Ladder diagram (LD)

TM240

Perfection in Automation

www.br-automation.com

Requirements

Training modules:	TM210 – The Basics of Automation Studio
	TM211 – Automation Studio Online Communication
	TM213 – Automation Runtime
	TM223 – Automation Studio Diagnostics
Software:	None
Hardware:	None

Table of contents

1. INTRODUCTION

The Ladder Diagram programming language is often referred to as relay ladder logic. It is a graphical language that is very popular for programming controller systems. Most manufacturers program their systems using the Ladder Diagram programming language.

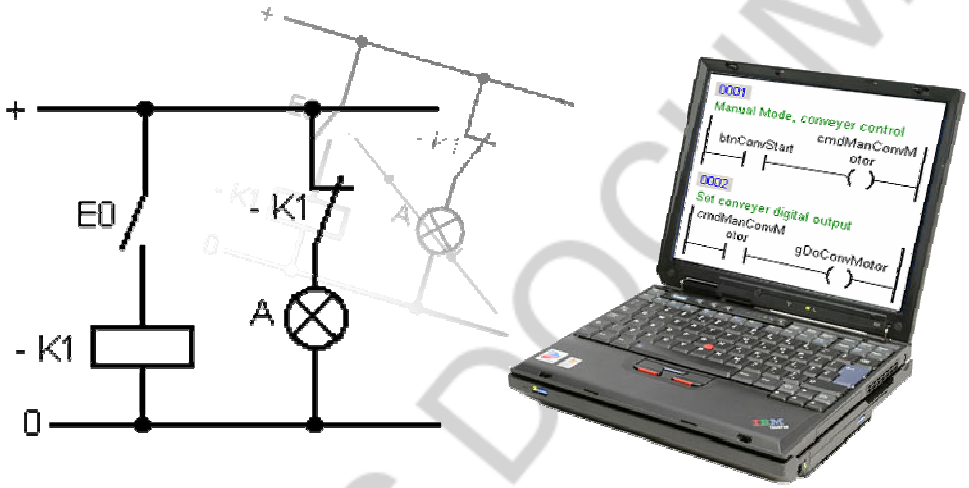


Fig. 1 Introduction

In the chapters that follow, you will get an overview of the history, advantages, and programming of ladder diagrams.
Examples will be given to give a better explanation of the individual functions available.

1.1 Objectives

You will get an overview of the possibilities available when programming with Ladder Diagram.

You will learn the fundamental elements of Ladder Diagram and the symbols for logic programming.

You will be able to develop flexible Ladder Diagram programs using the program flow control elements.

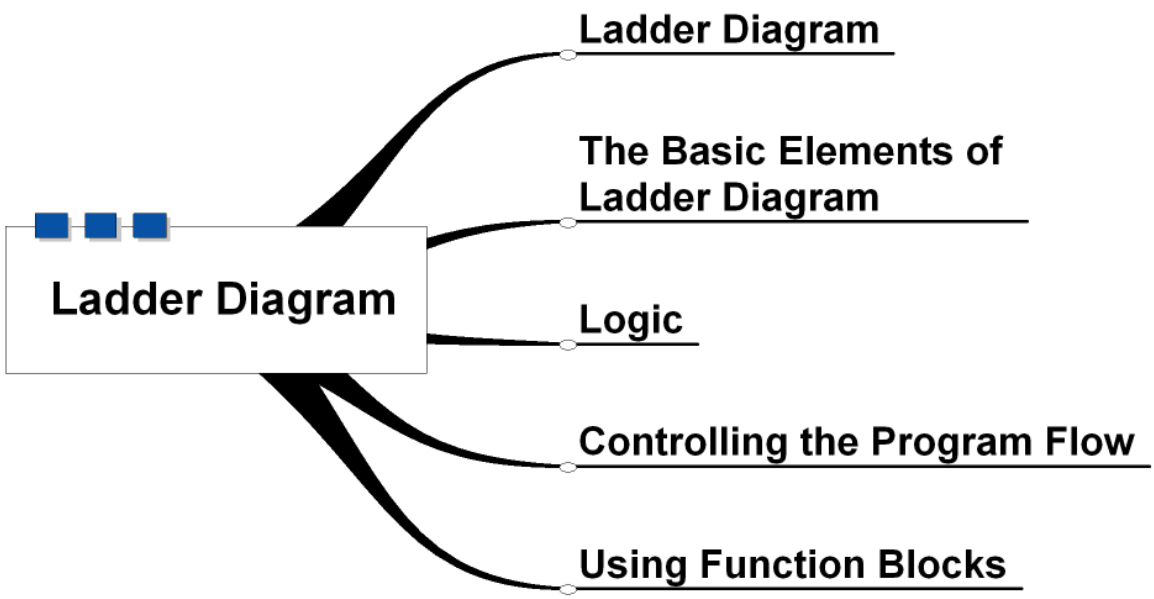


Fig. 2 Overview

2. LADDER DIAGRAM

2.1 The history of ladder diagrams

The original concept of the PLC (Programmable Logic Controller) was developed in the USA around 1968. The PLC concept was developed as a microprocessor-based, programmable substitute for hard-wired systems.

The PLC was based around the ladder diagram, which is a schematic representation of a logical control system based on relay circuitry.
At that time, the concept became a very fast way of quickly setting up and programming a simple logical control system with relatively little training. For simple logical control problems, ladder diagrams are ideal, easy to use, and easy to master. They are probably the main reason for the phenomenal success of PLCs in industry.

Many manufacturers base their programming systems on ladder diagrams. Unfortunately, the lack of any open standards meant that each vendor's system was slightly different. Many manufacturers often added "special operations" in order to increase functionality.

By the beginning of the 1990s, there were literally thousands of PLC manufacturers, each with their own programming systems and sets of instructions. Although the programs that were written for different systems were similar, the way that programs were structured, as well as the instruction sets used, varied from one supplier to another.

Hardware-specific peculiarities always seemed to crop up. This situation tended to tie users to a particular manufacturer.

In 1979, a working group was set up by the International Electrotechnical Commission (IEC) to create a common standard for PLCs. This working group decided to develop a new standard (to be known as IEC 61131) consisting of five separate parts.

Part III, "Programming Languages for PLCs", was published in 1993 and included the specification for PLC software. Part III covers the PLC configuration, programming, and data storage.

IEC61131-3 addresses most of the criticisms that are aimed at traditional PLC programming. It provides a framework for developing general PLC programs that do not require manufacturer-specific training. Most PLC and industrial control system manufacturers have now adopted this standard.

2.2 General information

Ladder Diagram (LD) is a graphics-based programming method. It is a symbolic representation of electronic circuits. Symbols were selected that actually looked similar to schematic symbols used for electric devices. For this reason, an electrician who has never seen a PLC can understand the Ladder Diagram programming language. In LD, the schematic symbols (contacts and coils) and the connection lines can be used to create the necessary logic.

B&R has adopted the 61131-3 standard for ladder diagram programming as well.

2.3 Properties

Ladder Diagram has the following characteristics:

- Graphical programming languages
- Similar to wiring diagram
- Simple and clear programming
- Intuitive to use
- Easy to find errors
- Complies with the IEC 61131-3 standard

2.4 Possibilities

The Ladder Diagram programming language provided with Automation Studio offers the following possibilities:

- Using digital inputs / outputs and internal Boolean variables
- Using analog inputs / outputs
- Using function blocks
- Program flow control (jumps, cancelling)
- Diagnostics tools

3. THE BASIC ELEMENTS OF LADDER DIAGRAM

Imagine that there is a vertical supply line called a "bus bar" on the left-hand side that continuously provides power. Lines that branch out to the right are known as "instruction lines".

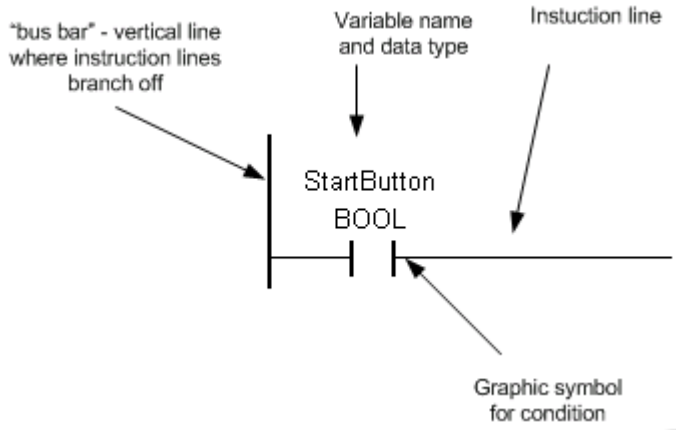


Fig. 3 Basic elements of Ladder Diagram

The ladder diagram itself consists of two basic parts. The left side holds the **condition logic**, while the right side contains the **instructions**. The logical combination of these conditions determines when and how an instruction on the right side is executed. The elements on the far right side are called coils (e.g. lamps, motors, relays, etc.).

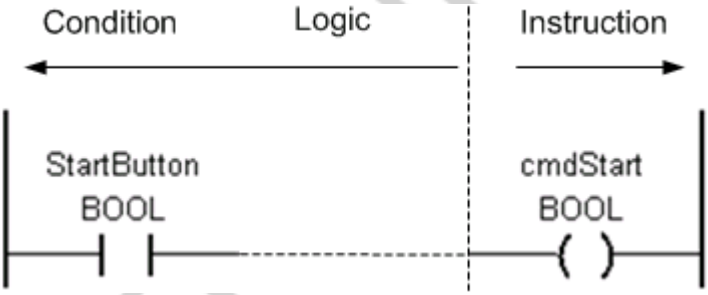


Fig. 4 Logical conditions and instructions

3.1 Networks

A network is a circuit that represents a specific function. It consists of **elements**, **branches**, and **blocks**. The network reflects a complete function and is the basic unit of a ladder diagram.

A complete Ladder Diagram program consists of several of these networks.

The beginning of the network is the **vertical line on the left (bus bar)**. If two or more circuits are connected by a vertical line, then they belong to the same network.
Up to **50 lines** and **50 columns** may exist in a network. The size of a complete ladder diagram is limited only by the amount of memory on the PC and the controller.

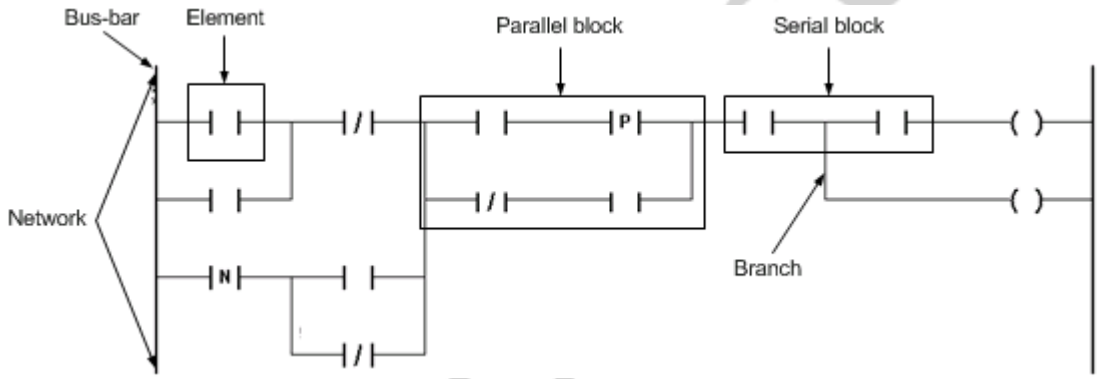


Fig. 5 Ladder diagram network

4. LADDER DIAGRAM SYMBOLS

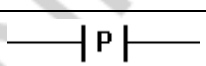
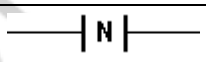
4.1 Contacts

A contact is one of many Ladder Diagram symbols. They can be placed in the first column and also form part of a link with other symbols. They cannot be placed on the right side; this area is reserved for coils. Contacts can be linked to the digital inputs / outputs of function blocks.

Linking contacts within a network can be assigned to one or more coils. Every contact is referenced by a variable name that has been or will be defined in the variable declaration window. Each contact–regardless of whether it's an input, output, or internal variable–can be used throughout the program whenever that particular condition needs to be evaluated.

The connection between contacts depends on the desired control logic. They can be placed in whatever series, parallel, or series/parallel configuration that is required to control a given output (coil).

Only variables of type BOOL can be assigned to contacts.

Type of contact	Symbol
Normally open	
Normally closed contact	
Positive edge	
Negative edge	
Both edges	

4.1.1 What are normally open and normally closed contacts?

In industrial environments, it's not uncommon to hear the terms **normally open** and **normally closed**. Both terms apply to words such as contacts, inputs, outputs, etc. (all combinations have the same meaning whether talking about inputs, outputs, contacts, etc.).

A **normally closed contact** will conduct electricity until it is pressed. A **normally open contact** won't conduct electricity until it is pressed down.

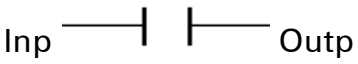
If a **normally closed contact** is selected, a bell will continously sound until someone pushes the bell switch. Pressing the switch opens up the contact and the electricity is cut off.
This behavior is reversed if using a **normally open contact**.

Fig. 6 Normally open and normally closed contacts

One example where normally closed contacts are used is in the safety door on machines. If this door is opened, the contact is interrupted and the supply circuit is interrupted. This can be done to prevent accidents.

Normally open and normally closed contacts can apply to outputs and sensors as well.

4.1.2 Normally open contact



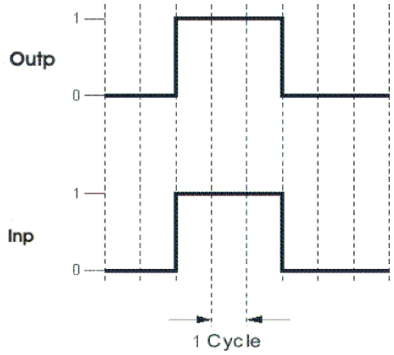
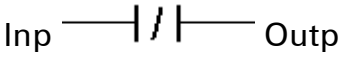


Fig. 7 Relation between an input and output

If the contact isn't pressed, electricity is not conducted and the logical state is **FALSE** (0).

When pressed, the physical state switches to "**ON**", and the instruction is **TRUE** (1).

4.1.3 Normally closed contact



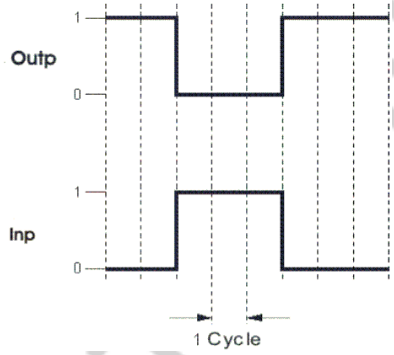


Fig. 8 Relation between an input and output

This symbol inverts the status of a variable (**BOOL**).

It is used when an input signal does **not** need to be present for the output to be set.

The state of the output is set to **FALSE** if the input is set to **TRUE**.

4.1.4 Positive edge

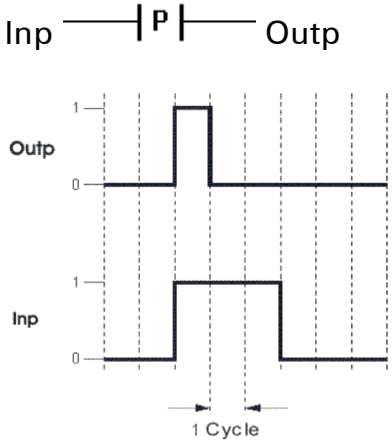


Fig. 9 Relation between an input and output

This symbol is used to form a positive edge of a digital signal.

When the value of a variable switches from **FALSE** to **TRUE**, i.e. a positive edge occurs, this contact returns **TRUE** for a cycle. This can be used to set / reset states or count the number of positive edges.

4.1.5 Negative edge

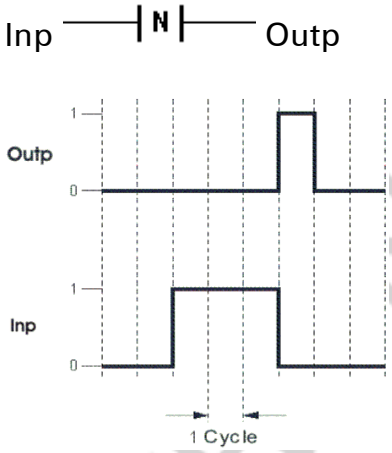


Fig. 10 Relation between an input and output

This symbol is used to form a negative edge of a digital signal.

If the value of a variable is switched from **TRUE** to **FALSE**, the status returns **TRUE** for a cycle. This can be used e.g. to set or reset outputs or count negative edges.

4.1.6 Positive and negative edge

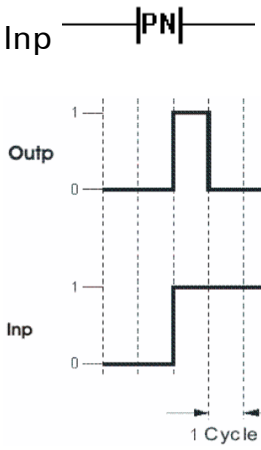


Fig. 11 Relation between an input and output

This symbol is used to form the positive and negative edge of a digital signal.

This behavior corresponds to a parallel switching of the positive and negative edge.

4.2 Coil

The coil is one of the basic Ladder Diagram elements. It is always placed on the right-hand side of the ladder diagram as an output. Coils can be connected to the right of contacts or to function block outputs. At least one coil must be present in a ladder diagram. Several parallel coils can also be used.

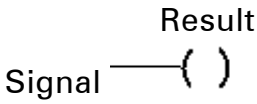
Each coil can be used for digital outputs or internal variables that will later serve as an input for an additional network in the program.

The contacts are always queried while the program is running; if logic continuity is found, then the contact is set to TRUE.

Only Boolean variables can be assigned to coils.

Type of contact	Symbol
Coil	—( )—
Negated coil	—(/)—
Set coil	—(S)—
Reset coil	—(R)—
Positive transition coil	—(P)—
Negative transition coil	—(N)—
Both edges	—(PN)—

4.2.1 Coil



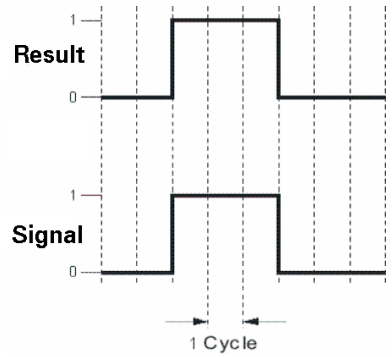
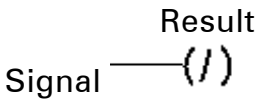


Fig. 12 Relation between an input and output

The coil is switched on if continuity is found.

4.2.2 Negated coil



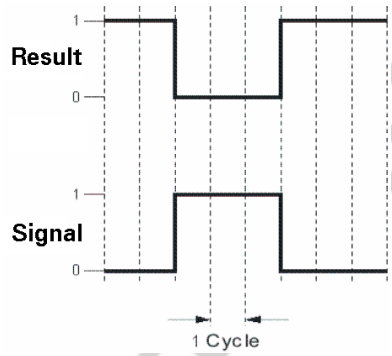


Fig. 13 Relation between an input and output

If continuity is found, the coil is switched off; otherwise, it is switched on.

4.2.3 Set

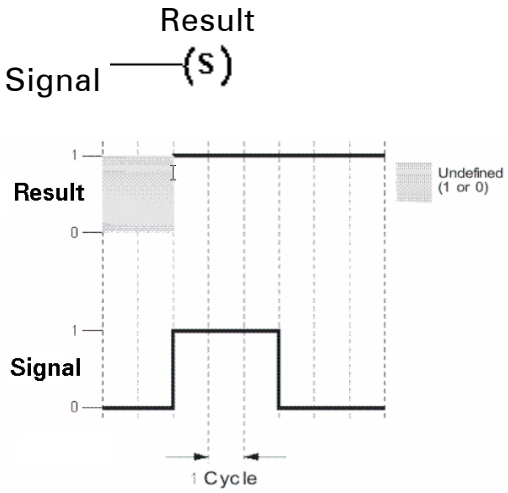


Fig. 14 Relation between an input and output

This coil sets a variable to **TRUE** if logical continuity is found.

This state remains until the variable is reset. For this reason, this coil is also referred to as conditionally setting.

4.2.4 Reset

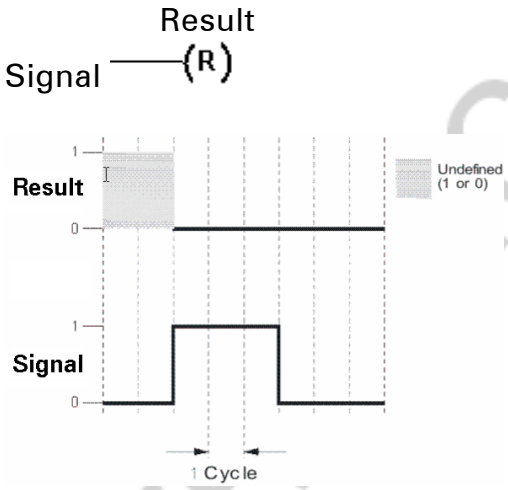


Fig. 15 Relation between an input and output

This coil sets a variable to **FALSE** if continuity is found.

4.2.5 Positive transition coil

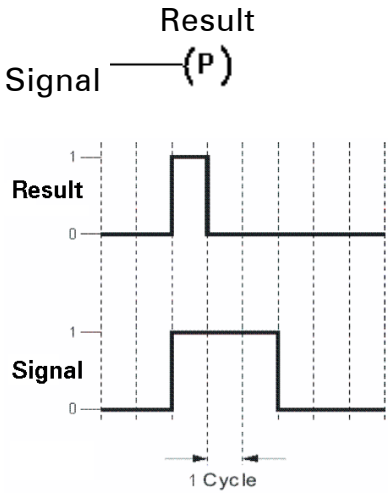


Fig. 16 Relation between an input and output

This coil sets a variable to **TRUE** for one cycle if continuity is found.

For all other cycles where continuity is found, the output remains set to **FALSE**.

4.2.6 Negative transition coil

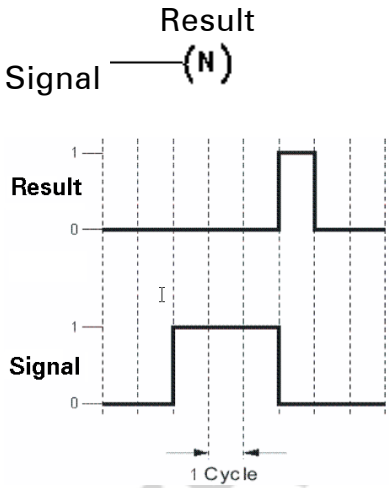
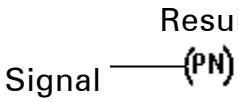


Fig. 17 Relation between an input and output

This coil sets a variable to **TRUE** for one cycle if **NO** continuity is found.

For all other cycles where no continuity is found, the value of the variable remains set to **FALSE**.

4.2.7 Positive and negative transition coil



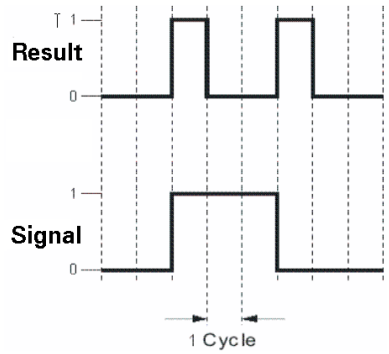


Fig. 18 Relation between an input and output

This coil unites the function of the positive and negative edge output.

Task: Part 1: Conveyor belt

In this training module, we will be creating an application for controlling a conveyor belt in four steps.

Create a program that controls the conveyor belt motor with the digital output "**gDoConvMotor**" by pressing the "**btnConvStart**" button. Don't connect the output directly with the input. Use the command variable "cmdManConvMotor" as an intermediate variable as shownin the image below.

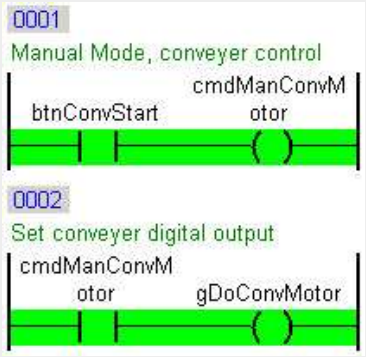


Fig. 20 Task 1 part1, operators

All new parts in the program are displayed in green.

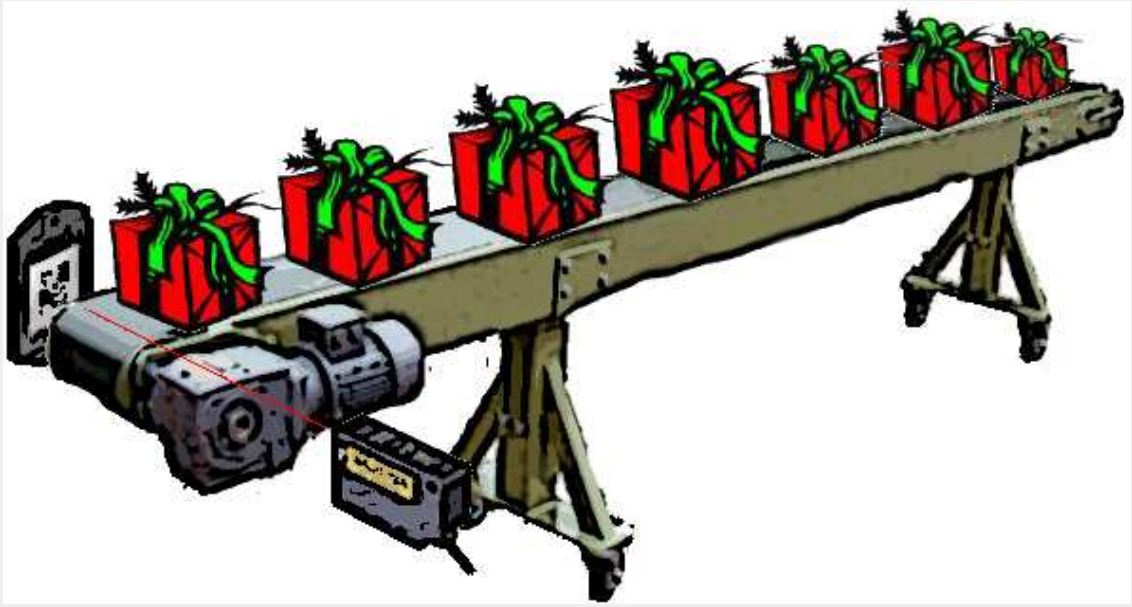


Fig. 20 Conveyor belt

5. LOGIC

5.1.1 AND operation

Fig. 21 Series blocks

If two or more contacts are connected in a series, the result is a logical **AND** operation.
When all of the conditions have been met, the output is set to TRUE.

Truth table:

Contact 1	Contact 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

5.1.2 OR operation

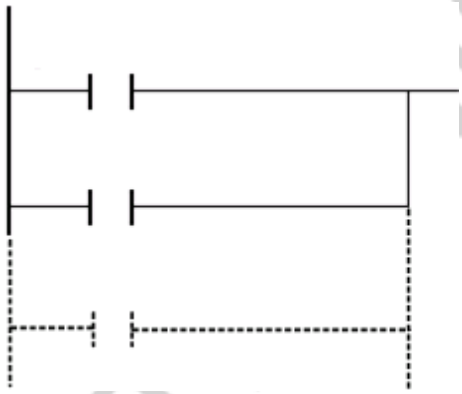


Fig. 22 Parallel blocks

A parallel block is equivalent to an **OR** operation.
If at least one of these parallel branches is **TRUE**, then the output is set to **TRUE**.

Truth table:

Contact 1	Contact 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

5.1.3 XOR operation

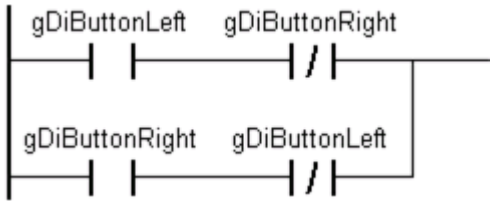


Fig. 23 Exclusive OR

The **Exclusive OR** operation is a combination of the logical **AND** and **OR** operations. If one of the two inputs is **TRUE**, then the output is also **TRUE**. If both inputs are **TRUE**, then the output is **FALSE**.

Truth table:

gDiButtonLeft	gDiButtonRight	Output
0	0	0
0	1	1
1	0	1
1	1	0

5.1.4 Branch

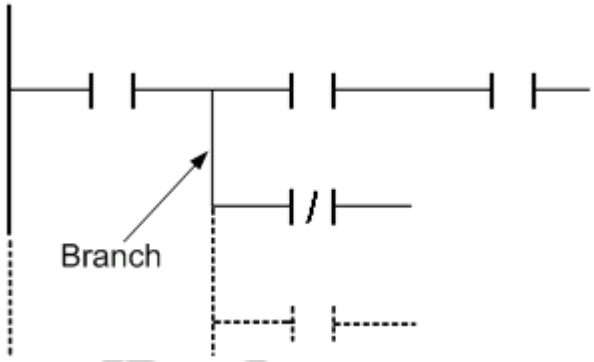


Fig. 24 Branch

In a network, a branch refers to a vertical line that connects two or more rows with one another.

5.1.5 Merge

A merge line is defined as another vertical line that runs parallel to a branch line and merges the branch circuits into a closed circuit (forming a parallel block).

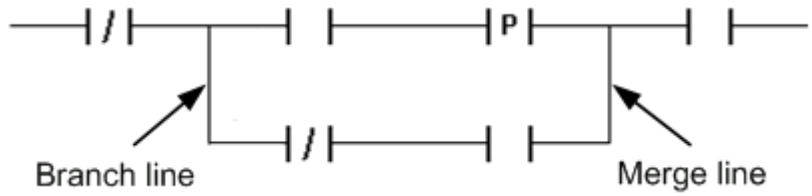


Fig. 25 Merge line

A single vertical line can be both a branch line and a merge line, as shown in the image below.

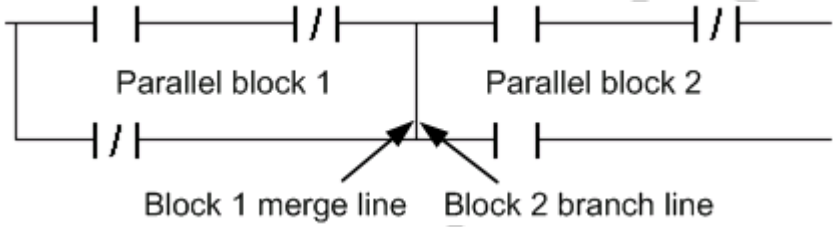


Fig. 26 Branch line and merge line

Task: Part 2: Conveyor belt

Right now, it's possible to control the conveyor belt by pressing a button. Now we want to add a few more functions for automatic mode.

Start the conveyor belt:

- If no material is detected by the final sensor of the "**gDiLoadConvEnd**" conveyor belt.
- If material is detected by the final sensor of the conveyor belt and the machine requests more material with the "**gDiMachAskMat**" digital input.

Stop the conveyor belt:

- If material is detected by the final sensor of the conveyor belt and the machine is not requesting any more material.

Your program may then appear like this:

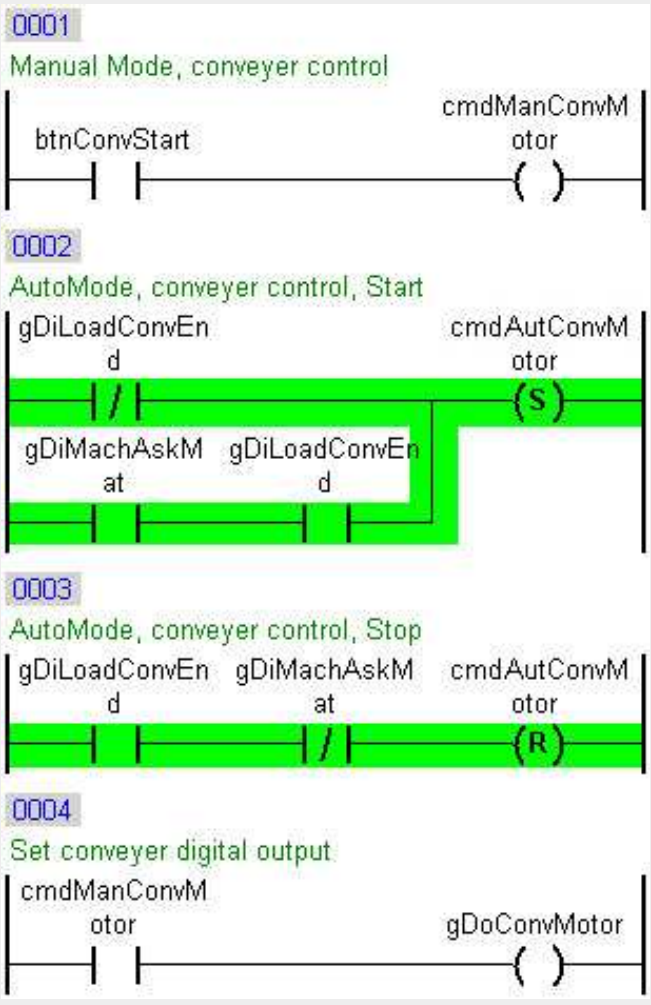


Fig. 27 Part 2 - Source code

6. CONTROLLING THE PROGRAM FLOW

6.1.1 Conditional jump

A conditional jump refers to a jump to a network with a symbolic name using a condition.

If the condition is TRUE, then the jump takes place. A jump label with a unique name must be present for each jump.

It is used to skip over networks in the program. This allows the program flow to be controlled efficiently. The program runtime is also reduced since networks are jumped over if they are not required.

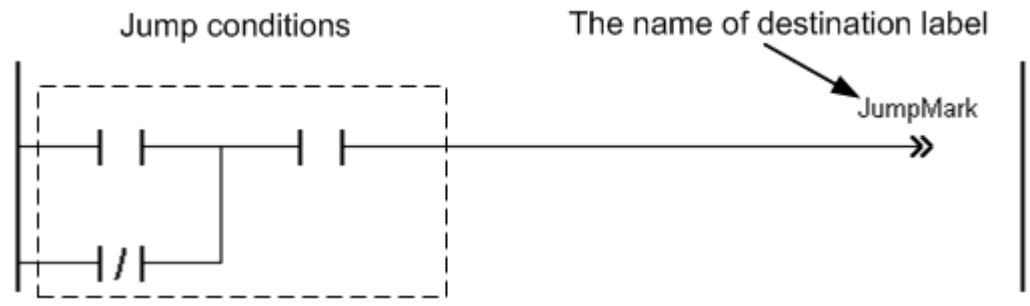


Fig. 28 Jump

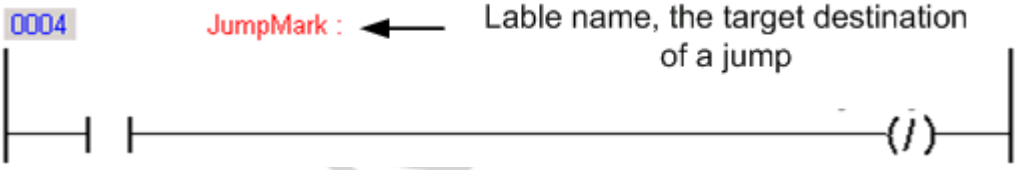


Fig. 29 Conditional jump

6.1.2 Return

The Return instruction is used to terminate the ladder diagram at a certain point. Any subsequent networks are no longer executed.

Fig. 30 Return

Task: Part 3: Conveyor belt

We will now add an input "**gDiAutoMode**" that we can use to switch between manual and automatic mode.

- Networks that belong to automatic mode are executed if "**gDiAutoMode**" is **TRUE**.
- Networks that belong to manual mode are only executed if "**gDiAutoMode**" is **FALSE**.
- Use conditional jumps.
- Use a new variable, "**cmdAutConvMotor**", in automatic mode.

Your program may then appear like this:

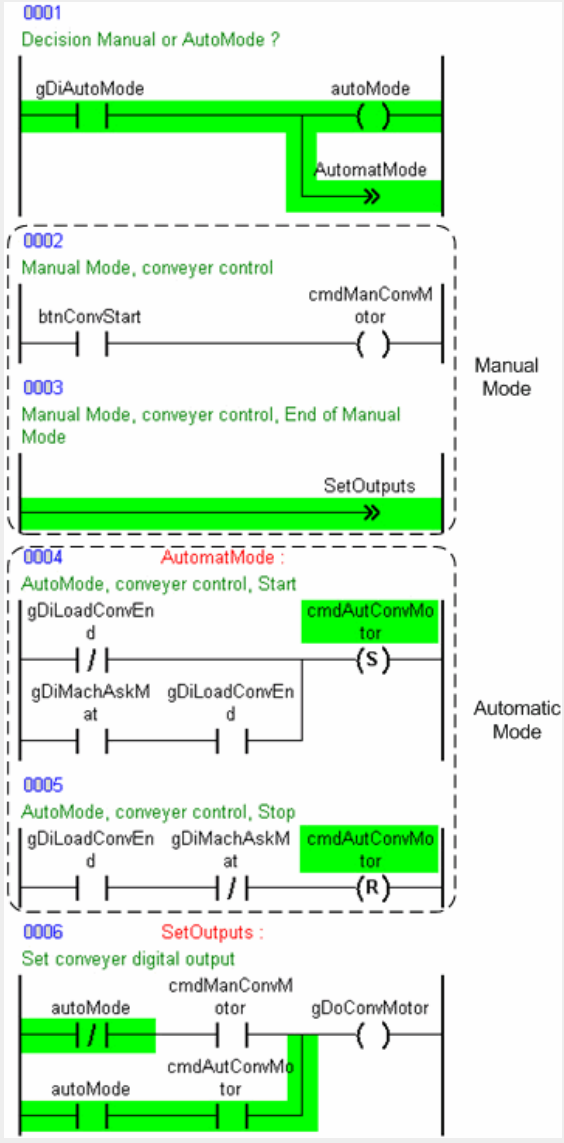


Fig. 31 Part 3 - Source code

7. USING FUNCTION BLOCKS

The Ladder Diagram editor in Automation Studio makes it possible to use function blocks.

If a function block is inserted, then input logic conditions are also represented by contact instructions driving the logic for the function block. A function block may have one or more coils as the outputs that store the status or result of a function. If a function block should be active at all times, then a connection can be made to the function block using a vertical line.

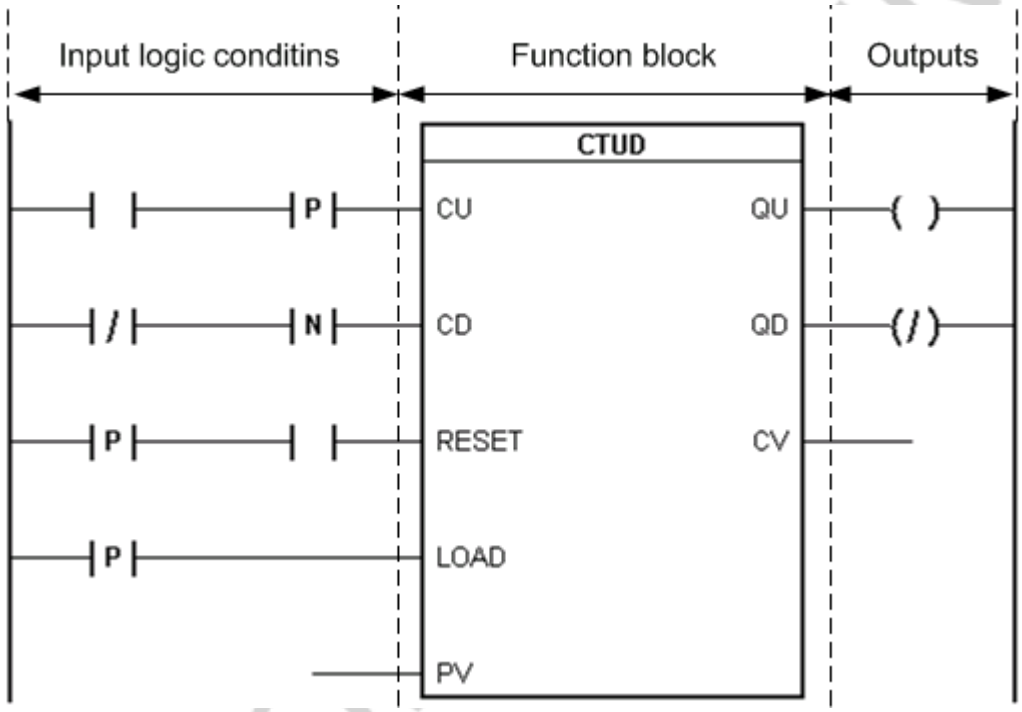


Fig. 32 Function block in Ladder Diagram

A function block can also have "analog" inputs and outputs. The icon or spacebar can be used to connected a variable.

Task: Part 4: Conveyor belt

Count the quantity of material that the conveyor belt is transporting in automatic mode. Use the **CTU** function block found in the **STANDARD** library.

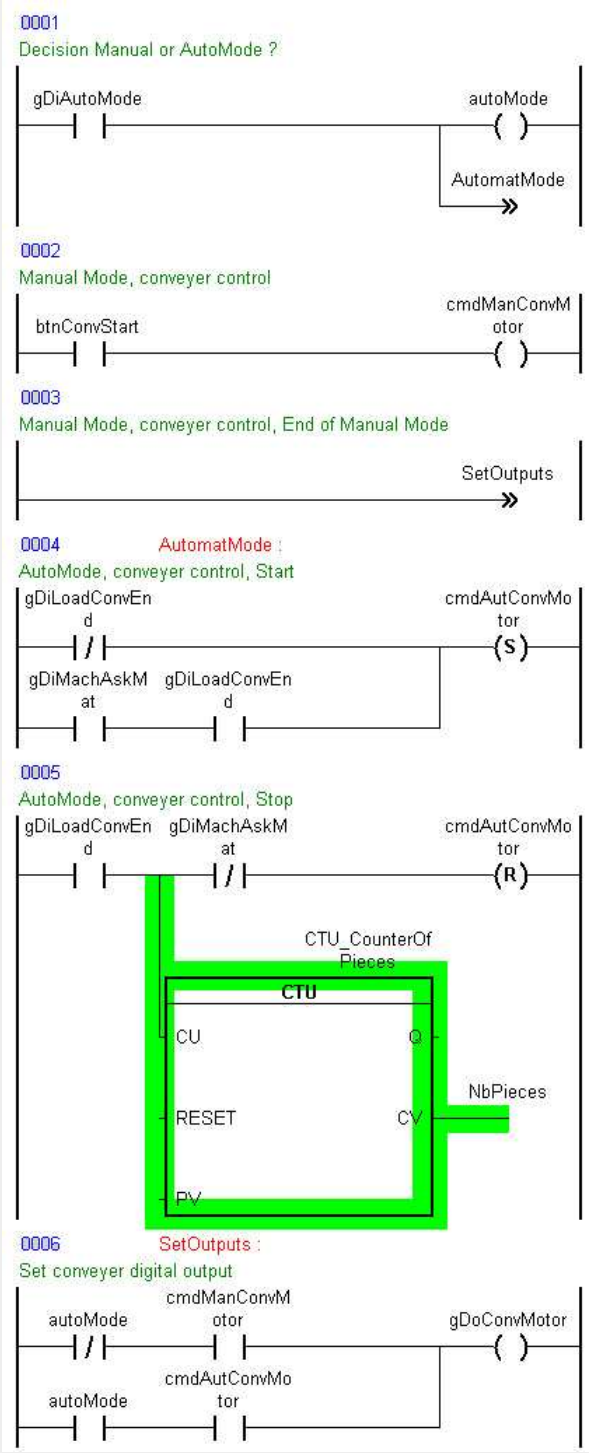


Fig. 33 Part 4 - Source doe

8. POWER FLOW

If logic continuity is present in a network, then its output is **TRUE**. Power flows from left to right in a network. Networks are executed one after the other except when the course is changed by returns or jumps.

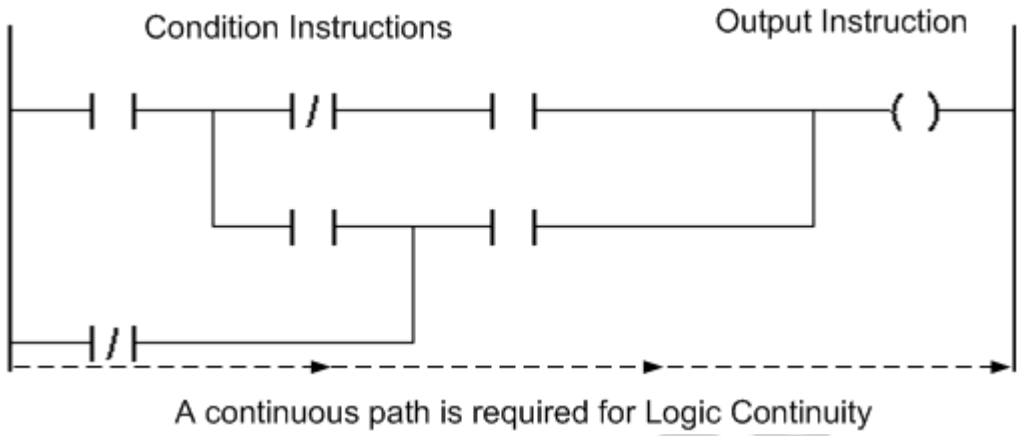


Fig. 34 Logic continuity

There are several different possibilities for logic continuity in this network.

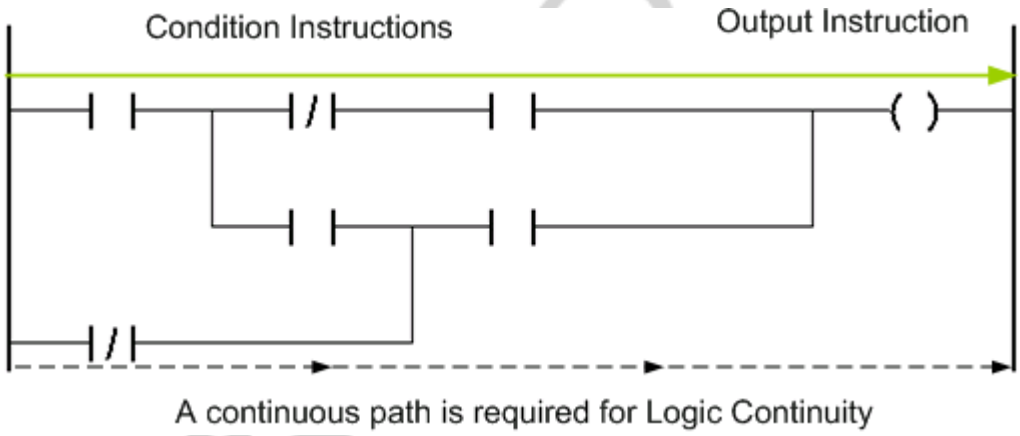


Fig. 35 Logical passage to the first command line

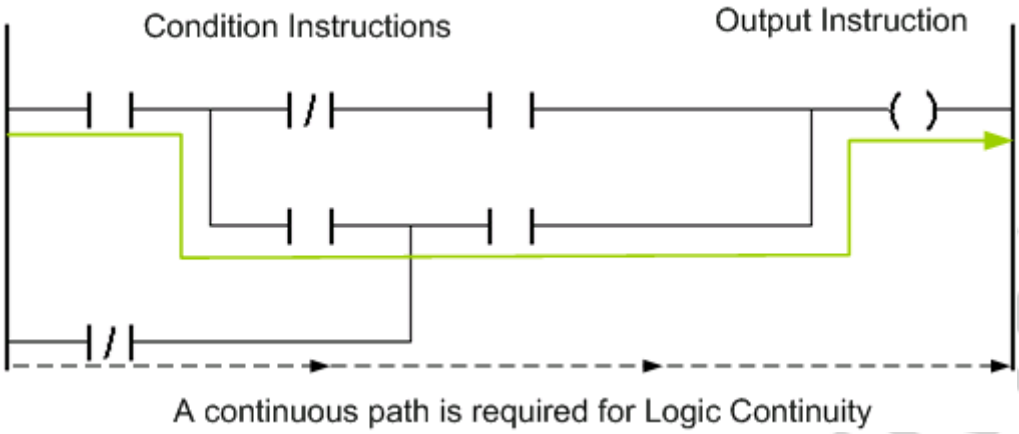


Fig. 36 The second continuous path

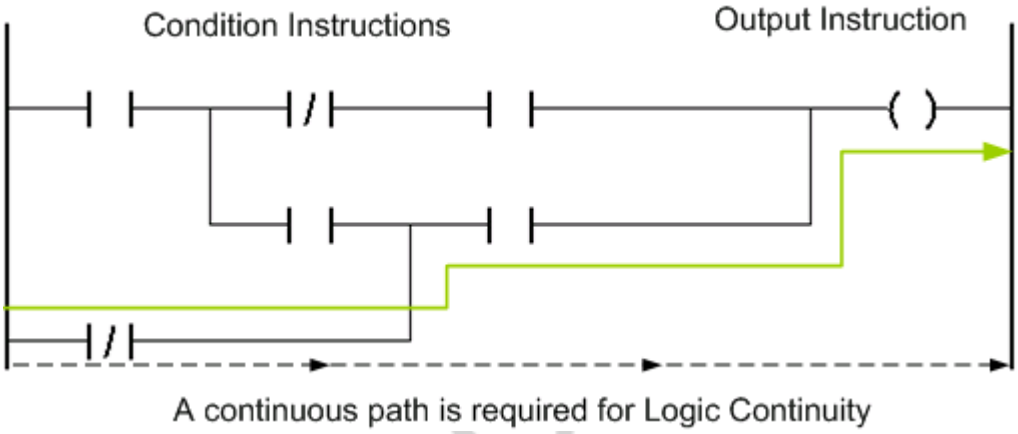


Fig. 37 The third continuous path

Unlike hard-wired relay logic, reversed power flow as shown in the image below is not possible in PLC logic.

If the logic calls for the implementation of reverse flow, the user must program it in with forward power flow to all contact elements.

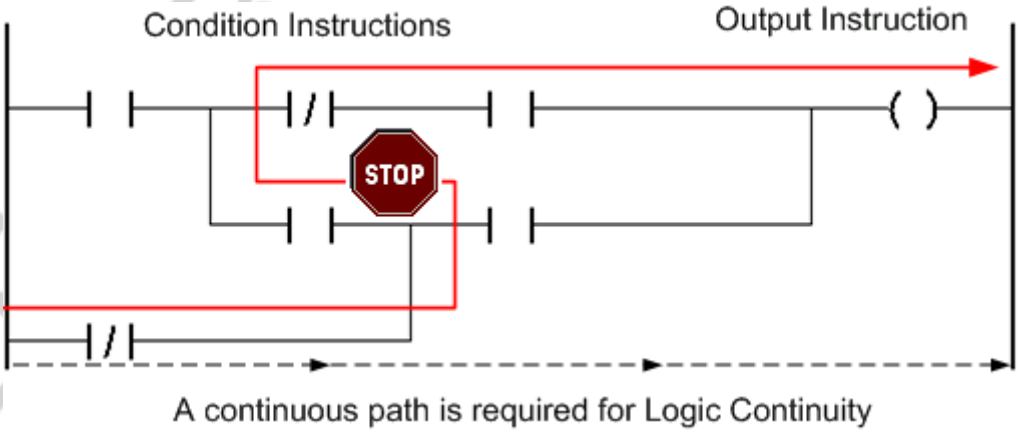


Fig. 38 A reverse continuity path is not possible

9. QUESTIONS AND EXERCISES

9.1 Questions

Why is the ladder diagram system so widely accepted? Why was it developed?

- It was developed as programmable logic that would serve as a subsitute for hard-wired relay logic.

When should Ladder Diagram be used?

- It is ideal for logical control tasks and operations.

What is a network?

- A network consists of at least one contact and one coil (input and output). If there is no connection between two rows, then they belong to two different networks. Networks are executed in the order they were programmed.

What is a contact in LD?

- The main purpose of a contact is to form a logical condition to control the output (coil).

What is a coil in LD?

- The coil receives the value that results from the logical conditions that arise.

9.2 Exercises

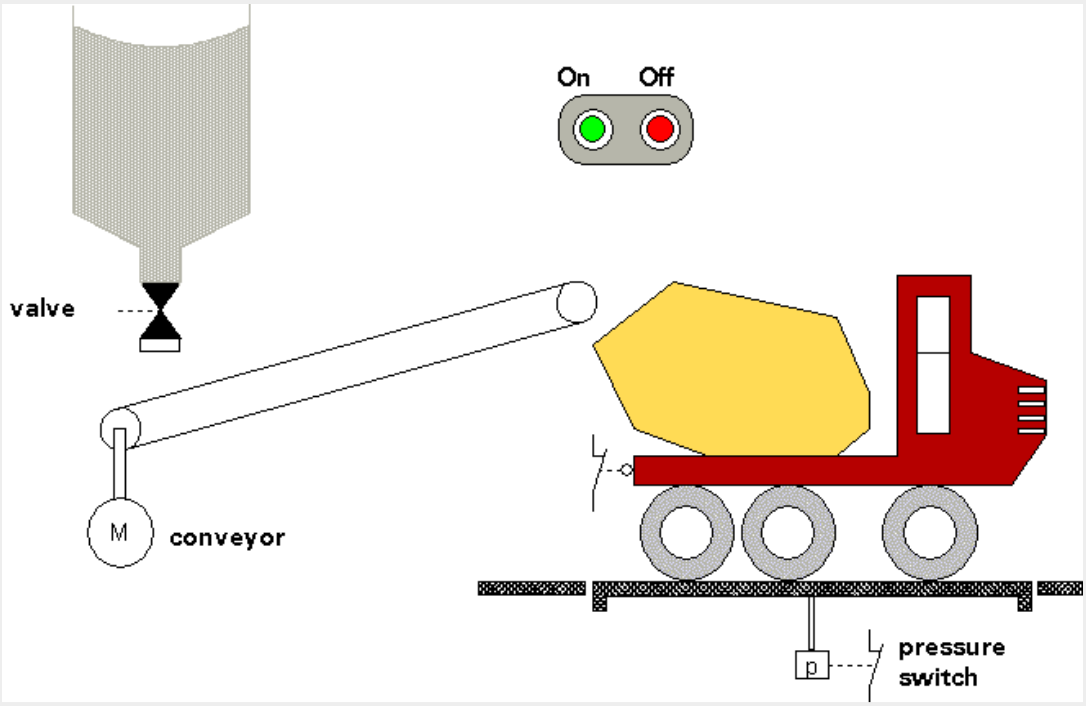
Task: Concrete filling system

In a cement mixing facility, cement is loaded onto a vehicle using a conveyor belt.

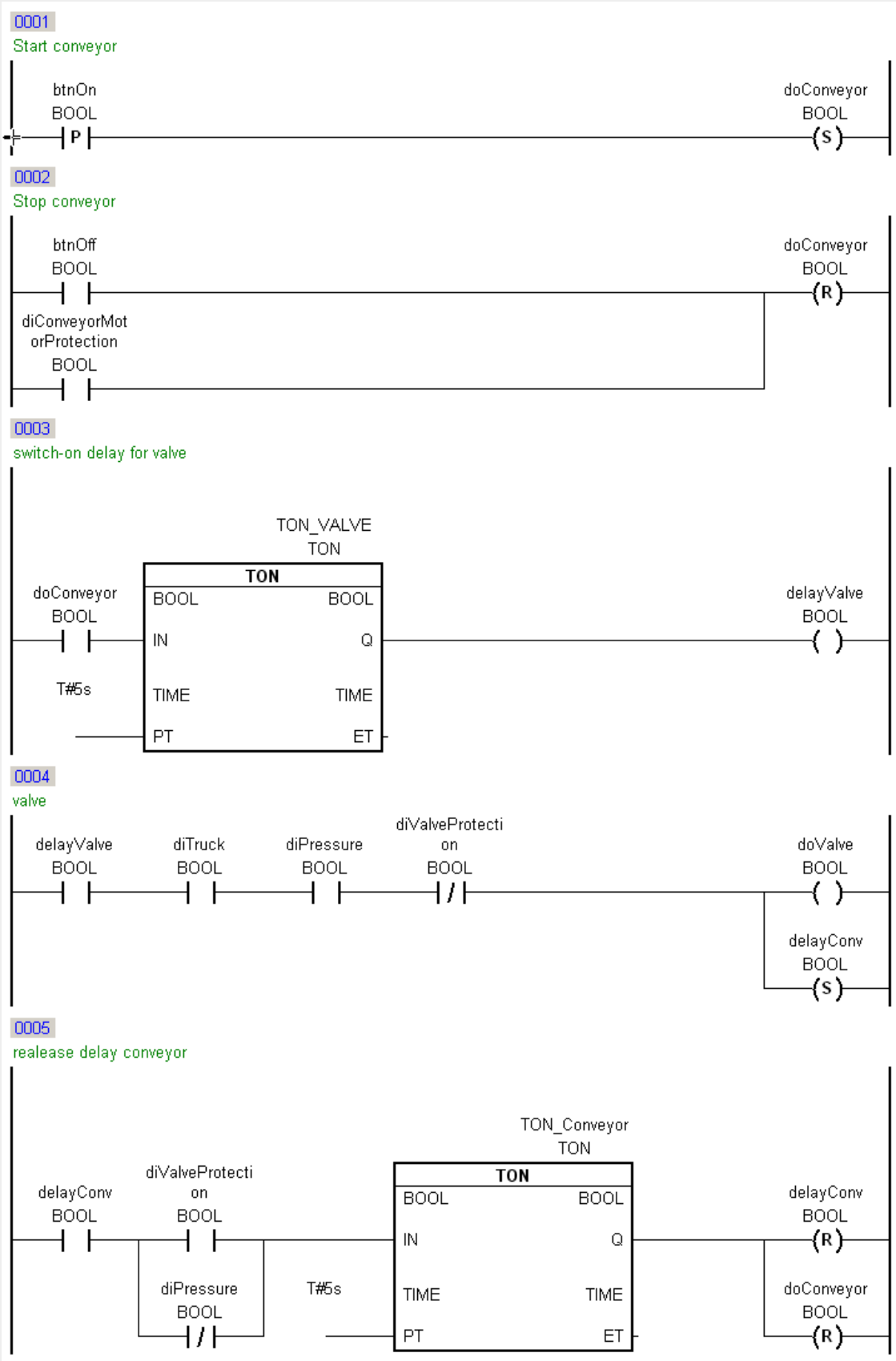
Filling begins by activating the On button (btnOn). The hydraulics activated using a magnet valve (doValve) can only be opened if the conveyor belt has been operating for 5 seconds and there is a vehicle under the belt (diTruck).

The magnet valve is switched off as soon as the permitted total weight of the vehicle has been reached (diPressure). However, the conveyor belt should continue to run for another 5 seconds.

The entire system is switched off immediately when the Off button is pressed (btnOff). If the conveyor belt is disrupted (diConveyorMotorProtection), the magnet value and the conveyor belt (doConveyor) should be switched off immediately. If the magnet value is disrupted (diValveProtection), it should be closed immediately, but the belt should run empty for another 5 seconds.



The ladder diagram could look like this:



10. SUMMARY

Programming with ladder diagram is still very popular. It was developed to program logical switches so that hard-wired relay logic could be replaced.

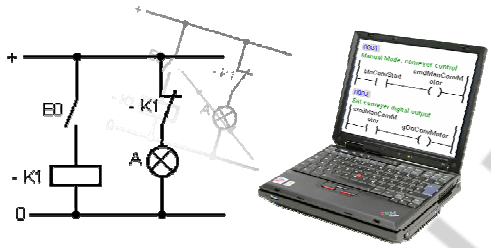


Fig. 39 Summary

Using analog signals and function blocks makes it possible to create high-powered applications using Ladder Diagram. Additional program flow control elements extend the function range.
The Automation Studio, program sequences can be traced using Power Flow. Colors are used to display the status of lines where are currently conducting electricity.

Overview of training modules

TM200 – B&R Company Presentation **
TM201 – B&R Product Spectrum **
TM210 – The Basics of Automation Studio
TM211 – Automation Studio Online Communication
TM212 – Automation Target **
TM213 – Automation Runtime
TM220 – The Service Technician on the Job
TM223 – Automation Studio Diagnostics
TM230 – Structured Software Generation
TM240 – Ladder Diagram (LAD)
TM241 – Function Block Diagram (FBD)
TM246 – Structured Text (ST)
TM247 – Automation Basic (AB)
TM248 – ANSI C
TM250 – Memory Management and Data Storage
TM260 – Automation Studio Libraries I
TM261 – Closed Loop Control with LOOPCONR

TM400 – The Basics of Motion Control
TM410 – The Basics of ASiM
TM440 – ASiM Basic Functions
TM441 – ASiM Multi-Axis Functions
TM445 – ACOPOS ACP10 Software
TM450 – ACOPOS Control Concept and Adjustment
TM460 – Starting up Motors

TM500 – The Basics of Integrated Safety Technology
TM510 – ASiST SafeDESIGNER

TM600 – The Basics of Visualization
TM610 – The Basics of ASiV
TM630 – Visualization Programming Guide
TM640 – ASiV Alarm System
TM650 – ASiV Internationalization
TM660 – ASiV Remote
TM670 – ASiV Advanced

TM700 – Automation Net PVI
TM710 – PVI Communication
TM711 – PVI DLL Programming
TM712 – PVIServices
TM730 – PVI OPC

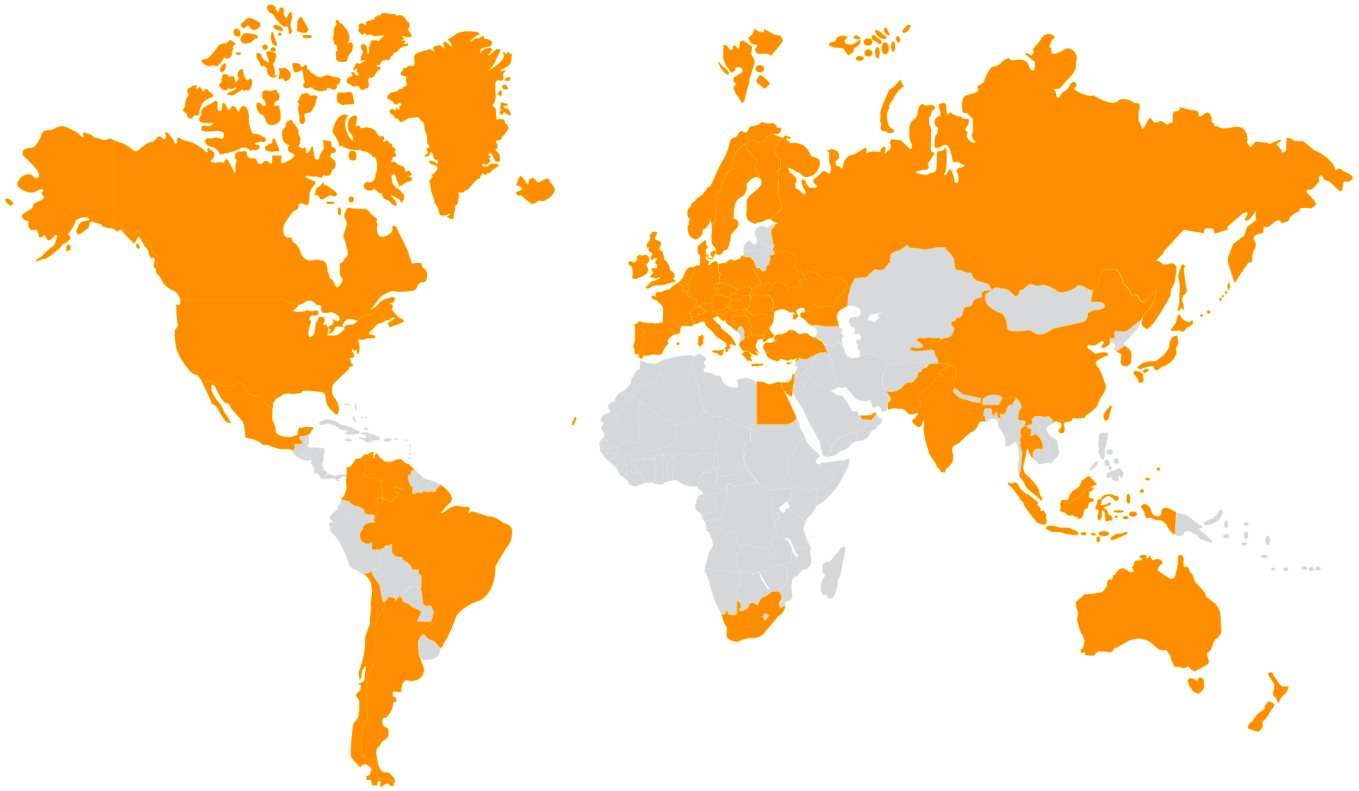
TM800 – APROL System Concept
TM810 – APROL Setup, Configuration and Recovery
TM811 – APROL Runtime System
TM812 – APROL Operator Management
TM813 – APROL XML Queries and Audit Trail
TM830 – APROL Project Engineering
TM840 – APROL Parameter Management and Recipes
TM850 – APROL Controller Configuration and INA
TM860 – APROL Library Engineering
TM865 – APROL Library Guide Book
TM870 – APROL Python Programming
TM890 – The Basics of LINUX

**) see Product Catalog

CORPORATE HEADQUARTERS

Bernecker + Rainer Industrie-Elektronik Ges.m.b.H.
B&R Strasse 1
5142 Eggelsberg
Austria
Tel.: +43 (0) 77 48 / 65 86 - 0
Fax: +43 (0) 77 48 / 65 86 - 26
info@br-automation.com
www.br-automation.com

140 offices in more than 55 countries - **www.br-automation.com/contact**



Australia • Argentina • Austria • Belarus • Belgium • Brazil • Bulgaria • Canada • Chile • China • Colombia • Croatia • Cyprus
Czech Republic • Denmark • Egypt • Emirates • Finland • France • Germany • Greece • Hungary • India • Indonesia
Ireland • Israel • Italy • Japan • Korea • Luxemburg • Kyrgyzstan • Malaysia • Mexico • The Netherlands • New Zealand
Norway • Pakistan • Poland • Portugal • Romania • Russia • Serbia • Singapore • Slovakia • Slovenia • South Africa
Spain • Sweden • Switzerland • Taiwan • Thailand • Turkey • Ukraine • United Kingdom • USA • Venezuela • Vietnam