ECET 4530

Industrial Motor Control

Introduction to Ladder Logic Programming I

(in the RSLogix environment)

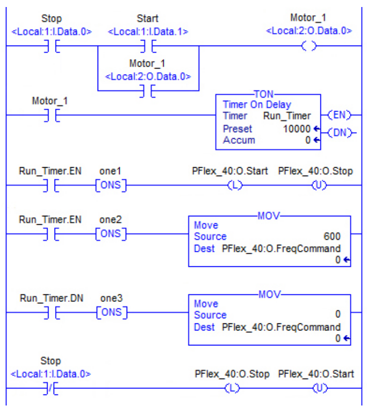
1

Ladder Logic Programming

<u>Ladder Logic</u> is one of many different languages* that can be utilized when programming a PLC.

Ladder Logic programs take the form of a <u>Ladder Diagram</u>, the contents and structure of which defines the PLC's operation.

A Graphical User Interface (GUI) is typically utilized in order to provide a simple drag-and-drop method for creation of the ladder diagram.



* – other languages include Function Block Diagram (FBD), Instruction List (IL), Structured Text (ST), Sequential Function Chart (SFC), Continuous Function Chart (CFC), as well as high-level programming languages like Automation Basic and C++.

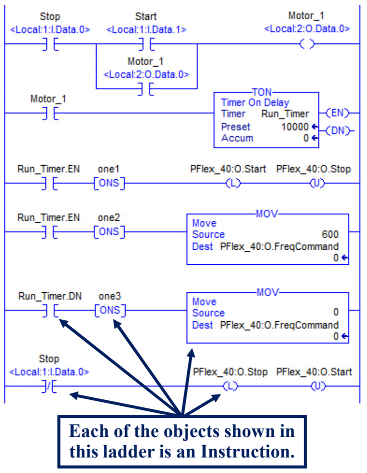
2

Presentation Format – PPT I

This presentation on Ladder Logic Programming will begin by introducing the overall structure of a Ladder Diagram.

Included within the introduction is a brief discussion on the general types of Instructions that will appear within a Ladder Diagram, and the overall flow of the program as it is executed.

Several commonly used Instructions will then be discussed in detail, including both the format of the instructions and their operation when utilized within a Ladder Logic Program.

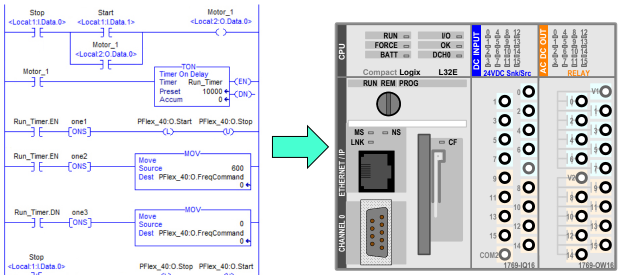

Stop <Local:1:I.Data.0>
Start <Local:1:I.Data.1>
Motor_1 <Local:2:O.Data.0>
Motor_1 <Local:2:O.Data.0>
Motor_1
TON Timer On Delay Timer Run_Timer Preset 10000 Accum 0
EN
DN
Run_Timer.EN one1 ONS
PFlex_40:O.Start L
PFlex_40:O.Stop U
Run_Timer.EN one2 ONS
MOV Move Source 600 Dest PFlex_40:O.FreqCommand 0
Run_Timer.DN one3 ONS
MOV Move Source 0 Dest PFlex_40:O.FreqCommand 0
Stop <Local:1:I.Data.0>
PFlex_40:O.Stop L
PFlex_40:O.Start U
Each of the objects shown in this ladder is an Instruction.

3

Presentation Format – PPT II

The second part of this presentation will focus on the practical implementation of Ladder Logic Programming in order to control the operation of a PLC.

The method for linking the operation of the PLC's Input and Output Ports to the operation of the Instructions within a Ladder Diagram will first be presented, after which several basic control system tasks will be implemented with the use of Ladder Logic Programming.



Allen-Bradley CompactLogix PLC

4

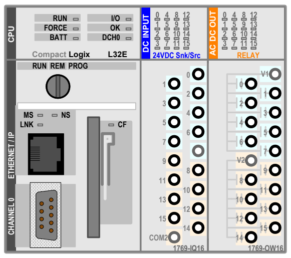
RSLogix/Studio 5000

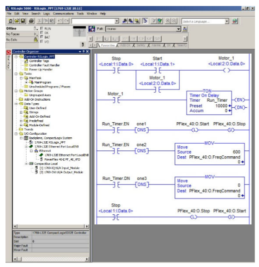
<u>RSLogix 5000</u>* is a platform developed by Rockwell Software for the programming of Allen-Bradley PLCs.

RSLogix 5000 (v.21) is shown in this presentation since we will utilize that software to program the PLCs in the lab.

Note that Ladder Logic is not exclusive to the RSLogix platform.

It is a standardized programming language (IEC 61131-3) that has been adopted by all PLC manufacturers because of its structural similarity to relay-logic-based control circuits.





RSLogix 5000 Software Platform

*** – Studio 5000 is the new version of the RSLogix platform that was developed to support PLCs with multi-core processors. Although this presentation discusses RSLogix 5000, the contents also applies to the Studio 5000 platform.**

(*Part A*)

Ladder Diagrams

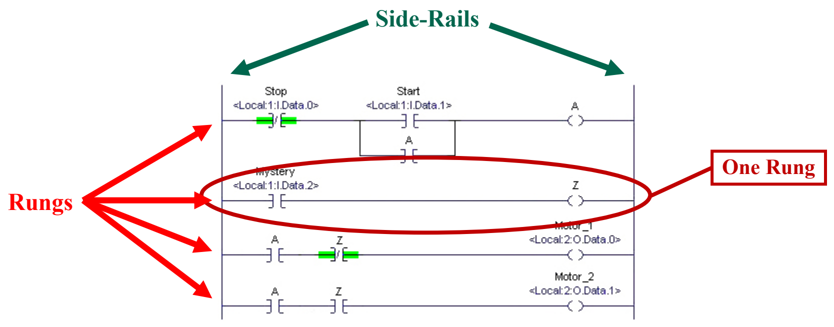
♦

Layout, Instruction Types, Evaluation, and Order of Execution

Ladder Diagrams

Ladder Diagrams are graphical representations of a ladder logic program.

They are named such because of their ladder-like appearance, with two vertical Side-Rails and multiple horizontal Rungs.



7

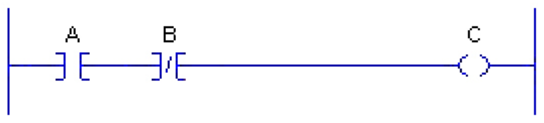
Ladder Diagrams – Rungs

The Rungs of the Ladder Diagram contain multiple Instructions that, when combined together, can provide the function of one or more lines of code in a text-based programming language.

For example, the function

let C=1 if (A=1 and B=0) else let C=0

can be implemented using either Ladder-Logic or C++ as:



Ladder Logic Rung

```
If ( A == 1 && B == 0) {
   C = 1
}
else {
   C = 0
}
```

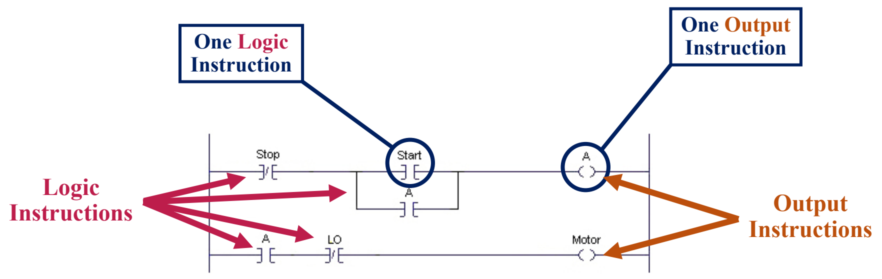
C++ If-Else Statement

8

Ladder Logic Instructions

Ladder Logic **Instructions** can be separated into two primary categories:

- Output Instructions
- Logic (Input) Instructions



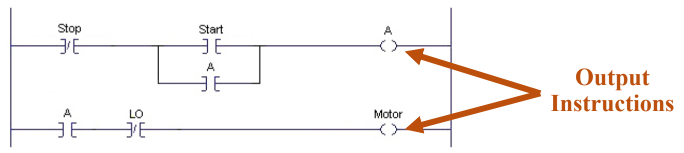
9

Output Instructions

Output Instructions perform a task.

The task performed may be simple or complex depending on the specific instruction, such as:

- Changing the Value of a Number Stored in Memory
- Turning ON or OFF one of the PLC's Output Ports
- Operating as a Timer or a Counter
- Energizing or Changing the Output Frequency of a VFD



10

Output Instructions

Output Instructions perform a task.

The task performed may be simple or complex depending on the specific instruction, such as:

- Changing the Value of a Number Stored in Memory
- Turning ON or OFF one of the PLC's Output Ports
- Operating as a Timer or a Counter
- Energizing or Changing the Output Frequency of a VFD

Note that all of these tasks involve manipulating the value of numbers that are stored in the memory of either the PLC or an external device.

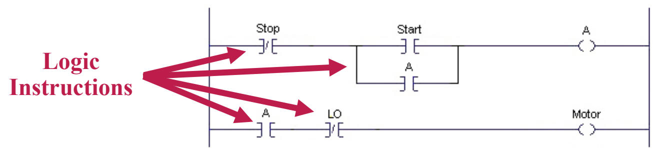
11

Logic Instructions

Logic Instructions provide the functional logic that controls the operation of the Output Instructions.

A Logic Instruction takes on either a TRUE or a FALSE state depending on the current state of its associated parameters.

In general, the "associated parameters" are one or more values stored in specific memory locations that are directly linked to a specific logic instruction.



12

Rung Requirements

Each rung must have at least one Output Instruction.

An Output Instruction must occupy the right-most position* on the rung.

Multiple Output Instruction may be placed in-series and/or in-parallel on the same rung.

Multiple Logic Instructions may be placed in-series and/or in-parallel on the same rung.

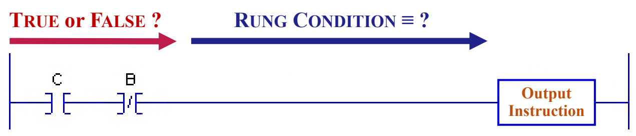
Each rung is not required to have any Logic Instructions.

* – If multiple Output Instructions are placed on the same rung, Logic Instructions may be placed between those instructions provided that the right-most position on the rung is occupied by an Output Instruction. Note that rungs configured in this manner will not be covered in this presentation.

Rung Condition

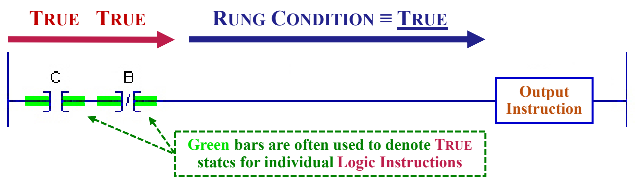
Rung Condition is a TRUE/FALSE logic state that is based on the state and placement of a rung's Logic Instructions and whether or not those instructions provide at least one path through Logic Instructions that are all TRUE, beginning from the left side-rail and progressing to the right.

It is actually the Rung Condition that governs the operation of any Output Instructions located on a specific rung.



TRUE Rung Condition

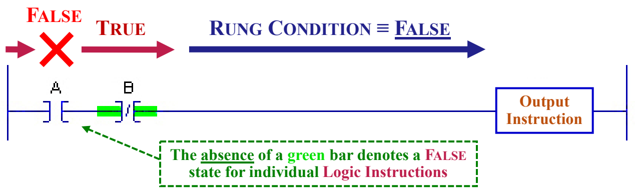
An Output Instruction experiences a TRUE Rung Condition when the rung's Logic Instructions provide a "TRUE path" from the left-hand side-rail of the ladder to the left side of the Output Instruction.


TRUE TRUE
RUNG CONDITION ≡ TRUE
C
B
Output Instruction
Green bars are often used to denote TRUE states for individual Logic Instructions

15

FALSE Rung Condition

An Output Instruction experiences a FALSE Rung Condition when the rung's Logic Instructions prevent a "TRUE path" from the left-hand side-rail of the ladder to the left side of the Output Instruction.

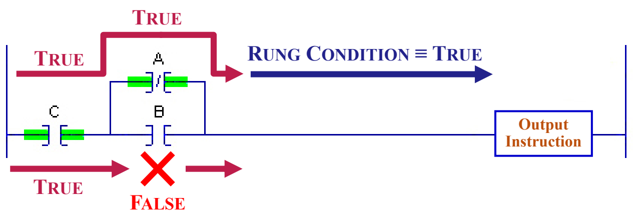

FALSE
TRUE
RUNG CONDITION ≡ FALSE
A
B
Output Instruction
The absence of a green bar denotes a FALSE state for individual Logic Instructions

16

Rung Condition

Note that the Logic Instructions placed on a rung may offer more than one potential path from the left-hand rail to an Output Instruction.

As long as at least one "TRUE path" exists, the Rung Condition for an Output Instruction is considered to be TRUE.

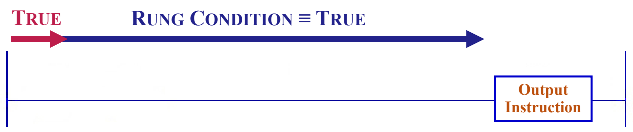


17

Empty Rung Condition

When determining the Rung Condition, a rung is assumed to begin with a TRUE logic state at the left side-rail.

Because of this, a rung with no Logic Instructions will always return a TRUE Rung Condition.



18

Rung Condition Evaluation Order

Rung Condition is <u>not</u> affected by an Output Instruction when progressing from left-to-right across a specific rung.

Thus, if a rung contains multiple Output Instructions, then:

- the Rung Condition for the left-most Output Instruction is determined first and the operation that instruction is completed, after which
- the process repeats for each additional Output Instruction on the rung until the right side-rail is reached.

19

Rung Condition Evaluation Order

Even if the operation of an Output Instruction would change the state of the previously-evaluated Logic Instructions, those Logic Instructions will <u>not be re-evaluated</u> and the previously-determined Rung Condition (to the left) will <u>remain unchanged</u> <u>until</u> the next time the rung is executed.

On the other hand, an Output Instruction may affect the state of any Logic Instructions that are placed to its right on a rung, in-turn possibly affecting the Rung Condition that is experienced by any additional Output Instructions that are also placed to its right[*].

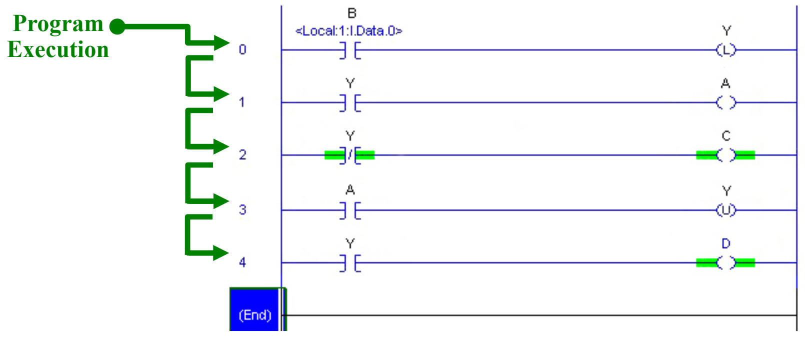
[*] – This is a complex situation that will not be covered in this presentation.

20

Ladder Logic Program Execution

When a PLC is switched to RUN mode*, the controller executes its ladder sequentially from top (Rung 0) to bottom (End), in the order that the rungs appear within the diagram.

* – The PLC has two primary modes of operation, PROGRAM and RUN. When switched to PROGRAM mode, the controller stops executing its Ladder Logic program.

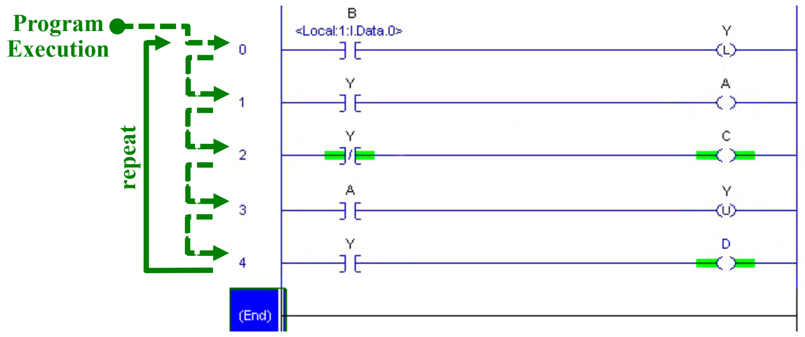


21

Ladder Logic Program Execution

When the controller reaches the End rung, it jumps back to the top of the ladder (Rung 0) and the process begins again.

The controller will repetitively keep stepping through the ladder as long as the PLC remains in RUN mode.

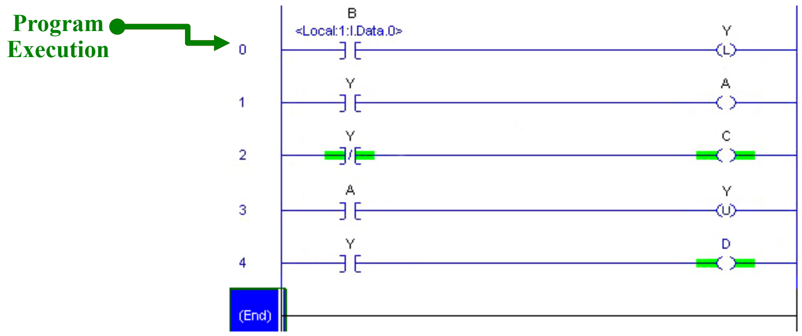


22

Individual Rung Execution

Thus, beginning with Rung 0, the controller:

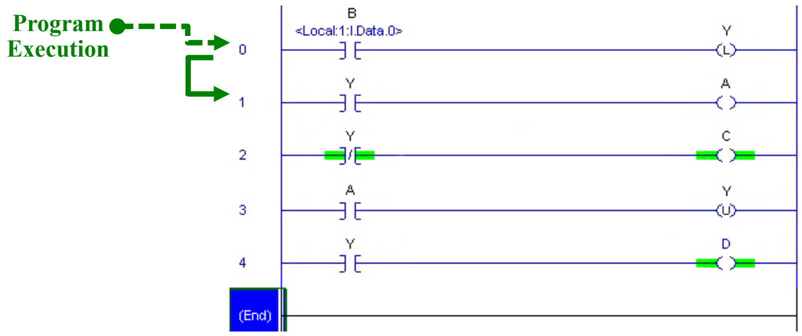
- **Determines the Rung-Condition based on the state of the rung's Logic Instructions, and**
- **Completes any required tasks based on the Rung-Condition and the Output Instruction(s) on that rung.**



23

Ladder Diagram Order of Execution

Once the controller completes the execution of a rung, it then moves to and executes the next rung by determining the Rung Condition on that rung and then applying the results to that rung's Output Instruction(s).

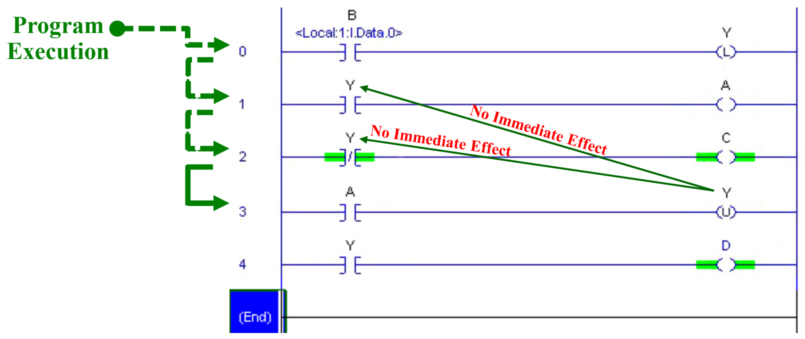


24

Ladder Diagram Order of Execution

Although the execution of a rung may affect the state of the Logic Instructions on any previously-executed rungs, those changes are not acted upon until those rungs are re-executed.

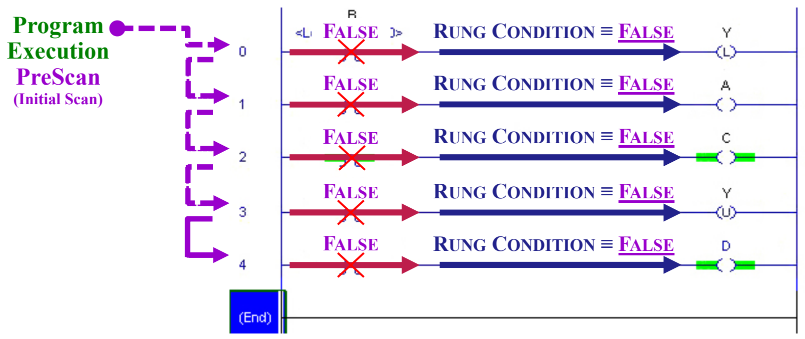
I.e. – Once executed, a rung will not be re-evaluated until after the entire ladder has been executed and the controller sequentially steps through to that rung again.



25

PreScan Mode

Note that, if the PLC transitions from Program → Run mode or if the PLC powers up in Run mode, the first (initial) scan of the ladder diagram is executed in PreScan mode, during which all Logic Instructions return a False state, in-turn resulting in all False Rung Conditions.

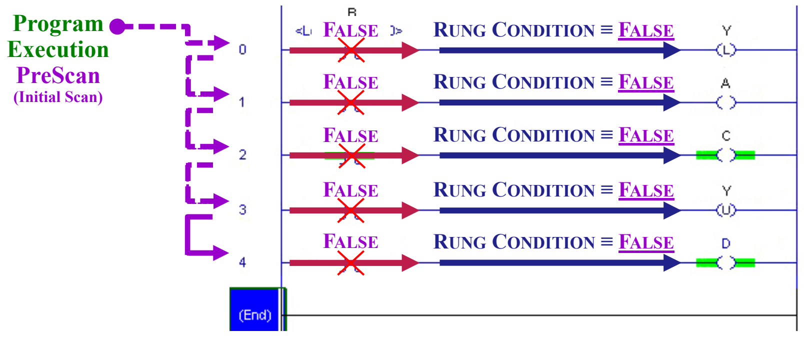


26

PRESCAN Mode

Thus, for the first scan of the ladder, the operation of every Output Instruction is based upon a FALSE Rung Condition.

The significance of PRESCAN mode will not be apparent until the detailed operation of several Instructions is understood.



27

(*Part B*)

Ladder-Logic Instructions

♦

Detailed Operation and Examples

28

Ladder Logic Instructions

The detailed operation of three, commonly used, Ladder Logic Instructions will now be presented:

- **-] [- the Examine if Closed (XIC) instruction,**
- **-]/[- the Examine if Open (XIO) instruction, and**
- **-( )- the Output Energize (OTE) instruction.**

Note – an analogy is often made between the look/operation of these instructions and that of a relay's Normally-Open (NO) contact, Normally-Closed (NC) contact, and Field Coil.

Although this analogy may be utilized during the associated lecture for this material, the analogy will NOT be discussed within this presentation because it can lead to several common misconceptions regarding their operation within a Ladder Diagram; instead, they will always be presented and discussed as "Instructions within a Ladder Logic Program".

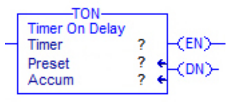
29

Icons Used for Ladder Instructions

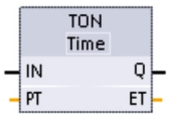
Note that the <u>icon</u> shown in this presentation for a various instruction will be the version displayed within the RSLogix (Allen-Bradley / Rockwell Software) environment.

Although other manufacturers (Siemens, Automation Direct, etc.) may use different icons, the overall operation of the various instructions should be consistent across the various platforms.

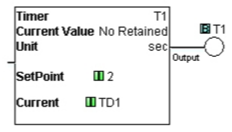
Shown below are some of the various Icons utilized for an "On-Delay Timer" in a Ladder Diagram:



RSLogix (Allen-Bradley)

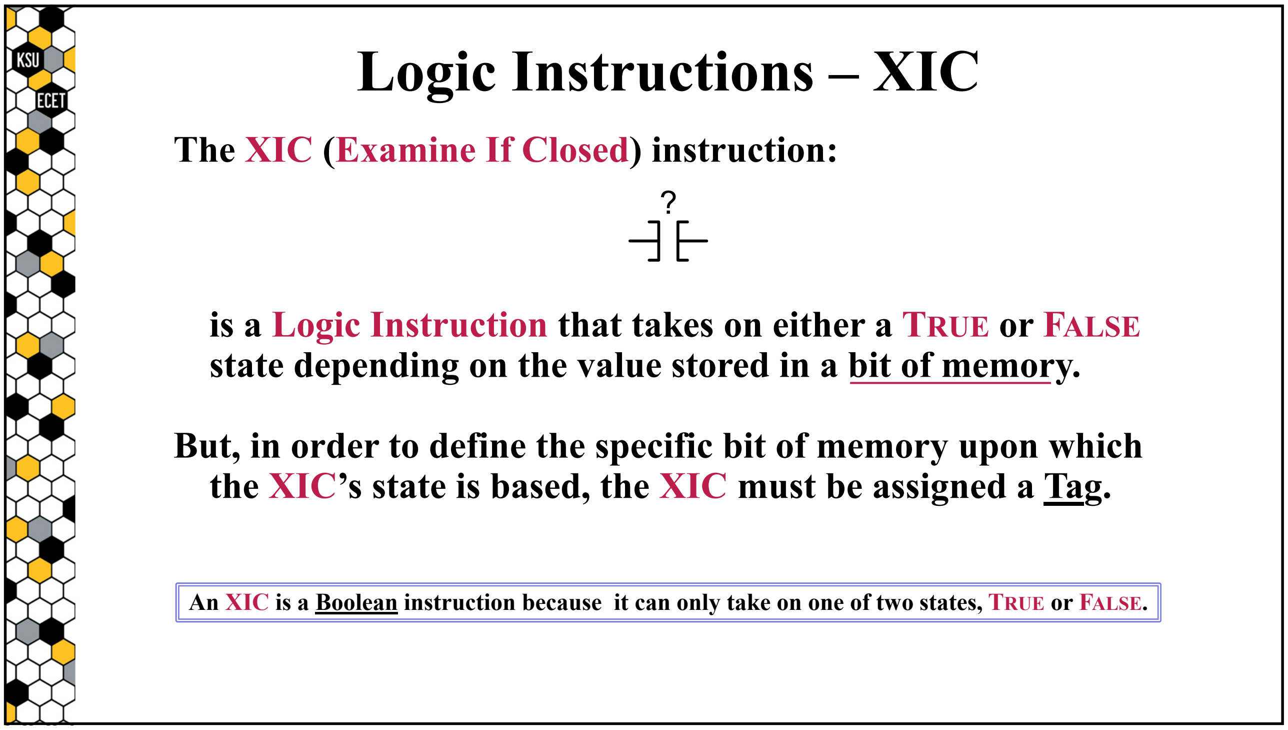


Siemens



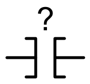
Automation Direct

30



Logic Instructions – XIC

The **XIC** (**Examine If Closed**) instruction:

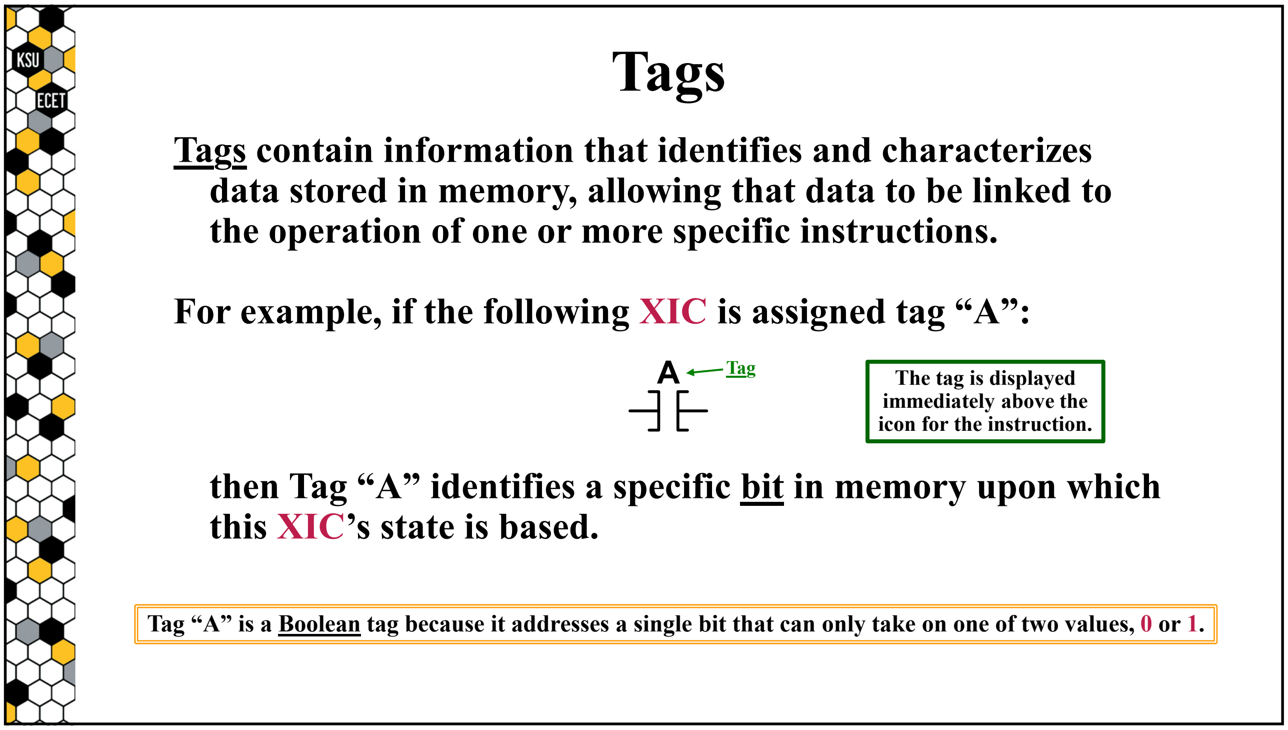


is a **Logic Instruction** that takes on either a **TRUE** or **FALSE** state depending on the value stored in a **bit of memory**.

But, in order to define the specific bit of memory upon which the **XIC**'s state is based, the **XIC** must be assigned a **Tag**.

An **XIC** is a **Boolean** instruction because it can only take on one of two states, **TRUE** or **FALSE**.

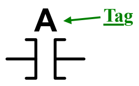
31



Tags

Tags contain information that identifies and characterizes data stored in memory, allowing that data to be linked to the operation of one or more specific instructions.

For example, if the following **XIC** is assigned tag "A":



The tag is displayed immediately above the icon for the instruction.

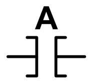
then Tag "A" identifies a specific **bit** in memory upon which this **XIC**'s state is based.

Tag "A" is a **Boolean** tag because it addresses a single bit that can only take on one of two values, **0** or **1**.

32

Logic Instructions – XIC

If tag "A" is assigned to the XIC (Examine If Closed) instruction:



then the logic state (TRUE or FALSE) of that instruction is defined as follows:

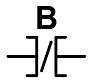
When evaluated: (by the controller)	If bit <u>A=1</u>, then XIC-A ➡ <u>TRUE</u>
	If bit <u>A=0</u>, then XIC-A ➡ <u>FALSE</u>

The XIC is TRUE when A=1.

33

Logic Instructions – XIO

The XIO (Examine If Open) Logic Instruction:



An XIO is also a <u>Boolean</u> instruction.

takes on the opposite state compared to that of an XIC.

If tag "B" is assigned to the XIO, then the logic state of this instruction (TRUE or FALSE) is defined as follows:

When evaluated: (by the controller)	If bit <u>B=0</u>, then XIO-B ➡ <u>TRUE</u>
	If bit <u>B=1</u>, then XIO-B ➡ <u>FALSE</u>

The XIO is TRUE when B≠1.

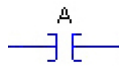
34

Instructions – Icons vs. Current State

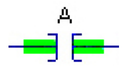
When a PLC is in RUN mode and its ladder diagram is being displayed (in real time) within the RSLogix environment:

- Instructions are always displayed by their standard icon
- The icons of Logic Instructions that return a TRUE state are highlighted with a green-bar behind their icons.

Thus, given an XIC (A —] [—), it will be displayed as follows by the RSLogix software depending on its current logic state:



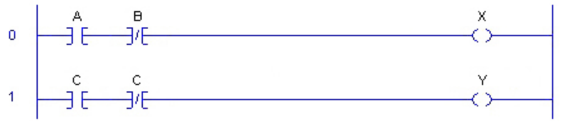
(when the XIC is FALSE)



(when the XIC is TRUE)

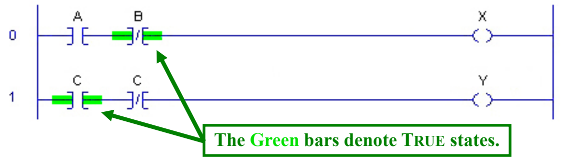
Instructions – Icons vs. Current State

For example, given the bit values: A = 0 B = 0 C = 1
and the following rungs:



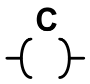
If displayed while the PLC is in RUN mode, then:

Rung-0:	XIC-A ➡ FALSE,	XIO-B ➡ TRUE
Rung-1:	XIC-C ➡ TRUE,	XIO-C ➡ FALSE



Output Instructions – OTE

The following Output Instruction is an OTE (Output Energize) instruction that has been assigned tag "C":



An OTE will either set or reset (store a 1 or 0 in) the bit identified by its assigned tag based on the Rung Condition as follows:

If the Rung Condition is TRUE, then OTE-C sets bit C ➡ 1

(when evaluated by the controller)

If the Rung Condition is FALSE, then OTE-C resets bit C ➡ 0

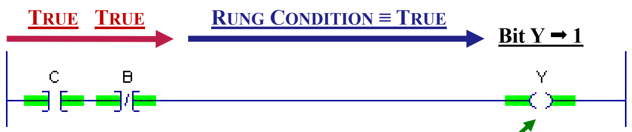
If a bit is "set", it is changed to a one (1).
If a bit is "reset", it is changed to a zero (0).

37

OTE – TRUE Rung Condition

If the Rung-Condition for an OTE is TRUE, then the OTE will set its assigned bit to 1.

Thus, given the bit values: A = 0 B = 0 C = 1
when the following rung is executed, both XIC-C and XIO-B will be evaluated TRUE, resulting in a TRUE Rung Condition, in-turn causing the OTE to set bit Y ➡ 1.



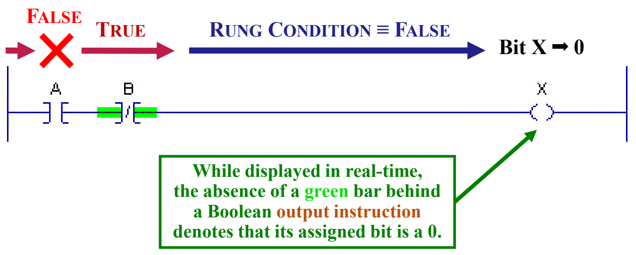
While displayed in real-time, a green bar behind a Boolean output instruction denotes that its assigned bit is a 1.

38

OTE – FALSE Rung Condition

If the Rung-Condition for an OTE is FALSE, then the OTE will reset its assigned bit to 0.

Thus, given the bit values: A = 0 B = 0 C = 1 when the following rung is executed, both XIC-A will be evaluated FALSE, resulting in a FALSE Rung Condition, in-turn causing the OTE to reset bit X ➡ 0.

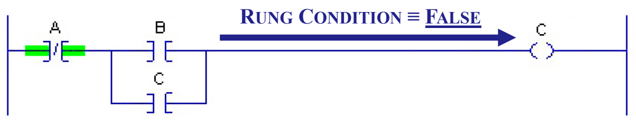

FALSE
TRUE
RUNG CONDITION ≡ FALSE
Bit X ➡ 0
A
B
X
While displayed in real-time, the absence of a green bar behind a Boolean output instruction denotes that its assigned bit is a 0.

39

OTE with Hold-In Example (Part 1)

Part 1 of Hold-In Example

Look closely at the following rung from a ladder diagram:



If bits A, B, and C are all zero (A=B=C=0) when the rung is executed, XIC-B and XIC-C will both be evaluated FALSE, resulting in a FALSE Rung Condition.

Since bit C was already a 0, it isn't changed.

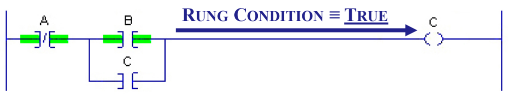
Since the Rung Condition is FALSE, OTE-C resets bit C ➡ 0, and since bit C=0, the results will be the same the next time the rung is executed provided bit B remains zero (B=0).

40

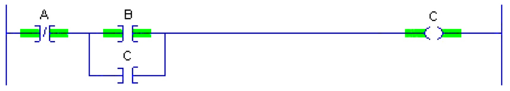
OTE with Hold-In Example (Part 2)

Given the same rung, what if bit B changes to a one (B ➡ 1) and then the rung is executed again (assuming A = C = 0)?

Then, the next time the rung is executed, XIO-A and XIC-B will both be evaluated TRUE, resulting in a TRUE Rung Condition, and…



since the Rung Condition is now TRUE, OTE-C sets bit C ➡ 1.

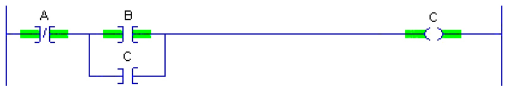


(continued on the next slide)

41

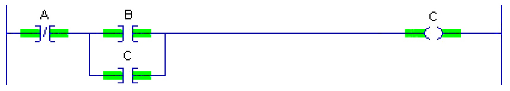
OTE with Hold-In Example (Part 2)

Note that, although bit C is now one (C = 1), XIC-C is still shown to be FALSE (no green bar).



This is to highlight the fact that, once the state of a specific Logic Instruction is evaluated, it will not be re-evaluated until the rung is executed again.

But, once the rung is executed again, assuming that A and B remain unchanged, the rung will appear as:



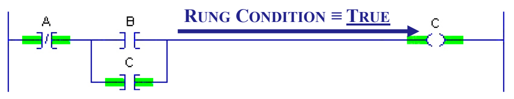
42

OTE with Hold-In Example (Part 3)

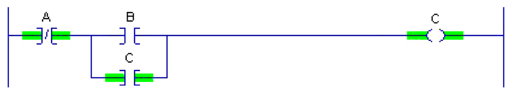
Part 3 of Hold-In Example

Now, given the current state of the rung (A = 0, B = C = 1), what if bit B resets (B ➡ 0) and then the rung is executed again?

Although XIC-B now evaluates FALSE, XIO-A and XIC-C still provide a TRUE path, maintaining the TRUE Rung Condition, and…



since the Rung Condition stays TRUE, OTE-C keeps bit C set.



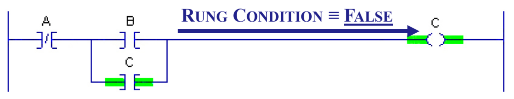
43

OTE with Hold-In Example (Part 4)

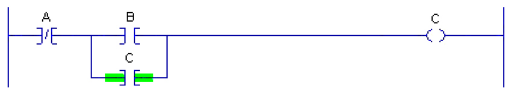
Part 4 of Hold-In Example

But, given the current state of the rung (A = B = 0, C = 1), what if bit A changes to a one (A ➡ 1) and the rung is executed again?

The next time the rung executes, XIO-A will be evaluated FALSE, resulting in a FALSE Rung Condition, and



since the Rung Condition is FALSE, OTE-C resets bit C ➡ 0.



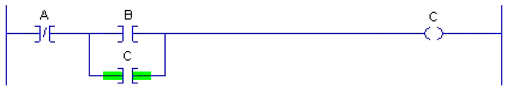
(continued on the next slide)

44

OTE with Hold-In Example (Part 4)

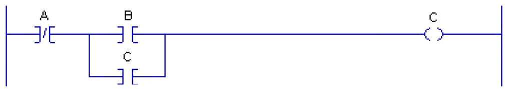
Part 4 of Hold-In Example (continued)

Note that, although C is now a zero (C = 0), XIC-C was still shown to be TRUE (with a green bar).



This is also to highlight that the state of the Logic Instruction is not re-evaluated until the rung is executed again.

But, if the rung is executed again and XIC-C is re-evaluated, the rung will appear as (assuming A and B remain unchanged):



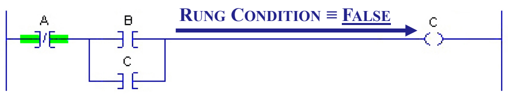
45

OTE with Hold-In Example (Part 5)

Part 5 of Hold-In Example

And finally, what if bit A resets (A ➡ 0) before the rung executes again?

The next time the rung executes, XIO-A will be evaluated TRUE, but the Rung Condition will remain FALSE .



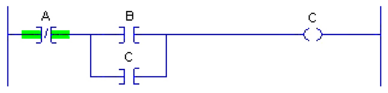
Thus, the rung is back to the same overall state at that existed at the beginning of this example.

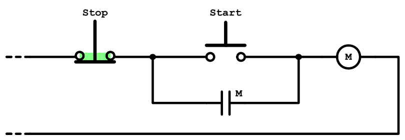
46

OTE with Hold-In Example (Analysis)

Analysis of Hold-In Example

Does the operation of the rung in this example remind you of the operation of a basic stop-start motor controller?





This rung-structure is often used in a PLC-based control system for which a "hold-in" function is required.

The trick is to somehow associate the value of bit A with the state of a "Stop" button, the value of bit B with the state of a "Start" button, and the value of bit C with the state of a contactor's field coil.

How this is done will be discussed in Part B of this presentation.

47

Ladder Logic Instructions

Although many programming tasks can be completed using just the following three instructions:

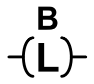
- -] [- the Examine if Closed (XIC) instruction,
- -]/[- the Examine if Open (XIO) instruction, and
- -( )- the Output Energize (OTE) instruction.

a handful of other instructions will also be presented in order to provide a solid foundation of Ladder Logic knowledge upon which the required programming tasks for a wide variety of PLC-based motor control systems can be implemented.

48

Output Latch (OTL)

OTL – Output Latch



An Output Latch (OTL) is an Output Instruction that can set a bit (B ➡ 1) but cannot reset a bit back to zero.

If the Rung-Condition is TRUE, then OTL-B sets* bit B ➡ 1
(when evaluated by the controller)

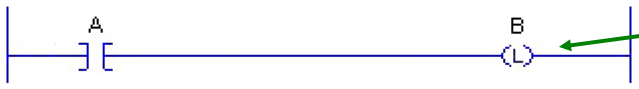
If the Rung-Condition is FALSE, then OTL-B does nothing.

* – It is often stated that an OTL "latches" a bit. The problem with this statement is that, in electronic circuits, a latch is a device that "sets and holds" a voltage at a specific level. It is true that the OTL can "set" a bit, but it doesn't "hold" the bit at a value of one. Instead, it doesn't have the ability to "reset" a bit, so even if the Rung Condition returns FALSE, the bit will remain set until it is actively reset by another instruction.

49

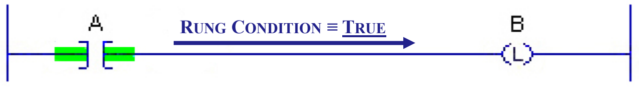
OTL Example

Given the following rung that contains on OTL (Output Latch):

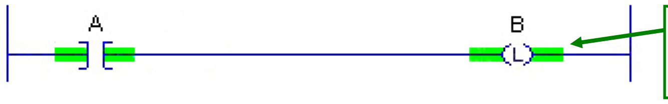


The absence of a green bar behind a Boolean output instruction denotes that its assigned bit is a 0.

If bit A changes to a one (A = 1), then the next time that the rung is executed, XIC-A will be evaluated TRUE, resulting in a TRUE Rung Condition, and



since the Rung Condition is TRUE, OTL-B sets bit B ➡ 1.

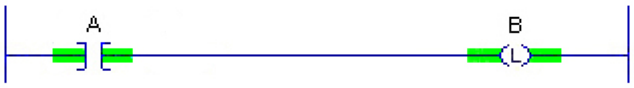


A green bar behind a Boolean output instruction denotes that its assigned bit is a 1.

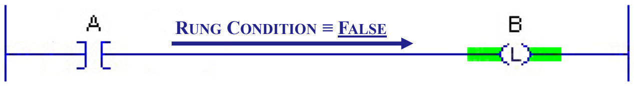
50

OTL Example

On the other hand, given the following rung:



If bit A resets (A = 0), then the next time that the rung is executed, XIC-A will be evaluated FALSE, resulting in a FALSE Rung Condition.

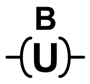


But, OTL-B does nothing when the Rung Condition is FALSE, so bit B remains set (B = 1).

51

Output Unlatch (OTU)

OTU – Output Unlatch



An Output Unlatch (OTU) is an Output Instruction that can reset a bit (B ➡ 0) but cannot set a bit to a one.

If the Rung-Condition is TRUE, then OTU-B resets* bit B ➡ 0 (when evaluated by the controller)

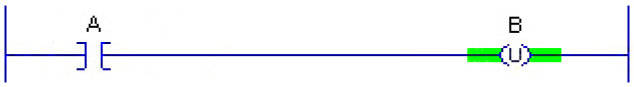
If the Rung-Condition is FALSE, then OTU-B does nothing.

* – Note that the OTU resets a bit when the Rung Condition becomes TRUE. On the other hand, an OTE resets a bit when the Rung Condition becomes FALSE. This is a critical distinction between the abilities of OTUs and OTEs to reset a bit, and often provides a challenge for beginning Ladder Logic programmers.

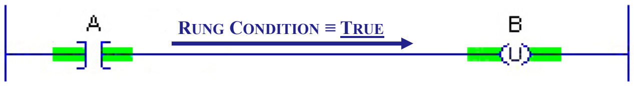
52

OTU Example

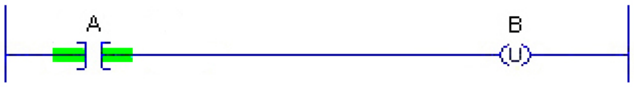
Given a rung that contains on OTU (Output Unlatch) (A=0, B=1):



If bit A changes to a one (A= 1), then the next time that the rung is executed, XIC-A will be evaluated TRUE, resulting in a TRUE Rung Condition, and



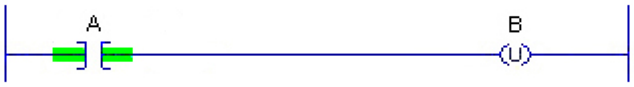
since the Rung Condition is TRUE, OTU-B resets bit B ➡ 0.



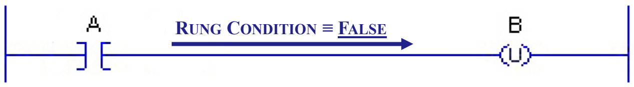
53

OTU Example

Given the same rung when A=1 and B=0:



If bit A resets (A= 0), then the next time that the rung is executed, XIC-A will be evaluated FALSE, resulting in a FALSE Rung Condition.

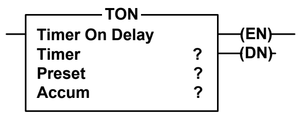


But, OTU-B does nothing when the Rung Condition is FALSE, so bit B <u>remains</u> reset (B=0).

54

On-Delay Timer (TON)

TON – On-Delay Timer



The <u>On-Delay Timer</u> (TON) is an Output Instruction that functions as a "non-retentive" timer.

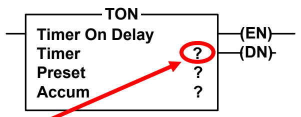
A <u>non-retentive timer</u> is a timer that does not retain its count (accumulator value) if the timer is disabled.

(I.e. – the timer resets to its initial conditions when it is disabled)

55

TON – Base Tag & Sub-Tags

TON – On-Delay Timer



When a <u>Base Tag</u> is created for the TON, the name of which appears in the *Timer* field, several <u>sub-Tags</u> are automatically created by the RSLogix software:

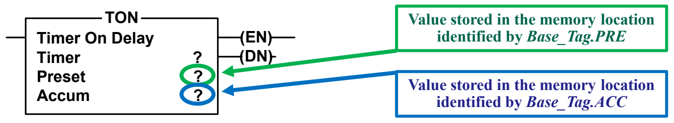
The sub-Tags include:

Base_Tag.PRE	***Base_Tag.EN***	***Base_Tag.TT***
Base_Tag.ACC	***Base_Tag.DN***	

56

TON – Preset & Accumulator

TON – On-Delay Timer


TON
Timer On Delay
Timer ?
Preset ?
Accum ?
EN
DN
Value stored in the memory location identified by *Base_Tag.PRE*
Value stored in the memory location identified by *Base_Tag.ACC*

In addition to the *Timer* field that contains the Base Tag name, there are two user-defined fields shown in the icon:

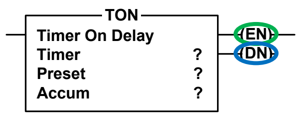
- *Preset* (*.PRE*) – Contains the time-delay value (specified in msec) up to which the TON will count.
- *Accum* (*.ACC*) – Contains the initial time value (also in msec).

Note that the *Accum* field displays the current value stored in the accumulator when viewed "online" (in real time).

57

TON – Operational State Bits

TON – On-Delay Timer



The *.TT* bit is not shown on the TON icon

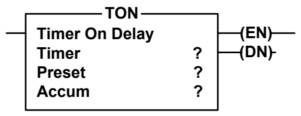
Three sub-Tags characterize the TON's operational state:

- *Base_Tag.EN* – The *.EN* (*Enable*) bit is set to 1 when the TON is enabled and reset to 0 when it's disabled.
- *Base_Tag.DN* – The *.DN* (*Done*) bit is set to 1 when the *Accum* equals the *Preset*, otherwise it's reset to 0.
- *Base_Tag.TT* – The *.TT* (*Time Transitioning*) bit is set to 1 when the TON is actively counting, otherwise it's reset to 0.

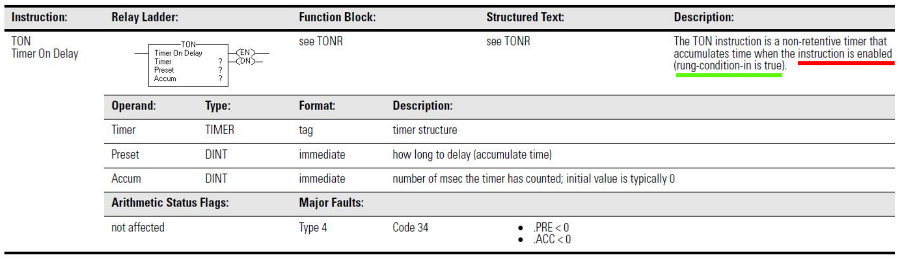
58

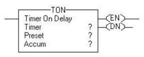
TON – Documentation

TON – On-Delay Timer



The following information can be found by using the Help tab within the RSLogix 5000 software:



Instruction:	Relay Ladder:	Function Block:	Structured Text:	Description:
TON Timer On Delay		see TONR	see TONR	The TON instruction is a non-retentive timer that accumulates time when the instruction is enabled (rung-condition-in is true).

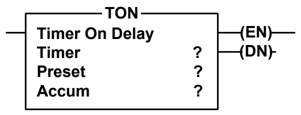
Operand:	Type:	Format:	Description:
Timer	TIMER	tag	timer structure
Preset	DINT	immediate	how long to delay (accumulate time)
Accum	DINT	immediate	number of msec the timer has counted; initial value is typically 0

Arithmetic Status Flags:	Major Faults:		
not affected	Type 4	Code 34	• .PRE < 0 • .ACC < 0

59

TON – Detailed Operation

TON – On-Delay Timer



The TON is "enabled" when its Rung Condition is TRUE.

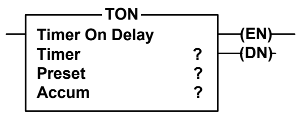
Once enabled, its *Enable* bit is set (.*EN* ➡ 1) and the TON begins "actively" incrementing (accumulating time).

Note that, if the Rung Condition goes FALSE, the TON is disabled, causing its accumulator to return to its initial value (the default is 0) and the *Enable* bit to reset (.*EN* ➡ 0).

60

TON – Detailed Operation

TON – On-Delay Timer



As long as the TON remains enabled, its accumulator will continue incrementing until it reaches the *Preset* value.

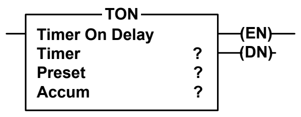
Note that, while the TON is "actively" accumulating time, the *Time Transitioning* bit will be set ($.TT \rightarrow 1$).

But, if the TON is not accumulating time, either because it's disabled or its accumulator has reached the *Preset* value, then the $.TT$ bit will be reset ($.TT \rightarrow 0$).

61

TON – Detailed Operation

TON – On-Delay Timer



Once the TON's accumulator reaches the *Preset* value, it will stop incrementing and the *Accum* value will remain equal to the *Preset* value as long as the timer is enabled.

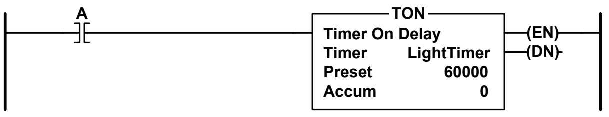
Additionally, as long as $.ACC = .PRE$, the timer's *Done* bit will be set ($.DN \rightarrow 1$).

But, if for any reason $.ACC \neq .PRE$, such as the timer being disabled or reset, the *Done* bit will be reset ($.DN \rightarrow 0$).

62

TON Example

Given the TON that has been placed on a rung with XIC-A:



If the timer is configured as follows:

- the name of the Base Tag is “LightTimer”,
- the *Preset* value is 60,000 (60,000 msec = 60 sec), and
- the initial *Accum* value is 0.

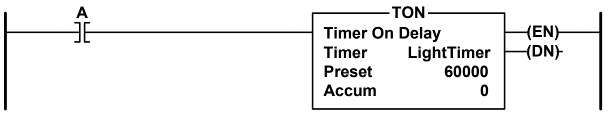
describe, in detail:

1) the timer’s <u>initial conditions</u> while bit A = 0,
2) the operation of the timer when <u>bit A is set</u> (A ➡ 1), and
3) the operation of the timer if <u>bit A is reset</u> (A ➡ 0).

63

TON Example – Initial Conditions

Given the TON that has been placed on a rung with XIC-A:



LightTimer.EN = 0
LightTimer.TT = 0
LightTimer.DN = 0

If bit A is initially zero (A = 0), then:

- XIC-A ≡ FALSE and the Rung Condition is FALSE, and
- the TON is disabled.

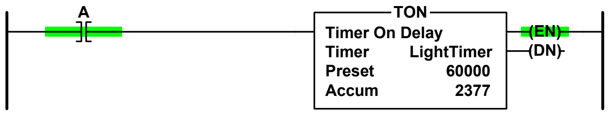
Since the timer is disabled:

- the status bits *Enable*, *Time Transitioning*, and *Done* will all be <u>zero</u> (.*EN* = 0, .*TT* = 0, .*DN* = 0), and
- the accumulator value will remain at <u>zero</u> (.*ACC* = 0).

64

TON Example – Timer Enabled

Given the TON that has been placed on a rung with XIC-A:



LightTimer.EN = 1
LightTimer.TT = 1
LightTimer.DN = 0

If bit A ➡ 1, then the next time the rung is executed, XIC-A will be evaluated TRUE, making the Rung Condition TRUE, and the TON will be <u>enabled</u>.

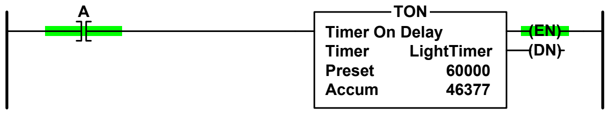
When the <u>timer is enabled</u>:

- the *Enable* bit will be <u>set</u> (.*EN* ➡ 1),
- the accumulator will begin <u>incrementing</u>, and
- the *Time Transitioning* bit will be <u>set</u> (.*TT* ➡ 1).

65

TON Example – Time Transitioning

Given the TON that has been placed on a rung with XIC-A:



LightTimer.EN = 1
LightTimer.TT = 1
LightTimer.DN = 0

As long as the TON remains enabled and the *Accum* value is less than the Preset value (.*ACC* < 60,000), then accumulator will keep <u>incrementing</u> until it reaches the *Preset* value.

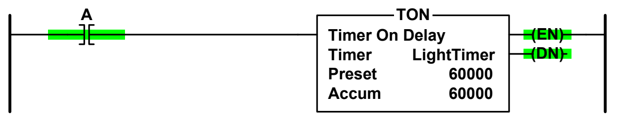
Thus, <u>while the accumulator is incrementing</u>:

- the *Enable* bit will remain <u>set</u> (.*EN* = 1), and
- the *Time Transitioning* bit will remain <u>set</u> (.*TT* = 1).

66

TON Example – Timer Done

Given the TON that has been placed on a rung with XIC-A:



LightTimer.EN = 1
LightTimer.TT = 0
LightTimer.DN = 1

When the ***Accum* value reaches the Preset value** (.*ACC*=60,000), which will occur 60 seconds after the timer is enabled:

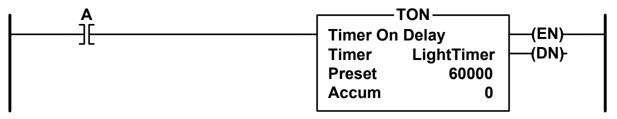
- the accumulator stops incrementing and remains at 60,000,
- the *Enable* bit will remain set (.*EN*=1),
- the *Time Transitioning* bit will reset (.*TT* ➡ 0), and
- the *Done* bit will be set (.*DN* ➡ 1).

The timer will remain in this "done" state until it is either disabled (i.e. – the Rung Condition goes FALSE) or reset by a RES instruction.

67

TON Example – Timer Done

Given the TON that has been placed on a rung with XIC-A:



LightTimer.EN = 0
LightTimer.TT = 0
LightTimer.DN = 0

At any point in time, if bit A ➡ 0, then the next time the rung is executed, XIC-A will be evaluated FALSE, making the Rung Condition FALSE, and the TON will be disabled.

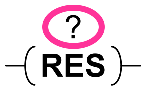
If the timer is disabled:

- the *Enable* bit will be reset (.*EN* ➡ 0),
- the accumulator will reset back to its initial value,
- the *Done* bit will be reset (.*DN* ➡ 0), and
- the *Time Transitioning* bit will be reset (.*TT* ➡ 0) if it was set.

68

Reset (RES)

RES – Reset



The Reset instruction is an Output Instruction that can be used to reset the accumulator (*.ACC*) and the status bits (*.EN*, *.DN*, *.TT*) of a timer or a counter.

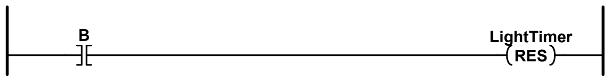
The Base Tag (name) associated with either the timer or counter that the RES instruction is being used to reset must be placed in the Reset's tag field.

For example, a RES instruction could be used to reset LightTimer's accumulator if assigned the tag LightTimer.

69

RES – Operation

RES – Reset



The operation of the Reset instruction is based upon the Rung Condition on the rung.

When the rung is executed:

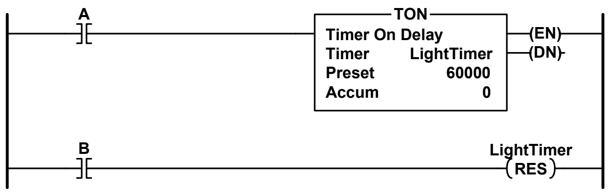
if the Rung Condition is TRUE, the RES will reset the status bits of its assigned timer, and it will reset the timer's accumulator back to its initial value, or

if the Rung Condition is FALSE, the RES does nothing.

70

RES Example (with TON)

Given the following rungs that contain a TON and a RES:



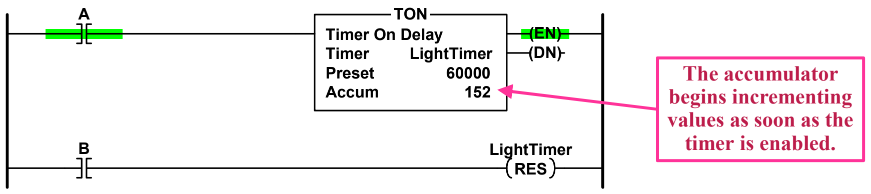
If bits A and B are initially both zero (A = B = 0):

- ⇨ **the Rung Conditions for both rungs are FALSE**
- ⇨ **the TON is disabled**
- ⇨ **the RES instruction does nothing**

71

RES Example – TON Enabled

Given the following rungs that contain a TON and a RES:


A
TON
Timer On Delay
Timer LightTimer
Preset 60000
Accum 152
EN
DN
B
LightTimer
RES
The accumulator begins incrementing values as soon as the timer is enabled.

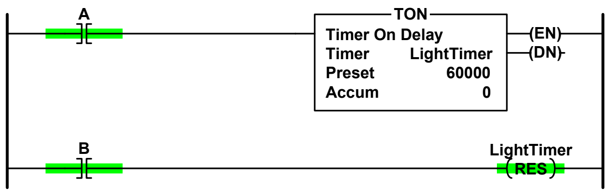
If bit A ➡ 1

- ⇨ **XIC-A ≡ TRUE**
- ⇨ **the Rung Condition for the TON is TRUE**
- ⇨ **TON-LightTimer is enabled**
- ⇨ **the timer's *Enable* bit is set (*.EN* ➡ 1)**
- ⇨ **the timer's accumulator begins incrementing**
- ⇨ **the timer's *Time Transitioning* bit is set (*.TT* ➡ 1)**

72

RES Example – TON Reset

Given the following rungs that contain a TON and a RES:



If bit B ➡ 1 ⇨ XIC-B ≡ TRUE

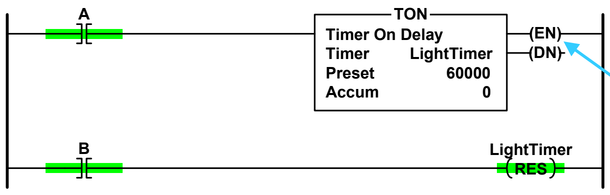
⇨ the Rung Condition for the RES is TRUE

⇨ the RES resets LightTimer's status bits and it resets LightTimer's accumulator back to its initial value

73

RES Example – TON Reset

Given the following rungs that contain a TON and a RES:



Note that, since the software used to display the state of the ladder in "real-time" only has a limited refresh rate, the *.EN* bit may be highlighted green as if set, unhighlighted as if reset, or intermittently change back and forth depending on the actual layout of the ladder diagram.

If bits A and B both remain set (A = B = 1):

⇨ when the TON's rung is executed, the timer's *.EN* bit will be set, and the timer will begin accumulating, but

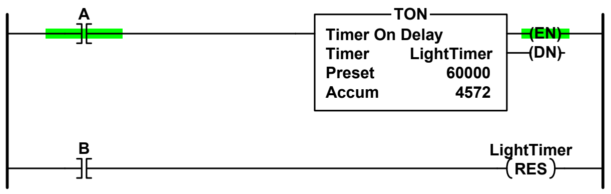
⇨ when the RES's rung is executed, the RES timer will be reset again, and the process will repeat.

This can be confusing if the two rungs are not located next to each other in the ladder diagram because it might appear as if the TON is disabled despite its TRUE Rung Condition.

74

RES Example – TON re-Enabled

Given the following rungs that contain a TON and a RES:



If bit B ➡ 0

- ⇨ XIC-B ≡ FALSE
- ⇨ the Rung Condition for the RES is FALSE
- ⇨ the RES does nothing
- ⇨ since the Rung Condition for the timer is TRUE, the next time the TON’s rung is executed, the timer will be re-enabled, and its accumulator will begin incrementing again.

75

Numerical Comparison Instructions

There are many Logic Instructions available the return either TRUE or FALSE states based on a comparison of two values.

These instructions include:

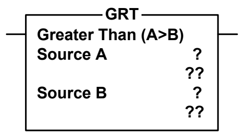
- GRT – Greater Than
- GEQ – Greater Than or Equal To
- LES – Less Than
- LEQ – Less Than or Equal To
- EQU – Equal To
- NEQ – Not Equal To

Only the Greater Than (GRT) instruction is covered in this presentation since the other comparison-type instructions function similarly.

76

Greater Than (GRT)

GRT – Greater Than



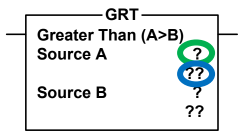
The GRT (Greater Than) instruction is used to compare the values of two numbers, A and B.

- If A > B, then the GRT returns a TRUE state.
- If A≤B, then the GRT returns a FALSE state.

77

GRT – Configuration

GRT – Greater Than



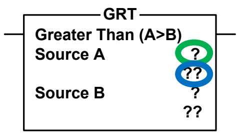
The GRT (Greater Than) instruction is used to compare the values of two numbers, A and B.

- Source A – This field contains either a numerical value for A or a tag name relating to the data that contains the value for A.
- If the Source A field contains a tag name, the second field shows the value currently identified by that tag.

78

GRT – Configuration

GRT – Greater Than



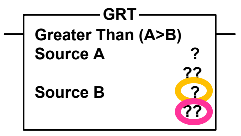
The GRT (Greater Than) instruction is used to compare the values of two numbers, A and B.

- Source A – This field contains either a numerical value for A or a tag name relating to the data that contains the value for A.
- If the Source A field contains a tag name, the field immediately below Source A displays the value currently stored in the tag assigned to Source A.

79

GRT – Configuration

GRT – Greater Than



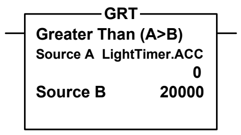
The GRT (Greater Than) instruction is used to compare the values of two numbers, A and B.

- Source B – This field contains either a numerical value for B or a tag name relating to the data that contains the value for B.
- If the Source B field contains a tag name, the field immediately below Source B displays the value currently stored in the tag assigned to Source B.

80

GRT Example

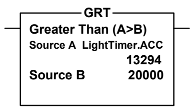
The following GRT is configured to compare the value stored in LightTimer's accumulator to the number 20,000.



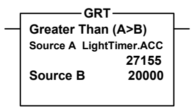
Note that the value stored in LightTimer's accumulator is displayed in the field immediately below Source A as shown in the following examples:

Thus, when the GRT is evaluated:

- If *LightTimer.ACC* is ≤ 20000, the GRT will return a FALSE state.



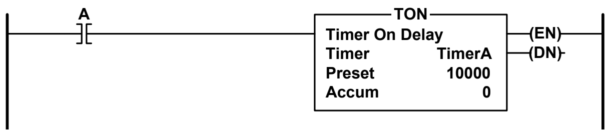
- While *LightTimer.ACC* is > 20000, the GRT will return a TRUE state.



81

Self-Repeating Timer Example

What if you have a 10-second timer that, after it is "enabled", needs to count to 10 seconds and then <u>automatically</u> <u>reset</u> back to zero and begin counting again, and will continue to do so until it is "disabled"?

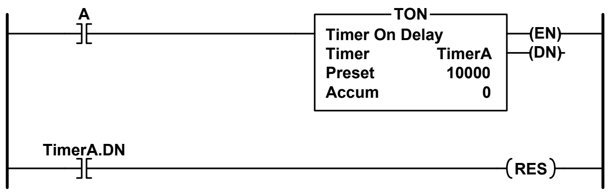


To perform this task, the timer's *.DN* bit can be utilized to trigger a reset of the timer whenever the timer's Accumulator reaches the Preset value (i.e. – it is "done" counting).

82

Self-Repeating Timer Example

Given the following rungs that contain a TON and a RES:



If bit A is initially zero (A = 0):

⇨ XIC-A ≡ FALSE

⇨ the Rung Condition for the TON is FALSE

⇨ the TON is disabled (and thus *TimerA.EN* = 0)

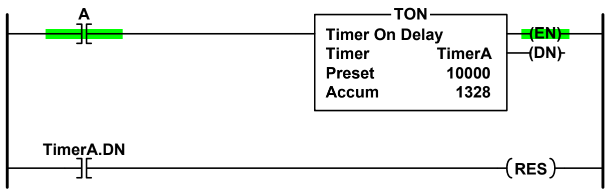
⇨ XIC-TimerA ≡ FALSE

⇨ the Rung Condition for the RES is FALSE

83

Self-Repeating Timer Example

Timer initially Enabled



If bit A ➡ 1

⇨ XIC-A ≡ TRUE

⇨ the Rung Condition for the TON is TRUE

⇨ TON-TimerA is <u>enabled</u>

⇨ the timer's *Enable* bit is <u>set</u> (.*EN* ➡ 1)

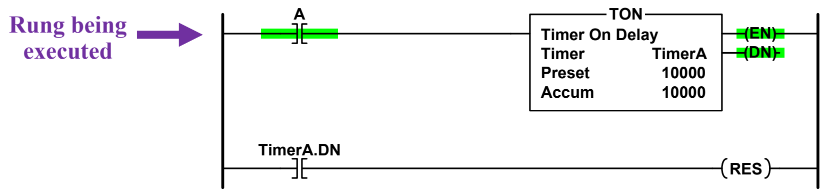
⇨ the timer's accumulator begins <u>incrementing</u>

⇨ the timer's *Time Transitioning* bit is <u>set</u> (.*TT* ➡ 1)

84

Self-Repeating Timer Example

Timer reaches its Preset Value



When *TimerA.ACC* increments to 10000:

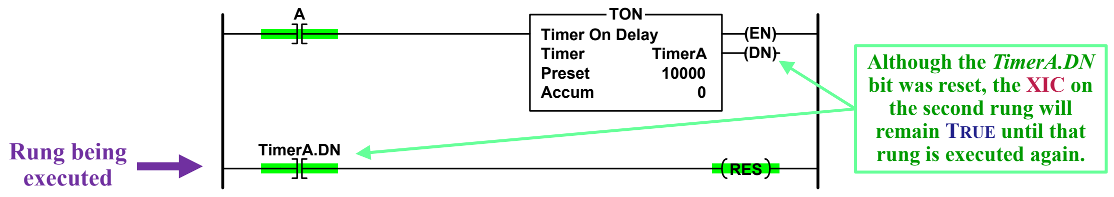
⇨ the timer's *Done* bit is <u>set</u> (.*DN* ➡ 1)

Note that, even though the .*DN* bit has been set, the state of the XIC on the second rung will not change to TRUE until the RES's rung is executed again (after the .DN bit was set).

85

Self-Repeating Timer Example

Timer is Reset


Rung being executed
A
TON
Timer On Delay
Timer TimerA
Preset 10000
Accum 0
(EN)
(DN)
TimerA.DN
(RES)
Although the *TimerA.DN* bit was reset, the XIC on the second rung will remain TRUE until that rung is executed again.

The next time the RES's rung is executed (after the .*DN* bit was set):

⇨ XIC-TimerA.DN ≡ TRUE

⇨ the Rung Condition for the RES is TRUE

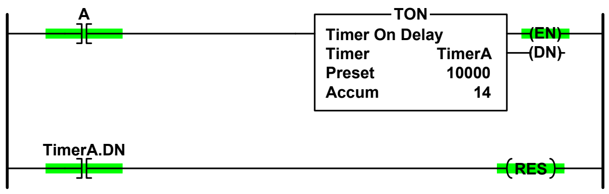
⇨ the RES <u>resets</u> TimerA's status bits and it <u>resets</u> TimerA's accumulator back to its initial value

86

Self-Repeating Timer Example

Timer is re-Enabled

Rung being executed



The next time the TON's rung is executed (after the TON was reset):

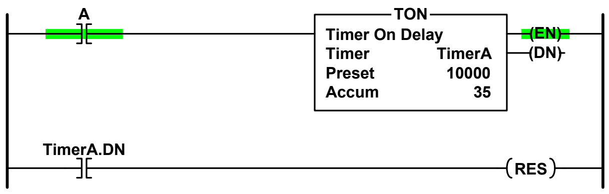
- ⇨ XIC-A ≡ TRUE (it never changed to FALSE)
- ⇨ the Rung Condition for the TON is still TRUE
- ⇨ TON-TimerA is re-<u>enabled</u>, its *.EN* bit is <u>set</u>, its accumulator begins <u>incrementing</u> again, and its *.TT* bit is <u>set</u>

87

Self-Repeating Timer Example

After the Timer is re-Enabled

Rung being executed



The next time the RES's rung is executed (after the TON was reset):

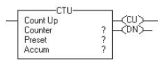
- ⇨ XIC-TimerA ≡ TRUE
- ⇨ the Rung Condition for the RES is FALSE
- ⇨ the RES does nothing

As long as bit A remains set (A = 1), the TON will remain enabled and it will continue incrementing until it's "Done" again, at which time the RES will reset the timer again. This process will repeat indefinitely.

88

Other Instructions – Counter Up (CTU)

The CTU (Counter Up) is an Output Instruction whose function is similar to that of the On-Delay Timer:

Instruction:	Relay Ladder:	Function Block:	Structured Text:	Description:
CTU Counter Up		see CTUD	see CTUD	The CTU instruction counts upward.

Operand:	Type:	Format:	Description:
Counter	COUNTER	tag	counter structure
Preset	DINT	immediate	how high to count
Accum	DINT	immediate	number of times the counter has counted; initial value is typically 0

Arithmetic Status Flags:	Major Faults:
not affected	none

The Counter Up (CTU) is useful for keeping track of the number of specific events that occur, such as the number of items passing by an optical detector.

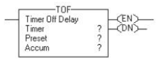
When the CTU's rung-condition becomes TRUE, the CTU's accumulator is incremented by one (1).

Note that, for the CTU to count up again, the CTU's rung-condition must first become FALSE and then TRUE again.

89

Off-Delay Timer (TOF)

TOF – OFF Delay Timer

Instruction:	Relay Ladder:	Function Block:	Structured Text:	Description:
TOF Timer Off Delay		see TOFR	see TOFR	The TOF instruction is a non-retentive timer that accumulates time when the instruction is enabled (rung-condition-in is false).

Operand:	Type:	Format:	Description:
Timer	TIMER	tag	timer structure
Preset	DINT	immediate	how long to delay (accumulate time)
Accum	DINT	immediate	number of msec the timer has counted; initial value is typically 0

Arithmetic Status Flags:	Major Faults:		
not affected	Type 4	Code 34	• .PRE < 0 • .ACC < 0

The Off-Delay timer (TOF) works similar to the On-Delay timer (TON) except that the TOF is enabled when its rung-condition becomes FALSE.

90