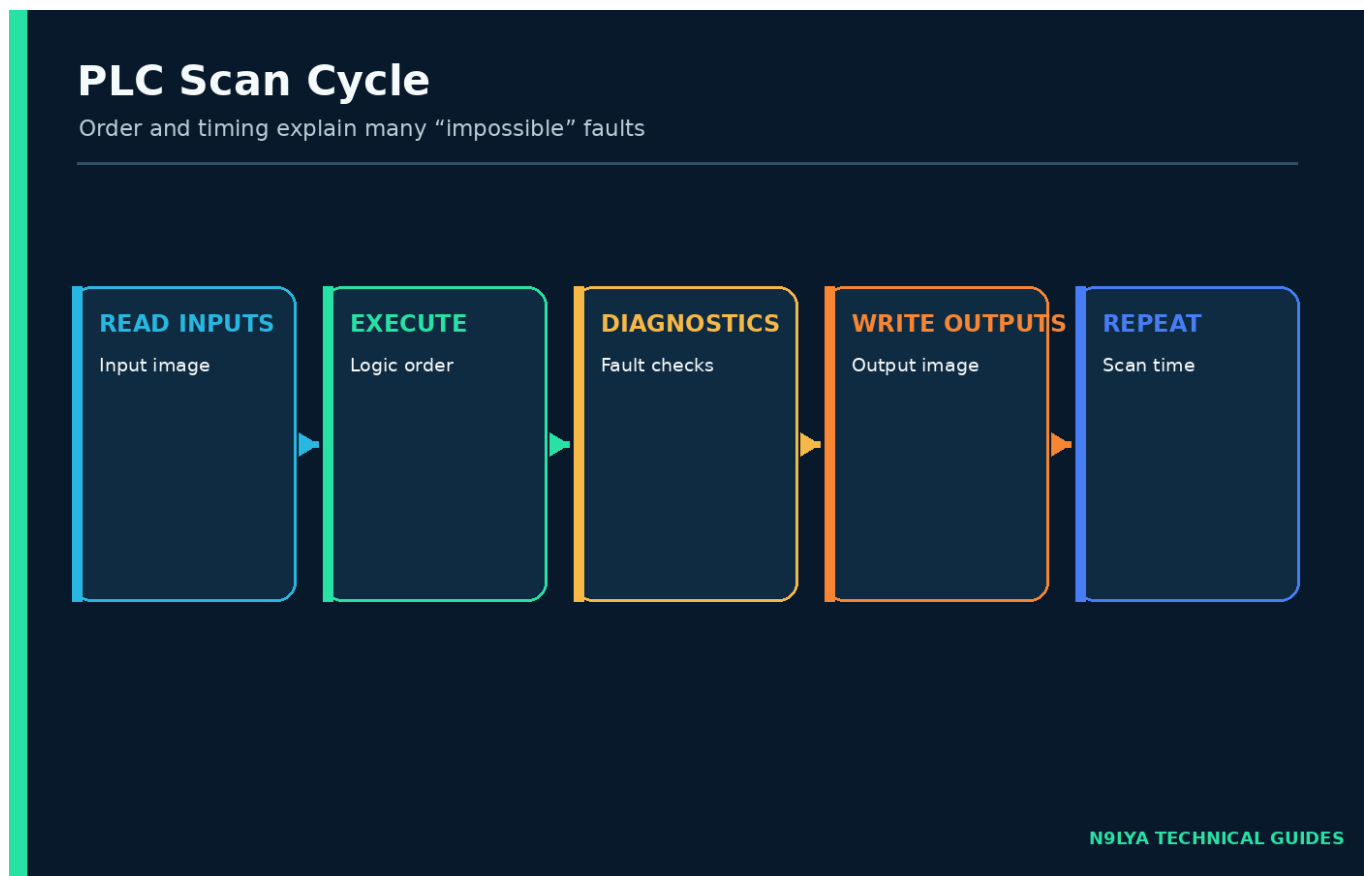


Figure 3: The PLC scan cycle

Understanding exactly when things happen inside a PLC's scan cycle resolves more "impossible" bugs than any other single piece of PLC knowledge.

The Three (or Four) Phases

Input scan. At the start of each cycle, the processor reads the physical state of every input point and copies it into an internal memory table — the **input image table**. Critically: the program does

not read inputs directly from the field during logic execution. It reads this snapshot, taken once at the start of the scan.

Logic scan. The processor executes the entire program, rung by rung (or block by block), top to bottom, left to right, using the input image table and any internal memory (timers, counters, internal bits) as its data. Output instructions during this phase don't energize physical outputs immediately — they write to an **output image table**.

Output scan. At the end of the logic scan, the processor copies the output image table out to the physical output modules, all at once.

Housekeeping/overhead. Communications servicing, diagnostics, and any background tasks happen here, and this is part of why scan time isn't perfectly constant — it varies with communication load, forcing, and online program edits.

Why This Matters

An input that changes state mid-scan doesn't affect that scan. If a fast sensor pulses on and back off between input scans, the PLC may never see it — this is the entire reason high-speed counter modules exist, to catch transitions faster than the normal scan can reliably see.

Multiple rungs referencing the same input all see the same value within one scan, even if that input physically changed halfway through logic execution. This is a feature, not a bug — it prevents a program from seeing inconsistent input states within a single pass, which would make logic unpredictable.

An output written early in the program and again later in the same scan — the last write wins, because only the final state of the output image table gets copied out at the output scan. If you have two rungs both driving the same output coil (which most platforms will flag as a problem, but not all, and not always clearly), the second one in scan order determines the real output.

Scan Time

Scan time is the total time for one complete cycle, and it varies with program size, the number of I/O points, communication overhead, and how much of the program is actually being executed (some instructions can be conditionally skipped). Typical scan times for moderate-sized programs run from single-digit milliseconds to a few tens of milliseconds. For most process control, this is vastly faster than the physical process changes, so it's effectively instantaneous from the process's point of view.

Where scan time matters directly: fast mechanical processes — a flying shear, a high-speed sorting system — where the physical event can complete in less time than a scan takes. This is where high-speed I/O, dedicated motion modules, or moving the time-critical function to hardware (a photoeye directly triggering a solenoid through dedicated fast logic) becomes necessary instead of relying on the standard scan.

Watchdog Timers

Every PLC runs a **watchdog timer** — if a scan takes longer than a configured maximum (something has gone wrong: an infinite loop condition, excessive online editing load, a genuine fault), the processor faults itself rather than risk an unpredictably slow, unsafe control loop. This is a safety feature, not a nuisance, even when it interrupts you at an inconvenient moment.

Bench Check

1. A photoeye pulses for 2ms. The PLC's scan time is 15ms. Explain, using the scan cycle, why the PLC might completely miss that pulse, and name a hardware solution.
2. Two rungs in a program both write to output O:2/3, with different conditions. Which one determines the actual physical output state, and why?
3. Why does a PLC use an input image table instead of reading physical inputs fresh every time a rung references them?

Part II — Programming

Chapter 5: Ladder Logic Fundamentals

Ladder logic looks the way it does for a historical reason worth understanding before the syntax, because the history is the explanation.

Where It Came From

Before PLCs, industrial control was built from actual relays — physical electromechanical switches wired together in panels, sometimes hundreds of them, to implement the control sequence for a machine. An electrician reading the wiring diagram for a relay panel saw something that looked like a ladder: two vertical rails representing the power supply, and horizontal “rungs” between them, each rung a series/parallel combination of contacts controlling a coil.

When PLCs arrived in the late 1960s (the standard origin story: General Motors wanted a way to reprogram control logic without rewiring relay panels every time a car model changed), the earliest ones were deliberately designed to be programmed in a language that looked exactly like those relay diagrams, so the electricians and technicians who already knew how to read a relay ladder diagram could read and troubleshoot the new PLC-based systems with minimal retraining. That design decision from fifty-plus years ago is why ladder logic still looks like it does today, even on platforms that also support far more modern languages.

The Basic Elements

Contacts represent conditions — an input, an internal bit, the state of a timer or counter. A **normally open (NO)** contact passes logic “power” through the rung when the referenced bit is true (on). A **normally closed (NC)** contact does the opposite — passes power when the referenced bit is false (off).

Coils represent the result of a rung’s logic — an output, or an internal bit used elsewhere in the program. When the rung’s contact logic evaluates true, the coil “energizes.”

A basic rung:

```
--[ I:1/0 ]----- ( 0:2/0 )--  
    Start PB           Motor Run
```

Read left to right: if input I:1/0 (a start pushbutton) is true, energize output O:2/0 (a motor run coil).

Series and Parallel

Contacts in series behave like an AND: all must be true for power to reach the coil.

```
--[ I:1/0 ]--[ I:1/1 ]----- ( 0:2/0 )--  
    Start PB   Guard Closed      Motor Run
```

Both the start pushbutton AND the guard-closed switch must be true.

Contacts in parallel (a “branch”) behave like an OR:

```
--[ I:1/0 ]----- ( 0:2/0 )--  
    |--[ 0:2/0 ]--|  
    Seal-in
```

If the start pushbutton is true, OR the output’s own seal-in contact is already true (because it was just energized), the coil stays energized. This is the classic **seal-in** or **latching** circuit — the most fundamental pattern in ladder logic, because it’s how you make a momentary pushbutton produce a maintained output.

Normally Closed Contacts and the Mental Trap

A normally closed contact in a rung reads as “this is true when the referenced condition is false” — but people new to ladder logic frequently confuse the physical device’s normally-open/normally-closed wiring with the programmed contact’s NO/NC symbol, and those are two separate decisions that can combine in four ways. A physically normally-closed safety switch, programmed with a normally-open contact instruction, gives you a rung that’s true when the switch is not actuated — which is exactly backward from what you usually want for a safety circuit, and is a genuinely common source of commissioning bugs. Always trace both the physical device type and the instruction type together, not just one.

Beyond Seal-In: Common Patterns Worth Knowing Cold

A handful of ladder patterns recur constantly enough that they’re worth being able to write from memory, the way you’d know a common phrase in a language you speak fluently.

One-shot (edge detection). Sometimes you need a rung to fire exactly once when a condition transitions from false to true, not continuously while it stays true — incrementing a counter once per part detected, not once per scan the sensor happens to still be looking at that part. A one-shot instruction (often called ONS) passes power for exactly one scan on a false-to-true transition, then blocks it on subsequent scans until the input goes false and true again. Without it, a condition held true for several scans in a row would trigger whatever follows it several times over, which is rarely what you actually want.

Interlocking a reversing starter. A motor that runs forward and reverse needs its two contactors electrically and programmatically prevented from ever energizing simultaneously — both physically (a mechanical interlock on the starter hardware itself, which should never be treated as optional) and in the program (each direction’s rung carries a normally-closed contact referencing the other direction’s output, so commanding reverse while forward is still actually running is blocked in software as well as hardware). Two independent layers of protection here, not one, because either one failing alone shouldn’t be enough to short the motor windings across the line.

Flip-flop (alternate on successive pulses). Useful for a single pushbutton that should alternate a output on and off with successive presses — implemented either with a dedicated flip-flop instruction where the platform provides one, or built from a one-shot feeding a rung that inverts the output’s own current state on each pulse.

First-scan logic. Most platforms provide a bit that’s true only on the very first scan after the processor transitions to run mode — useful for one-time initialization (setting a startup default value into a register, for instance) that shouldn’t repeat every subsequent scan.

Stop Buttons and Fail-Safe Wiring

A stop button should always be wired normally closed and programmed with a normally-open instruction. This means, in the rung, the stop condition reads true (allowing the circuit to run) when the physical button is unpressed and the wire is intact. If the wire to a normally-closed stop button breaks, or the button fails, the circuit sees a stop condition and shuts down — a safe failure. Wire it the other way (normally open button, so the rung logic needs an NC instruction to make it work the same way) and a broken wire or failed button looks identical to “nobody pressed stop” — the machine keeps running with no way to stop it electrically. This single wiring convention, done wrong, has caused real injuries. Chapter 20 goes further into fail-safe design for genuinely safety-rated circuits; this is the baseline habit for ordinary control circuits.

Bench Check

1. Sketch (in text notation) a seal-in circuit for a pump that starts on a momentary Start button and stops on a momentary Stop button.
2. Explain why a stop button should be wired normally closed rather than normally open, in terms of what happens if the wire breaks.

End of the free sample

PLC and Industrial Automation — Second Edition · by Jerome M. Kutche

That was the first 12 of 48 pages. The full book continues with the practical detail, worked examples and recovery steps that don't fit in a sample.

Get the full book (\$5.99, PDF + EPUB):

<https://booksrus.biz/product/plc-and-industrial-automation/>

Also on Gumroad:

<https://n9lya4.gumroad.com/l/hjcfjz>



All 17 Second Edition guides in one bundle for \$89.99:

booksrus.biz/product/n9lya-technical-guides-complete-bundle

This sample may be shared freely in unmodified form. © Jerome M. Kutche. Second Edition.