CSIKÓS SÁNDOR

PLC PROGRAMMING

UNIVERSITY OF SZEGED
FACULTY OF ENGINEERING
SZEGED, 2014

PLC PROGRAMMING

Supported by

Instrument for Pre-accession Assistance (IPA)
HUSRB/1203/221/075 project
JOINT DEVELOPMENT OF CURRICULA AND TEACHING MATERIALS OF MECHANICAL ENGINEER ON MSc LEVEL

Written by

Csikós Sándor

Editorial work:

Csikós Sándor

Lectured by

Dr. Gyeviki János PhD

Published by
University of Szeged, Faculty of Engineering – Szeged (HUNGARY), 2014

ISBN xxx-xxx-xxx-xxx

CONTENT

FOREWORD

Programmable Logic Controllers (PLCs) have been used in industrial applications for more than 35 years and today most of the automated manufacturing systems use PLCs as control units. Nowadays the potential physical platform to realize a supervisor in industry is the PLC.

Its high reliability, high stability, friendly programming environment and complementing with the touch-type man-machine interface, make various industrial controls more convenient and more visual, economic and reliable.

This book is designed to help people understand the inner workings of PLC-s. To design and write a well written program with **RSLogix 5000**.

1. WHAT IS A PLC?

The PLC, also known as programmable controller is defined by the National Electrical Manufacturers Association (NEMA) in 1978 as:

"A digitally operating electronic apparatus which uses a programmable memory for the internal storage of instructions by implementing specific functions such as logic, sequencing, timing, counting, and arithmetic to control, through digital or analog input/output modules, various types of machines or processes."

1.1. PLC-s then and now (A short history of programmable logic controllers)

Prior to the existence of the PLCs, the control of machines was done through hard-wired systems. These systems consisted of relays, cam timers, drum sequencers and dedicated closed loop controllers in the control panels. Unfortunately, for big projects, the amount of relays counted in thousands, because of this the control panels were quite large. When a change in the system was required, the task was expensive and time consuming.

The American automotive manufacturing industry was the first to realize the need for replacing relay-based controls. In 1968, Bedford Associates, of Bedford Massachusetts, created the first PLC (named 084), for GM Hydramatic (division of General Motors). The PLC was designed to replace hard-wired controls. Because PLCs were created for the needs of the automotive industry, it needed to withstand harsh conditions, such as dust, moisture, heat and cold.

The Modicon (MOdular DIgital CONtroller), company created by Bedford Associates, was dedicated to developing, manufacturing, selling and servicing PLCs.

1.2. How the PLC works

The first PLCs were programmed with a technique that was based on relay logic wiring schematics. This eliminated the need to teach the electricians, technicians and engineers how to program a computer. This method has stuck and it is the most common technique for programming PLCs today.

When a process is controlled by a PLC it uses inputs from sensors to make decisions and update outputs to drive actuators, as shown in Figure 1. The process is a real process that will change over time. Actuators will drive the system to new states (or modes of operation). This means that the controller is limited by the sensors available.

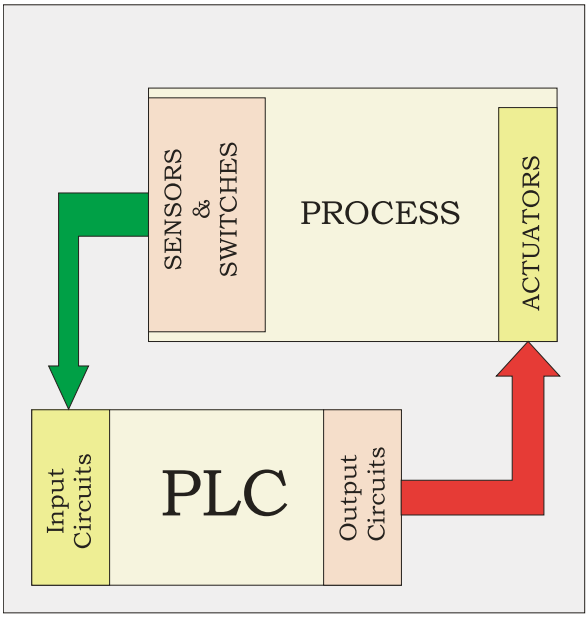


Figure 1 Connection between the PLC and the controlled process

The control loop is a continuous cycle of the PLC reading inputs, solving the ladder logic, and then changing the outputs. Like any computer this does not happen instantly. Figure 2 shows the basic operation cycle of a PLC. When power is turned on initially the PLC does a quick sanity check to ensure that the hardware is working properly. If there is a problem the PLC will halt and indicate there is an error. For example, if the PLC power is dropping and about to go off this will result in one type of fault.

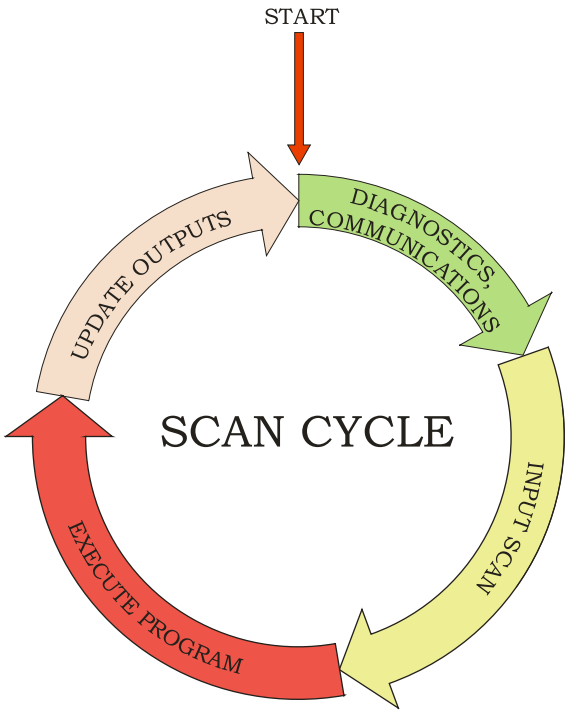


Figure 2 A PLC scan cycle

If the PLC passes the sanity check it will then scan (read) all the inputs. After the inputs values are stored in memory the ladder logic will be scanned (solved) using the stored values - not the current values. This is done to prevent logic problems when inputs change during the ladder logic scan. During execution the output states are stored in a different location in memory. When the ladder logic scan is complete the outputs will be updated (the output values will be changed) (Figure 3). After this the system goes back to do a sanity check, and the loop continues indefinitely. Unlike normal computers, the entire program will be run every scan. The typical time for each of the stages is in the order of milliseconds.

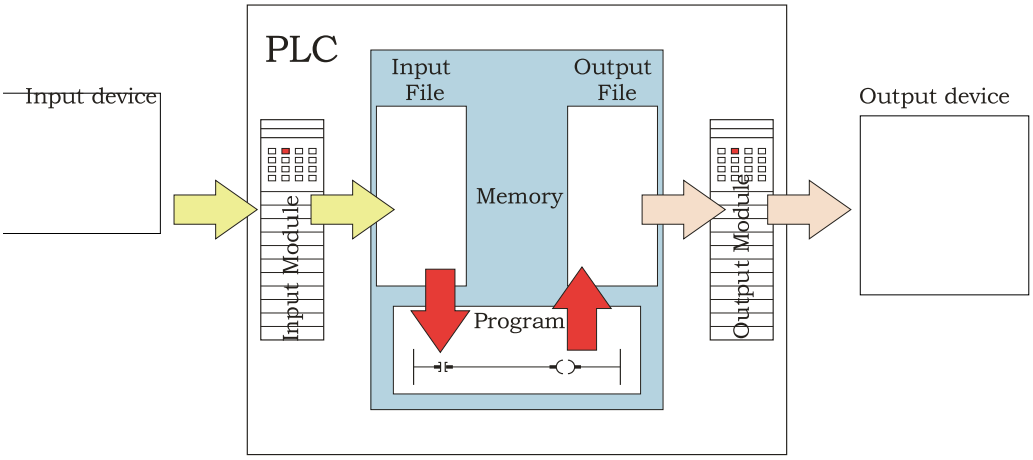


Figure 3 The work flow of a PLC

1.3. The architecture of a PLC

A PLC consists of five main parts: CPU, Memory, Power Supply, I/O and Communication port, which are shown in Figure 4.

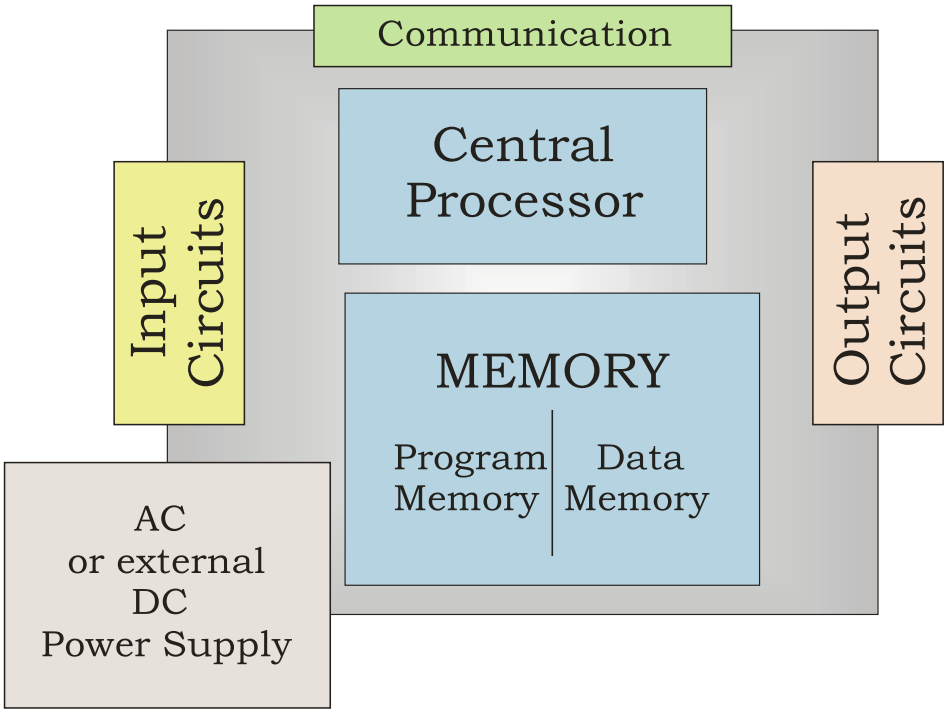


Figure 4 The main parts of a PLC

EPROM memory is used to store the program instructions in the PLC. RAM memory is used to store the states of the inputs and outputs. A computer or hand-held programmer can be used to load and save the programs into the PLC.

The central processor receives, decodes, stores and processes information and executes the PLC program that is stored in memory.

Every PLC must have some means of receiving and interpreting signals from real-world sensors known as inputs, as well as be able to effect control over real world control elements known as outputs. Inputs are devices that supply a signal/data to a PLC. Typical examples of inputs are push buttons, switches, encoders and measurement devices. Outputs are devices that wait for a signal or data from the PLC to perform their control functions. Typical examples of outputs are solenoids, lights, horns, motors and valves. This is 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 physical input and output modules can be discrete or analog and can be selected and specified when purchasing the PLC depending on the number of the required I/O lines. Discrete modules have two states on and off while analog modules have a finite number of values that they can take over a certain range. The number of these values is determined by the resolution of the analog module.

1.3.1. Discrete I/O modules

Discrete field inputs devices have normally open (N.O.) and/or normally closed (N.C.) contacts. The simplest example are pushbuttons which can be purchased with either N.O. or N.C. mechanical contacts. "Normally" implies the state of the contacts when the device is NOT activated. The discrete I/O modules connects field inputs devices of the ON/OFF nature like limit switches, push button switches, solenoids, solenoid valves or electro-mechanical relay etc. Each discrete I/O module needs a supply voltage source. Since these voltages can be of different magnitude or types, I/O modules are available at various AC and DC voltages ratings as shown in Table 1. Furthermore, the inputs and outputs are connected to LED's to indicate the state and operation of the I/O module.

Table 1 Common ratings for discrete I/O interface modules ordered by popularity

Interface input module	Interface output module
12-24 V DC	120 V AC
100-120 V AC	24 V DC
10-60 V DC	12-48 V AC
12-24 V AC/DC	12-48 V DC
5 V DC (TTL)	5V DC (TTL)
200-240 V AC	230 V AC
48 V DC	
24 V AC	

There are trade-offs depending which voltage we choose:

- DC voltages are usually lower, and therefore safer (i.e., 12-24V).
- DC inputs are very fast, AC inputs require a longer on-time. For example, a 50 Hz wave may require up to 1/50 s for reasonable recognition.
- DC voltages can be connected to larger variety of electrical systems.
- AC signals are more immune to noise than DC, so they are suited to long distances, and noisy (magnetic) environments.
- AC power is easier and less expensive to supply to equipment.
- AC signals are very common in many existing automation devices.

Input Module:

The input module connects the input terminals to the rest of the system. Each terminal is usually electrically isolated from the internal electronics by optocouplers. This is a way of passing on the value of the input by use of a

light emitting diode and phototransistor. The optocouplers protect the other parts of the PLC from voltage spikes that would damage them. Thanks to the optocouplers even if the input module breaks down the rest of the system does not suffer harm. This does not mean that optocouplers make a PLC indestructible, they can withstand voltages up to 10 kV-s before letting it affect the rest of the system.

Input modules usually do not supply power so an external power supply is required to power the inputs. If there are more power supplies in a system they need to share a common wire (the terminal labeled 0 V). There are two types of discrete input modules current-sinking and current-sourcing. 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.

A sinking input, also known as PNP input requires a device that sources "pushes" the current. The device accepting (conventional flow) current into its control terminal is said to be sinking current (Figure 5). While a sourcing input also known as NPN input requires a device that sinks "pulls" the current. The device sending (conventional flow) current out of its control terminal to some other device(s) is said to be sourcing current (Figure 6).

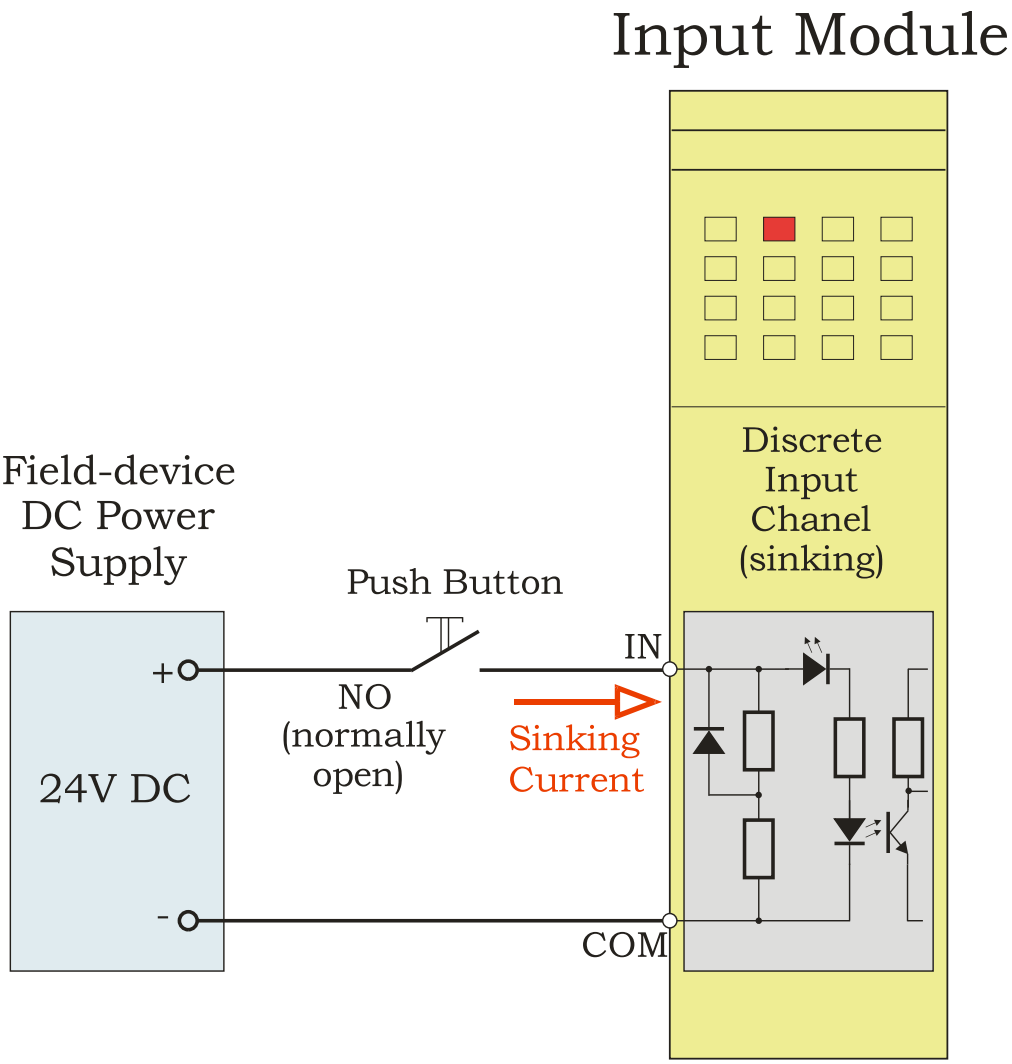


Figure 5 Sinking (PNP) input module

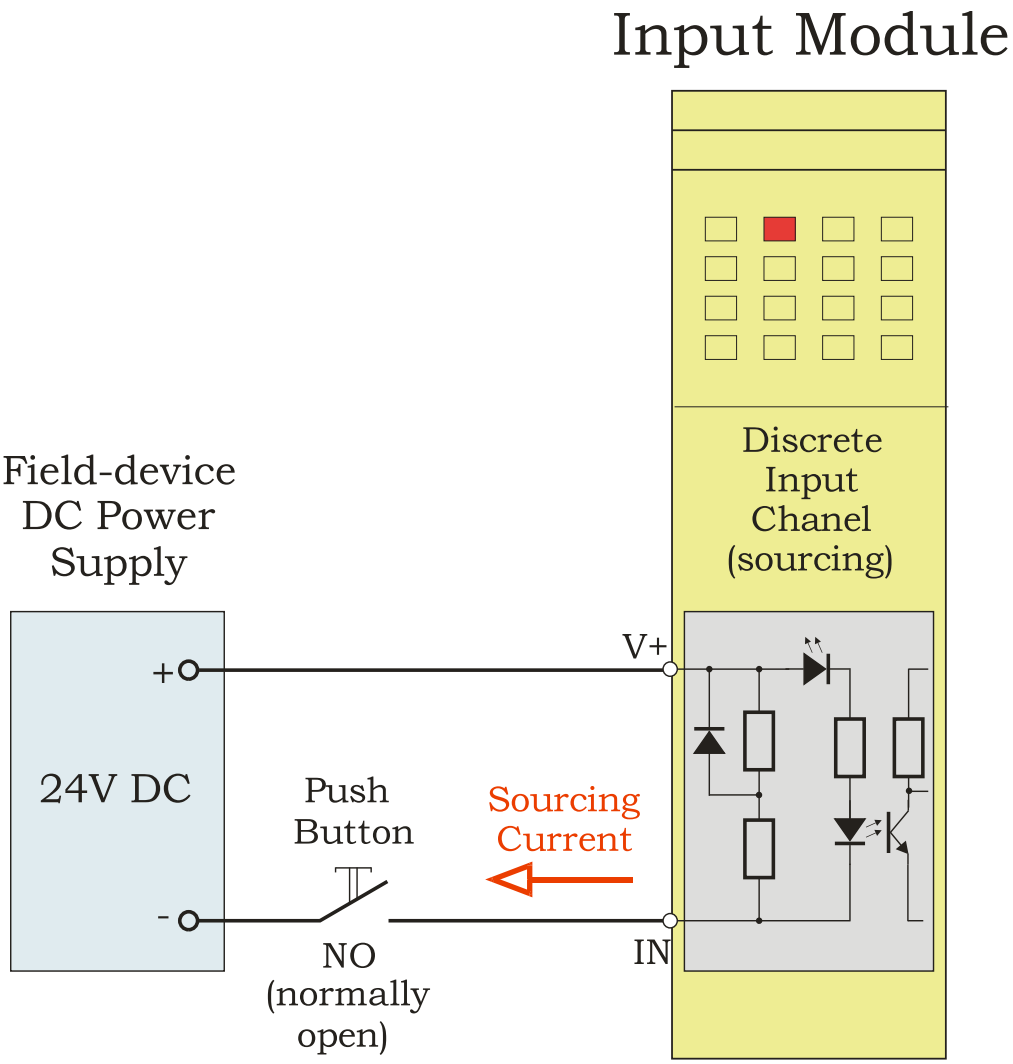


Figure 6 Sourcing (NPN) input module

Output Module:

The output module contains switches activated by the CPU in order to connect two terminals and so allows current to flow into the external circuit. Care must be taken not to overload the contacts. The switch may be a transistor or a relay. Examples of discrete control devices are Indicator lamps, solenoid valves, and motor contactors (starters). In a manner similar to discrete inputs, a PLC connects to any number of different discrete final control devices through a discrete output channel. 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. Alternatively, small electromechanical relays may be used to interface the PLC’s output bits to real-world electrical control devices.

As with input modules output modules usually do not supply power, so an external power supply is required to power the outputs. If there are more power supplies in a system they need to share a common wire (the terminal labeled 0 V). There are two types of discrete output modules current-sinking (NPN) and current-sourcing (PNP). Notice the difference between the sourcing input and sourcing output transistors. This is because in the case of the input the signal supplying device's transistor is dominant.

A sinking output sinks "pulls" the current through the load. The device accepting (conventional flow) current into its control terminal is said to be sinking current (Figure 7). While a sourcing output also known as PNP output sources "pushes" the current through the load. The device sending (conventional flow) current out of its control terminal to some other device(s) is said to be sourcing current (Figure 8).

Relay output modules can switch both DC and AC currents. Relay output modules usually have separated groups. Therefore, different voltages can be applied to each group as the specific application requires. These modules can be easily identified by the clicking sound they make. They are the most universal, but since they use mechanical switches they may be slow for some applications. Figure 9 shows a relay output module.

Triac output modules are used to drive AC loads at high switching speeds. Figure 10 shows a triac output module.

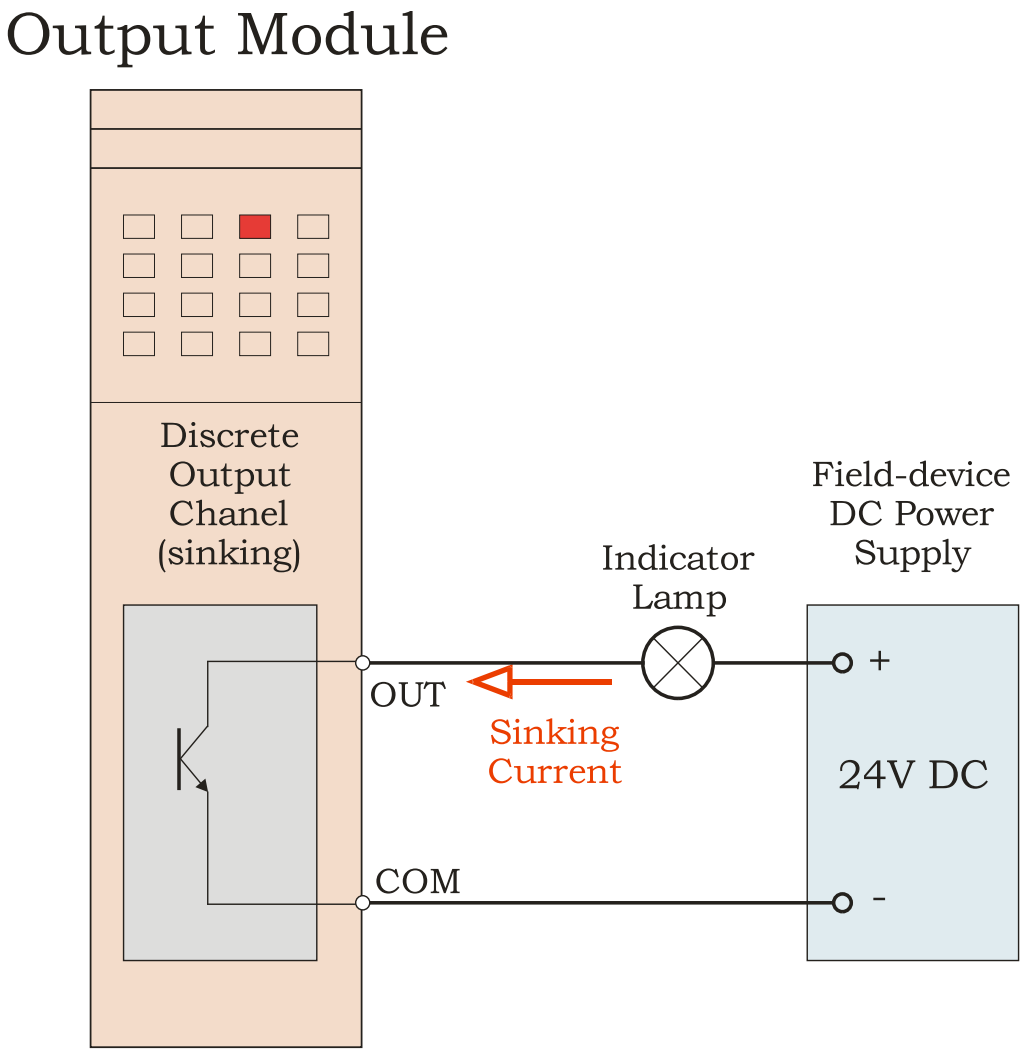


Figure 7 Sinking (NPN) output module

Output Module

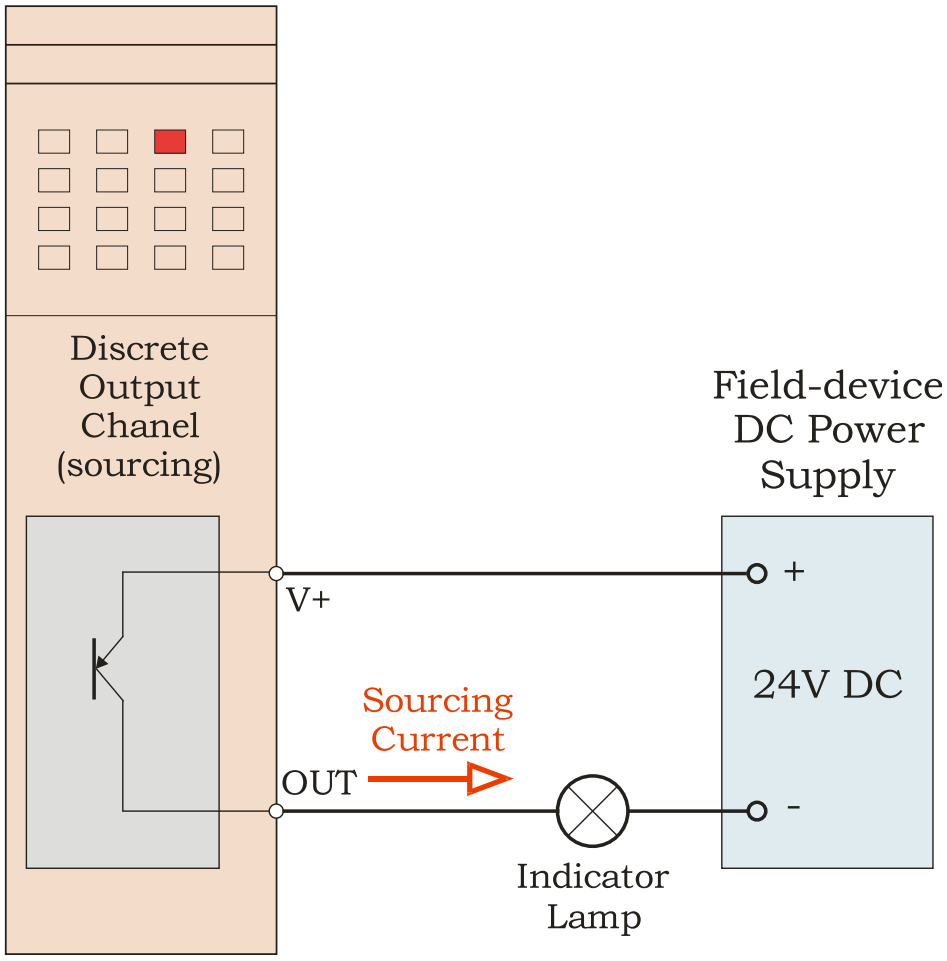


Figure 8 Sourcing (PNP) output module

Output Module

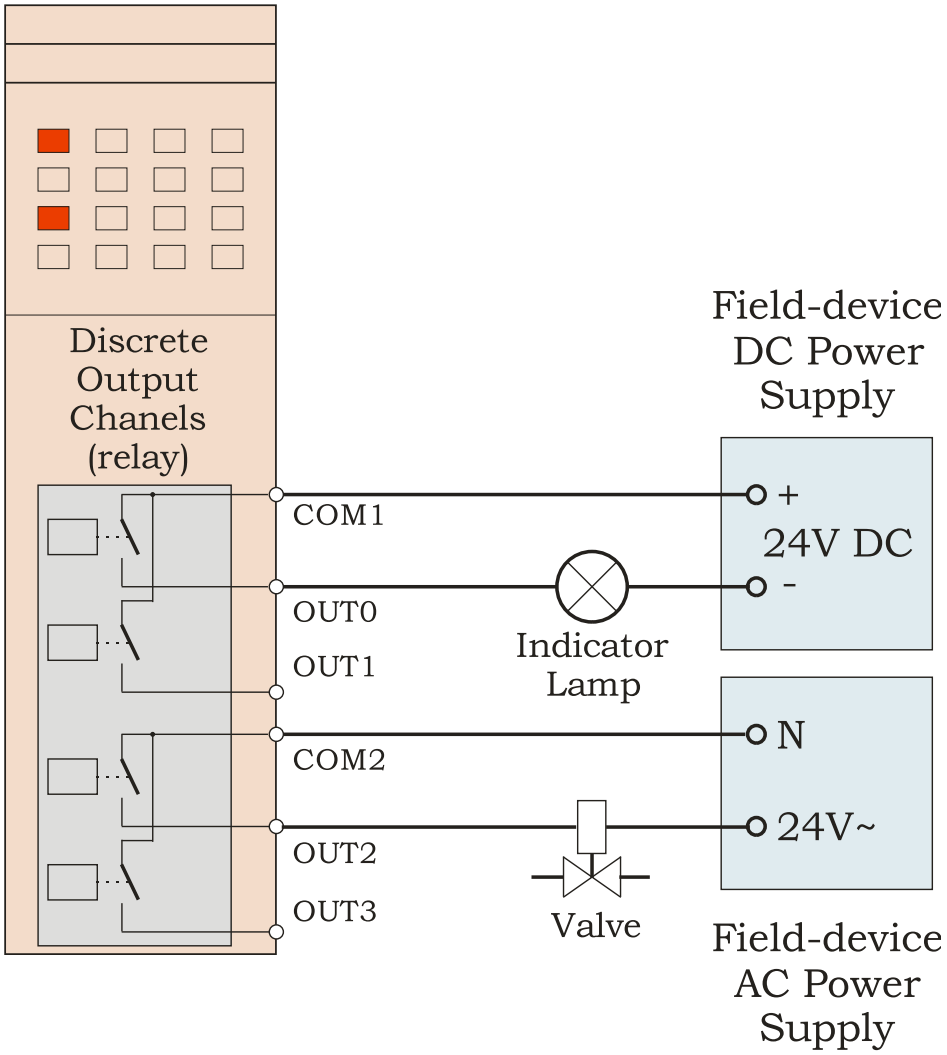


Figure 9 Relay output module

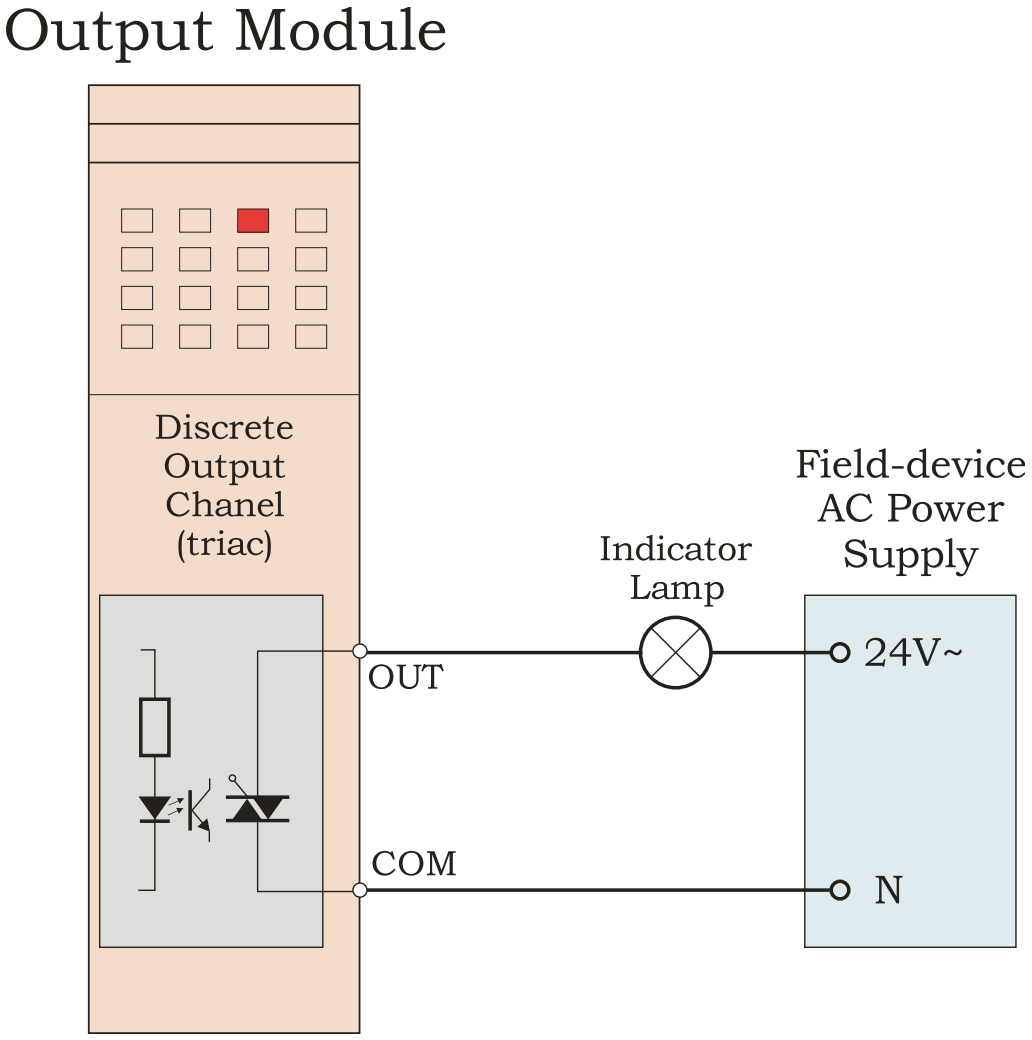


Figure 10 Triac output module

1.3.2. Analog I/O modules

Analog I/O modules differ from discrete I/O modules in the number of states they can have and their range. To translate real signals to the PLC and vice-versa the PLC uses converters. Analog inputs have an analog to digital converter (ADC) while analog outputs have a digital to analog converter (DAC). The number of states the converter can recognize or create is characterized by its resolution. If the modules resolution is n bit then number of states is 2^n. This is the number by with you divide the range of the module to obtain the value of 1 bit. Typical analog I/O ranges can be 0-10 V, 0-20 mA, 4-20 mA.

If you have an analog module with a range of 0-10 V and a resolution of 10 bits that means the module has 1024 states. So 1 bit changes the value by 0,009766 V.

1.4. Programming languages

There exists a large number of languages for PLCs because each manufacturer develops their own languages for their controller. However there are five (mostly) standardized languages:

- Structured text (**ST**)
- Instruction list (**IL**)
- Ladder diagram (**LD**)
- Function block diagram (**FBD**)
- Sequential function chart (SFC)

Besides "structured text" and "sequential function chart" the others are oriented on the logic of the circuits. This is reflected in the form of programming. Some languages are better suited for some tasks. Figure 11 shows which languages are suited for which task. Most new PLC-s know at least the **LD**, **ST** and **SFC** languages and allow you to combine them, so you can use the language most suited for your needs.

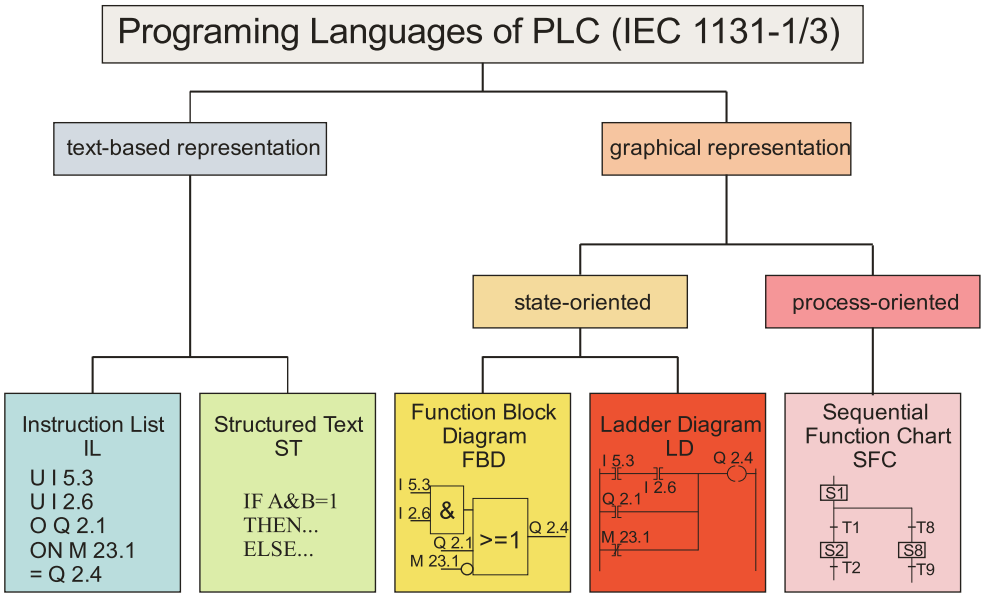


Figure 11 Which tasks are suited which language

To discuss further topics of programming we are going to use Ladder Logic (**LL**) this is not the same as the **LD** language. In **LL** we have 2 power rails (left power rail, right power rail). The power rails simulate the power supply lines. Input instructions are entered on the left, output instructions are entered on the right. We activate outputs by having a logical continuity to the output. Logical continuity in a ladder rung occurs when there is a continuous path of **TRUE** conditions from the left power rail to the output instruction(s). Let's see what instructions this language has. The instructions symbols are depicted on Figures 12-14. Each instruction has a memory location assigned to it.

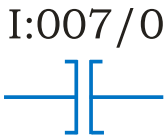


Figure 12 Normally open contact (input)

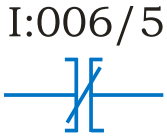


Figure 13 Normally closed contact (input)

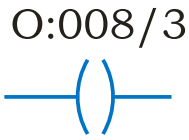


Figure 14 Coil (output)

Normally open contact reads its memory location. If the memory value is **1** the instruction is evaluated **TRUE** otherwise the instruction is **FALSE**.

Normally closed contact reads its memory location. If the memory value is **0** the instruction is evaluated **TRUE** otherwise the instruction is **FALSE**.

Coil writes a memory location. If there is a logical continuity to this instruction its memory location will be **1** otherwise it will be **0**.

To show which instruction is evaluated **TRUE** we will highlighted it green. An example of a **LL** program can be seen in Figure 15. To interpret this

diagram, imagine that the power is on the vertical line on the left hand side, we call this the hot rail. On the right hand side is the neutral rail. In Figure 15 there are three rungs, and on each rung there are combinations of inputs and outputs. If the inputs are opened or closed in the right combination the power can flow from the hot rail, through the inputs, to power the outputs, and finally to the neutral rail. An input can come from a sensor, switch, or any other type of sensor. An output will be some device outside the PLC that is switched on or off, such as lights or motors.

In the top rung the contacts are normally open and there is no logical continuity since **input2** is false. In the middle rung we see a normally closed contact with no logical continuity. In the bottom rung we have an example of logical continuity.

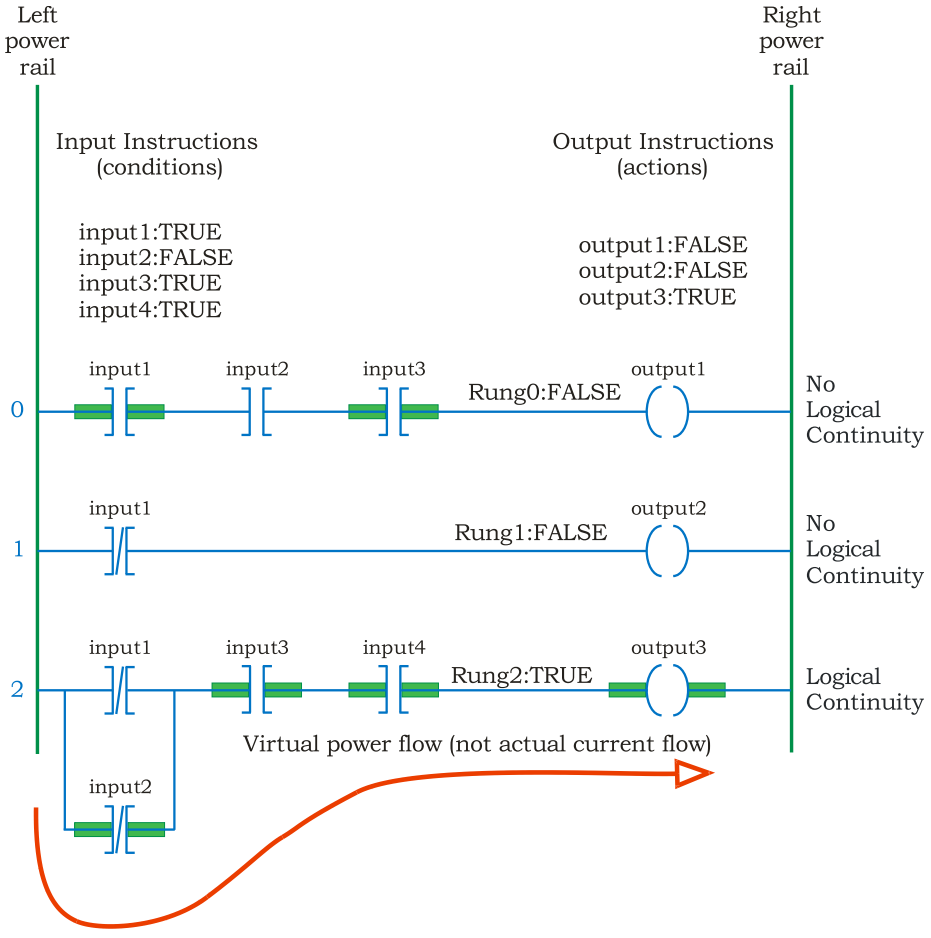


Figure 15 Sample **LL** program

Let us examine a simple PLC input circuit consisting of a power supply and a switches. Electrical continuity in an input circuit occurs when there is a complete path for the current to flow. If an input is active (i.e., there is electrical continuity), the corresponding bit in the input image table will be set to a **1**. If there is no electrical continuity, the bit is reset to a **0**. In the input scan phase the PLC copies the status of ALL of the input terminals to the input image table (Figure 16).

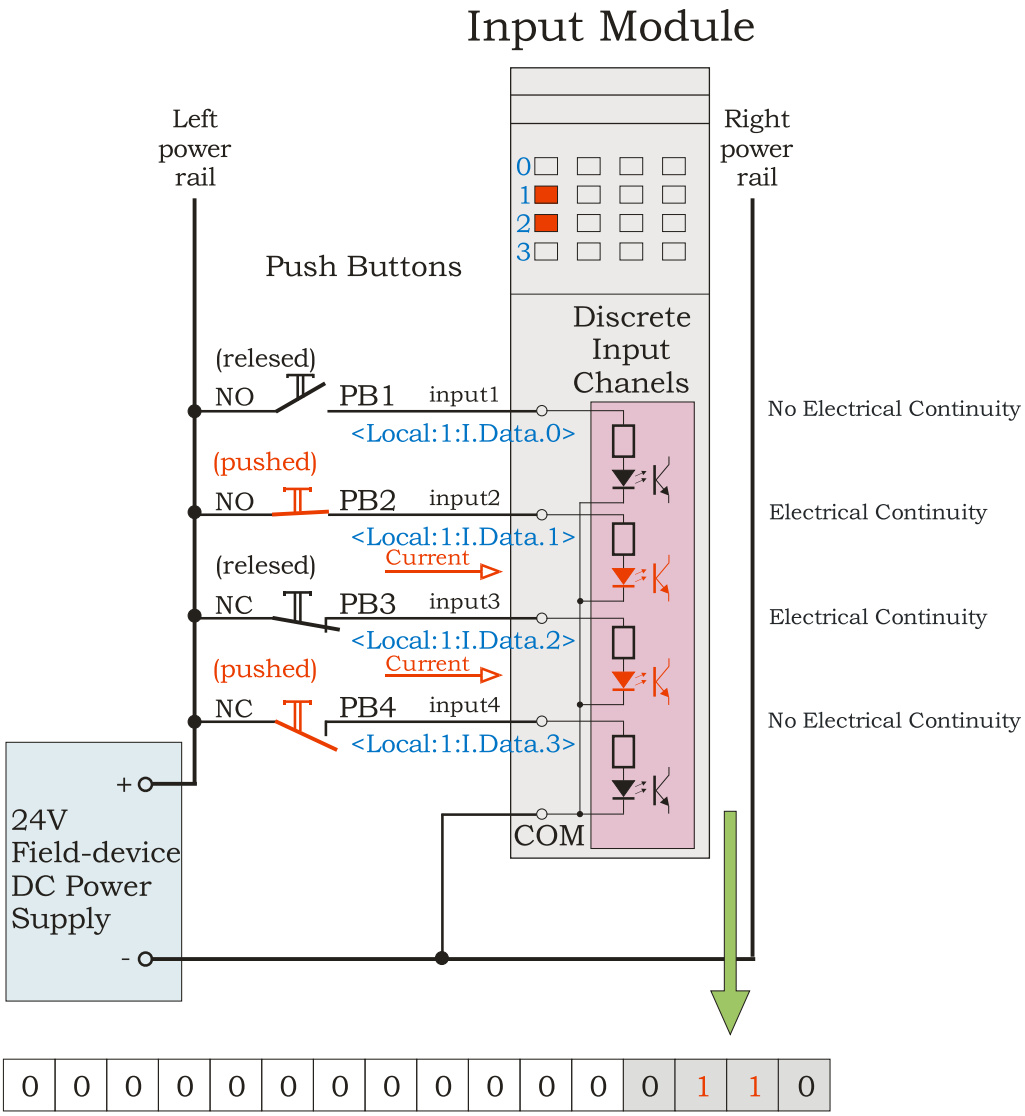


Figure 16 Scan input phase

The PLC program logically connects the input devices to the output actuators. In the execute program phase the PLC executes the program using the values in the input image table to update the output image table (Figure 17).

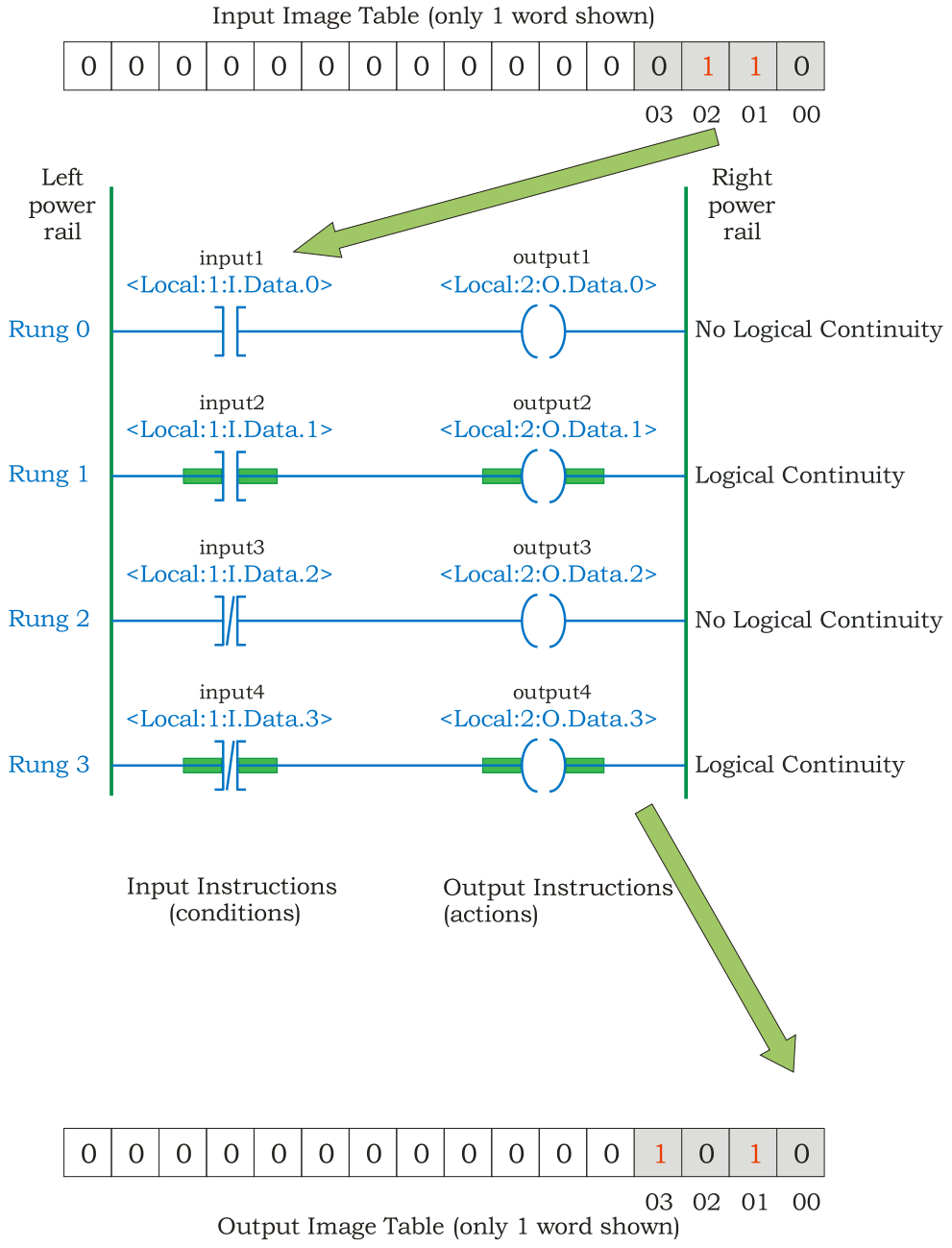


Figure 17 Execute program phase

Every discrete output is assigned to a specific bit in the PLC's memory (output image table). In order for an output to turn on, its associated bit must first be set to **1**. In the update outputs phase the PLC copies the output image table status to the ALL of the output terminals (discrete output circuits) (Figure 18).

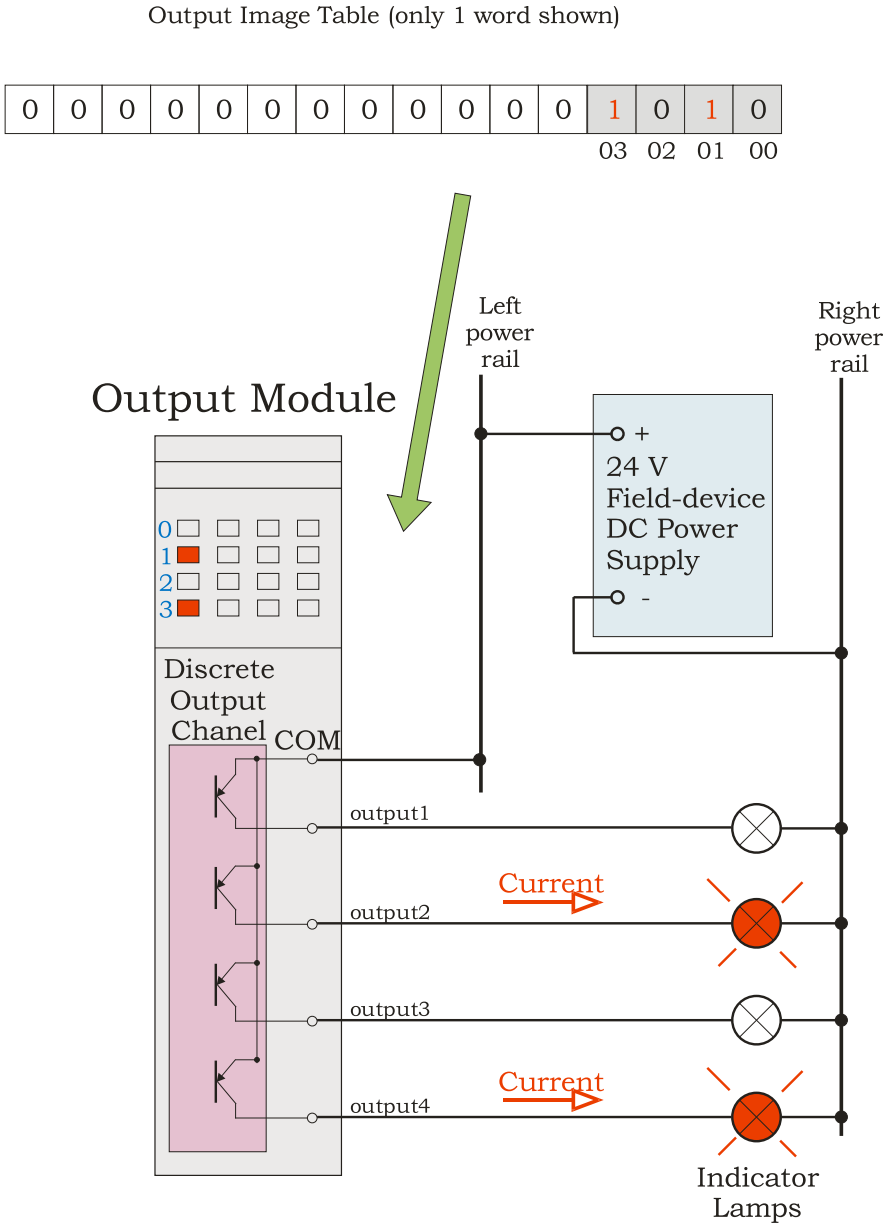


Figure 18 Update outputs phase

1.5. PLC operation modes

PLC-s has 2 types of operation modes. Depending what we want to do we must set the PLC in the appropriate mode. The modes can be set on the PLC with a switch or from the programming software. These modes are:

- **Program**: In this mode you can upload the program to the PLC. The program will not run.
- **Run**: In this mode the program is running. The PLC cannot be programmed.

The PLC programming software also has operation modes. These modes are:

- **Online**: You need to be in this mode to upload, download a program or monitor the PLC.
- **Offline**: You need to be in this mode to write or modify a program.
- **Monitor**: In this mode you can watch the states of the PLC's inputs, outputs and memory. This is the mode you use for debugging.

2. PROGRAMMING WITH THE RSLOGIX 5000

2.1. Inputs, Outputs and Basic Operations

In this section we will learn how to set up and write a simple program and what are the most basic instructions and what they do. We will assume that you will know how to set up communications with the help of **RSLinx** and have already done so.

There are 2 input and 3 output instructions you need to know to start plc programming.

- Examine If Open: input instruction. Is true when memory address is false.
- Examine If Closed: input instruction. Is true when memory address is true.
- Output Energize: output instruction. Sets the memory address to true if the condition is true, false if the condition is false.
- Output Latch: output instruction. Sets the memory address to true if the condition is true.
- Output Unlatch: output instruction. Sets the memory address false if the condition is true.

With these 5 instructions you can create fairly complex programs. In the next segment we will see examples for the use of these instructions.

2.1.1. Setting Up and Writing Our First Program

First start the **RSLogix 5000** program. Once it loads create a new program by clicking **New** (Figure 19). A new window will pop up (Figure 20), here select the PLC type, the firmware version and type in the program name. Once you are done click **OK**. **RSLogix 5000** will now create the basic structure of the program this might take a while. Once it is done select the path to the PLC by clicking **RSWho** (Figure 21) and selecting the PLC you wish to program. When the path is selected we can start writing the program by opening the **Main Routine** under the **Main Program** tag (Figure 22).

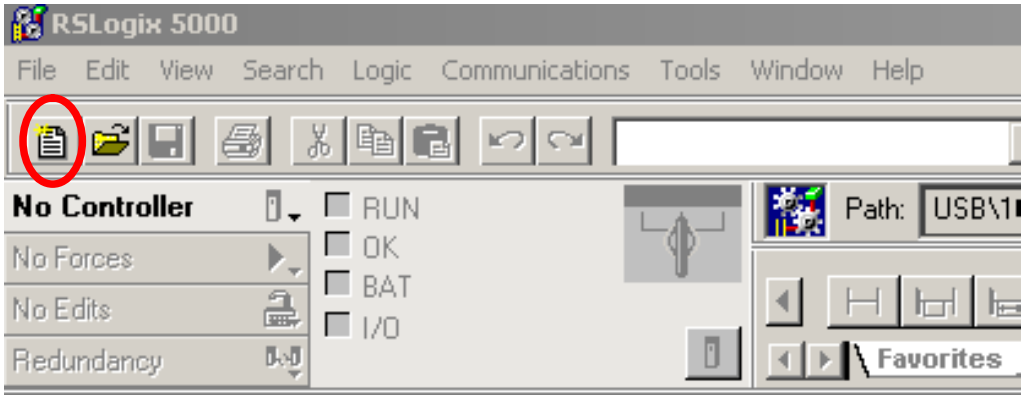


Figure 19 Create a new program

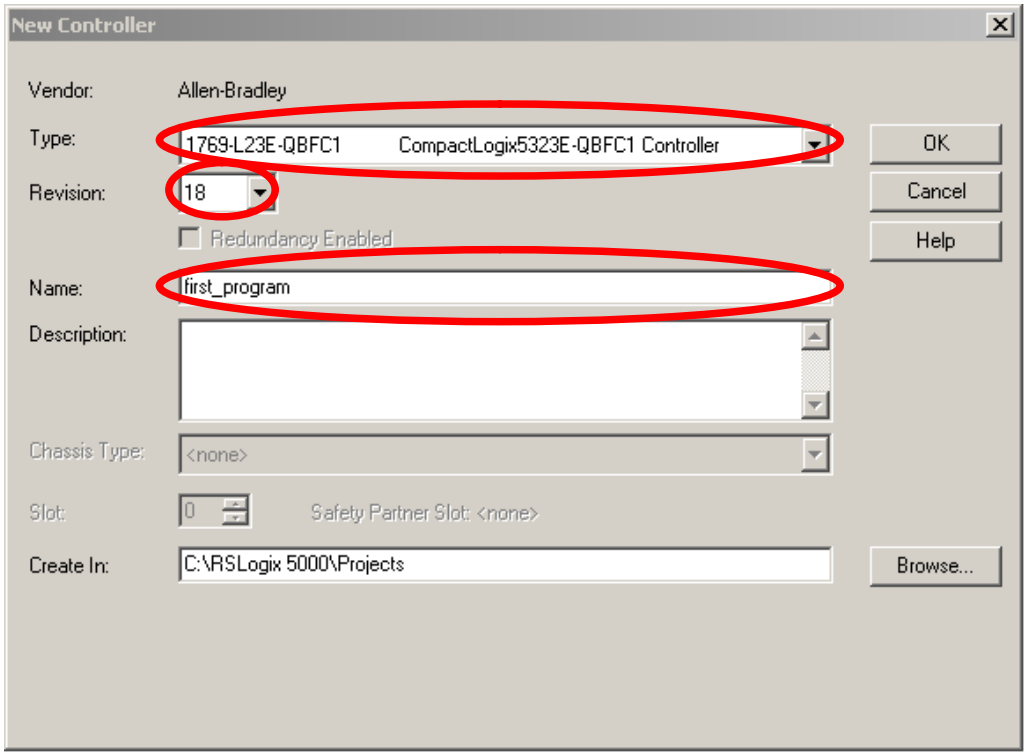


Figure 20 Select PLC type and name the program

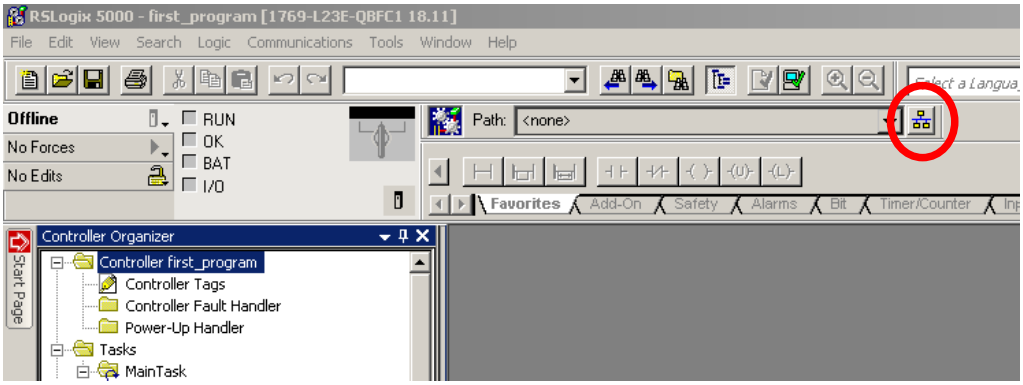


Figure 21 Select the path to the PLC

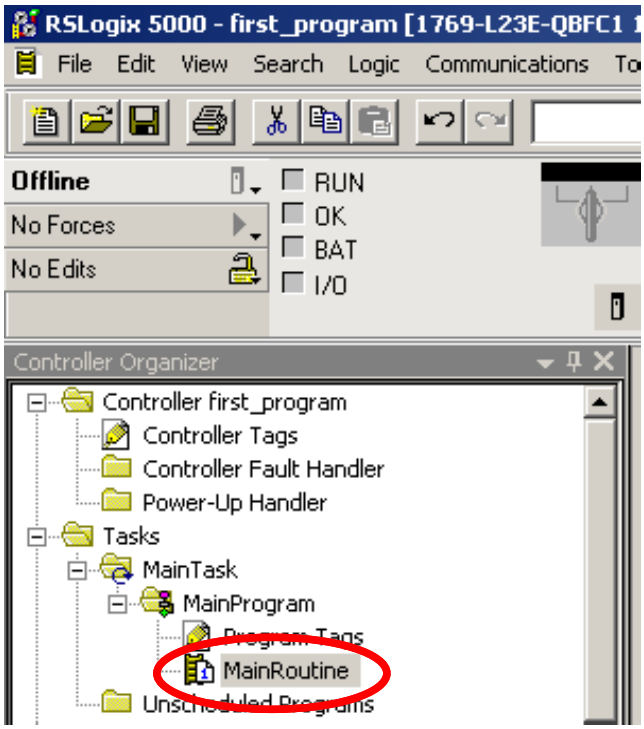


Figure 22 Open the **MainRoutine**

To write a PLC program simply pull the instructions from the instruction bar onto the chosen place on the rung. Each line has to have at least one output instruction (Figure 23) these are at the right side of the rungs. Once you place an input or output instruction you have to assign a **tag** to it. This

tag will define what is connected to it this can be a physical input or output. It can also be a bit in memory or a number among many things. We will go into these in more detail later, but first place the output energize instruction at the end of the first rung. Once it is there right click on the question mark above it and select **New Tag** (Figure 24) or press **ctrl+w** (Figure 25). A new window pops up, here we can give a name, a type and select if this will be a **base** tag or an **alias** for something. To connect this output to a physical output select alias and select the physical output you wish to activate. If this value will not have a physical connection select **base**. Right now we want this to be the first physical output so we will select that (Figure 26) and name it "**output1**". Remember as in most programming languages the first is labeled with 0. Once you have done this the program is ready to be downloaded to the PLC, you can also verify the program to see if it contains semantic errors (Figure 27).

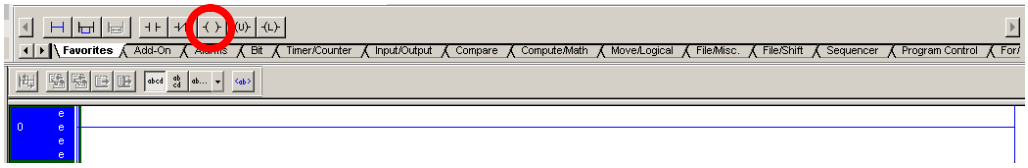


Figure 23 Put the output energize instruction at the end of the first rung.

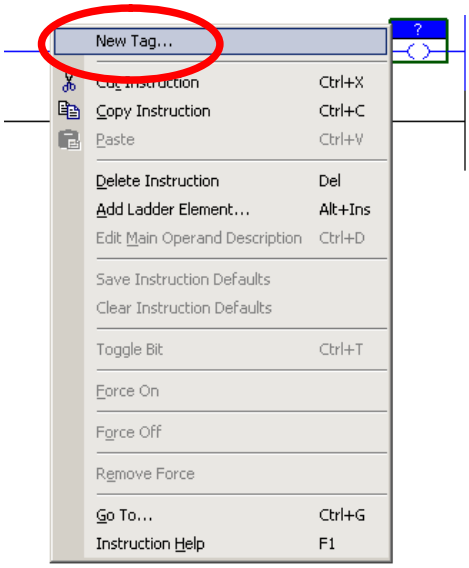


Figure 24 Create a **New Tag**

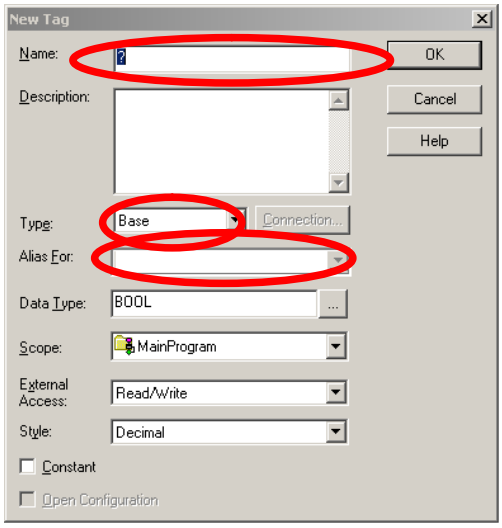


Figure 25 Name the tag and select its type

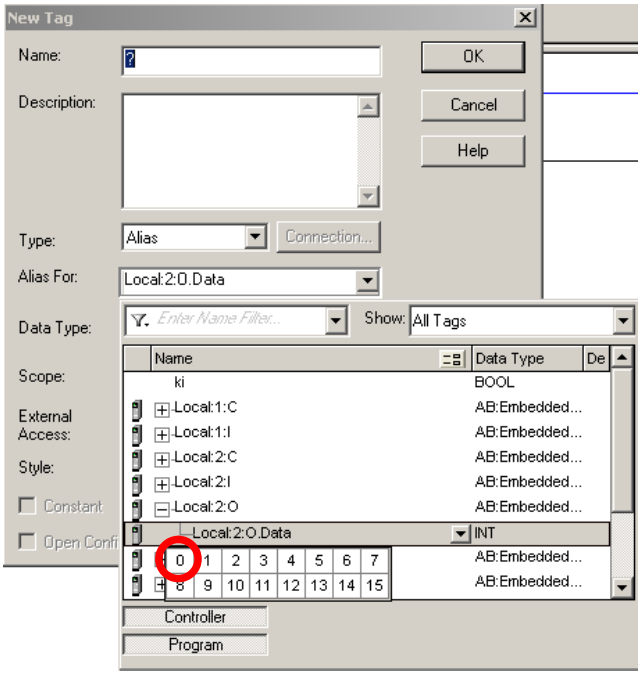


Figure 26 Select what the tag is an **alias** for

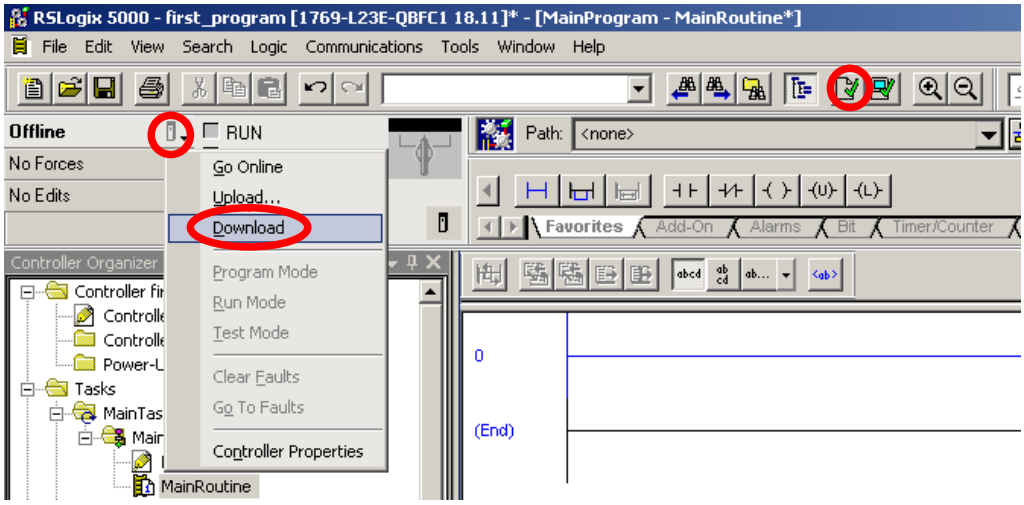


Figure 27 Verify and download the program

Once you press download a series of prompts will appear asking you if you want to change modes. Change to **RUN** mode at the end and select **Go Online** this will enable you to see the program in action and is your main tool for debugging the programs you write. However you can only modify the program when it is **Offline**. Next let's add a condition to the beginning of the line. No conditions for a branch is the same as always true it will always execute. The first condition we will look at is Examine If Closed (**XIC**) (Figure 28). This will be true if the tag assigned to it holds the value of **true** otherwise it is **false**. Drag the symbol to the beginning of the rung (Figure 29).

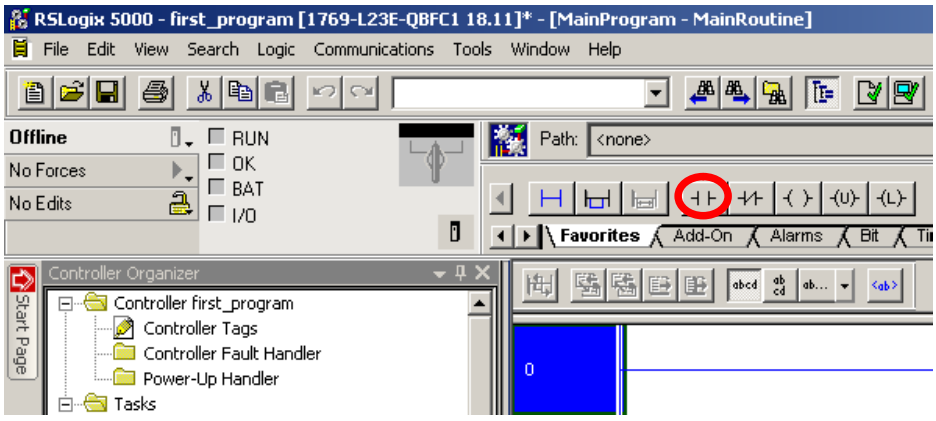


Figure 28 Examine If Closed

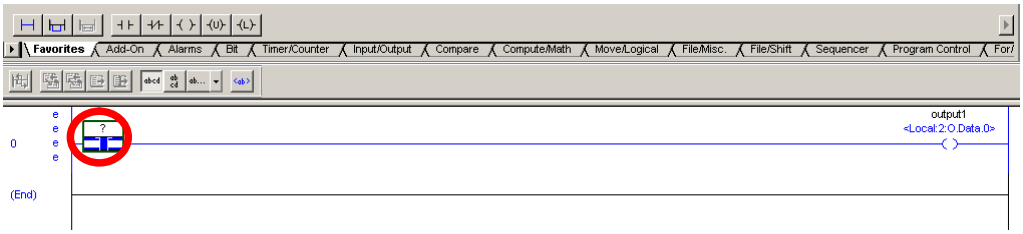


Figure 29 Place Examine If Closed

Finally add a **tag** to the condition and set it to be the **alias** for the first input similarly as you did with the output (Figure 30).

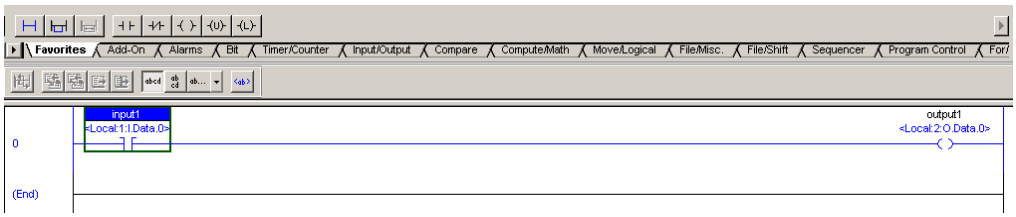


Figure 30 Add a **tag** to Examine If Closed, the condition is not green indicating that it is false

Once you have downloaded the program to the PLC and set it to **RUN** mode. The controller will work according to the truth table in (table 2).

Table 2 The truth table for the program

input1	output1
0	0
1	1

The next condition we will look at is Examine If Open (**XIO**) (Figure 31). This will be true if the tag assigned to it holds the value of false otherwise it is false. Drag the symbol to the beginning of the rung in place of XIC (Figure 32). Finally assign the same tag of input1 to it (Figure 33).

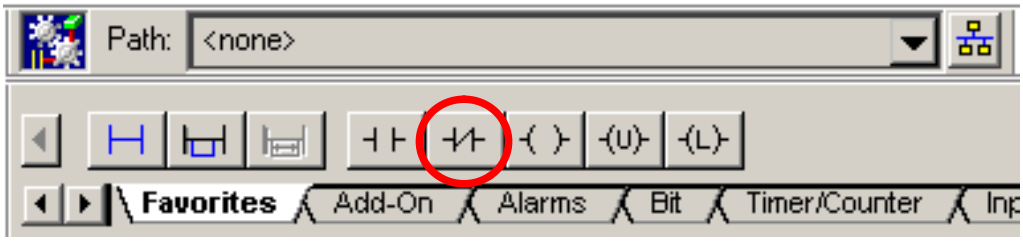


Figure 31 Examine If Open

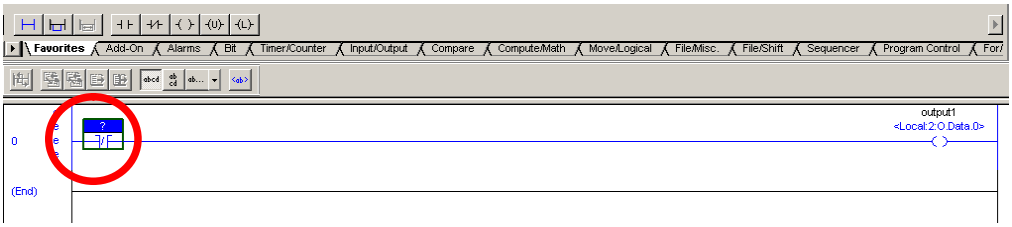


Figure 32 Place Examine If Open

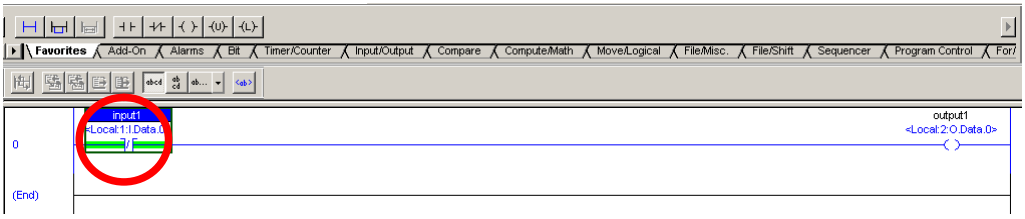


Figure 33 Add a tag to Examine If Open, the condition is green indicating that it is true

Once you have downloaded the program to the PLC and set it to **RUN**. The controller will work according to the truth table in (table 3).

Table 3 The truth table for the program

input1	output1
0	1
1	0

The Figure 34 shows a review how inputs (XIO and XIC) and outputs (OTE) respond during a scan.

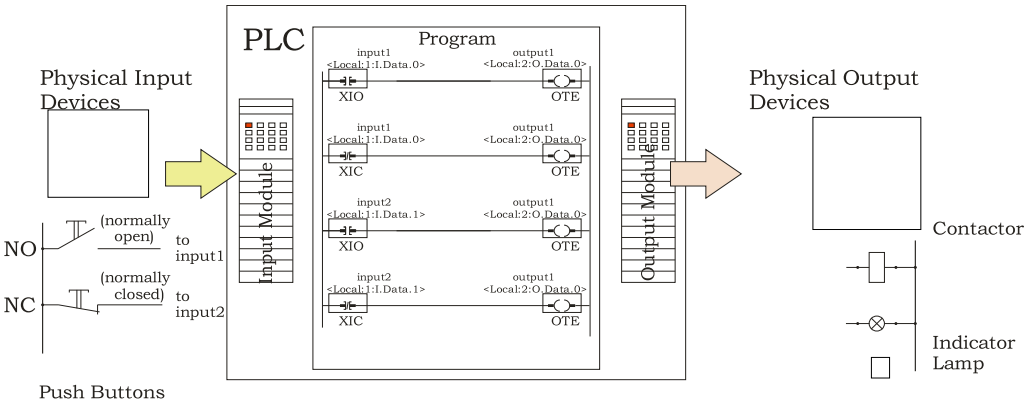


Figure 34 Review how inputs (XIO and XIC) and outputs (OTE) respond during a scan.

The next condition we will look at is a logical **AND**. To create a logical **AND** between two or more conditions simply place them in series (Figure 35). As with the logical **AND** when all the conditions are evaluated true (all of them are green) the output instruction(s) will activate. Table 4 shows the truth table for the program on Figure 35.

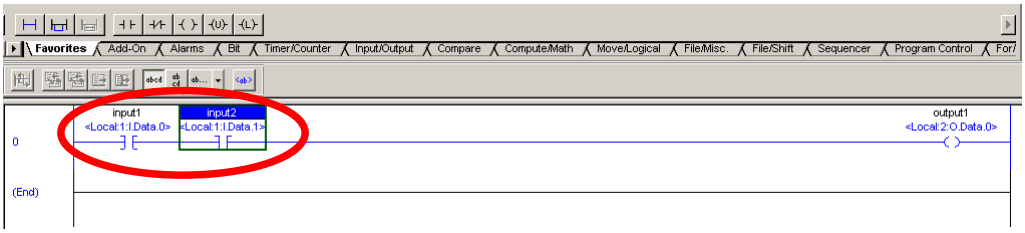


Figure 35 A logical **AND** condition between two conditions

Table 4 The truth table for the program

input1	input2	output1
0	0	0
0	1	0
1	0	0
1	1	1

The next condition we will look at is a logical **OR**. To create a logical **OR** between two or more conditions simply place them in parallel using the branch instruction (Figure 36). Place a branch instruction on the rung (Figure 37) and place the conditions on the branches (Figure 38). As with the logical **OR** when at least one of the conditions is evaluated **true** (one of them is green) the output instruction(s) will activate. Table 5 shows the truth table for the program on Figure 38.

Figure 36 The **branch** instruction

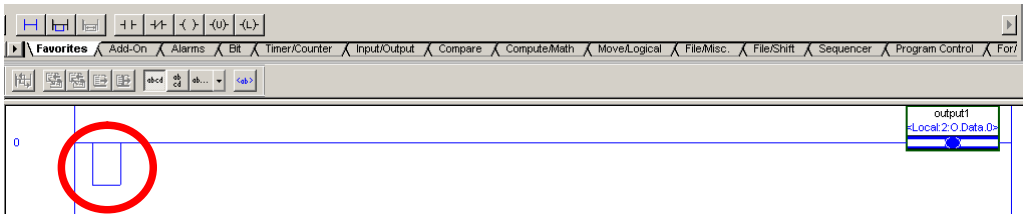


Figure 37 Place the **branch** instruction

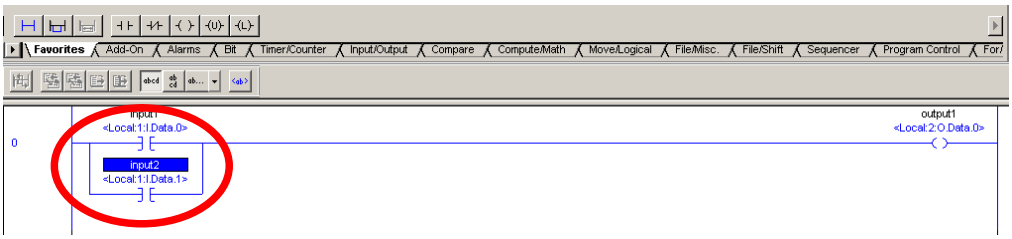


Figure 38 A logical **OR** condition between two conditions

Table 5 The truth table for the program

input1	input2	output1
0	0	0
0	1	1
1	0	1
1	1	1

Using the same **branch** instruction you can activate more than one output instructions at the same time.

This brings us to one of the most important circuits in automation the self holding circuit (Figure 39).

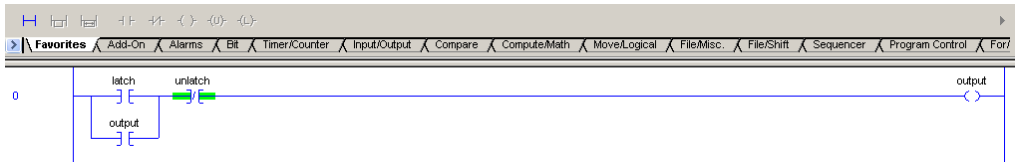


Figure 39 The self holding circuit

The self holding circuit is a circuit that has 2 inputs, feedback and an output. Since the PLC works in cycles the feedback is the previous state of the output. **Latch** sets the output to **true** and the feedback ensures that the output stays **true**. **Unlatch** resets the output to **false** by cutting the off the flow from the latch and the feedback. Table 5 shows the truth table for the self holding circuit.

Table 6 The truth table for the self holding circuit

Latch	Unlatch	Current output state	New output state
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

This configuration is used so much that there are two output instructions that save time in its implementation. These are called Output Latch (**OTL**) and Output Unlatch (**OTU**).

These are the basic conditions and configurations, there are many more conditions and output instructions that we will get into later.

2.1.2. Memory Bits

As with most PLCs there are some **tags** that serve a special purpose, these tags do not turn green even when they are active. The main one is the first scan flag, which is active for only the first cycle after you switch the PLC to run mode. Here is a list of these flags.

- **S:FS**

  The **first scan** of the program

- **S:MINOR**

This is set to true when the PLC has a minor fault

- **S:V**

  This is set true when the value you are trying to store cannot fit into its destination because it is eider greater than the maximum storable value or smaller than the minimum storable value. Each time this happens the PLC also sets **S:MINOR** true.

- **S:Z**

  This is set true when the instructions destination value is **0**.

- **S:N**

  This is set true when the instructions destination value is **negative**.

- **S:C**

  This is set true when an arithmetic instruction causes carry or a borrow outside of the data type.

2.1.3. Exercise I: Garage Door

Let's examine a real world problem namely a garage door system. On Figure 40 we see the system in question. Figure 41 shows the solution.

Figure 40 Garage door system

The inputs are shown in table 7.

Table 7 Inputs of the garage door system

Inputs (switching elements)	Name	Type	Identifier
Pushbutton	Open	NO	Local:1:I.Data.0
Pushbutton	Close	NO	Local:1:I.Data.1
Pushbutton	Stop	NC	Local:1:I.Data.2
Limit switch	LS1	NC	Local:1:I.Data.3
Limit switch	LS2	NC	Local:1:I.Data.4

The outputs are shown in table 8:

Table 8 Outputs of the garage door system

Output devices	Name	Identifier
Contactor	Motor UP	Local:2:O.Data.0
Contactor	Motor DOWN	Local:2:O.Data.1
Indicator lamp	AJAR	Local:2:O.Data.2
Indicator lamp	OPEN	Local:2:O.Data.3
Indicator lamp	SHUT	Local:2:O.Data.4

The task we are charged with is the following:

- Once you press the **OPEN** button the door will go up until **LS1** signals that the top has been reached.

- Once you press the **CLOSE** button the door will go down until **LS2** signals that the bottom has been reached.
- You do not have to keep pressing the buttons to keep the selected action active.
- Pressing the **STOP** button halts the current action.
- The **Motor UP** switch and **Motor DOWN** switch can't be active at the same time.
- The lamps signal the current state. **OPEN** when the door is fully open, **SHUT** when the door is fully shut and **AJAR** when the door is not moving in eider direction and it is not in eider end position.

Figure 41 Garage door system PLC program

When you are writing PLC programs it helps to follow this simple convention. Each line should have 1 output instruction that controls a physical output. When using the Output Energize (**OTE**) instruction do not use the same **tag** more than once in a program. Since the last one will overwrite the previous ones. All conditions that influence this output should be used as conditions for it. This may look difficult at first, but used properly they make programs clearer to see through.

Line 0 is responsible for opening the door. It is a self holding configuration. Thus if the output is activated in a cycle it will be activated in the next one. To deactivate this output you need at least one condition in series that can break the condition. Here we have 3 the **Stop** button, **LS1** and the **Motor**

DOWN. The **Stop** button is NC this means it is always active except when pressed. When you it is pressed the output is made false. The same thing happens with **LS1**. The **Motor DOWN** is a bit different. It is there to make sure **Motor UP** can't be activated if **Motor down** is active. This is called cross latching.

Line 1 is very similar to line 0. Line 2 activates the **OPEN** lamp when **LS1** gives a signal. Line 3 activates the **SHUT** lamp when **LS2** gives a signal. Line 4 activates the **AJAR** lamp when the motor is not going up and it's not going down and **LS1** is not active and **LS2** is not active. In other words when the motor has stopped and is not in the end positions.

2.1.4. Exercise II: Silo

Figure 42 shows the silo system. Figure 43 shows the solution.

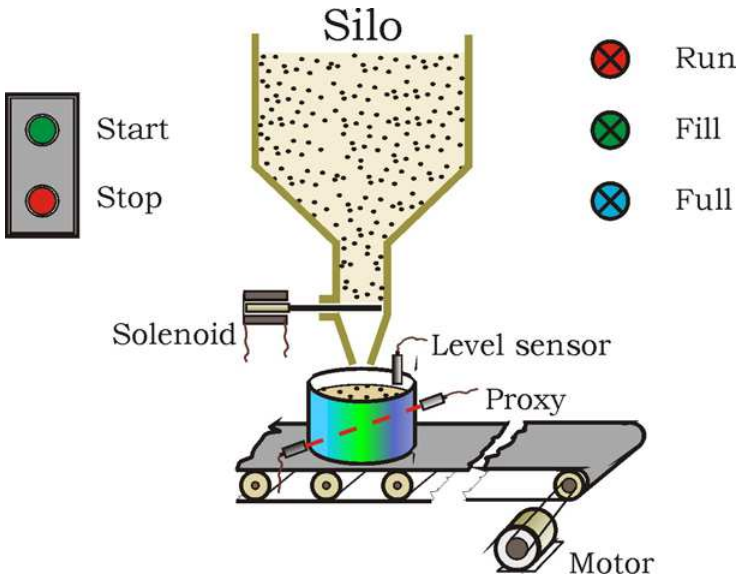


Figure 42 Silo system

The inputs are shown in table 9.

Table 9 Inputs of the silo system

Inputs (switching elements)	Name	Type	Identifier
Pushbutton	Start	NO	Local:1:I.Data.0
Pushbutton	Stop	NC	Local:1:I.Data.1
Photo swich	Proxy	NO	Local:1:I.Data.2
Level sensor	LS	NO	Local:1:I.Data.3

The outputs are shown in table 10.

Table 10 Outputs of the silo system

Output devices	Name	Identifier
Contactor	Motor	Local:2:O.Data.0
Contactor	Solenoid	Local:2:O.Data.1
Indicator lamp	Run	Local:2:O.Data.2
Indicator lamp	Fill	Local:2:O.Data.3
Indicator lamp	Full	Local:2:O.Data.4

The task we are charged with is the following:

- When you press the **Start** button the **Run** lamp turns on and the conveyor starts.
- When the box reaches the **Proxy** sensor the conveyor stops.

- When the box stops the **Solenoid** activates and starts filling the box.
- While the box is being filled the **Fill** lamp turns on.
- When **LS** (level sensor) indicates the box is full deactivate the **Solenoid** to stop the filling.
- While the level sensor indicates the box is full the full lamp turns on.
- When the box is full the conveyor starts again and brings the next box and the process starts again.
- When you press the **Stop** button the process stops. When you press the **Start** button the process continues where it was stopped.

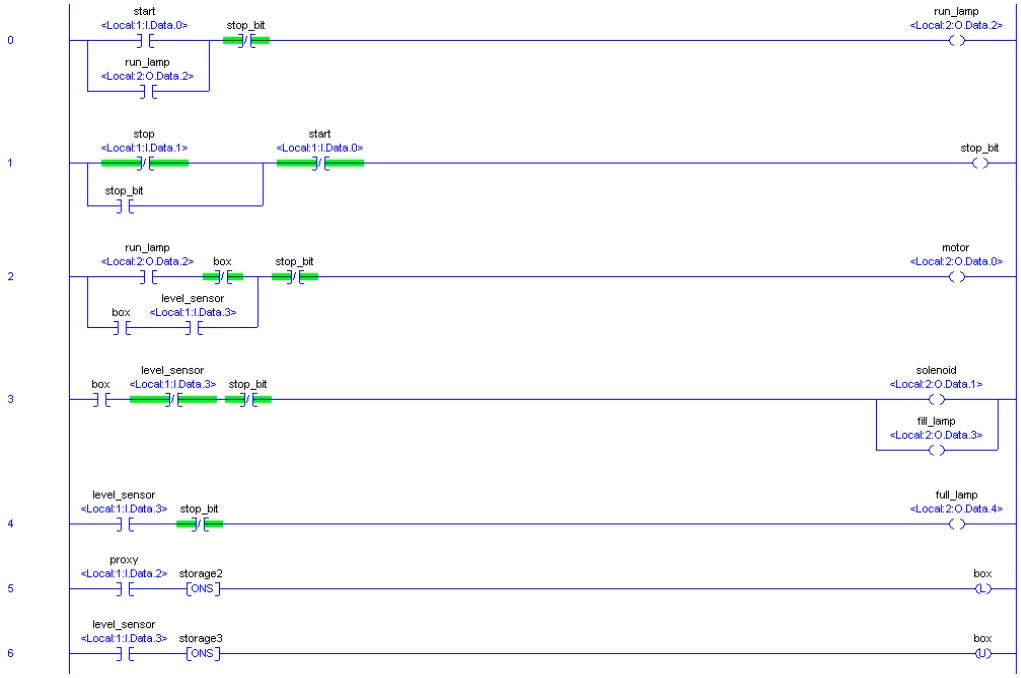


Figure 43 Silo system PLC program

In this program there are some instructions that are new to us let's examine these. Line 1, 5 and 6 contain the instruction **One Shot** (**ONS**). This instruction detects the rising edge of the conditions before it and needs a storage bit. To detect the falling edge of a condition simply negate it. Since it detects rising edges its output instruction will only activate for one cycle. Line 5 contains the output instruction output latch (**OTL**). It works similar

to output energize (**OTE**) except once activated it sets the output high and keeps it high. Line 6 contains the complement to this output unlatch (**OTU**). When the condition to output unlatch become true it sets the output low. These two work similarly to a self holding circuit.

Let's analyze the program:

Line 0 controls the **Run** lamp. When the **Start** button is pressed the **Run** lamp becomes active. When the **stop bit** is active it becomes inactive.

Line 1 controls the **stop_bit**. The **stop_bit** is activated by the **Stop** button. The **Start** button deactivates the **stop_bit**. The **stop_bit** is used in lines 0, 2 and 3 to disable those outputs when it is active.

Line 2 controls the **Motor**. Here we use a bit labeled box. When we are running and there is no box and when there is a box and it is full the **Motor** for the conveyor will run. The **stop_bit** deactivates this output.

Line 3 controls the **Solenoid** valve and the **Fill** lamp. These have to work at the same time. They turn on when there is a box and the level sensor indicates it is not full. The **stop_bit** deactivates this output.

Line 4 controls the **Full** lamp. It becomes active when the **Level sensor** indicates the box is full. The **stop_bit** deactivates this output.

Line 5 latches the bit that indicates there is a box. The rising edge of the **Proxy** sensor latches the box bit. This is not deactivated by the **stop_bit**. We need it unchanged to keep our position when we restart the machine.

Line 6 unlatches the bit that indicates there is a box. The falling edge of the **Level sensor** unlatches the **box bit**.

2.2. It's PLC Time!

With the instructions we have learned till now we can deal with something instantly, but we can't do anything that is time related. To handle events that require timing the PLC uses timers. Timers have a time base which in most cases is millisecond based. The smallest time it can measure is 1 ms. Some PLCs have timers that have a time base of 10, 100, 1000 ms. In this section we will learn about the different types of timers and how they are used.

2.2.1. Types of Timers and Their Uses

There are 3 types of timers you can use in **RSLogix 5000 Timer On Delay** (**TON**), **Timer Off Delay** (**TOF**), **Retentive Timer On Delay** (**RTO**). Each

timer has a **tag**, **preset** value (**PRE**) and **accumulator** (ACC). The **preset** value holds the value in milliseconds for how long the timer should wait. The **accumulator** holds the value in milliseconds for which the timer was active. The timer also has bits that indicate its state. These are the **enable bit** (**EN**), **done bit** (**DN**), **timing bit** (**TT**).

Timer On Delay (**TON**). The operation of a timer on delay is depicted on Figure 44. The timer delays the rising edge of the input signal by the preset value, if the input signal is active for the preset time. The enable bit (**EN**) is active while the input signal is active. The timer timing bit (**TT**) is active while the timer is active and has not reached the preset value. The done bit (**DN**) is active while the input signal is active and the timer reached the preset value. Once the input signal is gone the timer resets.

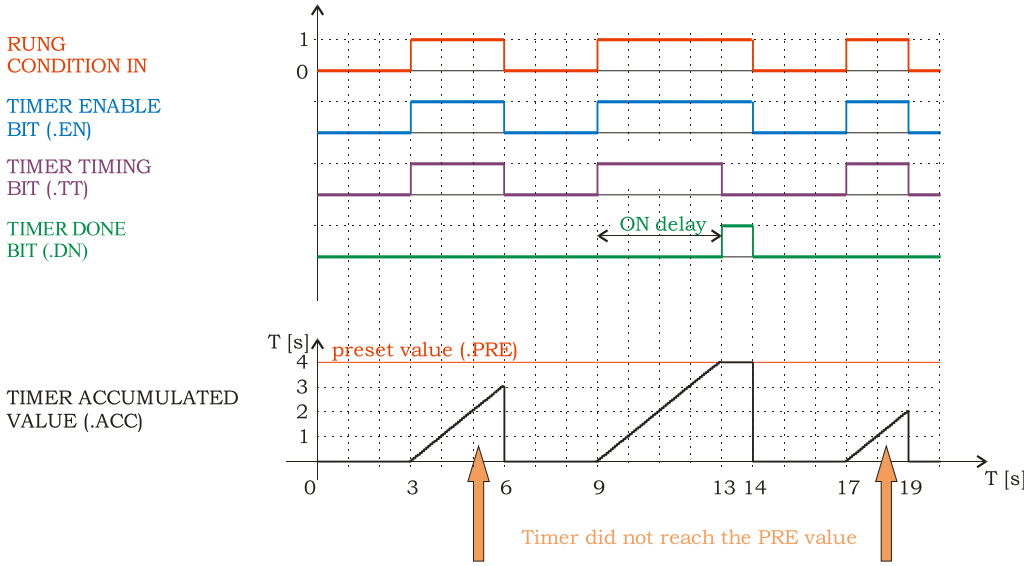


Figure 44 Timer on delay

Timer Off Delay (**TOF**). The operation of a timer on delay is depicted on Figure 45. The timer delays the falling edge of the input signal by the preset value, if the input signal is inactive for the preset time. The enable bit (**EN**) is active while the input signal is active. The timer timing bit (**TT**) is active while the timer is active and has not reached the **preset** value. The done bit (**DN**) is active while the timer has not reached the **preset** value. Once the input signal returns the timer resets.

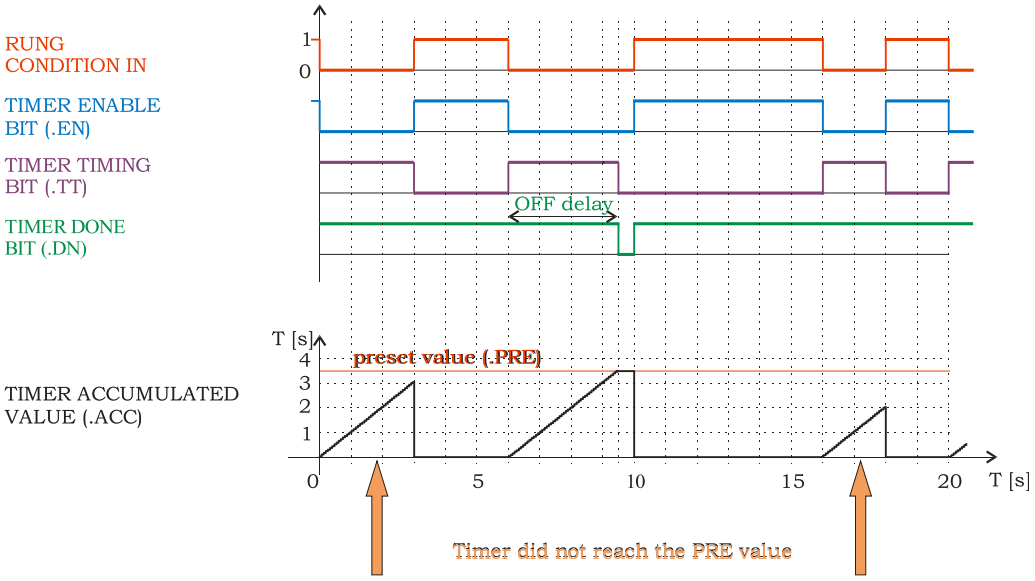


Figure 45 Timer off delay

Retentive Timer On Delay (**RTO**). The operation of a timer on delay is depicted on Figure 46. The timer delays the rising edge of the input signal by the preset value, if the input signal is active for the preset time. The enable bit (**EN**) is active while the input signal is active. The timer timing bit (**TT**) is active while the timer is active and has not reached the **preset** value. The done bit (**DN**) is active while the input signal is active and the timer reached the **preset** value. Once the input signal is gone the timer does not reset.

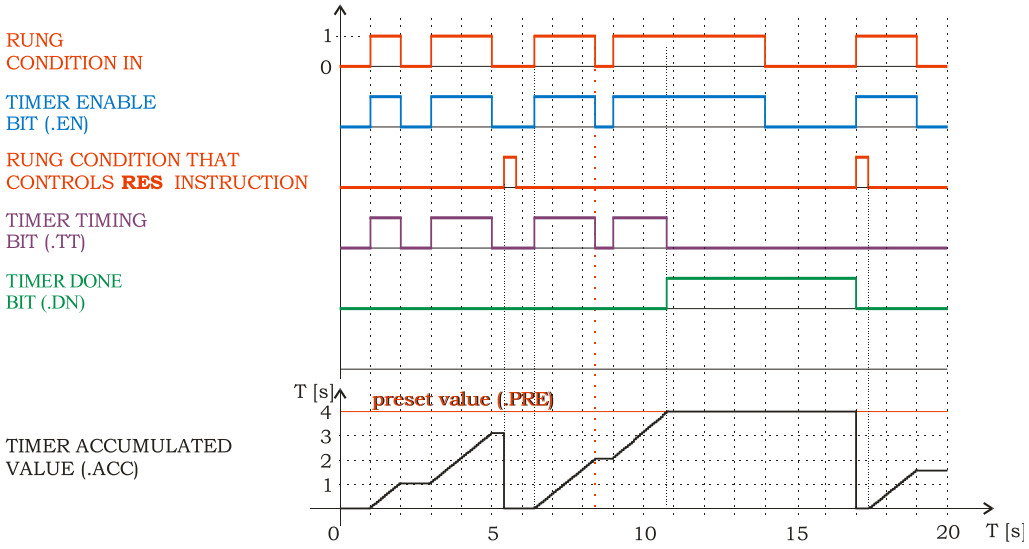


Figure 46 Retentive timer on delay

Next let's see how to use timers. Timers can be found under the **Timer/Counter** tab of the instruction bar (Figure 47). We will examine the other instructions later. Now add the **TON** instruction to the rung. Figure 48 shows the field for the **tag**. First add a **tag** to the timer. Next set the **preset time** (Figure 49).

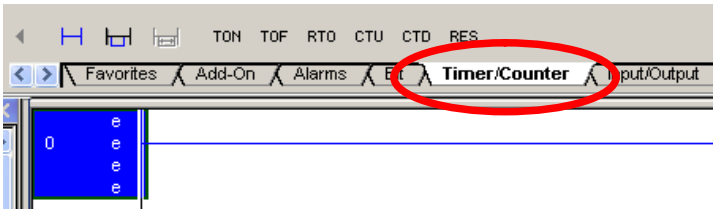


Figure 47 The Timer/Counter tab

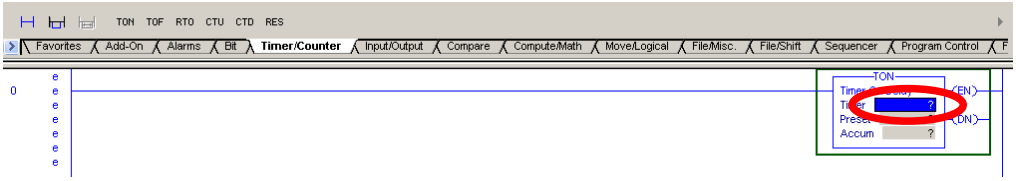


Figure 48 Set the timer tag

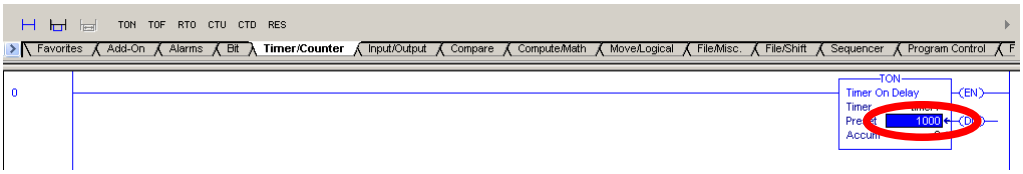


Figure 49 Set the preset time

Let's add a condition to this line that will reset the timer once the preset time is reached (Figure 50).

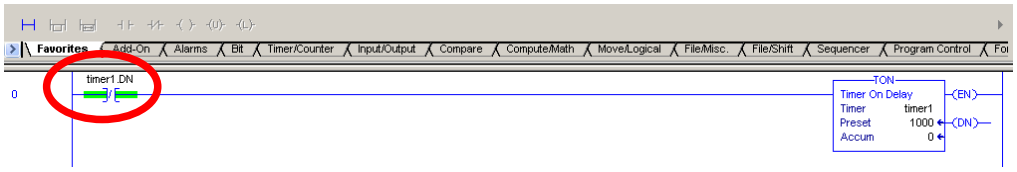


Figure 50 Add reset condition for the timer

To access the status bits and values of the timer simply use the tag name add a dot "." and the field name {**PRE**, **ACC**, **EN**, **DN**, **TT**}. Make sure that the field you use is applicable to the instruction. You can't use {**PRE**, **ACC**} on bit instructions. A common application of timers is the generation of a square wave. Figure 51 shows a program that generates a square wave with a 50% duty cycle and a period of 1s. The program works as follows. Timer starts timing since its condition (not **timer1.DN**)is true. Once timer reaches the preset time **timer.DN** becomes active. **Timer.DN** starts **timer1**. Once **timer1** reaches the preset time **timer1.dn** becomes active. This resets both **timer** and **timer1**. The process repeats endlessly.

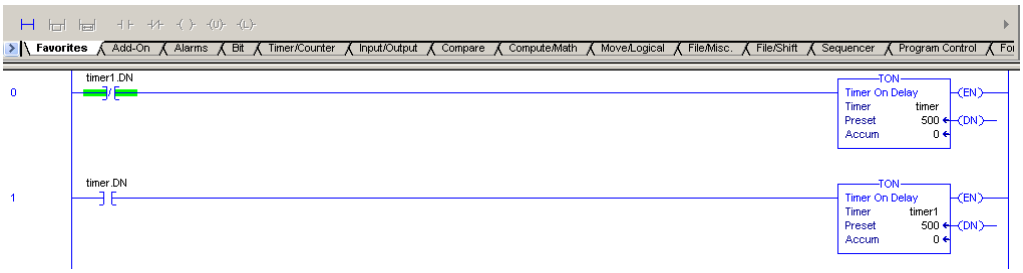


Figure 51 Square wave generating program

2.2.2. Exercise III: Traffic Control

In this exercise you will have to program a traffic light at a crossroad with separate lights for the pedestrians. Figure 52 shows a time diagram of how the lights at the crossing are supposed to work. The lights for the pedestrians turn on the next green light for the appropriate direction after its button has been pressed. Figure 53 shows the traffic control system. Figure 54 and Figure 55 show the complete program.

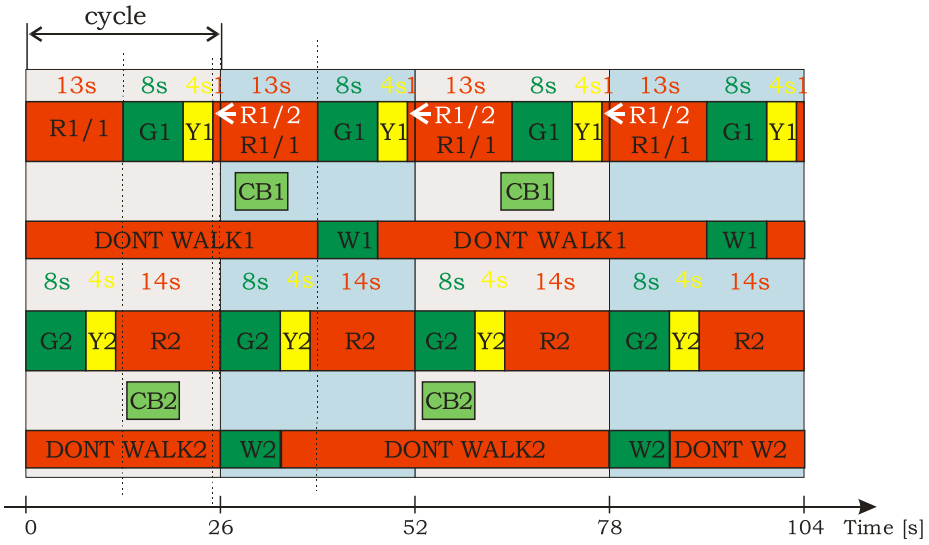


Figure 52 Time diagram for traffic control system

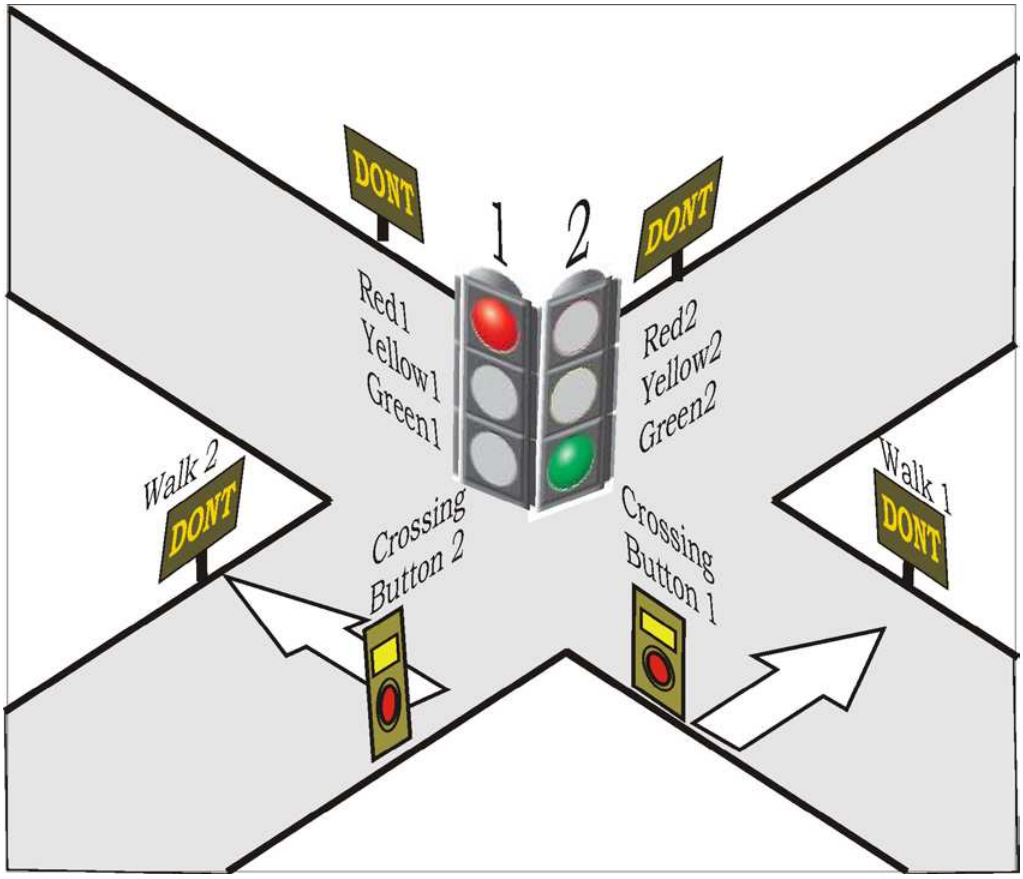


Figure 53 Traffic control system

The inputs are shown in table 11.

Table 11 Inputs of the traffic light system

Inputs (switching elements)	Name	Type	Identifier
Pushbutton	Crossing Button1	NO	Local:1:I.Data.0
Pushbutton	Crossing Button2	NO	Local:1:I.Data.1

The outputs are shown in table 12.

Table 12 Outputs of the traffic light system

Output devices	Name	Identifier
Traffic Light	Red1	Local:2:O.Data.0
Traffic Light	Yellow1	Local:2:O.Data.1
Traffic Light	Green1	Local:2:O.Data.2
Traffic Light	Red2	Local:2:O.Data.3
Traffic Light	Yellow2	Local:2:O.Data.4
Traffic Light	Green2	Local:2:O.Data.5
Traffic Light	Walk1	Local:2:O.Data.6
Traffic Light	Walk2	Local:2:O.Data.7

The task we are charged with is the following:

- **Red1**, **Red2**, **Yellow1**, **Yellow2**, **Green1**, **Green2** should follow the time diagram on Figure 52.
- **Walk1** should only be active after **Crossing Button1** was pressed.
- **Walk1** should be active the next time **Green1** is active for the duration that **Green1** is active.
- **Walk2** should only be active after **Crossing Button2** was pressed.
- **Walk2** should be active the next time **Green2** is active for the duration that **Green2** is active.

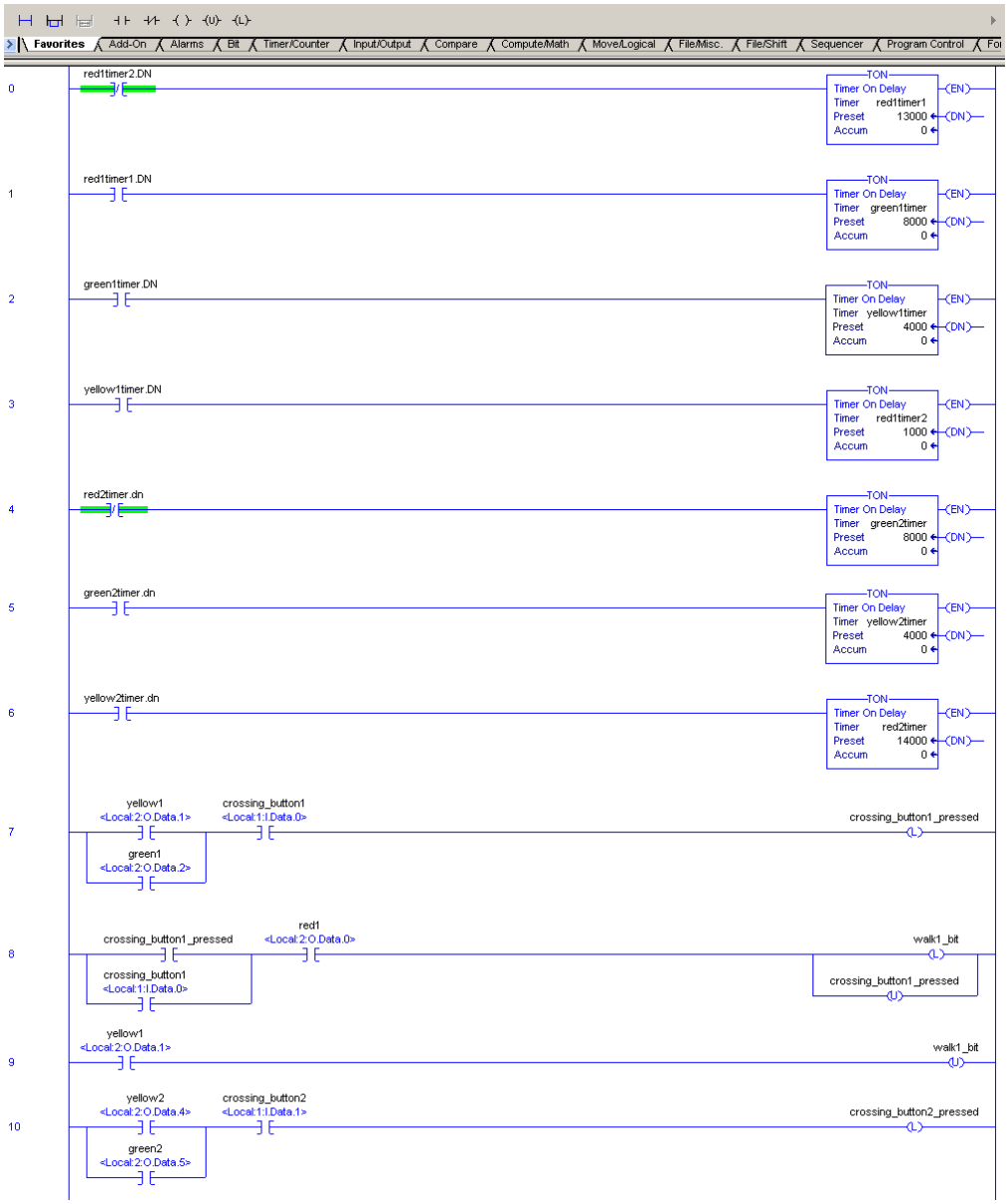


Figure 54 Traffic control system solution

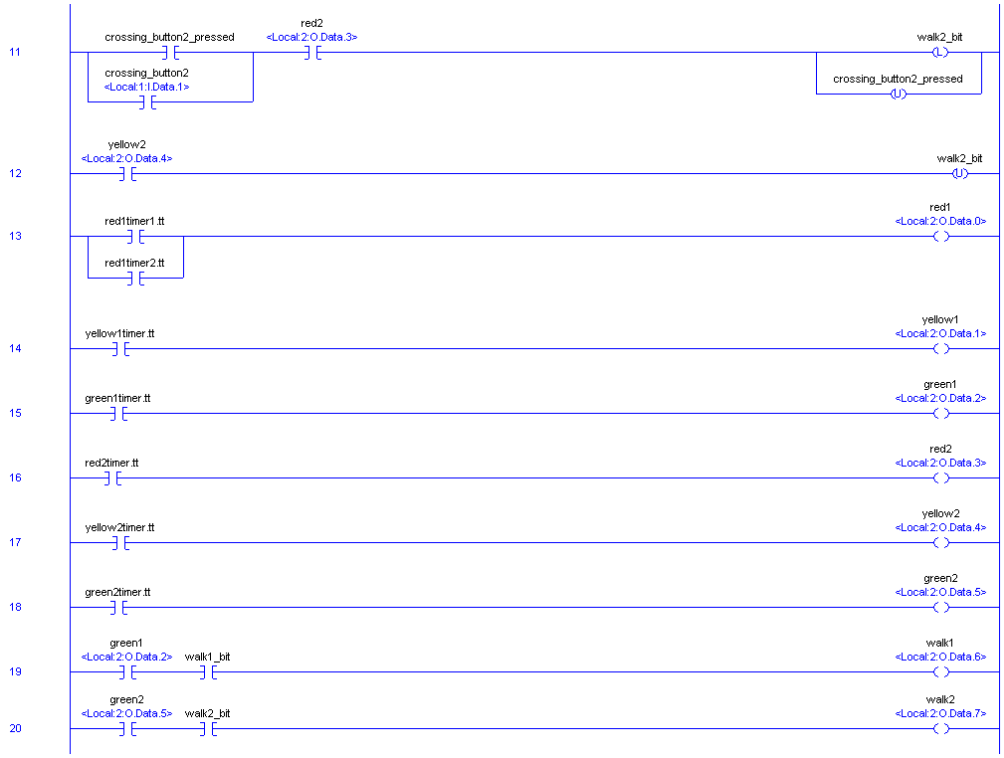


Figure 55 Traffic control system solution cont.

Let's analyze the program. Lines 0-6 has two sets of chained timers. The first timer chain is for the outputs **Red1**, **Yellow1**, **Green1**. The second timer chain is for the outputs **Red2**, **Yellow2**, **Green2**. These two chains are independent of one another.

Lines 7-9 hold the logic for **Crossing Button1**. We can press the **Crossing Button1** while the light is red, yellow or green. When it's yellow or green we set the **crossing_button1_pressed** bit. As the name suggests this bit stores the information that **Crossing Button1** was pressed. If we press the button while the light is red or we have already pressed the button before we set a bit called **walk1_bit** we also reset **crossing_button1_pressed**. The **walk1_bit** holds the information that we need to turn on the **Walk1** lamp the next time **Green1** is lit. Line 9 will reset the **walk1_bit** on the next **Yellow** lamp. We do this to make sure that once we press the crossing button the walk lamp will only stay active for one cycle.

Lines 10-12 are similar to lines 7-9. The only exception is they are for the **Walk2** lamp.

Lines 13-20 control the outputs. As we discussed it is good practice to leave all the outputs to the end and only work with bits till then.

Line 13 controls the **Red1** light. This can be active on the beginning or the end of the cycle. So we use an **OR** condition to check that the **red1timer2** is timing or the **red1timer2** is timing.

Lines 14-18 control the lights **Yellow1**, **Green1**, **Red2**, **Yellow2**, **Green2** respectively. These need to be active while their timers are timing.

Line 19 controls the **Walk1** light. This will be active when both the **Green1** light and the **walk1_bit** is active.

Line 20 controls the **Walk2** light. This will be active when both the **Green2** light and the **walk2_bit** is active.

2.3. Dare to Compare

As we have seen timers basically hold numbers. Until now we only used the **DN**, **EN**, **TT** bits of a timer. You can't use the **ACC**, **PRE** numbers in a ladder diagram since you need logical **TRUE** or **FALSE** values. To use these values first we need to convert them. This is where comparison operations come in.

2.3.1. Comparison Operators

The comparison operations are conditions and are used the same way. They can be found under the **Compare** tab. These operations are:

- **EQU** - equal (Figure 56) is **TRUE** if source **A** and source **B** have the same value. Otherwise it is **FALSE**.

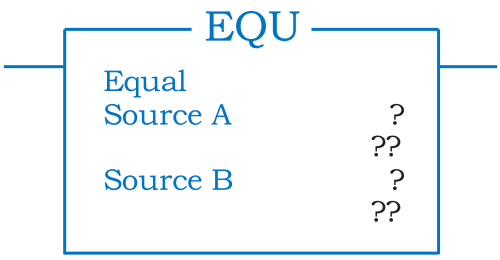


Figure 56 EQU instruction

- **NEQ** - not equal (Figure 57) is **FALSE** if source **A** and source **B** have the same value. Otherwise it is **TRUE**.

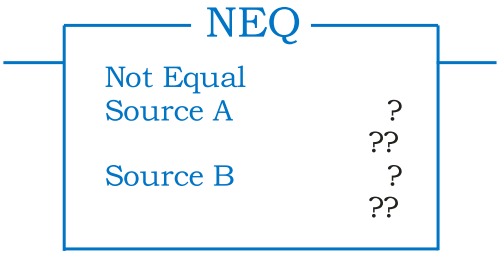


Figure 57 NEQ instruction

- **LES** - less than (Figure 58) is **TRUE** if source **A** is less than source **B**. Otherwise it is **FALSE**.

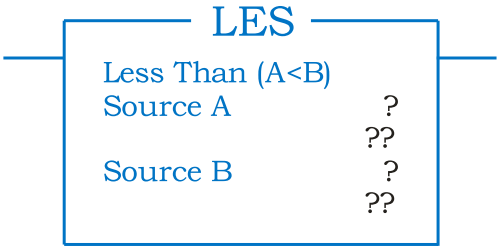


Figure 58 LES instruction

- **GRT** - greater than (Figure 59) is **TRUE** if source **A** is greater than source **B**. Otherwise it is **FALSE**.

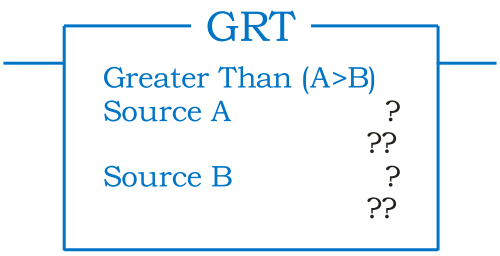


Figure 59 GRT instruction

- **LEQ** - less than or equal (Figure 60) is **TRUE** if source **A** is less than or equal to source **B**. Otherwise it is **FALSE**.

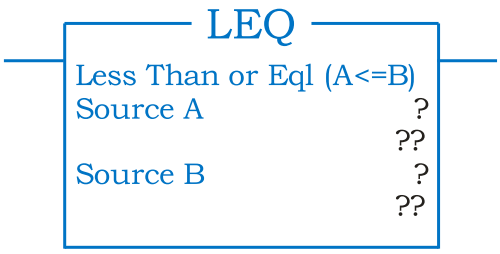


Figure 60 LEQ instruction

- **GEQ** - greater than or equal (Figure 61) is **TRUE** if source **A** is greater than or equal to source **B**. Otherwise it is **FALSE**.

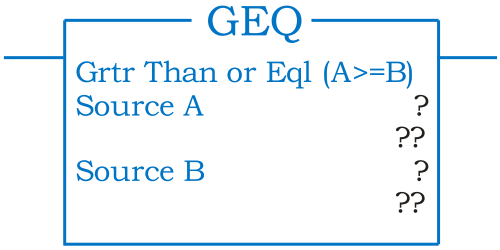


Figure 61 GEQ instruction

- **LIM** - limit test (Figure 62) is **TRUE** if the test value is greater than or equal to the low limit value and less than or equal to the high limit value. Otherwise it is FALSE.

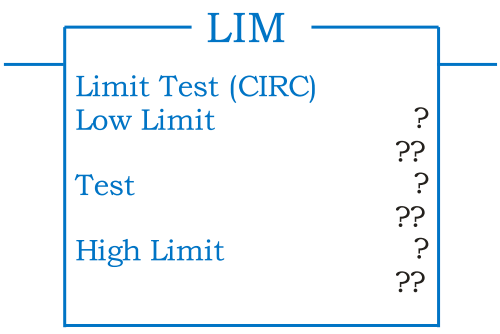


Figure 62 LIM instruction

- **CMP** - compare (Figure 63) is **TRUE** if expression typed in is **TRUE**. Otherwise it is **FALSE**.

Figure 63 CMP instruction

- **MEQ** - masked equal (Figure 64). Take the binary value of source, use the **AND** operation with the mask. If the resulting number the same value as the one in the compare field masked equal will be **TRUE**. Otherwise it is **FALSE**.

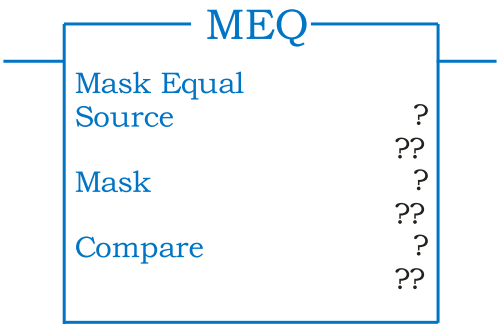


Figure 64 MEQ instruction

2.3.2. Exercise IV: Traffic Control Revisited

Let's revisit the precious exercise. This time we will use only one timer. Everything else will stay the same. Figure 65 and 66 show the complete program.

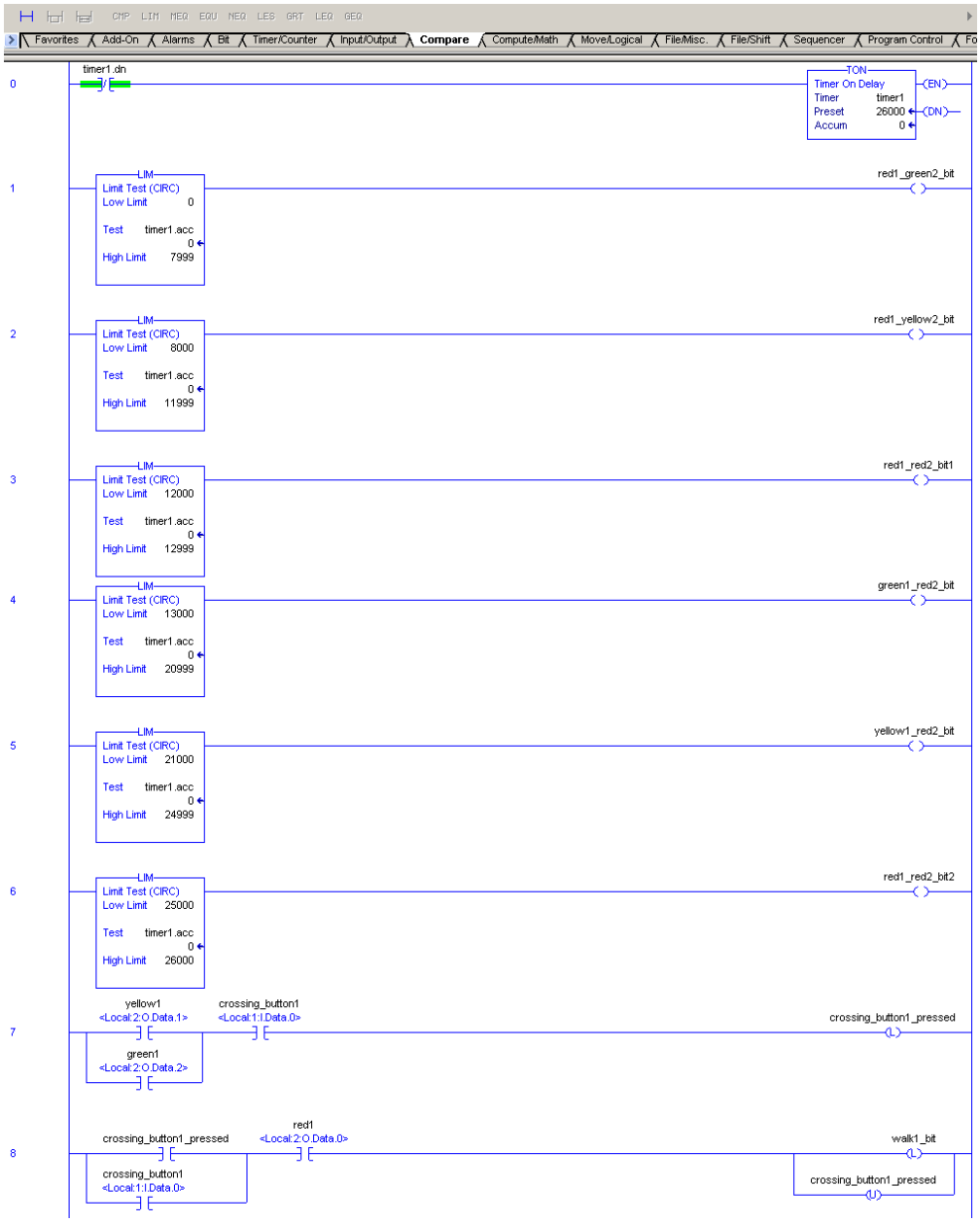


Figure 65 Solution to the problem

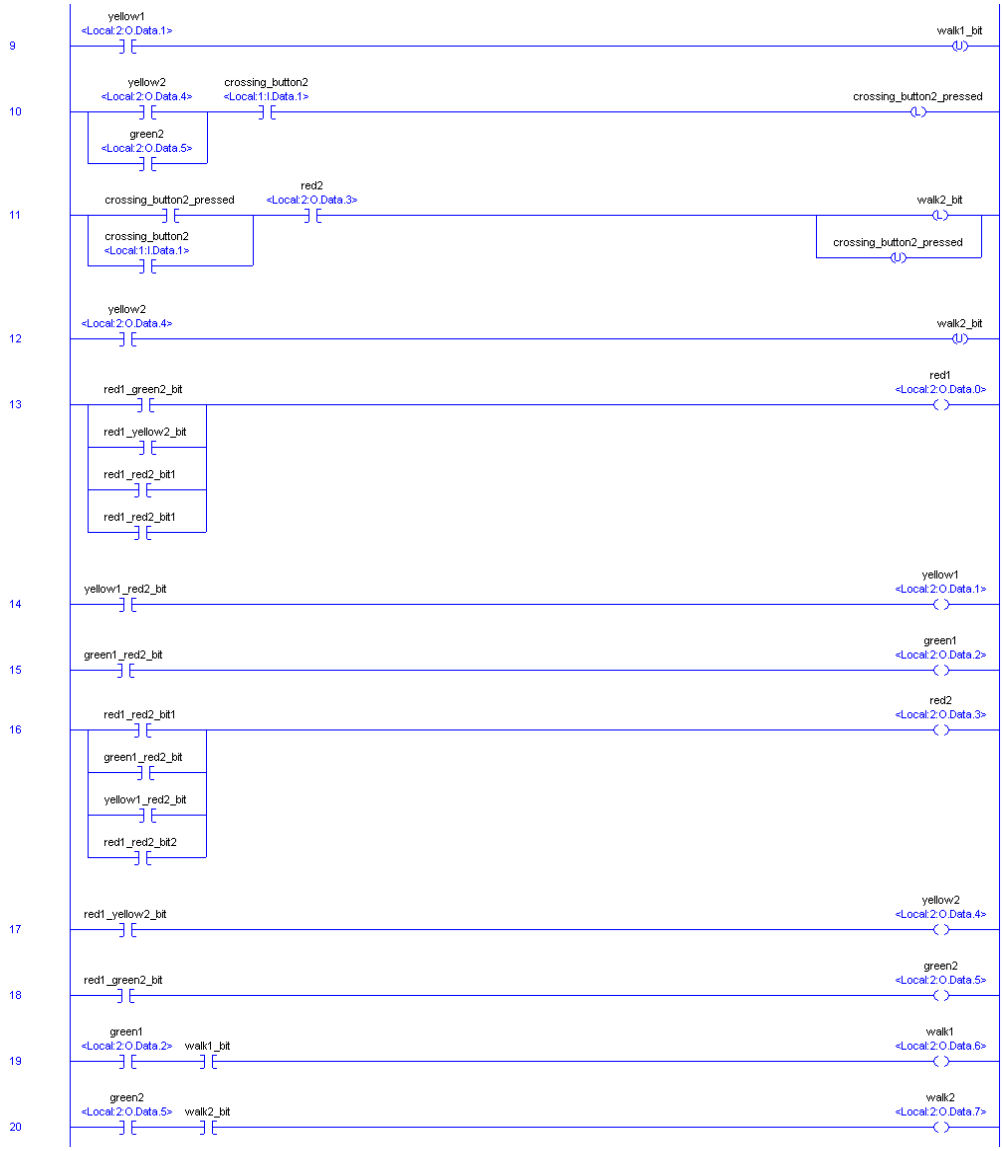


Figure 66 Solution to the problem cont.

Line 0 holds the timer that repeats every 26 s. Lines 1-6 separate the 26 s into segments that define which lights should be lit. This is most easily done with the limit test operation. There are alternatives such as using **GEQ** and **LES** in series.

2.4. I Can Count On You

In the same way as timers count time, counters count the number of times something occurred.

2.4.1. Types of Counters and Their Uses

There are 3 types of counters in **RSLogix 5000**. **Count Up** (**CTU**) and **Count Down** (**CTD**) can be used in ladder diagrams. **Count Up/Down** can be used in function block diagrams. It is omitted from the ladder diagram language because using the **CTU** and **CTD** together achieves the same result. Like timers counters have a **tag**, preset value (**PRE**) and accumulator (**ACC**). They also have bits that indicate their state. These states are overflow (**OV**), underflow (**UN**), done (**DN**) as well as count ((**CU**) for **CTU**, (**CD**) for **CTD**).

The **CTU** works as shown in Figure 67.

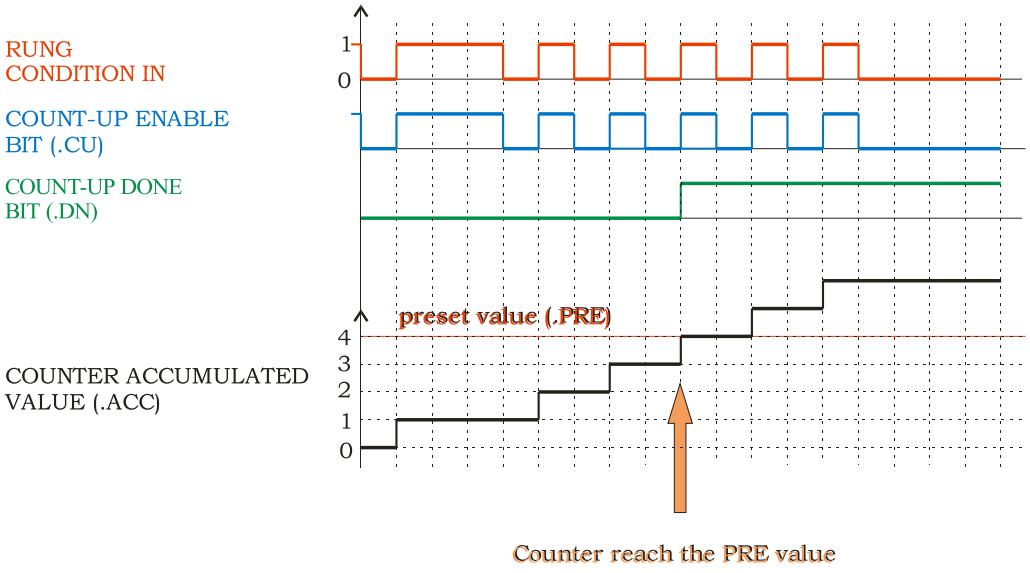


Figure 67 CTU

The **CTD** works as shown in Figure 68.

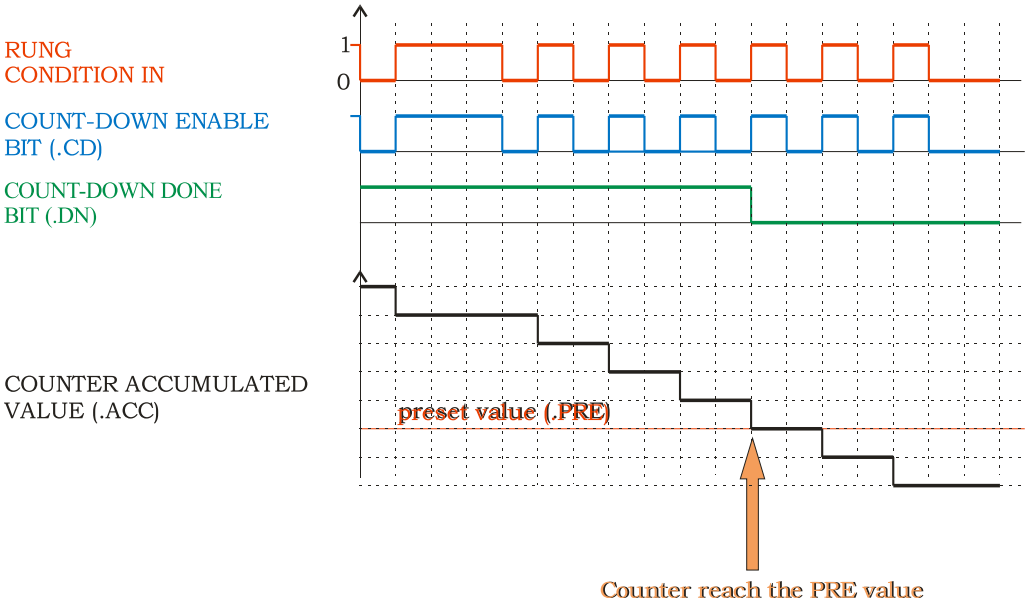


Figure 68 CTD

Now let's create a simple program that uses a counter. First go to the **Timer/Counter** tab (Figure 69). Add the **CTU** to the end of a rung and add a **tag** to the counter tag (Figure 70). Set the counter **Preset** value (Figure 71). Finally add a condition to count the occurrence of (Figure 72).

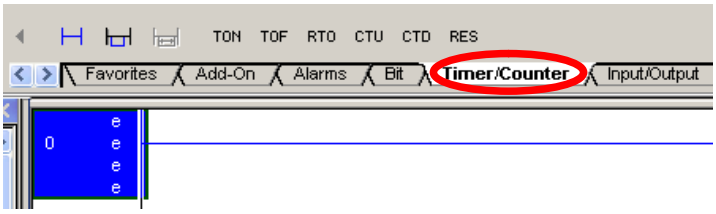


Figure 69 Timer/Counter tab

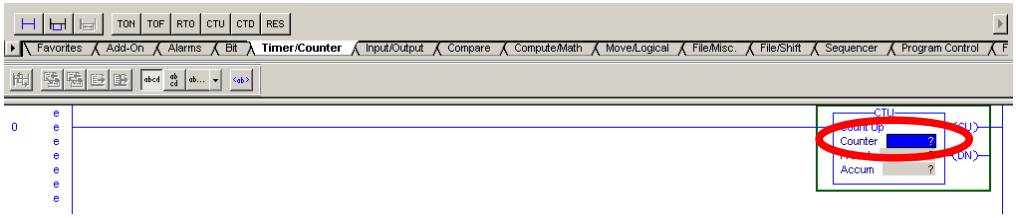


Figure 70 Add **CTU** to rung and add **tag**

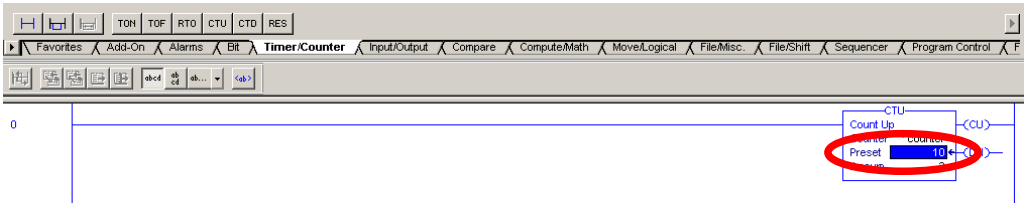


Figure 71 Set preset value

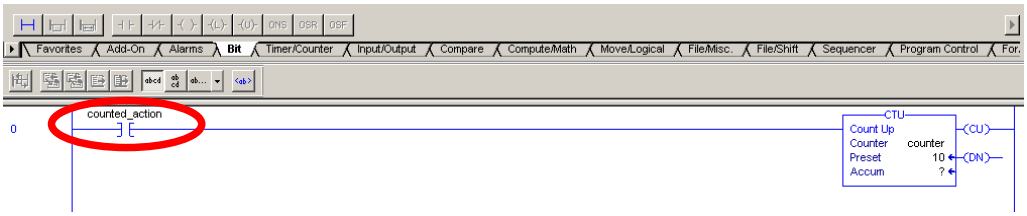


Figure 72 Add condition to count the occurrence of

This short program will count the occurrence of the **counted_action** tag. The counter will add one to the accumulator (ACC) on every rising edge of its condition. Once it reaches the preset value the done bit will be activated. As we have seen on Figure 67 once the done bit is true the counter will not stop counting. For every rising edge of the **counted_action** the accumulator will be incremented by one until it reaches the highest possible value (2,147,483,647). If we try to increment it further an overflow will occur. This will result in the overflow bit (**OV**) becoming **TRUE** and the value of **ACC** will become -2,147,483,648.

Similarly a **CTD** would subtract one from the **ACC** on every rising edge of its condition. Once it reaches -2,147,483,648 and we decrement it further an underflow will occur. This will result in the underflow bit (**UN**) becoming **TRUE** and the value of the **ACC** will become 2,147,483,647.

To reset the **ACC** value of a timer/counter use the **Reset** instruction. This instruction takes the tag of a timer/counter and sets its **ACC** to **0** when its condition is **TRUE**.

In those instances where we need to exceed these limits we can use cascading counters (Figure 73).

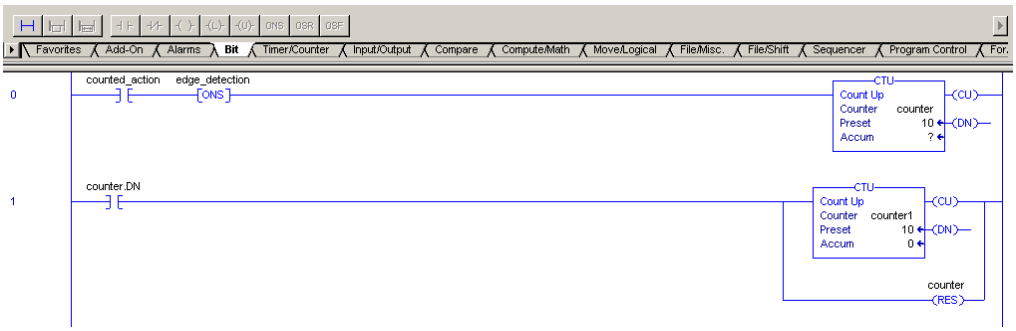


Figure 73 Cascading counters

Normally in the case of counters you do not need to use edge detection since it is already integrated into the counter. However when you are using the a counters **DN** bit to reset itself as shown in Figure 72 it will detect a false edge and the **ACC** will be set to **1** instead of **0**. To fix this simply add a **One Shot** instruction to the counter being reset.

2.4.2. Exercise V: Batch Mixing

In this exercise the task is to write a program to control a batch mixer which is shown on Figure 74.

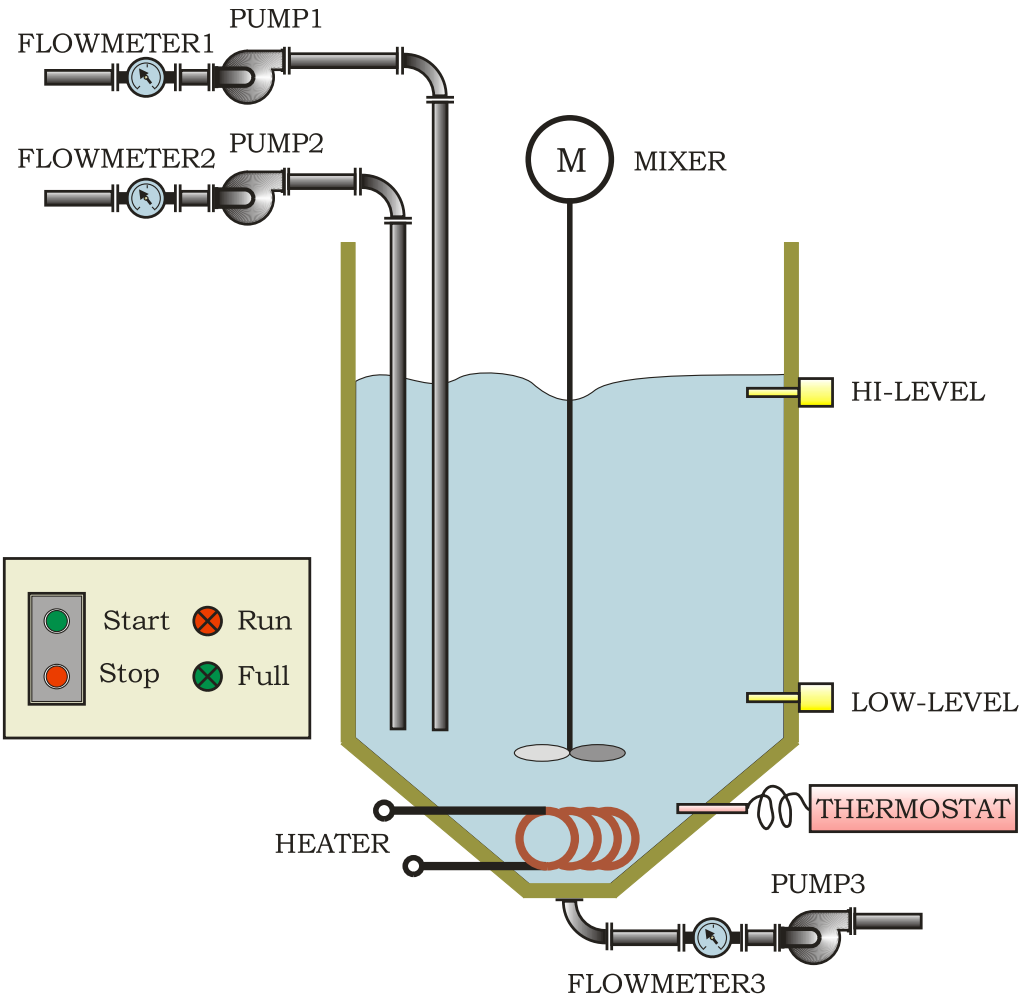


Figure 74 Batch mixer

The inputs are shown in table 13.

Table 13 Inputs of the batch mixer system

Inputs (switching elements)	Name	Type	Identifier
Pushbutton	Start	NO	Local:1:I.Data.0
Pushbutton	Stop	NC	Local:1:I.Data.1
Flowmeter	FLOWMETER1	NC	Local:1:I.Data.2
Flowmeter	FLOWMETER2	NC	Local:1:I.Data.3
Flowmeter	FLOWMETER3	NC	Local:1:I.Data.4
Level sensor	HI-LEVEL	NO	Local:1:I.Data.5
Level sensor	LOW-LEVEL	NO	Local:1:I.Data.6
Temperature sensor	THERMOSTAT	NO	Local:1:I.Data.7

The outputs are shown in table 14.

Table 14 Outputs of the batch mixer system

Output devices	Name	Identifier
Pump	PUMP1	Local:2:O.Data.0
Pump	PUMP2	Local:2:O.Data.1
Pump	PUMP3	Local:2:O.Data.2
Contactor	MIXER	Local:2:O.Data.3
Contactor	HEATER	Local:2:O.Data.4
Indicator Lamp	Run	Local:2:O.Data.5
Indicator Lamp	Full	Local:2:O.Data.6

The task we are charged with is the following:

- When the **Start** button is pressed the batch mixing will start.
- Fill the tank up with the fluids using **PUMP1** and **PUMP2**.
- As **PUMP1** and **PUMP2** are working **FLOWMETER1** and **FLOWMETER2** will indicate every liter of fluid with one pulse. Use these pulses to count the amount of fluid used. Make sure **PUMP1** stops after 80 liters.
- Once the tank is full heat and mix the batch.
- Once the heat is sufficient the **THERMOSTAT** will signal. Keep the temperature on this level while mixing.
- The mixing should stop after 60 seconds.
- Once the mixing is done stop heating and drain the tank completely using **PUMP3**.

- The machine should continuously work until stopped.
- While the machine is running indicate it with the **Run** lamp.
- When the tank is full **HI-LEVEL** will give a **TRUE** signal. Indicate this with the **Full** lamp.
- When the **Stop** button is pressed the process stops. When you press the **Start** button the process must continue where it left off.

Figure 75 shows the solution.

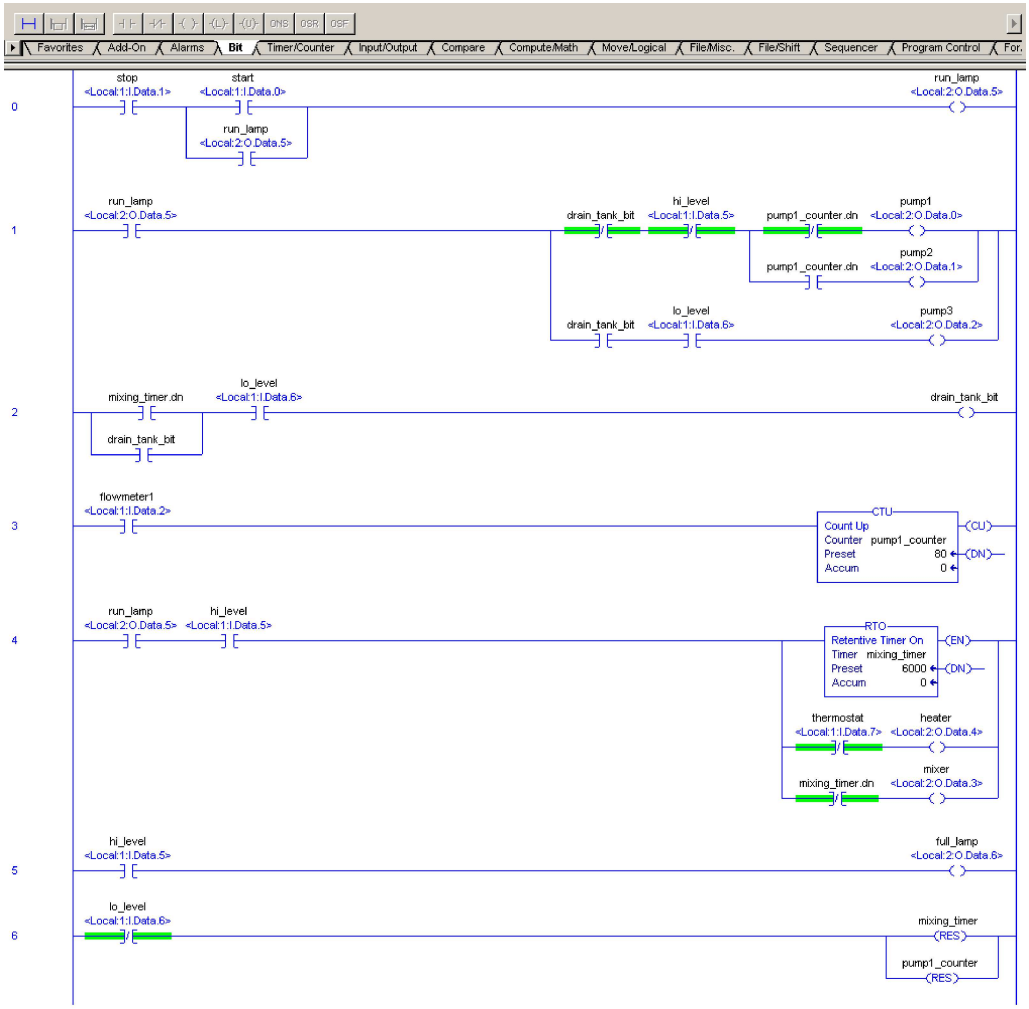


Figure 75 Batch mixer solution

Line 0 is a simple self holding circuit that operates the **Run** lamp. Since the **Stop** button is **NC** the **XIC** instruction is used to break the circuit.

Line 1 is responsible for filling and draining the tank. It uses the **run_lamp** to check if the **Start** button has been pressed and breaks into 2 parts. If the **drain_tank**_bit is false and the tank is not full it starts to fill the tank using first **PUMP1**. Once **PUMP1** has pumped 80 liters it is disabled and **PUMP2** fills the rest of the tank until the **HI-LEVEL** sensor disables **PUMP1** and **PUMP2**. Once the **drain_tank**_bit is **TRUE** the tank is drained using **PUMP3** until the **LO-LEVEL** sensor disables **PUMP3**.

Line 2 is also a self holding circuit. It is responsible for the **drain_tank_bit**. This bit is **TRUE** when the tank needs to be drained. The condition for this becomes **TRUE** when the **mixing_timer** reaches its preset value. Once the tank reaches the **LO-LEVEL** sensor the draining needs to stop.

Line 3 has the counter that measures the fluid pumped by **PUMP1**. Its preset value is set to 80 since that is required. If you need to mix a different ratio simply change the value.

Line 4 is responsible for the heating and mixing of the batch. Here we used a retentive timer. This way if the **Stop** button is pressed it will not lose its value. A normal timer may not be acceptable since each time it stops the timer will reset and the batch can be over mixed. The condition for heating is simply when the **THERMOSTAT** is off start heating, when it is on stop. This is the most basic control for heating. In other cases something more sophisticated should be used.

Line 5 handles the **Full** lamp.

Line 6 resets the counter and the timer once the tank is drained.

2.5. Advanced Stuff

In the previous sections we saw the basic instructions that are available for most PLCs. In this section we will see the full abilities of the **RSLogix 5000** programming environment. First up are math operations.

2.5.1. Math Operations

Math operations as their name suggest allow us to manipulate data in a mathematical way. Numbers are stored in memory and we access them with tags. To store a number a fix amount of memory needs to be set aside or in other words allocated. The size of the allocated memory determines the

range of values that can be stored there. Table 15 shows the data types associated with numbers and their ranges.

Table 15 Data types and ranges

Data Type	Negative	Positive
SINT	-127	127
INT	-32,768	32,767
DINT	-2,147,483,648	2,147,483,647
REAL	$-3.402823*10^{38}$	$1.1754944*10^{-38}$
	$-1.1754944*10^{-38}$	$3.402823*10^{38}$

Math operations can be found under **Compute/Math** (Figure 76).

Figure 76 Compute/Math tab

These operations are:

- **ADD** - (Figure 77) Adds the value of source **A** to source **B** and stores the result in **Dest**.

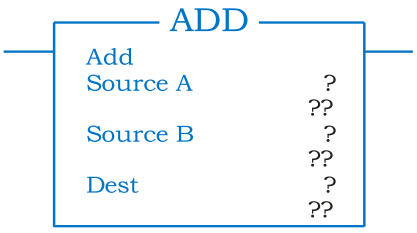


Figure 77 ADD instruction

- **SUB** - (Figure 78) Subtracts the value of source **B** from source **A** and stores the result in **Dest**.

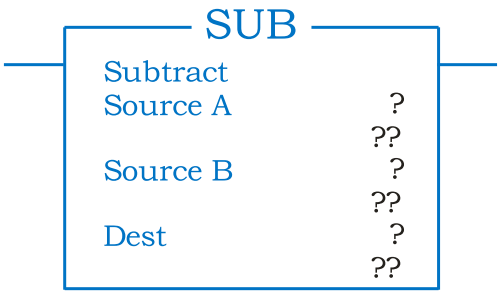


Figure 78 SUB instruction

- **MUL** - (Figure 79) Multiplies the value of source A by the value of source **B** and stores it in **Dest**.

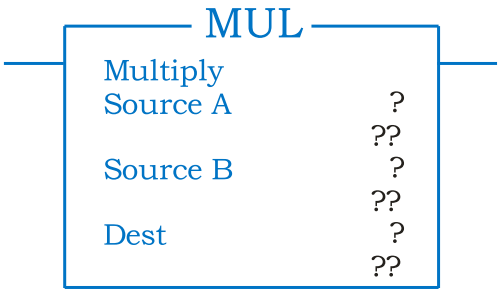


Figure 79 MUL instruction

- **DIV** - (Figure 80) Divides the value of source **A** by the value of source **B** and stores the result in **Dest**.

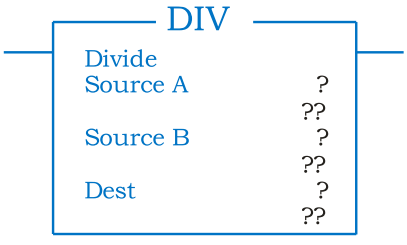


Figure 80 DIV instruction

- **MOD** - (Figure 81) Divides the value of source **A** with the value of source **B** and stores the remainder in **Dest**.

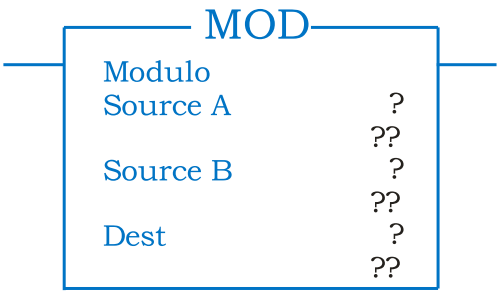


Figure 81 MOD instruction

- **SQR** - (Figure 82) Computes the square root of source and stores it in **Dest**.

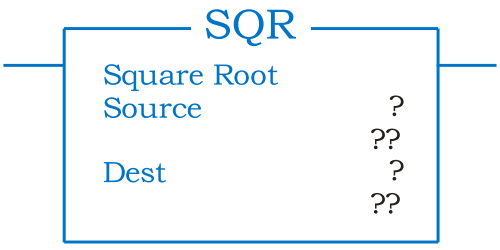


Figure 82 SQR instruction

- **NEG** - (Figure 83) Changes the sign of source and stores it in **Dest**.

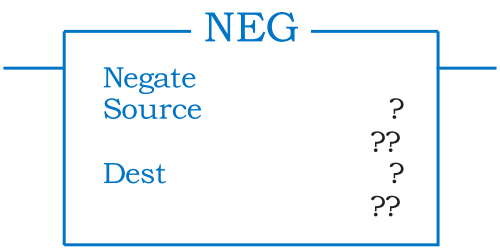


Figure 83 NEG instruction

- **ABS** - (Figure 84) Takes the absolute value of source and stores it in **Dest**.

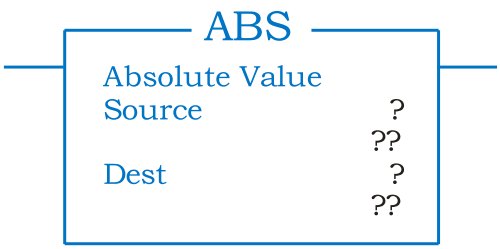


Figure 84 ABS instruction

- **CPT** - (Figure 85) Computes the arithmetic operations defined in the expression and stores it in **Dest**.

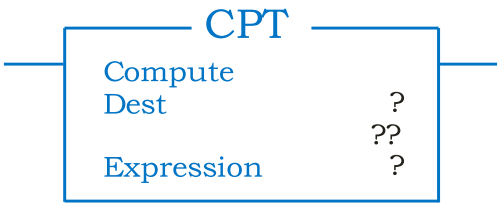


Figure 85 CPT instruction

Naturally since there are 4 data types it can occur that we have data in one data type and want to store it in another. In these cases data conversion happens in the following way:

- Before the instruction executes, it performs the following conversions:

  1) If one of the input operands is a **REAL** value, any **SINT**, **INT**, or **DINT** values convert to **REAL** values.

  2) If none of the input operands are a **REAL** value, any SINT or **INT** value converts to a **DINT** value.

- After instruction execution, the result (a **DINT** or **REAL** value) converts to the destination data type, if necessary.

Once an arithmetic operation has been completed the arithmetic status bits hold additional information about the result of the operation. The arithmetic status bits are:

- **S:N** sign. This bit is set to **TRUE** if the result is negative. The location of this bit can be seen on Figure 84.
- **S:C** carry. The carry flag represents the bit that would be in the data type if it were stored to a larger data type. It is not actually part of the data type. Figure 32 shows where integers store this data.
- **S:Z** zero. This bit is set to **TRUE** if the result is **0**.
- **S:V** overflow. This bit is set to **TRUE** if the result overflows the destination. Every time this bit changes from **FALSE** to **TRUE** it generates a minor fault. This also happens if you divide by **0**.

2.5.2. Logical Operations

We have seen logical operations before. You connect 2 or more conditions in series or parallel and they form a logical **AND** or **OR** instruction. Those operations are only performed on bits. These logical operations are performed on bytes. These logical operations are found under the **Move/Logical** tab (Figure 86).

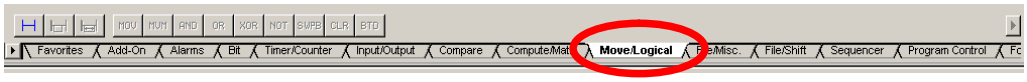


Figure 86 Move/Logical tab

These operations are:

- **AND** - (Figure 87) Bitwise **AND** the value of source **A** to source **B** and stores the result in **Dest**. An example can be seen on Figure 88.

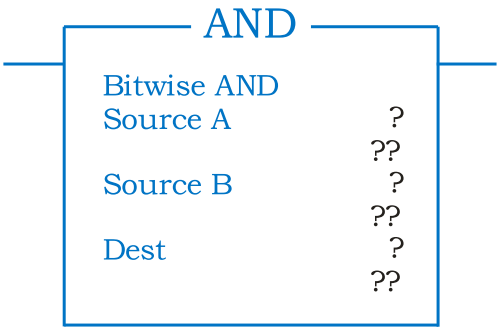


Figure 87 bitwise **AND** instruction

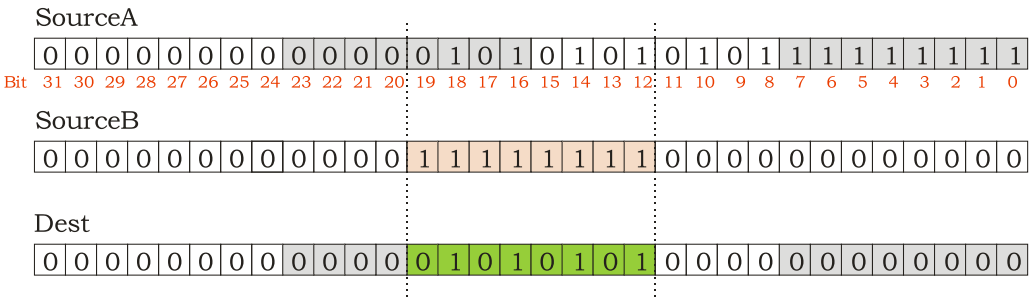


Figure 88 Example of bitwise **AND**

- **OR** - (Figure 89) Bitwise **OR** the value of source **A** to source **B** and stores the result in **Dest**. An example can be seen on Figure 90.

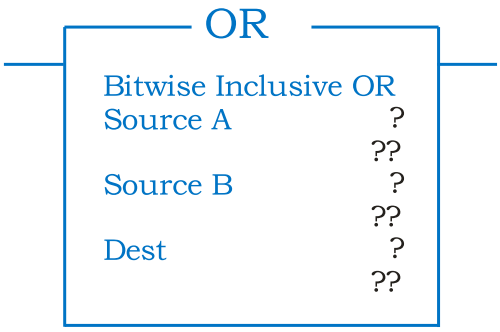


Figure 89 OR instruction

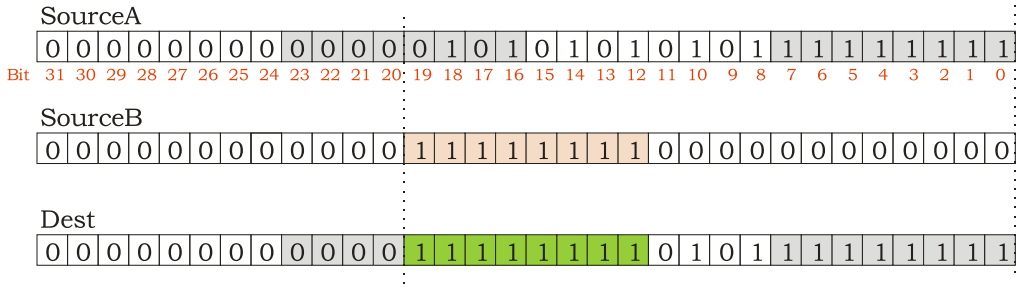


Figure 90 Example of bitwise **OR**

- **XOR** - (Figure 91) Bitwise **Exclusive OR** the value of source **A** to source **B** and stores the result in **Dest**. An example can be seen on Figure 92.

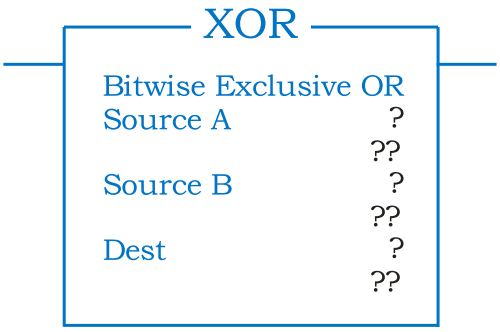


Figure 91 XOR instruction

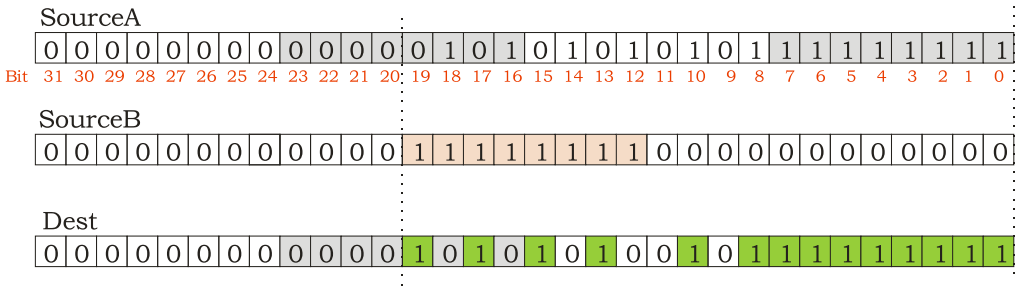


Figure 92 Example of bitwise **XOR**

- **NOT** - (Figure 93) Negates every bit of **Source** and stores the result in **Dest**. An example can be seen on Figure 94.

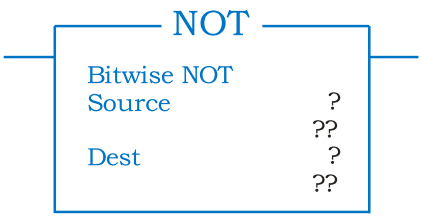


Figure 93 NOT instruction

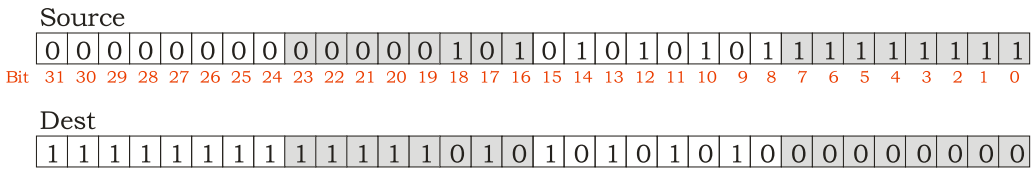


Figure 94 Example of **NOT**

- **MOV** - (Figure 95) Copies the value of **Source** to **Dest** and leaves the source unchanged.

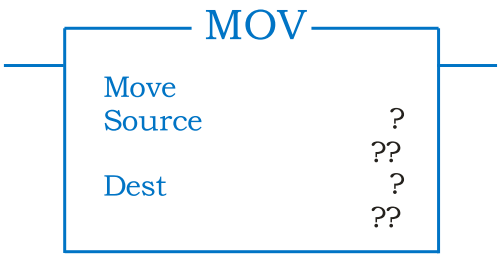


Figure 95 MOV instruction

- **MVM** - (Figure 96) Copies the value of **Source** to **Dest**, allows a portion of the source to be masked. Leaves the source unchanged. An example can be seen on Figure 97.

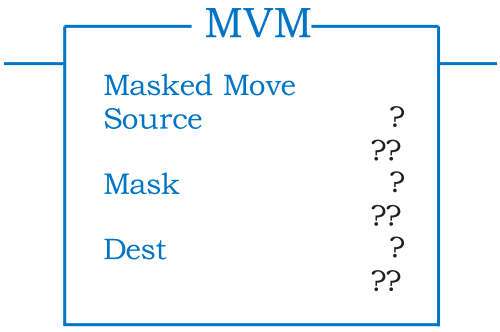


Figure 96 MVM instruction

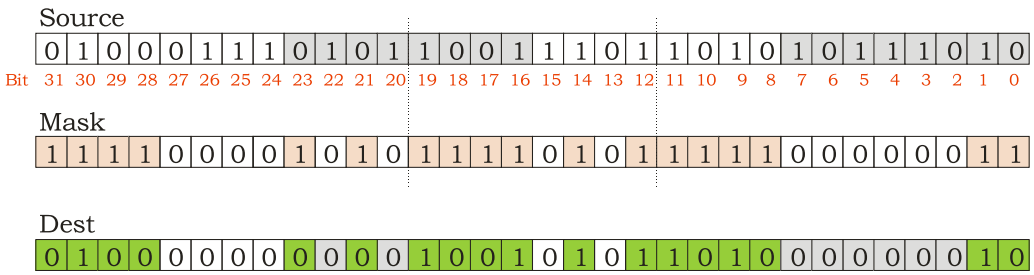


Figure 97 Example of **MVM**

- **CLR** - (Figure 98) Clears the **Dest** value (Sets it to **0**).

Figure 98 CLR instruction

- **SWPB** - (Figure 99) Rearranges the bytes of the **Source** and stores it in **Dest**. The swap has several orders these are: Reverse, Word, High/Low. Examples of these orders can be seen of Figure 100-102.

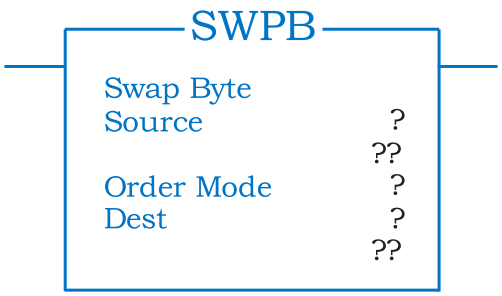


Figure 99 SWPB instruction

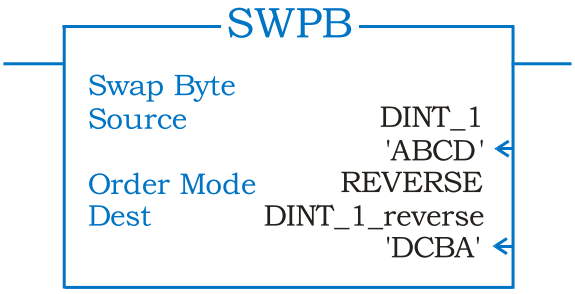


Figure 100 Example of Reverse byte swap

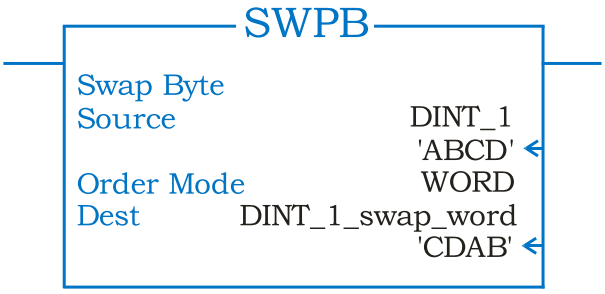


Figure 101 Example of Word byte swap

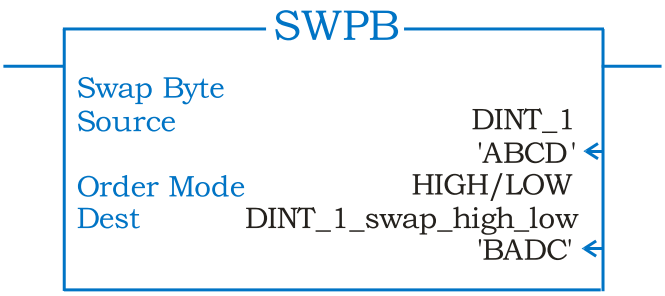


Figure 102 Example of High/Low byte swap

- **BTD** - (Figure 103) Copies the specified bits from the **Source**, shifts the bits to the appropriate position, and stores the bits into the **Dest**. Examples of the **BTD** instruction can be seen on Figures 104, 105.

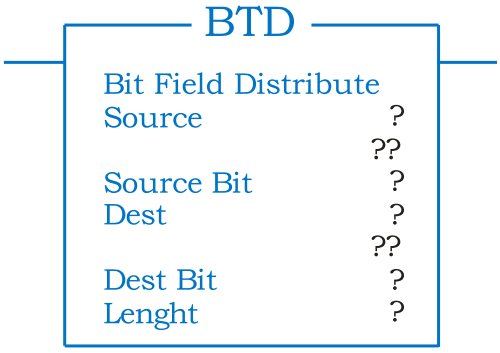
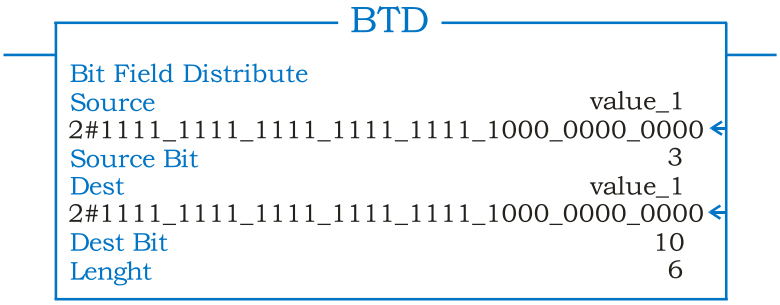


Figure 103 BTD instruction



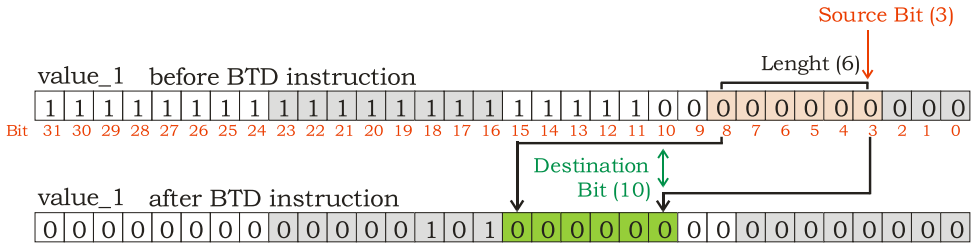
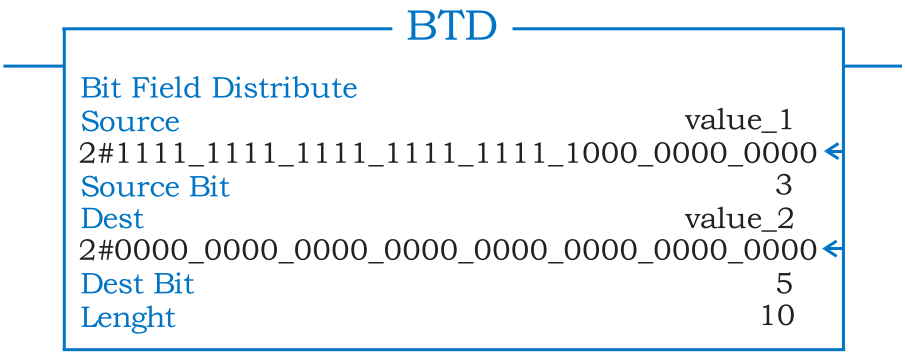


Figure 104 Example 1 of **BTD**



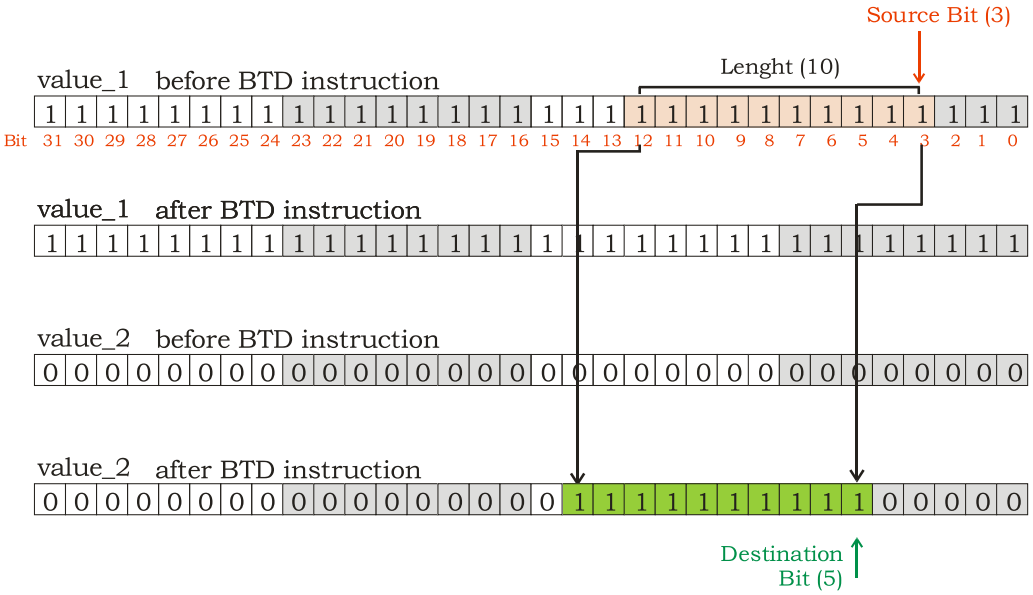


Figure 105 Example 2 of **BTD**

2.5.3. File operations

Up until now we have dealt with bits and numbers. Numbers were in the form of a single number or bytes. When we want to deal with a collection of numbers or larger groups of bytes we use files. Files are a location in memory that has a beginning, some content and an end. In PLC programming files are used to handle data that are grouped together such as an array. They are also used to buffer data. **FIFO** and **LIFO** structures are the most important data buffers used. File operations are found under 2

tabs **File/Misc.** (Figure 106) and **File/Shift** (Figure 107). File operations usually have a **Control** field. Place a tag in this field that identifies that instance of the instruction.

Figure 106 File/Misc. tab

Figure 107 File/Shift tab

The file operations are:

- **BSL** - (Figure 108) Bit Shift left shifts the specified bits within the Array one position left. An example can be seen on Figure 109. The **BSL** instruction has 5 attributes:
  - **EN** - Enable indicates the **BSL** instruction is enabled.
  - **DN** - Done indicates the instruction shifted.
  - **UL** - Unload is the instructions output. It holds the bit that was shifted out of range.
  - **ER** - Error is set when the **LEN<0**.
  - **LEN** - Length specifies the number of array bits to shift.

Figure 108 BSL instruction

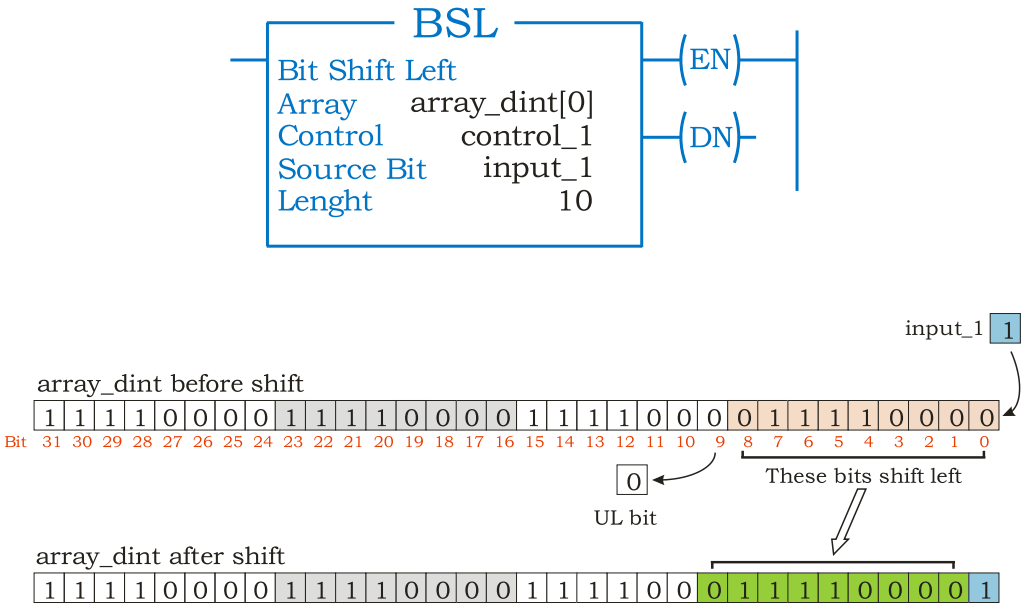


Figure 109 BSL example

- **BSR** - (Figure 110) Bits Shift Right shifts the specified bits within the Array one position right. An example can be seen on Figure 111. The **BSL** instruction has 5 attributes:
  - **EN** - Enable indicates the BSR instruction is enabled.
  - **DN** - Done indicates the instruction shifted.
  - **UL** - Unload is the instructions output. It holds the bit that was shifted out of range.
  - **ER** - Error is set when the LEN<0.
  - **LEN** - Length specifies the number of array bits to shift.

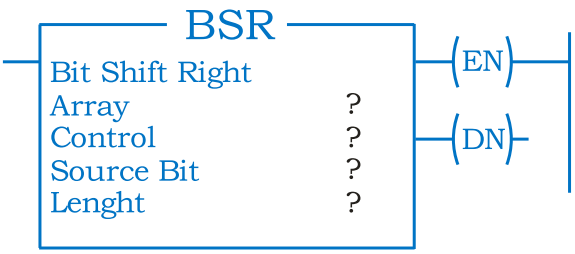


Figure 110 BSR instruction

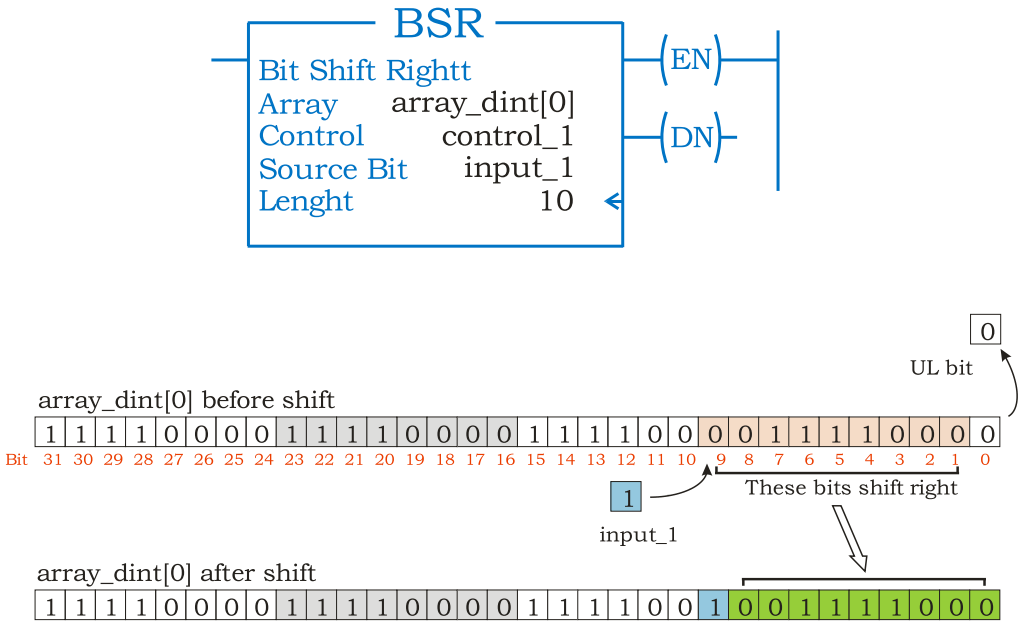


Figure 111 BSR example

- **FFL** - (Figure 112) **FIFO** Load copies the **Source** value to the **FIFO**. An example can be seen on Figure 113. **FFL** has 5 attributes:
  - **EN** - Enable indicates the **FFL** instruction is enabled.
  - **DN** - Done indicates the **FIFO** is full. This bit disables further loading while **POS** = **LEN**.
  - **EM** - Empty indicates the **FIFO** is empty.
  - **LEN** - Length specifies the maximum number of elements in the **FIFO**
  - **POS** - Position indentifies the place where to put the next value.

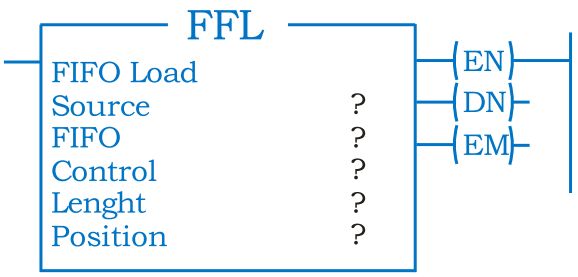
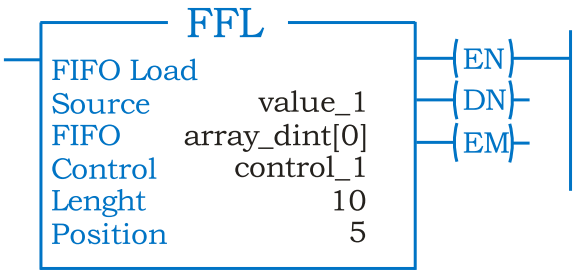


Figure 112 FFL instruction



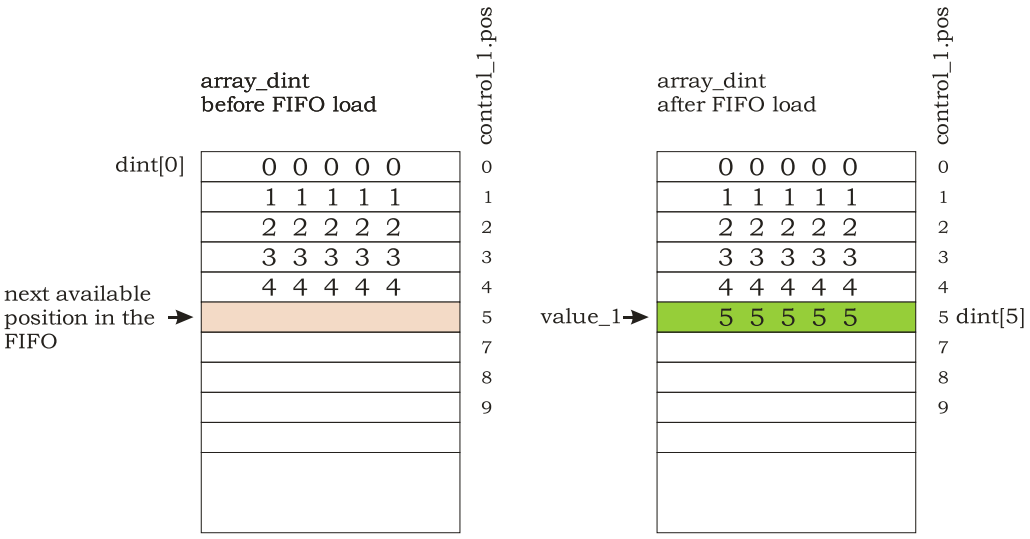


Figure 113 FFL example

- **FFU** - (Figure 114) **FIFO** Unload unloads the value from position 0 (first position) of the **FIFO** and stores that value in the Destination. The remaining data in the **FIFO** shifts down one position. An example can be seen on Figure 115. **FFL** has 5 attributes:
  - **EN** - Enable indicates the **FFU** instruction is enabled.
  - **DN** - Done indicates the **FIFO** is full. This bit disables further loading while **POS** = **LEN**.
  - **EM** - Empty indicates the **FIFO** is empty.
  - **LEN** - Length specifies the maximum number of elements in the **FIFO**
  - **POS** - Position indentifies the place where to put the next value.

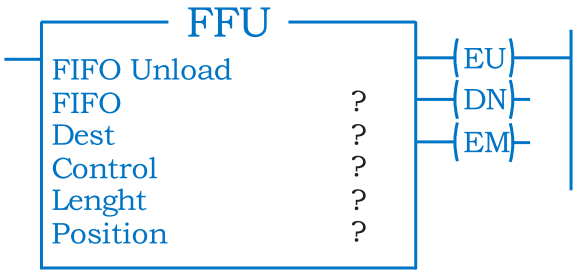
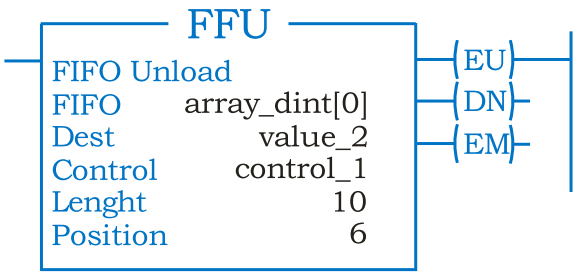


Figure 114 FFU instruction



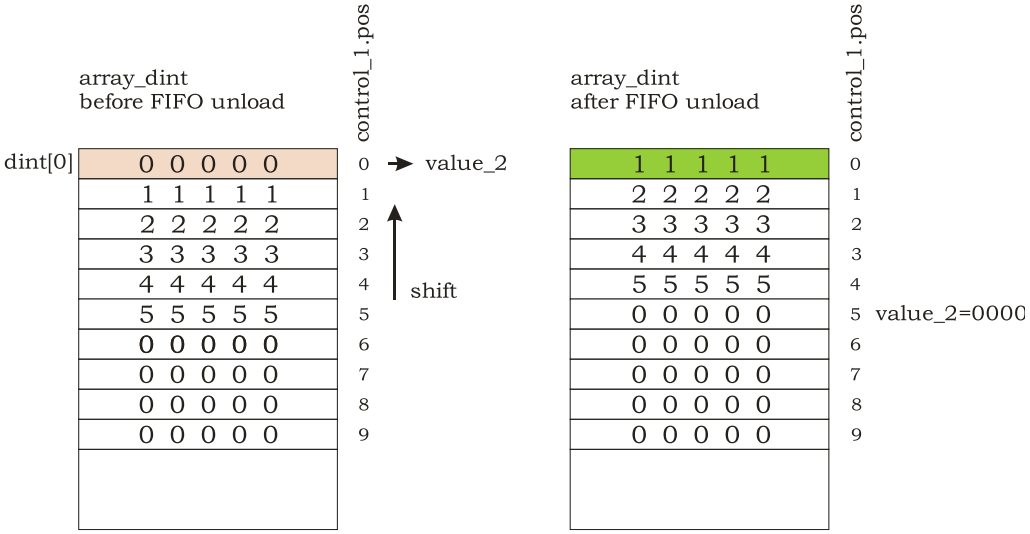


Figure 115 FFU example

- **LFL** - (Figure 116) **LIFO** Load copies the **Source** value to the **LIFO**. An example can be seen on Figure 117. **LFL** has 5 attributes:
  - **EN** - Enable indicates the **LFL** instruction is enabled.
  - **DN** - Done indicates the **LIFO** is full. This bit disables further loading while **POS** = **LEN**.
  - **EM** - Empty indicates the **LIFO** is empty.
  - **LEN** - Length specifies the maximum number of elements in the **LIFO**
  - **POS** - Position indentifies the place where to put the next value.

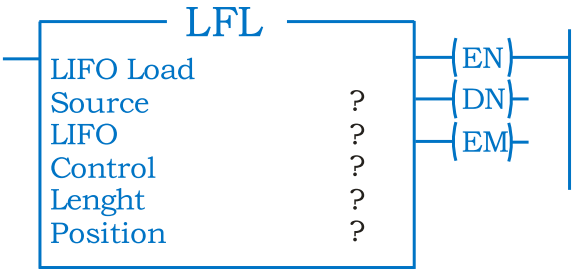


Figure 116 LFL instruction

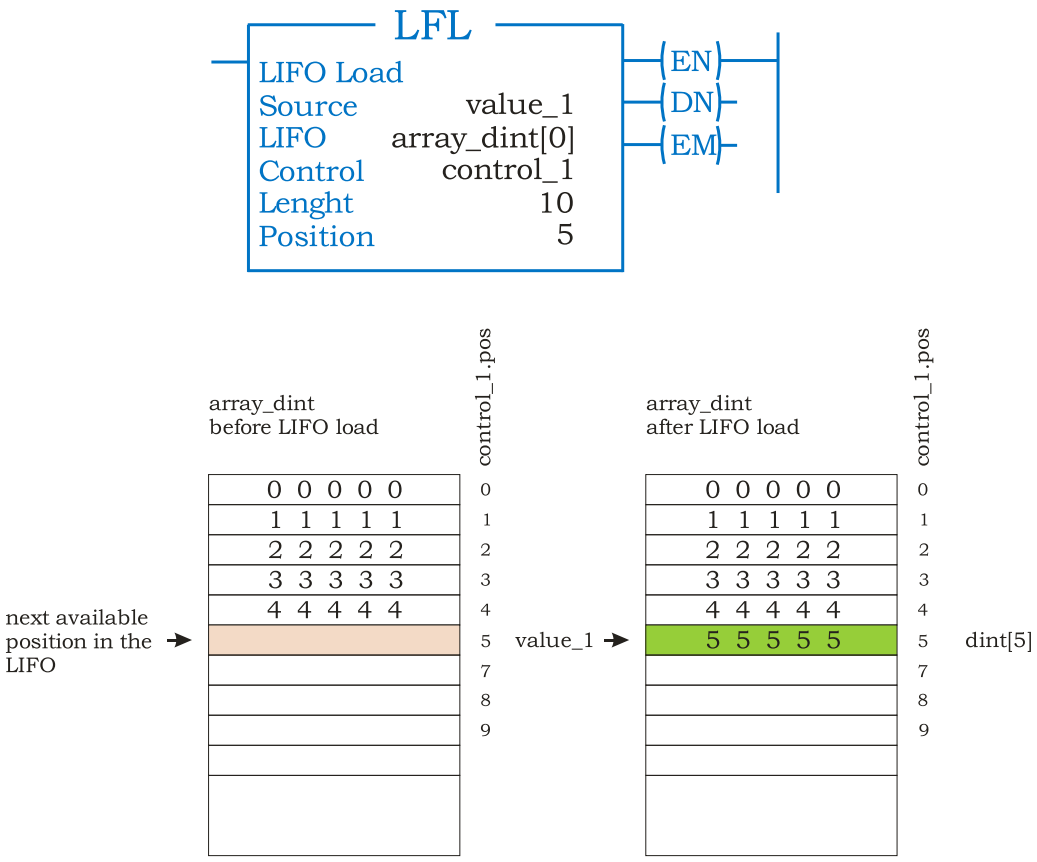


Figure 117 LFL example

- **LFU** - (Figure 118) **LIFO** Unload unloads the value at .POS of the **LIFO** to **Dest** and stores **0** in that location. An example can be seen on Figure 119. **LFL** has 5 attributes:
  - **EN** - Enable indicates the LFU instruction is enabled.
  - **DN** - Done indicates the LIFO is full. This bit disables further loading while **POS** = **LEN**.
  - **EM** - Empty indicates the **LIFO** is empty.
  - **LEN** - Length specifies the maximum number of elements in the **LIFO**
  - **POS** - Position indentifies the place where to put the next value.

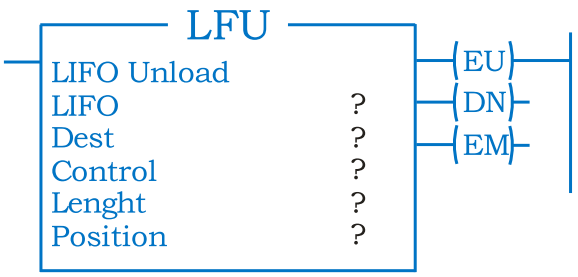
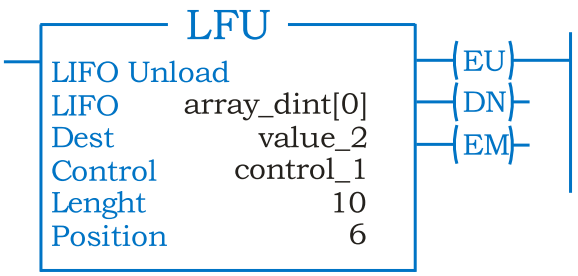


Figure 118 LFU instruction



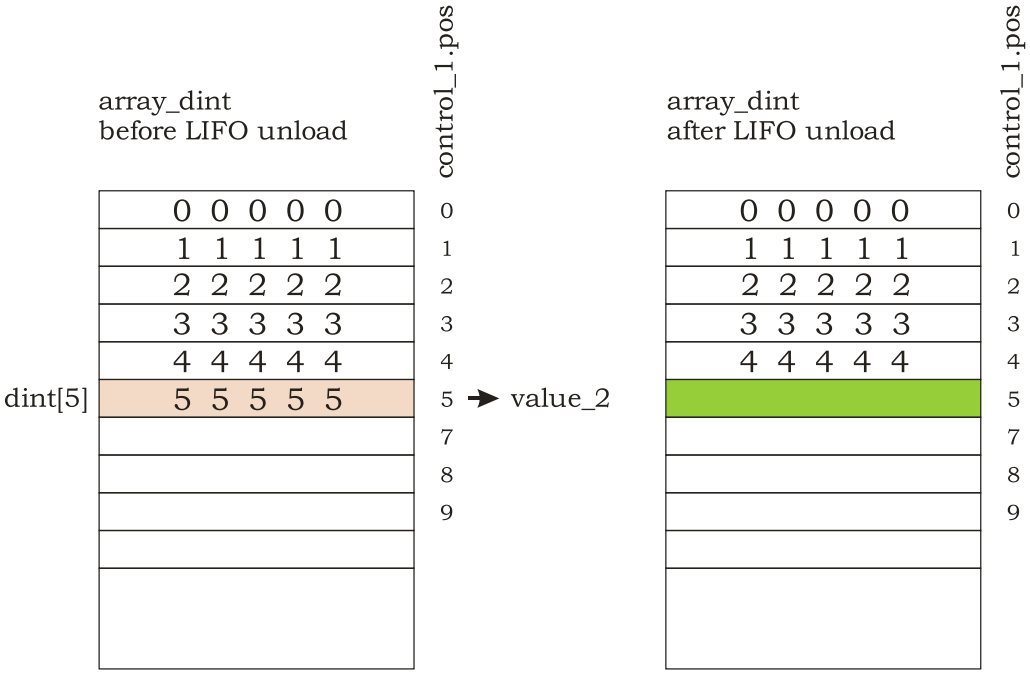


Figure 119 LFU example

- **SIZE** - (Figure 120) finds the **Size** of a dimension of an array. **Dim. to Vary** is the dimension to check (0, 1, 2). Size is a tag that holds the result.

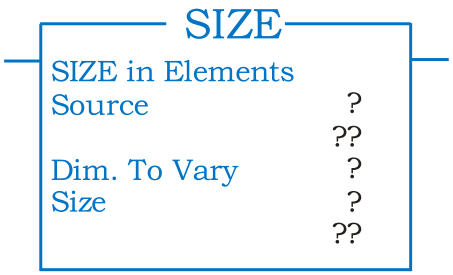


Figure 120 SIZE instruction

- **FLL** - (Figure 121) File Fill fills elements of an array with the **Source** value. The Source remains unchanged.

Figure 121 FLL instruction

- **COP** - (Figure 122) Copy File copies the value(s) in the **Source** to the values in the **Destination**. The Source remains unchanged.

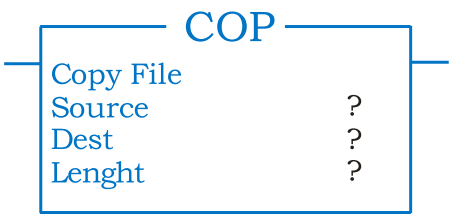


Figure 122 COP instruction

- **CPS** - (Figure 123) Synchronous Copy File copy the value(s) in the **Source** to the values in the **Destination**. While copying no I/O updates can change the data. The Source remains unchanged.

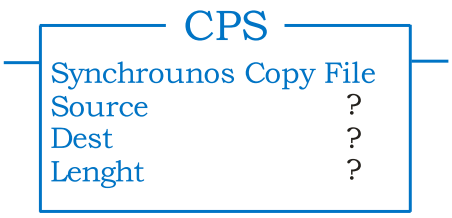


Figure 123 CPS instruction

- **FAL** - (Figure 124) File Arithmetic and Logic performs copy, arithmetic, logic, and function operations on data stored in an array. These instructions are the same as for the **CPT** except they work on arrays. **FAL** has 5 attributes:
  - **EN** - Enable indicates the **FAL** instruction is enabled.
  - **DN** - Done indicates the instruction has operated on the last element (**POS** = **LEN**).
  - **ER** - Error is set if the expression generates an overflow (**S:V** is set). The instruction stops executing while the **ER** bit is cleared.
  - **LEN** - Length specifies the number of elements in the array on which the FAL instruction operates.
  - **POS** - Position indentifies the position of the current element that the instruction is accessing.

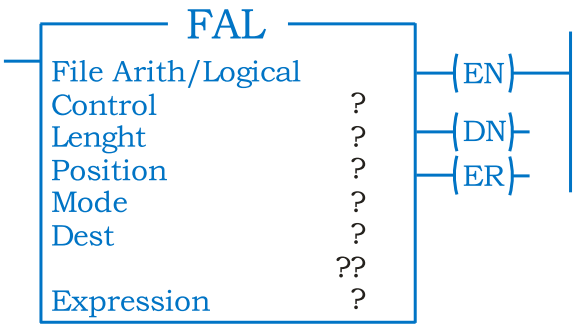


Figure 124 FAL instruction

- **FSC** - (Figure 125) File Search and Compare compares values in an array, element by element as defined by the expression. This instruction is the same as **CMP** except it works on arrays. **FSC** has 7 attributes these are:
  - **EN** - Enable indicates the **FSC** instruction is enabled.
  - **DN** - Done indicates the instruction has operated on the last element (**POS** = LEN).
  - **ER** - Error bit is not modified.
  - **IN** - Inhibit indicates the **FSC** instruction detected a true comparison. You must clear this bit to continue the search operation.
  - **FD** - Found indicates the **FSC** instruction detected a true comparison.
  - **LEN** - Length specifies the number of elements in the array on which the FSC instruction operates.
  - **POS** - Position indentifies the position of the current element that the instruction is accessing.

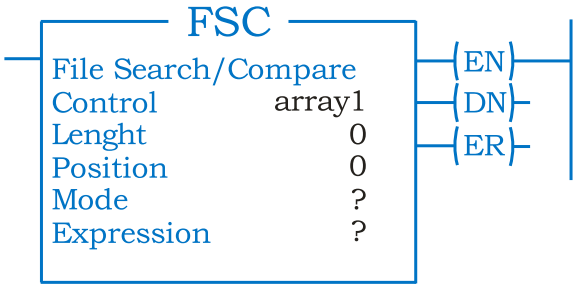


Figure 125 FSC instruction

- **SRT**- (Figure 126) File Sort sorts a set of values in one dimension (Dim to vary) of the array into ascending order. **SRT** has 5 attributes:
  - **EN** - Enable indicates the **SRT** instruction is enabled.
  - **DN** - Done indicates the instruction has operated on the last element (**POS** = **LEN**).
  - **ER** - Error is set when either **.LEN** < 0 or **.POS** < 0. Either of these conditions also generates a major fault.
  - **LEN** - Length specifies the number of elements in the array on which the **SRT** instruction operates.
  - **POS** - Position indentifies the position of the current element that the instruction is accessing.

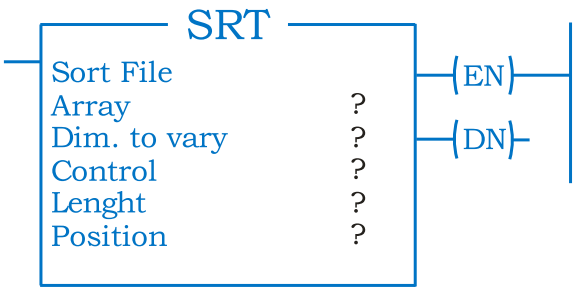


Figure 126 SRT instruction

- **AVE** - (Figure 127) Average calculates the average of a set of values. **AVE** has 5 attributes:
  - **EN** - Enable indicates the **AVE** instruction is enabled.
  - **DN** - Done indicates the instruction has operated on the last element (**POS** = **LEN**).
  - **ER** - The error bit is set when the instruction generates an overflow. The instruction stops executing until the program clears the **ER** bit. The **POS** value stores the position of the element that caused the overflow.
  - **LEN** - Length specifies the number of elements in the array on which the **AVE** instruction operates.
  - **POS** - Position indentifies the position of the current element that the instruction is accessing.

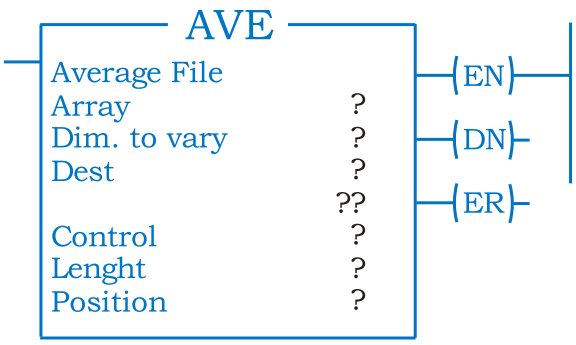


Figure 127 AVE instruction

- **STD** - (Figure 128) File Standard Deviation calculates the standard deviation of a set of values in one dimension of the Array and stores the result in the Destination. **STD** has 5 attributes:
    - **EN** - Enable indicates the STD instruction is enabled.
    - **DN** - Done indicates the instruction has operated on the last element (**POS** = **LEN**).
    - **ER** - The error bit is set when the instruction generates an overflow. The instruction stops executing until the program clears the **ER** bit. The **POS** value stores the position of the element that caused the overflow.
    - **LEN** - Length specifies the number of elements in the array on which the **STD** instruction operates.
    - **POS** - Position indentifies the position of the current element that the instruction is accessing.

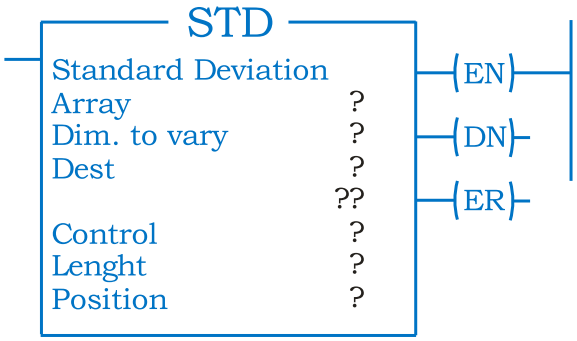


Figure 128 STD instruction

2.5.4. Program Control

As tasks become more complex the programs usually become larger and more difficult to manage. Programs can become too large to see through, some parts only need to execute once, some parts need to execute many times. This has always been true for programming. Depending on the programming language there are several ways to manage code. More advanced languages use functions. Languages that are close to the metal usually use things like labels and subroutines.

Labels are identifiers in the code that jump instructions can point to. They usually identify code that needs to be executed more than once. Once a line with a jumps to a label the flow of the program continues from there. This way you can skip certain parts of the code (Figure 129).

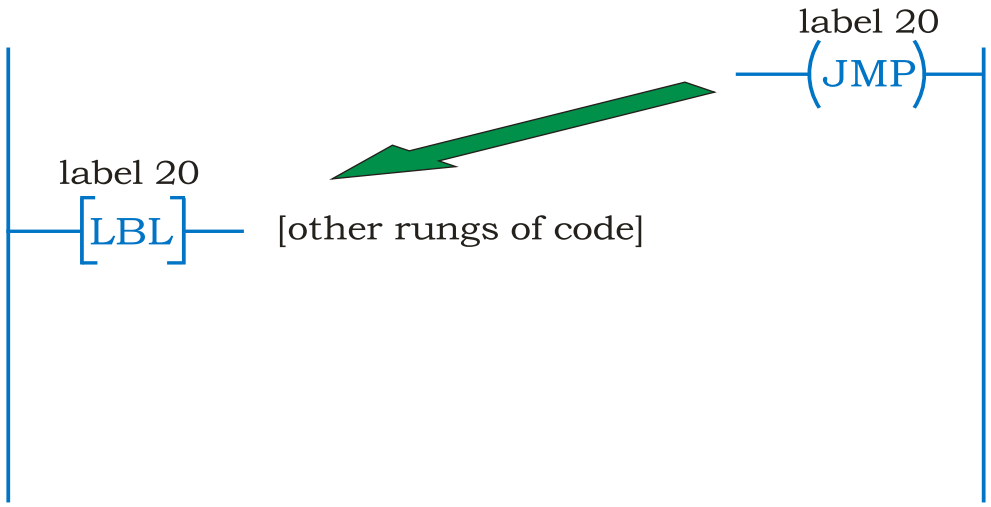


Figure 129 Jump to label

Subroutines are sections of code in a different program that can be called when needed. The main difference between a subroutine and a label is that you return from subroutines and continue where you left off. You can nest subroutines in subroutines. Each will return to the one that called it (Figure 130). In older PLCs subroutines do not accept input parameters. However in **RSLogix 5000** they do (Figure 131).

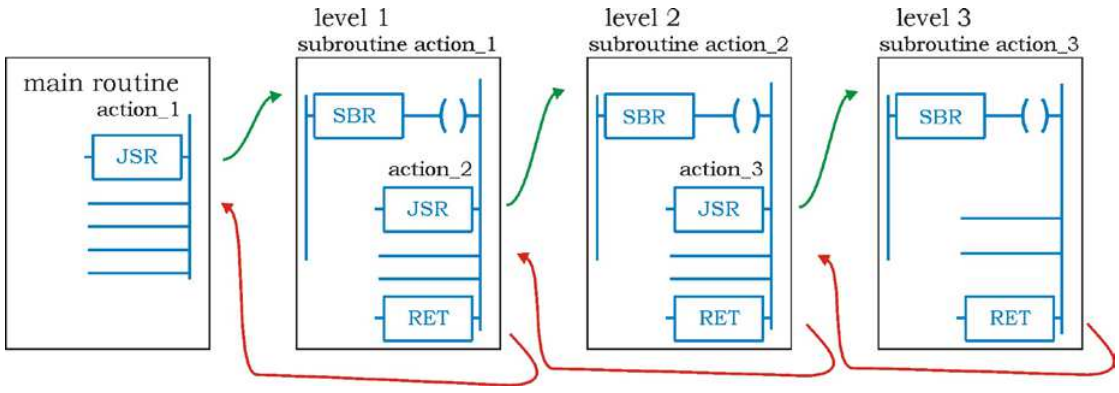


Figure 130 Nested subroutines

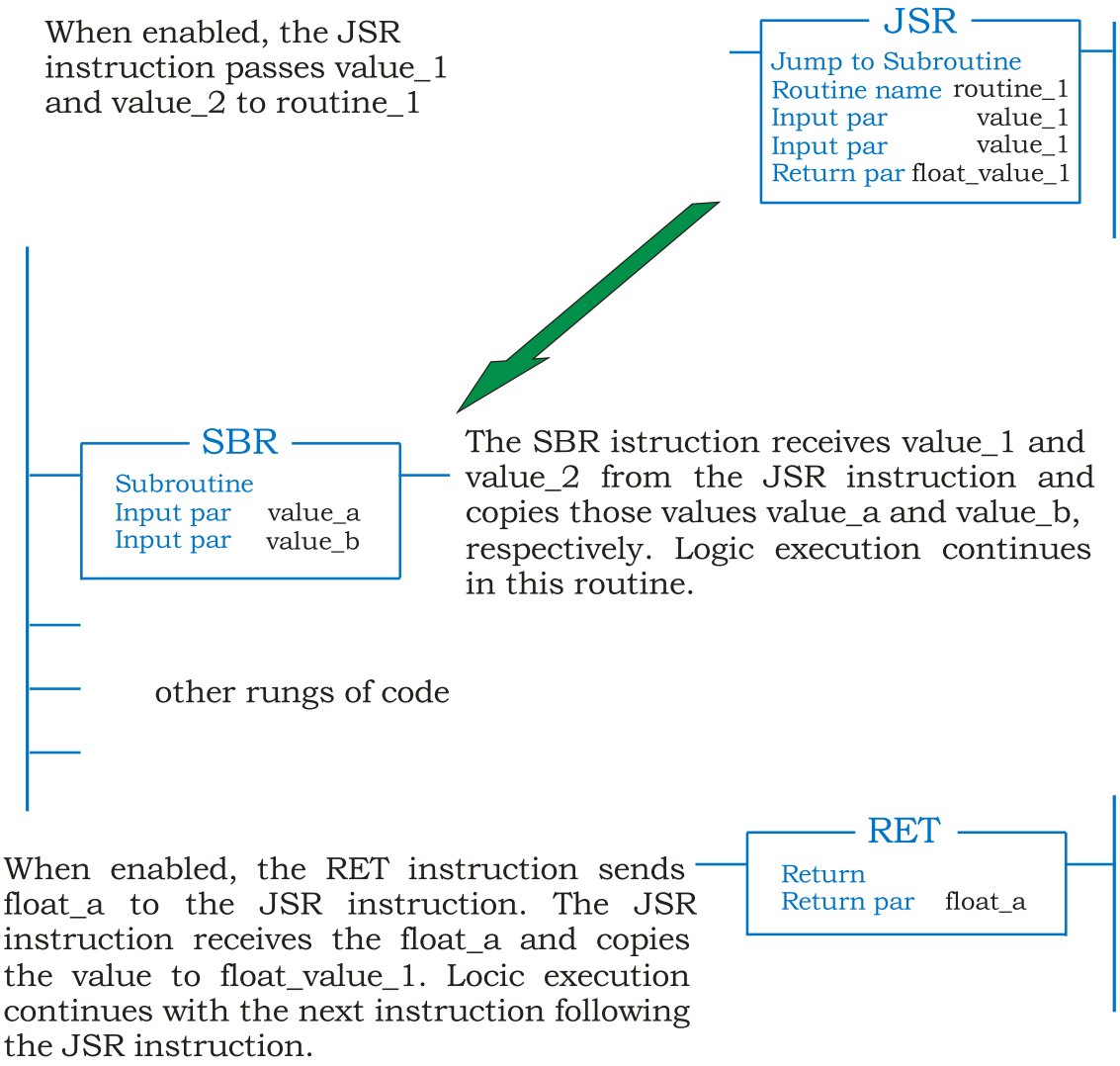


Figure 131 Subroutine execution

Figure 132 Program Control tab

Program control instructions can be found under the **Program Control** tab (Figure 132). These are:

- **LBL** - (Figure 133) Label identifies portion of code by label name for jump instructions.

Figure 133 LBL instruction

- **JMP** - (Figure 134) Jump jumps to a specified label and skips the code between. You can jump forward or back. If the jump condition is disabled it does not affect program execution.

Figure 134 JMP instruction

- **SBR** - (Figure 135) Subroutine signifies the beginning of a subroutine. It passes data to and executes a routine. Input par holds the tags for the subroutine. You can pass more than one input parameter.

Figure 135 SBR instruction

RET - (Figure 136) Return signifies the end of a subroutine and returns the result.

Figure 136 RET instruction

- **JSR** - (Figure 137) Jump To Subroutine jumps execution to a different routine and passes input parameters.

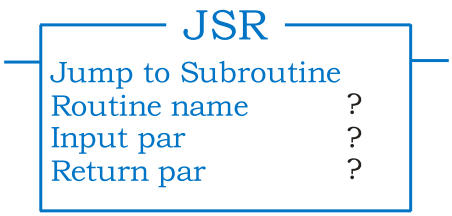


Figure 137 JSR instruction

- **MCR** - (Figure 138) Master Control Reset used in pairs, creates a program zone that can disable all rungs within the **MCR** instructions (Figure 139, 140).

Figure 138 MCR instruction

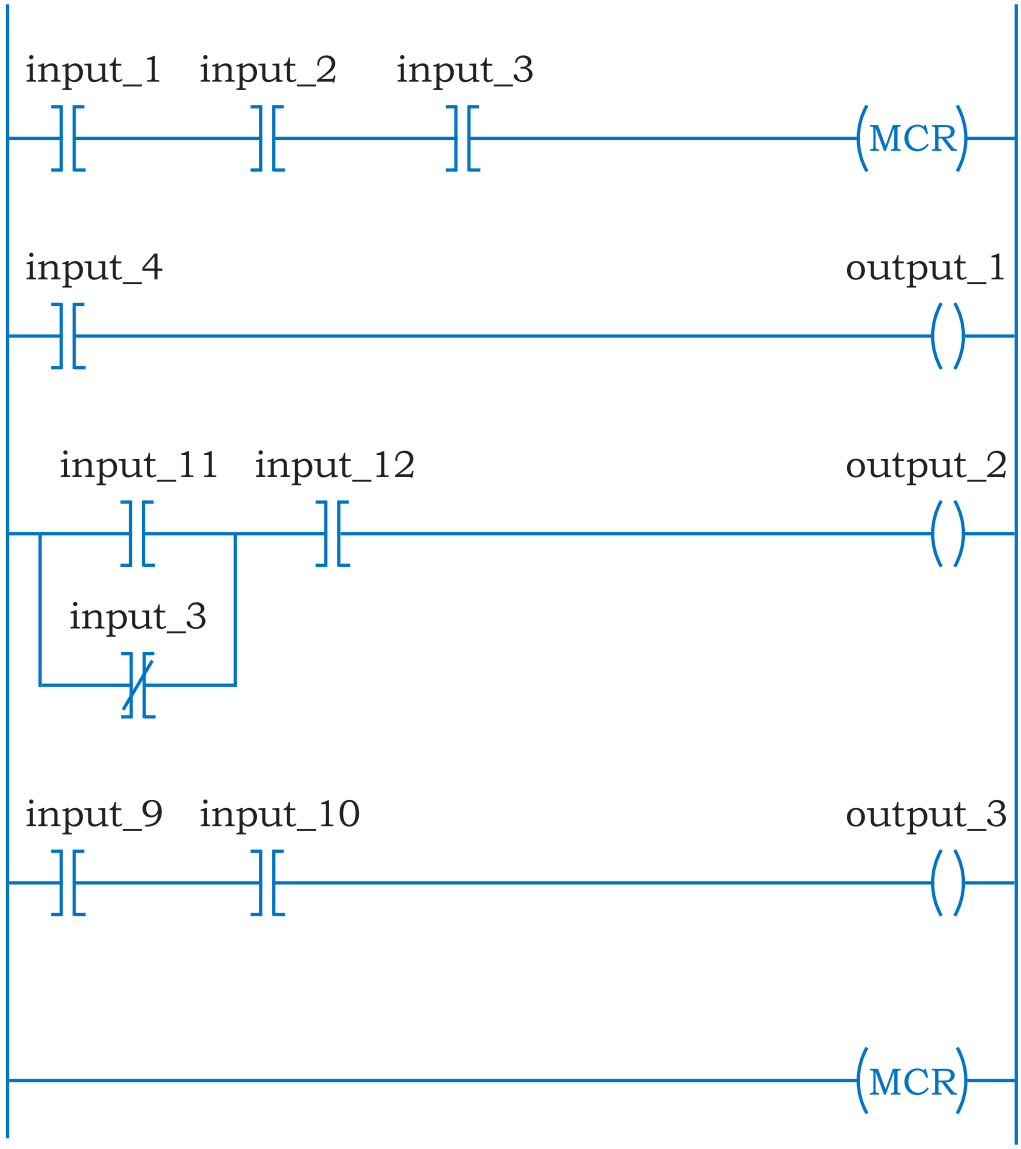


Figure 139 MCR example

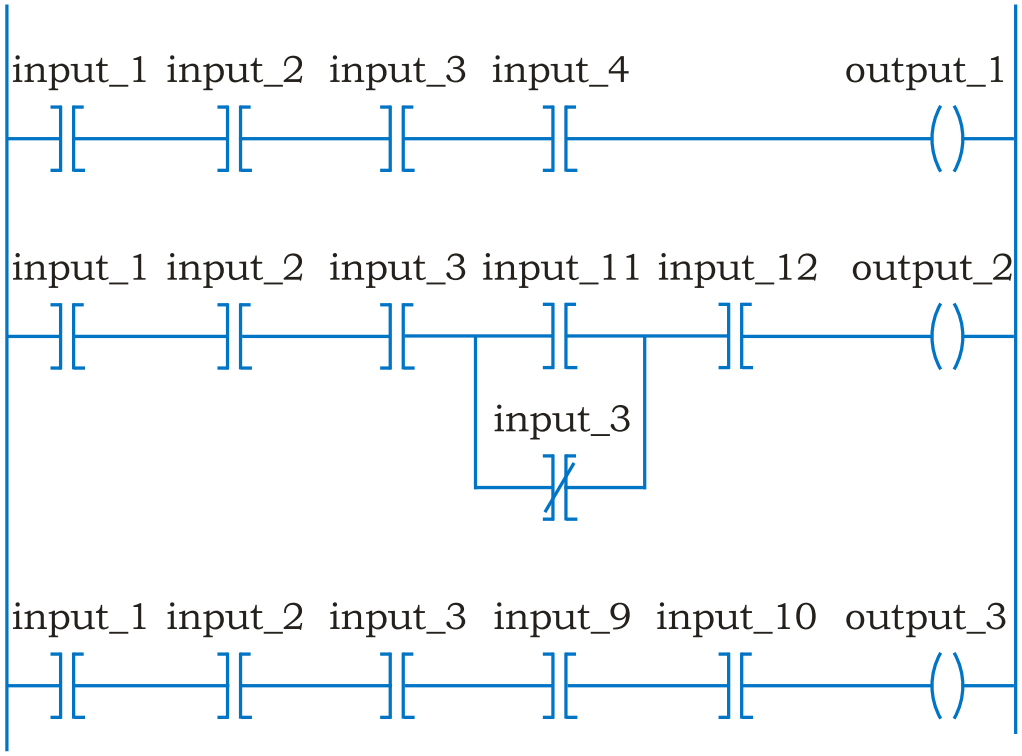


Figure 140 MCR example

These are the most commonly used program control instructions. Now let's see an example of their use.

2.5.5. Exercise VI: Bottling Line

In this exercise the task is to control a bottling plant (Figure 141).

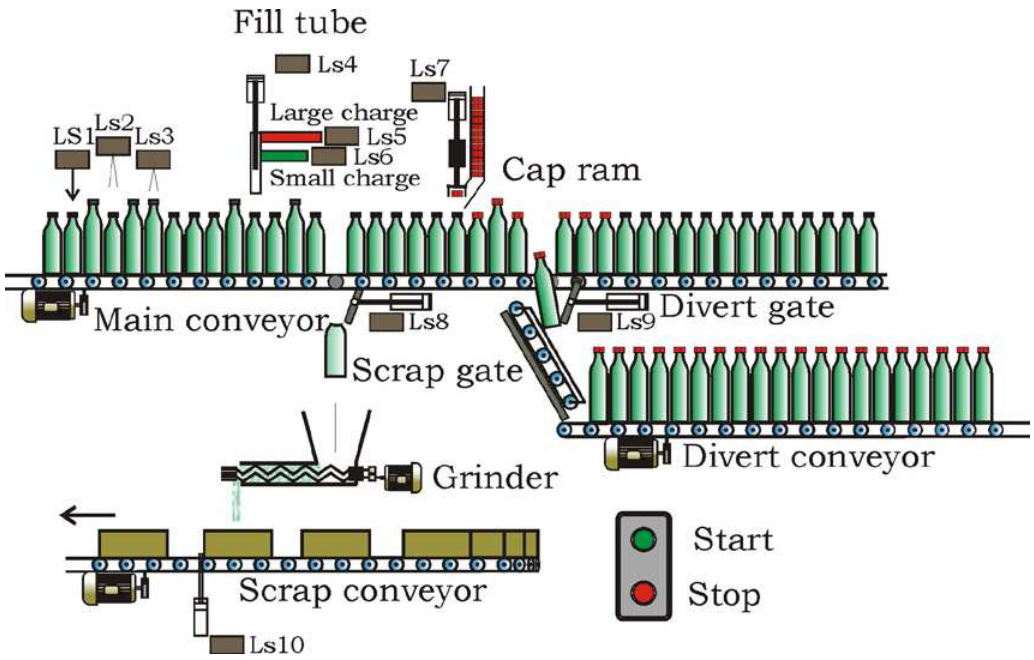


Figure 141 Bottling plant

The inputs are shown in table 16.

Table 16 Inputs of the bottling plant system

Inputs (switching elements)	Name	Type	Identifier
Pushbutton	Start	NO	Local:1:I.Data.0
Pushbutton	Stop	NC	Local:1:I.Data.1
Bottle exist	LS1	NO	Local:1:I.Data.2
Large bottle	LS2	NO	Local:1:I.Data.3
Broken bottle	LS3	NO	Local:1:I.Data.4
Fill tube out	LS4	NO	Local:1:I.Data.5
Large charge out	LS5	NO	Local:1:I.Data.6
Small charge out	LS6	NO	Local:1:I.Data.7
Cap ram out	LS7	NO	Local:1:I.Data.8
Scrap gate open	LS8	NO	Local:1:I.Data.9
Diver gate open	LS9	NO	Local:1:I.Data.10
Box	LS10	NO	Local:1:I.Data.11

The outputs are shown in table 17.

Table 17 Outputs of the bottling plant system

Output devices	Name	Identifier
Contactor	Main Conveyor	Local:2:O.Data.0
Contactor	Scrap Conveyor	Local:2:O.Data.1
Contactor	Divert Conveyor	Local:2:O.Data.2
Contactor	Grinder	Local:2:O.Data.3
Valve	Scrap Gate	Local:2:O.Data.4
Valve	Divert Gate	Local:2:O.Data.5
Valve	Fill Tube	Local:2:O.Data.6
Valve	Large Charge	Local:2:O.Data.7
Valve	Small Charge	Local:2:O.Data.8
Valve	Cap ram	Local:2:O.Data.9
Indicator Lamp	Run Lamp	Local:2:O.Data.10

The task we are charged with is the following:

- Operate the bottling plant to fill large bottles with large charges, small bottles with small charges, grind broken bottles and cap filled bottles.
- Small bottles have 2/3 times less material than large bottles. Keep this in mind when you grind them. Once ground up each box can hold the total volume of 9 large bottles.
- Start tracking the bottles from **LS1** (**0**). Count out when the bottle will reach the gates, fill tube, cap ram.
- The **Start** button starts the bottling process.
- The **Stop** button stops the bottling process.

- The **Run Lamp** is active while the bottling process is active.
- Small bottles remain on the main conveyor, while large bottles go to the divert conveyor

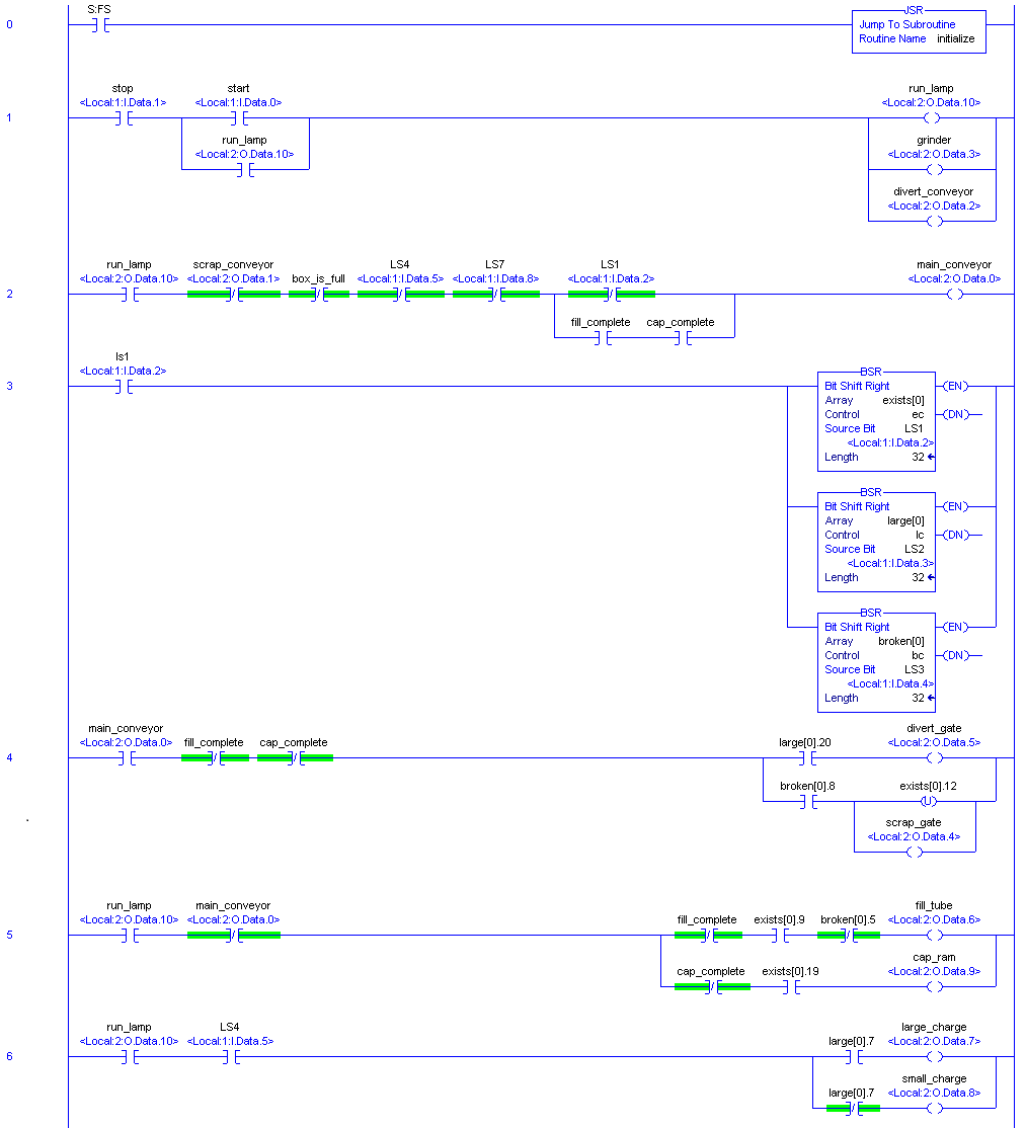


Figure 142 Bottling plant solution

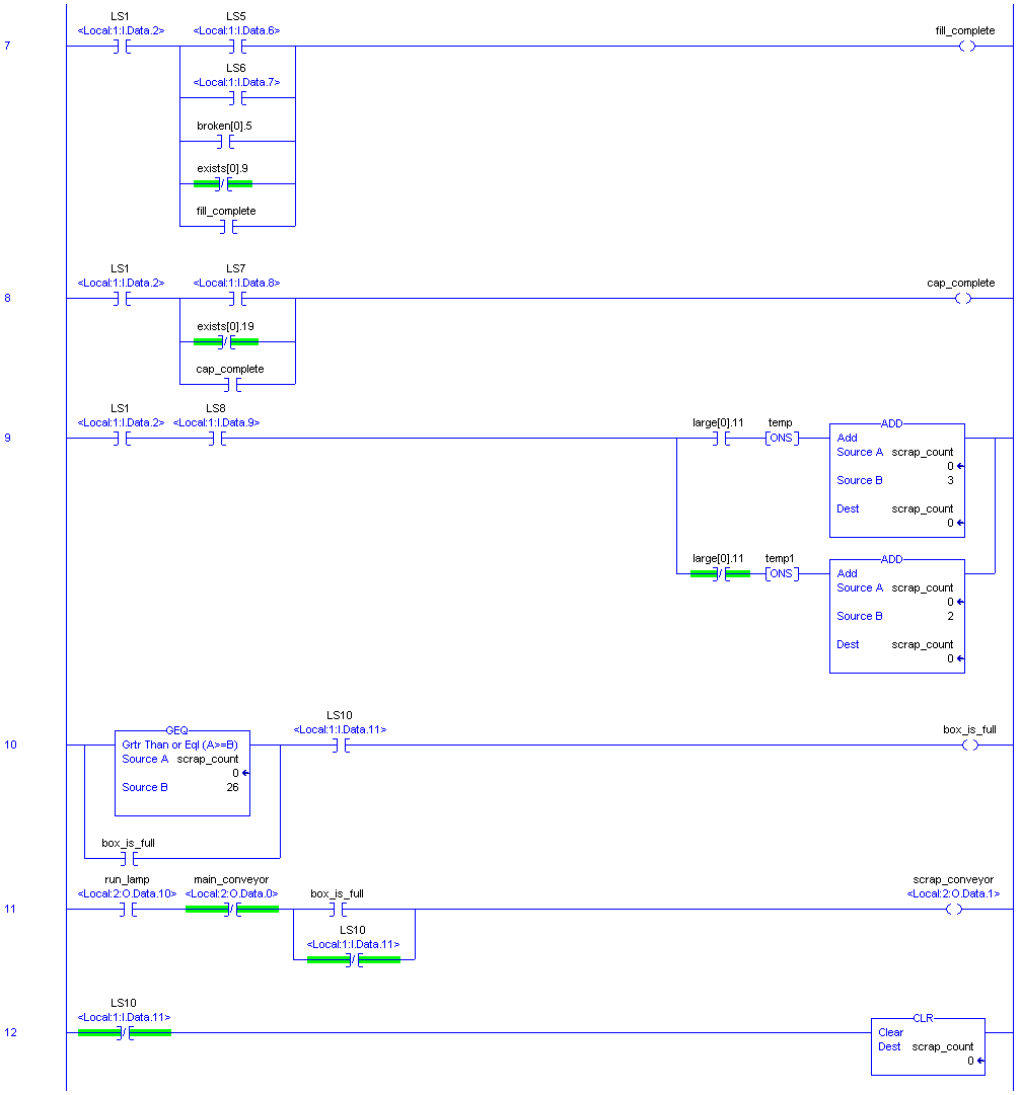


Figure 143 Bottling plant solution continued

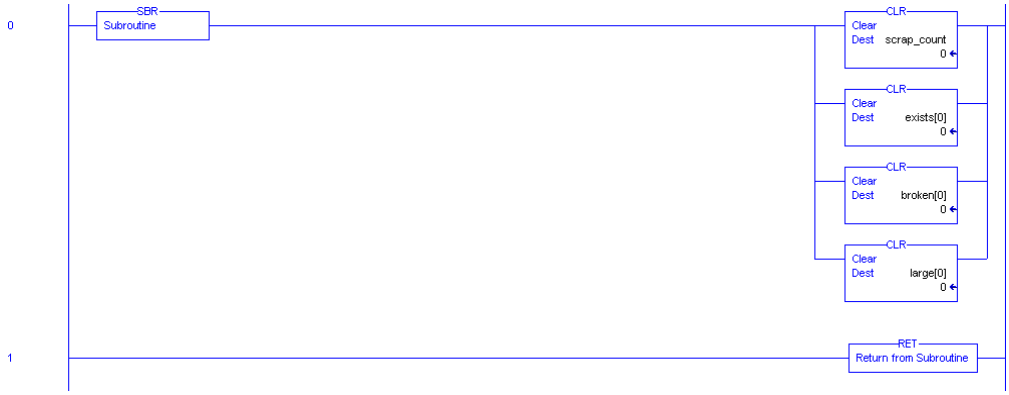


Figure 144 Initialize subroutine

Figures 142-144 show the solution to the exercise. Let's analyze the program.

Line 0 on the main program calls the initialize subroutine on the first scan. After the first scan it is not executed.

Line 1 is a self holding circuit which the **Start** button sets to **TRUE** and the **Stop** button sets to **FALSE**. It turns on the **Run Lamp**, the grinder and the divert conveyor.

Line 2 is responsible for the main conveyor. The main conveyor needs to run unless the fill tube or cap ram are extended since these will break bottles. We also need to stop the main conveyor if the box is full or the scrap conveyor is running since then the ground up bottles will end up on the factory floor.

Line 3 keeps track of the bottles. It uses the 3 **BSL** instructions to read data from the exists sensor (**LS1**), large bottle sensor (**LS2**) and broken bottle sensor (**LS3**). Each new bottle detected by **LS1** will trigger a shift. The **BSL** instructions are using 1 dimensional arrays with 1 **DINT** element. No more is needed since a **DINT** is 32 bits and we don't really need to track more than 24 bottles. If we needed to track more than 32 than we would use a 1 dimensional array with more than 1 **DINT** element.

Line 4 handles the scrap gate and divert gate. It also deletes a those bottles from the exists array which are ground up. We are checking **large[0].20** from since the gate is at the 21th bottle from the large sensor (**LS2**) and we want to know if there is a large bottle at the divert gate. Similarly we are

checking the **broken[0].8** since the scrap gate is at the 9th bottle from the broken sensor (**LS3**).

Line 5 handles the cap ram which is extended if there is a bottle under it. It also handles the fill tube which is extended if there is a not broken bottle under it.

Line 6 handles the bottle filling. If the fill tube is extended (LS4) it releases a large charge if there is a large bottle under the fill tube and a small charge if there is a small bottle under the filler.

Line 7 and 8 hold 2 bits which represent when the capping and filling are done.

Line 9 keeps track of the amount of ground up bottles in a box. It adds 3 for every large bottle and 2 for every small bottle.

Line 10 is a self holding circuit. It is responsible for a bit that shows when the box is full.

Line 11 is also a self holding circuit. It is responsible for the scrap conveyors movement.

Line 12 sets the amount of ground up bottles in the box to 0 once the new box arrives.

The subroutine has 2 lines but only the first one is important. It clears all arrays when the program starts.

Literatures

1. W. Bolton: W. Bolton: Programmable logic controllers 4th edition 2006. ELSEVIER ISBN-13: 978-0-7506-8112-4, ISBN-10: 0-7506-8112-8
2. Dr. Hugh Jack: Automating manufacturing systems with PLCs
3. Allen-Bradley company: Understanding and applying micro programmable controllers