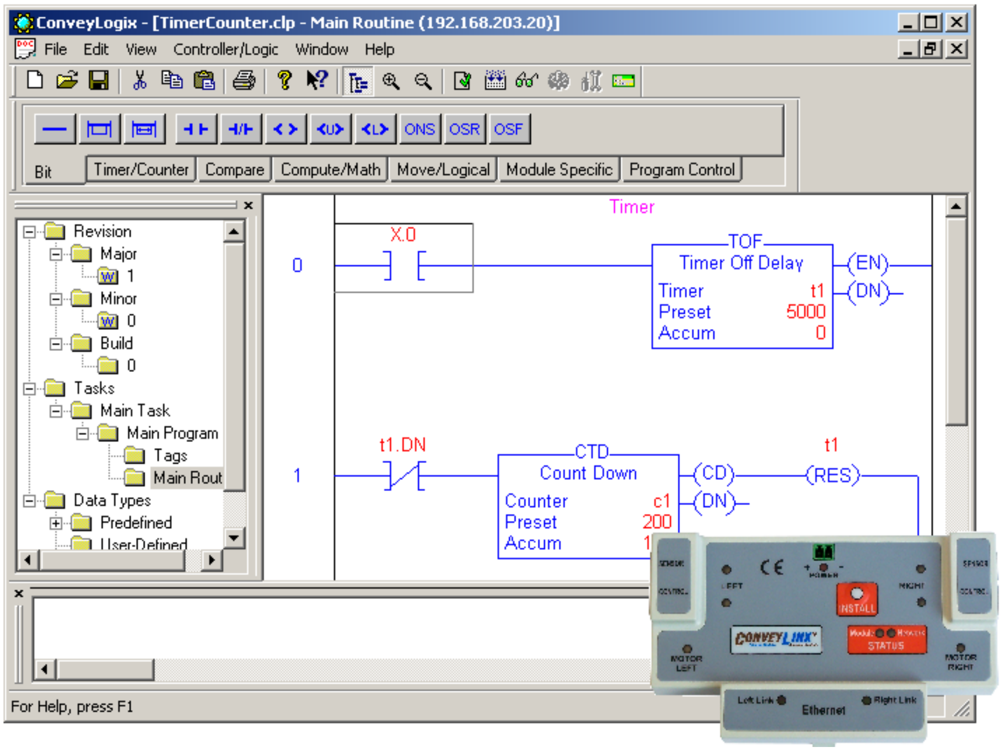
Programmer's Guide

Version 2.2

July 2016



Publication *ERSC*-1200

Important User Information

ConveyLinx ERSC modules contain ESD (Electrostatic Discharge) sensitive parts and components. Static control precautions are required when installing, testing, servicing or replacing these modules. Component damage may result if ESD control procedures are not followed. If you are not familiar with static control procedures, reference any applicable ESD protection handbook. Basic guidelines are:

- Touch a grounded object to discharge potential static
- Wear an approved grounding wrist strap
- Do not touch connectors or pins on component boards
- Do not touch circuit components inside the equipment
- Use a static-safe workstation, if available
- Store the equipment in appropriate static-safe packaging when not in use

Because of the variety of uses for the products described in this publication, those responsible for the application and use of this control equipment must satisfy themselves that all necessary steps have been taken to assure that each application and use meets all performance and safety requirements, including any applicable laws, regulations, codes, and standards

The illustrations, charts, sample programs and layout examples shown in this guide are intended solely for purposes of example. Since there are many variables and requirements associated with any particular installation, Insight Automation Inc. does not assume responsibility or liability (to include intellectual property liability) for actual use based on the examples shown in this publication

Summary of Changes

The following table summarizes the changes and updates made to this document since the last revision

Revision	Date	Change / Update
1.5	April 2014	Updates Global Contact Information
1.6	June 2014	Added Function Block and Structured Text Sections
2.1	April 2016	Added Standard Function Blocks, ConeyLinx-Ai Controller Tags, Appendix F
2.2	July 2016	Updated ConeyLinx and ConeyLinx-Ai Controller Tags, Added ConeyLinx-Ai2 Controller Tags and Appendix G

Global Contact Information

PULSEROLLER

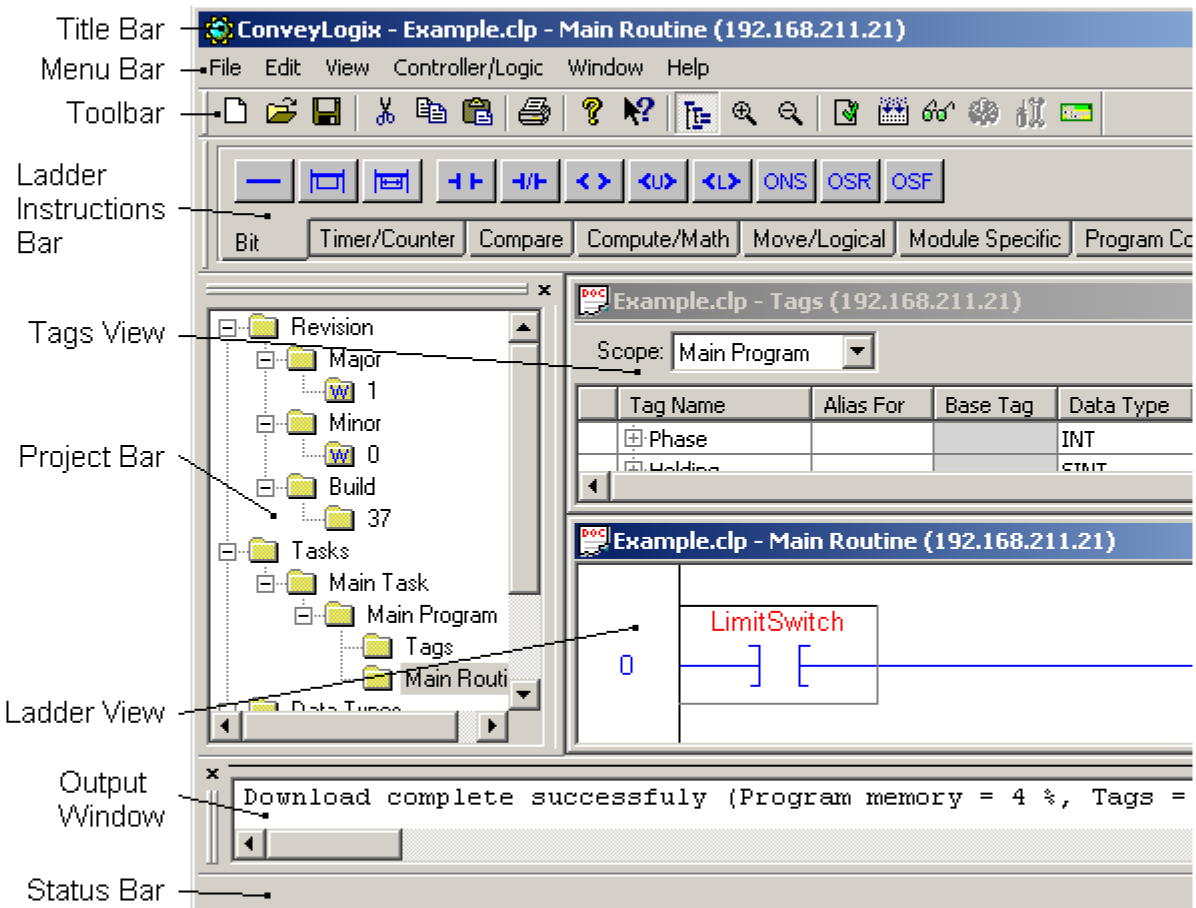
WWW.PULSEROLLER.COM
SALES@PULSEROLLER.COM
SUPPORT@PULSEROLLER.COM

Table of Contents

1. Getting Started

1.1 Screen Areas

To understand more easily how to work with ConveyLogix Programmer software, main screen areas are pointed on the picture and described below:



1.1.1 Title Bar

Title Bar displays the information of working project (file with extension .clp), selected view (Main Program or Tags), controller IP Address and Debug information (described in point 6).

1.1.2 Menu Bar

File Menu	Description	Icon	Shortcut
New	creates an untitled project		Ctrl+N
Open	opens an existing project		Ctrl+O
Close	closes the current project		
Save	saves the current project		Ctrl+S
Save As	saves the current project to a different file		
Print	prints ladder logic and/or Main Program Tags		Ctrl+P
Print Preview	preview ladder logic and/or Main Program Tags before printing		
Print Setup	setup printer properties		
Exit	quits the application		

Edit Menu	Description	Icon	Shortcut
Undo	undo the last action		Ctrl+Z
Cut	cuts the selection and put it to Clipboard		Ctrl+X
Copy	copies the selection and put it to Clipboard		Ctrl+C
Paste	pastes the Clipboard content to the selected location		Ctrl+V

Edit menu commands apply only to Main Program (Ladder View) operations.

View Menu	Description	Icon
Toolbar	hides/displays the Toolbar	
Status Bar	hides/displays the Status Bar	
Project Bar	hides/displays the Project Bar	
Zoom In	increase the zoom level of the Main Program (Ladder View)	
Zoom Out	decrease the zoom level of the Main Program (Ladder View)	

Controller/Logic Menu	Description	Icon
Verify Program	Verifies the Ladder program. The result of the operation is displayed in Output window.	
Download Program	downloads the project to controller with chosen IP Address	
Debug	puts ConveyLogix Programmer in Debug mode (described in point 6)	
Stop Debugging	puts ConveyLogix Programmer in Normal (editable) mode	
Program Mode	puts the controller in Program mode. In this mode controller stops execute the Ladder program	
Run Mode	puts the controller in Run mode. In this mode controller executes the Ladder program	
Controller Properties	opens the dialog box to change Controller Type and/or its IP Address (described in point 1.5)	

Program Mode and Run Mode menus are active only in ConveyLogix Programmer Debug mode.

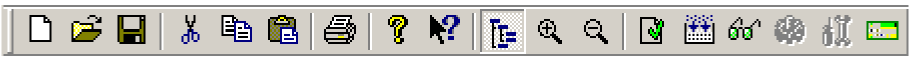
Window Menu

Window Menu contains the standard Windows menus to navigate between Main Program (Ladder View) and Tags (Tags View).

Help Menu	Description	Icon
Help Topics	opens the ConveyLogix Programmer user's guide	
About	opens the dialog box to display ConveyLogix Programmer version information	

1.1.3 Toolbar

Toolbar contains the shortcuts to some of the menus:



Icons meaning is described above in section 1.1.1 Title Bar.

1.1.4 Ladder Instruction Bar

Ladder Instruction Bar is enabled only in Main Program (Ladder view). It divided on several tabs by categories. Every tab contains relevant Ladder Instructions buttons as described in section 5.0 Ladder Logic Instructions).

1.1.5 Project Bar

Project Bar contains the information of the current project as described in section 1.3 Project Organization).

1.1.6 Tags View

Tag View is the window where you edit your tags.

1.1.7 Ladder View

Ladder view is the window where you edit your ladder logic.

1.1.8 Output Window

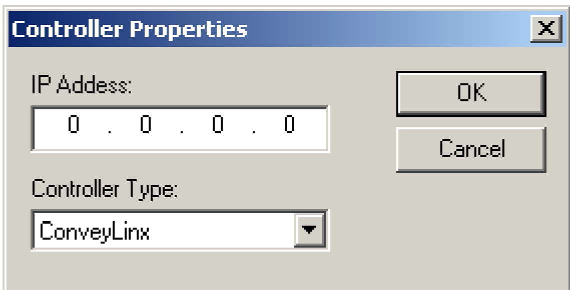
Output window displays the results of Download Program, Verify Program, runtime errors, etc.

1.1.9 Status Bar

The right side of the Status Bar provides ongoing status information and prompts as you use the software. The left side of the Status Bar provides information about Caps Lock, Num Lock and Scroll Lock keys.

1.2 Create a Project

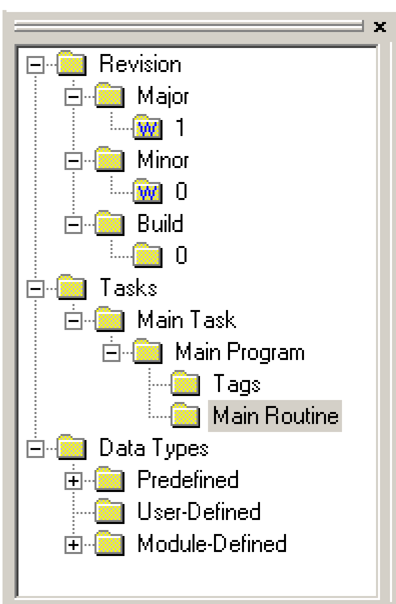
From the File menu, select New or click on 🗋 icon. The next dialog appears.



- Type the IP Address of the controller you need to work with.
- Choose the controller type – ConveyLinx or ConveyNet.
- Press OK button and a project called “Untitled” will be created.

1.3 Project Organization

The project organization is shown on Project Bar.



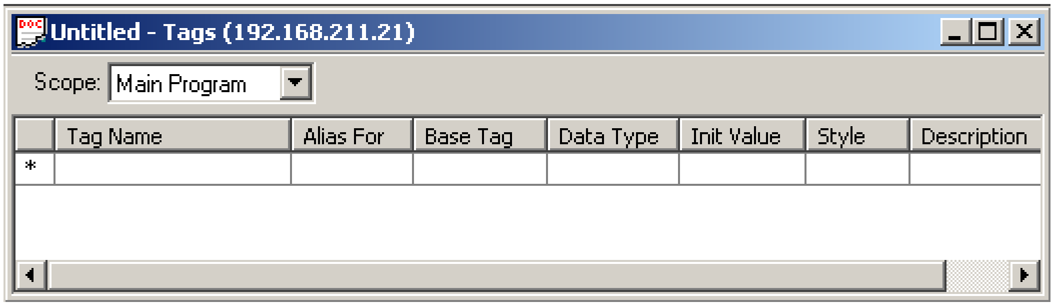
1.3.1 Revision

"Major" and "Minor" contain values as to the major and minor versions of the project and these fields are editable. Build contains a value which increments automatically during every Save operation.

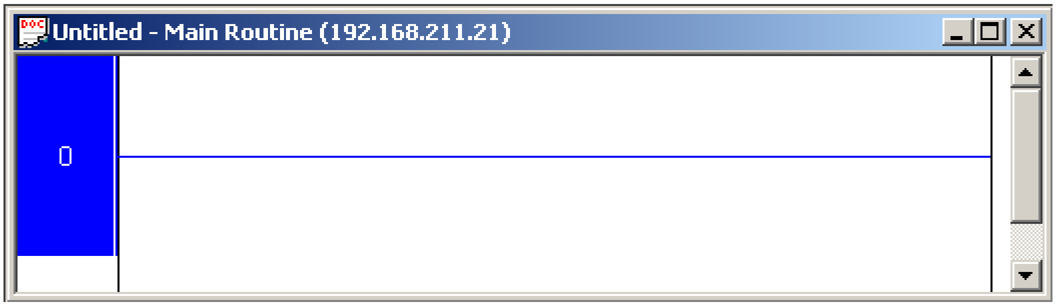
1.3.2 Tasks

ConveyLinx and ConveyNet controllers support only one task, called Main Task and run only one Program, called Main Program. Main Program represents by two views:

- Tags – double click to open Tags View. Tags View displays all information about tags.



- Main Routine – double click to open Ladder View. Ladder View displays all information about ladder diagram routine.



1.3.3 Data Types

Data Types are divided by three categories:

- Predefined – ConveyLogix supported data types.
- User-Defined – not supported.
- Module-Defined – controller supported data types.

1.4 Save, Close and Open a Project

To save a project, select File/Save menu or click on icon. If the project is Untitled, Save As dialog appears to choose your project name.

If you want to store a project with another name, select File/Save As menu.

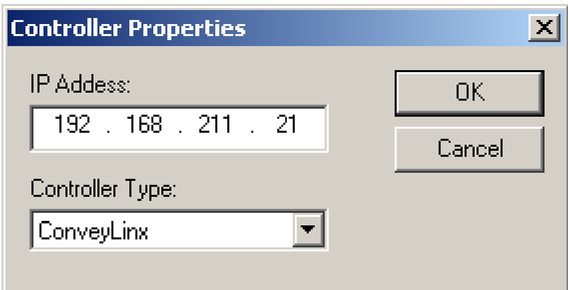
When the project is saves once, the every next save operation increases Build value.

To close the project, select File/Close menu.

To open a project, select File/Open menu or click on icon and select a file from disk.

1.5 Configure a Controller

To configure a controller, select Controller/Logic / Controller Properties menu or click on  icon. The next dialog appears.



- IP Address is the IP address of the controller you need to work with.
- Controller Type is a type of the controller – ConveyLinx or ConveyNet.
- Change the controller properties if you need and press OK button for cofirmation.
- If you change the Controller Type from ConveyLinx or ConveyNet or vice versa, you may lose some Controller Tags properties.
- With changing controller's IP Address from dialog above, you may download and debug the same ladder program to different controllers.

Example:

Let you have a network with three controllers with IP addresses 192.169.211.20, 192.169.211.21 and 192.169.211.22, which have to work with same ladder program.

- Change IP Address in dialog above to 192.169.211.20, then download and debug the ladder program.
- Then change IP Address in dialog above to 192.169.211.21, download and debug the ladder program.
- And then change IP Address to 192.169.211.22, download and debug the ladder program.

2.0 Organize Tags

2.1 Defining Tags

Tag is a named area of the controller's memory where data is stored. Tags are the basic mechanism for allocating memory, referencing data from logic, and monitoring data.

The controller uses the tag name internally and doesn't need to cross-reference a physical address.

The minimum memory allocation for a tag is a byte.

When you create a tag, you assign the following properties to the tag:

- Scope
- Tag Type
- Data Type

2.1.1 Scope

Tags might divide of two categories by Scope:

- Main Program Tags – user defined tags.
- Controller Tags – controller defined tags. They cannot be changed and depend from controller type. Controller tags are described in Appendix A – Controller Tags.

2.1.2 Tag Type

There are five types of tags that you can create:

- Base – refers to a normal tag (selected by default). This type of tag allows you to create your own internal data storage.
- Alias – allows you to assign your own name to an existing tag, structure tag member, or bit, and refers to a tag which references another tag with the same definition.
- Produce – refers to a tag that is produced by another controller whose data you want to use in this controller.
- Consumed – refers to a tag that is consumed by another controller.
- Non-volatile – power independent tags.

2.1.3 Data Type

The data type defines the type of data that a tag stores, such as a bit, integer, etc.

ConveyLogix Programmer supports four types of data:

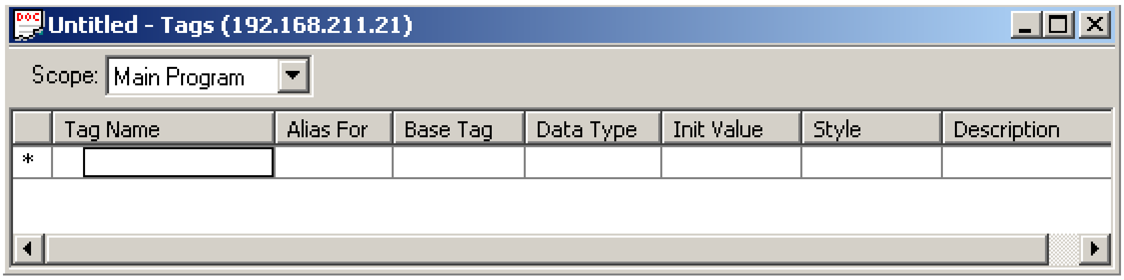
- Simple – BOOL, SINT, INT and DINT.

Data Type	Size	Range
BOOL	1 Bit	**0 or 1**
SINT	1 Byte	**-128 to +127**
INT	2 Bytes	**-32,768 to +32767**
DINT	4 Bytes	**-2,147,483,648 to +2,147,483,647**

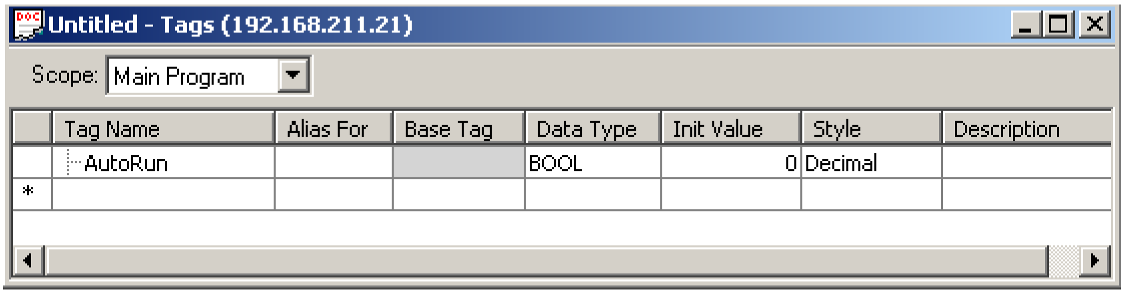
- Structure – a data type that is a combination of other data types. Structure is formatted to create a unique data type that matches a specific need. Within a structure, each individual data type is called a member. Like tags, members have a name and data type. ConveyLogix Programmer supports two predefined structures – TIMER and COUNTER for use with specific instructions such as timers, counters, etc. and one user-defined – Zone.
- Array – a numerically indexed sequence of elements of the same data type. In ConveyLogix Programmer, an array index starts at 0 and extends to the number of elements minus 1. An array can have up to 3 dimensions unless it is a member of a structure, where it can have only 1 dimension. An array tag occupies a contiguous block of memory in the controller with each element in sequence.

2.2 Create a Tag

Tags are created or edited in Tags View. Open Tags View by double click to Tags on Project Bar. To create a tag click into Tag Name area on the last row (marked with sign *):



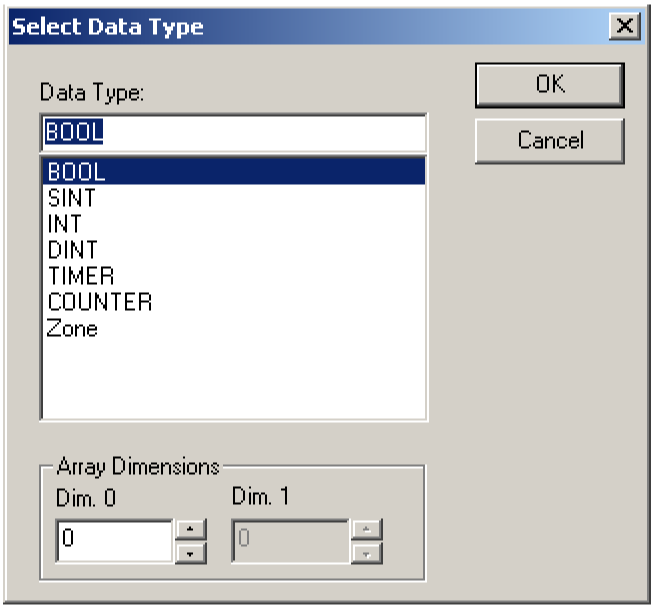
Type a name of the new tag and then press Enter key or click outside from the rectangle area.



The Tag has the next properties:

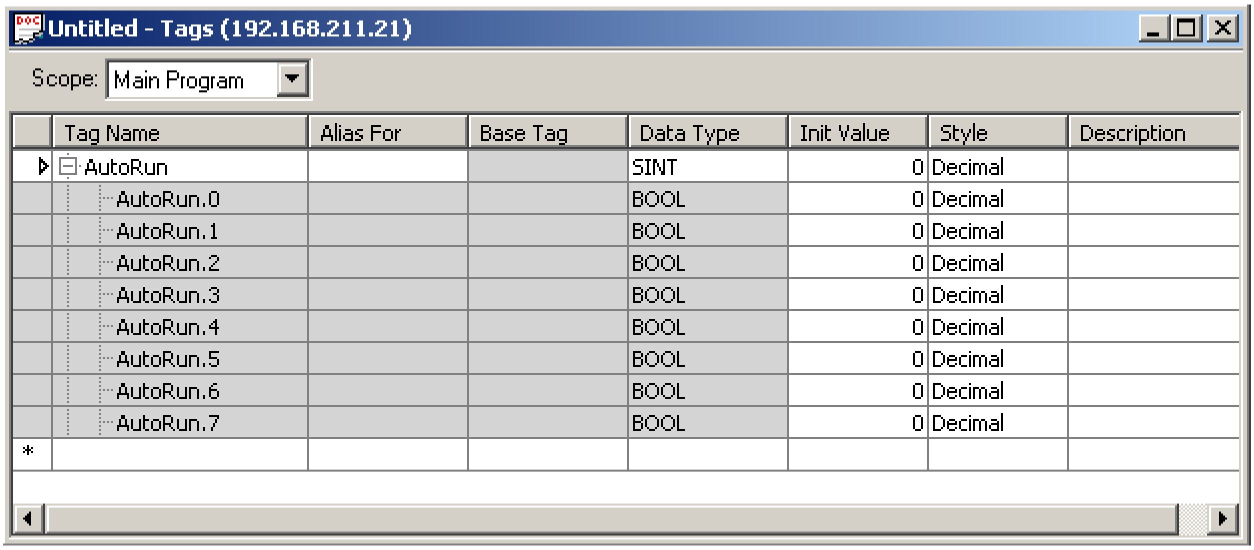
- Scope – to create a tag is allowed only for Main Program.
- Tag Name – unique alphanumeric name, excluding the symbols ".", ",", "[" and "]".
- Alias For – used to represent this tag to another (described in point 2.4).
- Base Tag – the original tag name, related to alias. In case that Alias For is not used, this field is disabled (grayed).
- Data Type – type of the data of the tag.

To change data type click on Data Type cell. The next dialog box appears:



- Choose a type from Data Type list and press OK button.
- If the chosen type is different from BOOL, the tag contains subtags, represent like a tree. If data type is a simple type the subtags are BOOL types. Count of subtags is equal of type length in bits.
- If data type is a structure, subtags are fields of the structure. Every field is represented down to BOOL types.

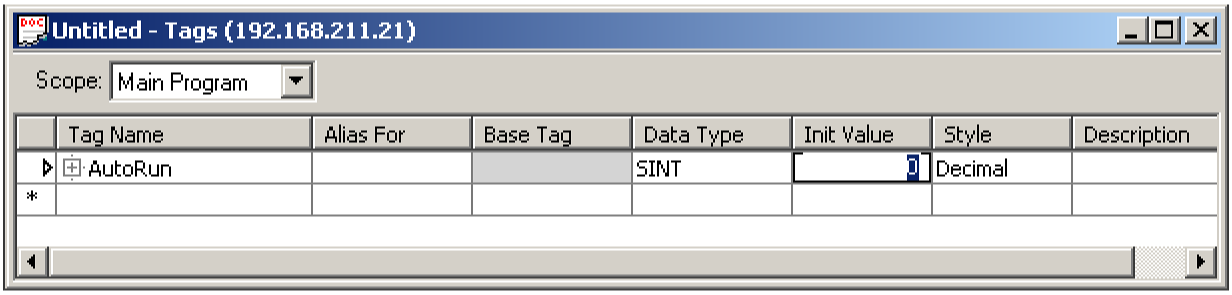
- If data type is an array, subtags are the elements of the array. Every field is represented down to BOOL types.
- For example choose data type as SINT. To see the subtags, click on "+" button (left of the tag name):



Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
AutoRun			SINT	0	Decimal	
AutoRun.0			BOOL	0	Decimal	
AutoRun.1			BOOL	0	Decimal	
AutoRun.2			BOOL	0	Decimal	
AutoRun.3			BOOL	0	Decimal	
AutoRun.4			BOOL	0	Decimal	
AutoRun.5			BOOL	0	Decimal	
AutoRun.6			BOOL	0	Decimal	
AutoRun.7			BOOL	0	Decimal	

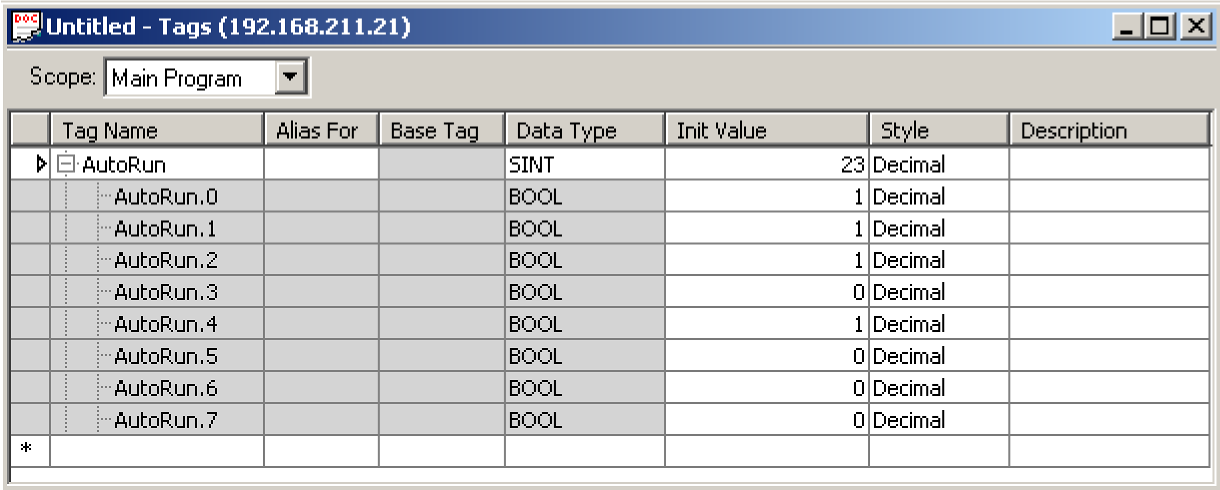
- Init Value – shows the initialize value of the tag, which is the start value when the controller power-up. Default value is 0.

To change this value, click on Init Value cell. Edit box is shown:



Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
AutoRun			SINT	0	Decimal	

- Type the new value and then confirm by pressing Enter or clicking outside the edit box area. To cancel typed Init Value changes, press Esc.
- If typed Init Value is not in the range, message box will appear. When you press OK, edit box will stay to correct or cancel the value.
- The new Init Value will be changed on corresponding subtags (if they exist).

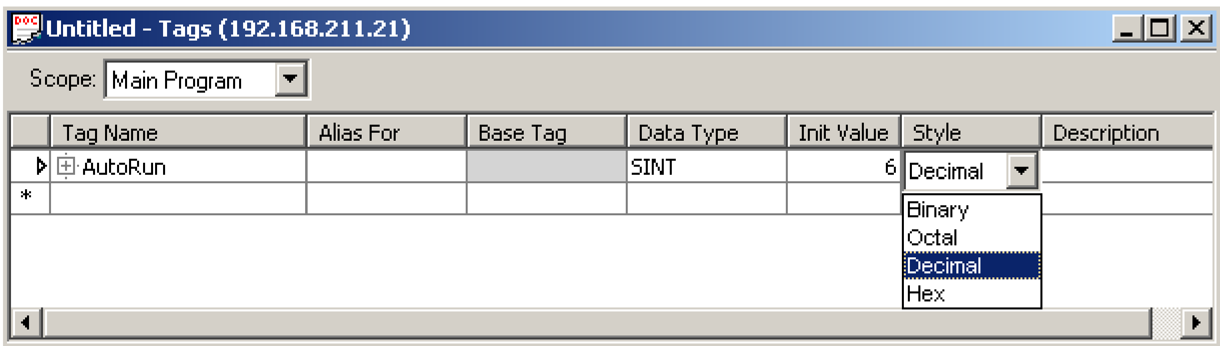


Likewise, if subtag is changed, change is reflected on corresponding tag.

- Style – the format that numeric values are displayed in.

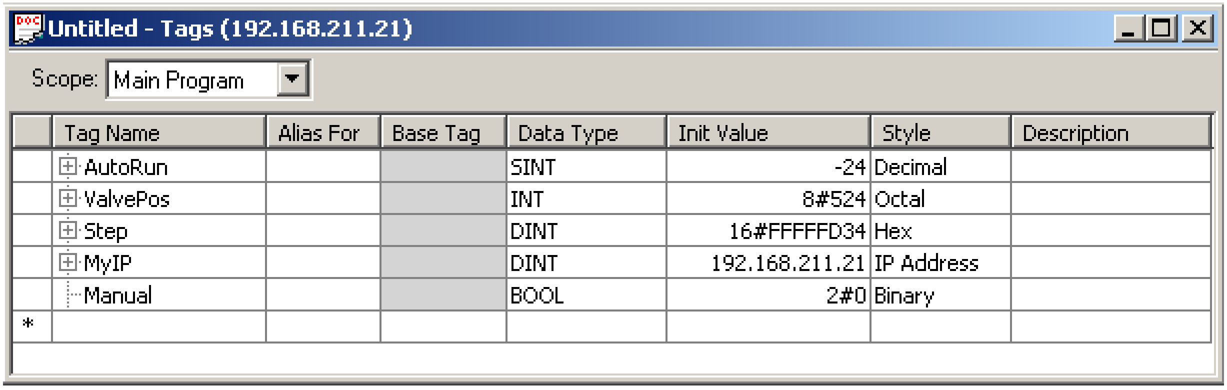
Style	Presentation	Example
Binary	2#	**2#1101**
Octal	8#	**8#47**
Decimal	Signed numeric value	**-5; 27**
Hex	16#	**16#FFFFFFFF**
IP Address	**IP Address**	**192.168.211.21**

To change the Tag style, click on Style cell. Combo box with permitted formats will appear. Open it and select desired style.



Confirm the selected style by pressing Enter or clicking outside the combo box area.

On the next picture are shown tags with simple data types and different styles.



Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
AutoRun			SINT	-24	Decimal	
ValvePos			INT	8#524	Octal	
Step			DINT	16#FFFFFD34	Hex	
MyIP			DINT	192.168.211.21	IP Address	
Manual			BOOL	2#0	Binary	
*						

On the next table, Data Type ranges for simple data types are shown for different styles:

Data Type/ Style	BOOL	SINT	INT	DINT
Binary	2#0 to 2#1	2#00000000 to 2#11111111	2#0000000000000000 to 2#1111111111111111	**2#0000000000000000 0000000000000000 to 2#1111111111111111 1111111111111111**
Octal	8#0 to 8#1	8#0 to 8#377	8#0 to 8#177777	**8#0 to 8#377777**
Decimal	0 to 1	-128 to 127	-32768 to 32767	**-2,147,483,648 to +2,147,483,647**
Hex	16#0 to 16#1	16#00 to 16#FF	16#0000 to 16#FFFF	**16#00000000 up to 16#FFFFFFFF**
IP Address	**Not used**	**Not used**	**Not used**	**0.0.0.0 to 255.255.255.255**

Init Value and Style are disabled for complex data types (structures and arrays).

If the Data Type is changed to type with smaller type length, and Init Value exceeds type range, the value is converted to new type.

Example:

Let Data Type of tag AutoRun is INT and Init Value is 16#FE17. Changing Data Type to SINT, Init Value will be changed to 16#17.

Correspondence from IP Address to number is explained in the next example.

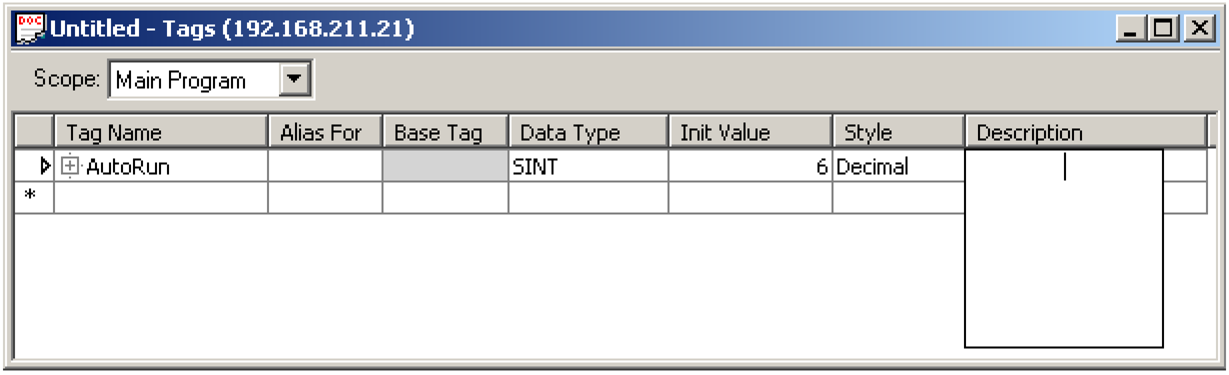
Example:

Lets have a tag *MyIP* (as shown on figure above) with Style IP Address and Init Value 192.168.211.21. If Style is changed to Hex, 16#D315C0A8 will displayed. Bytes respond to the next part of IP Address:

Most significant byte	D3 -> 211
…	15 -> 21
…	C0 -> 192
Least significant byte	A8 -> 168

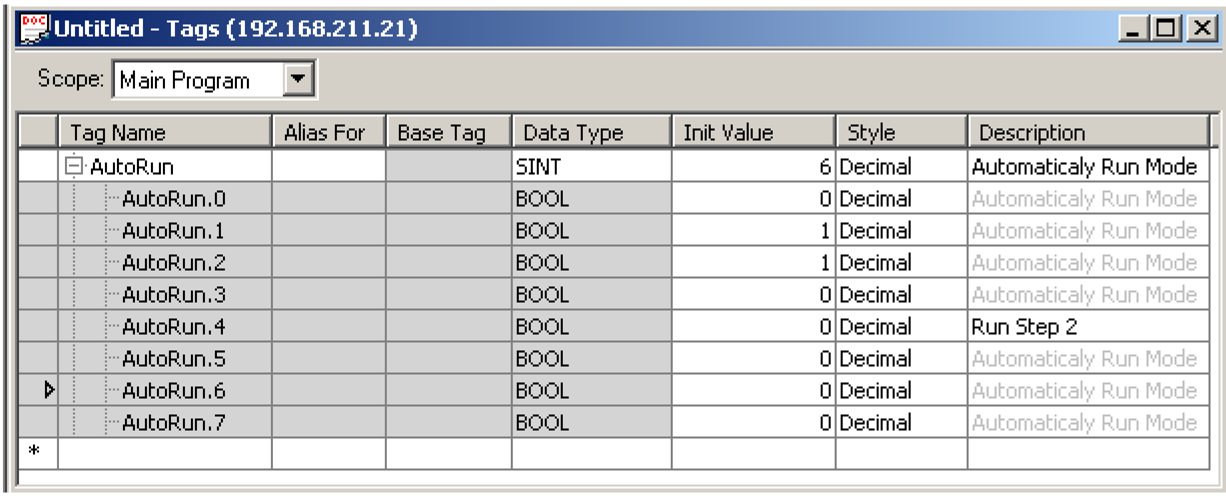
- Description – user text for better explanation of the tag.

To enter a description, click on Description cell. Edit box will arrear:



Type the description and then confirm by pressing Enter or clicking outside the edit box area.

All subtags inherit typed description. Inherited descriptions show in grey. If you type a description of subtag, its color will change to black (for example AutoRun.4 subtag).



	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	AutoRun			SINT	6	Decimal	Automaticaly Run Mode
	AutoRun.0			BOOL	0	Decimal	Automaticaly Run Mode
	AutoRun.1			BOOL	1	Decimal	Automaticaly Run Mode
	AutoRun.2			BOOL	1	Decimal	Automaticaly Run Mode
	AutoRun.3			BOOL	0	Decimal	Automaticaly Run Mode
	AutoRun.4			BOOL	0	Decimal	Run Step 2
	AutoRun.5			BOOL	0	Decimal	Automaticaly Run Mode
	AutoRun.6			BOOL	0	Decimal	Automaticaly Run Mode
	AutoRun.7			BOOL	0	Decimal	Automaticaly Run Mode
*							

2.3 Create an Array

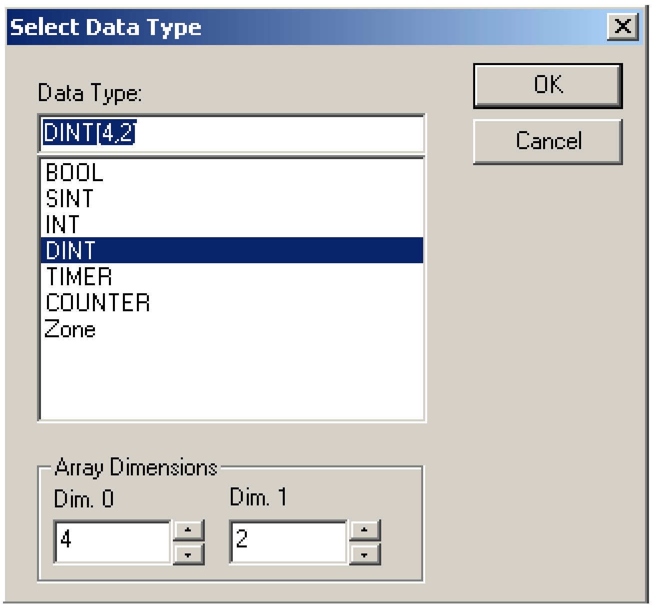
Array is a tag that contains a block of multiple pieces of data. Within an array, each individual piece of data is called an element. Each element uses the same data type.

An array tag occupies a contiguous block of memory in the controller, each element in sequence.

The Data may be organized into a block of 1 or 2 dimensions array.

An element within the array starts at 0 and extends to the number of elements minus 1 (zero based).

To create an array, click on Data Type cell of an existing tag. Select Data Type dialog will open. Choose Data Type and type the array dimensions.



Dim. 1 is the number of elements in the first dimension. If Dim.1 is zero, the next dimensions are disabled (grayed).

Dim. 2 is the number of elements in the second dimension. Choose OK button to confirm changes.

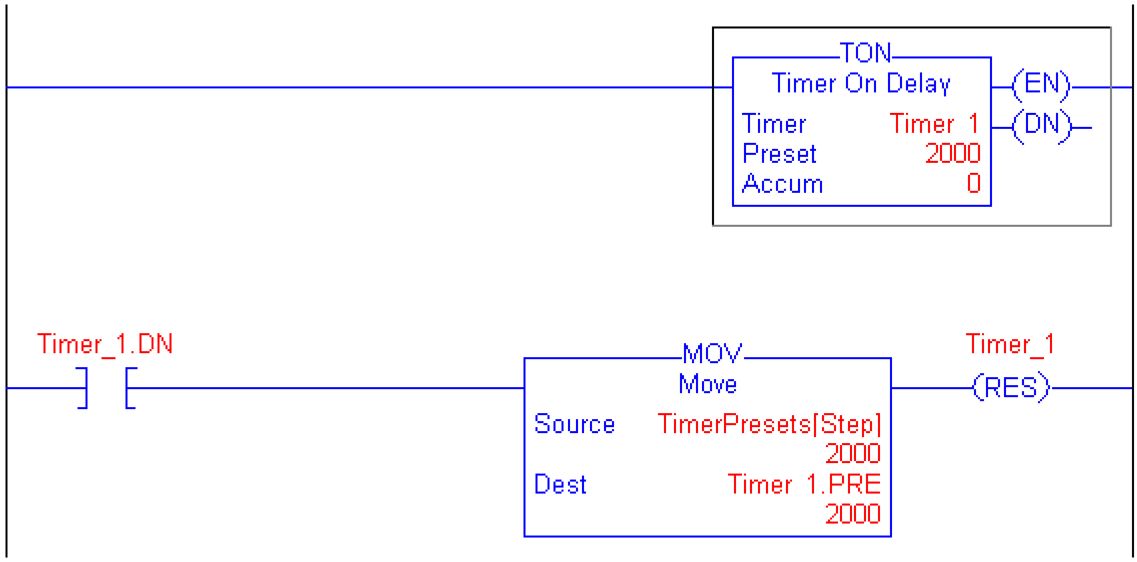
ConveyLogix and controllers can index arrays.

Example: Single dimension array

In this example, a single timer instruction times the duration of several steps. Each step requires a different preset value. Because all the values are the same data type (DINTs) an array is used.

Scope: Main Program

	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	⊞ Step			SINT	0	Decimal	
▶	⊞ Timer_1			TIMER	{...}		
	⊟ TimerPresets			DINT[4]	{...}		
	⊞ TimerPresets[0]			DINT	2000	Decimal	
	⊞ TimerPresets[1]			DINT	3000	Decimal	
	⊞ TimerPresets[2]			DINT	4000	Decimal	
	⊞ TimerPresets[3]			DINT	5000	Decimal	
*							



On MOV instruction Source operand indexes TimerPresets tag by Step. When Step = 0, TON instruction accumulate time to TimerPresets[0] = 2000 milliseconds. When Step = 1, TON instruction accumulate time to TimerPresets[1] = 3000 milliseconds and vice versa.

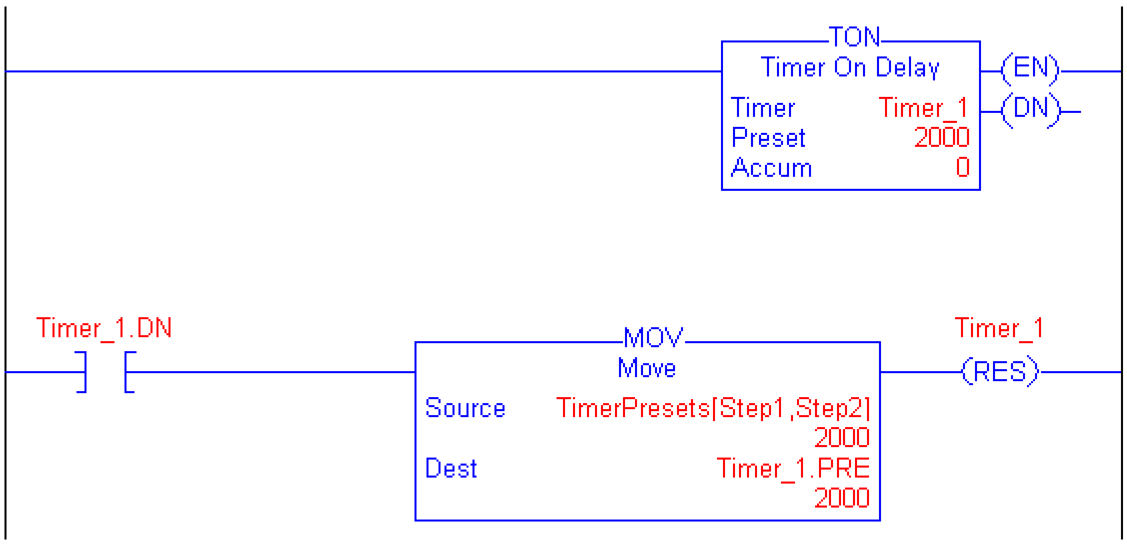
When Step is out of TimerPresets index range (Step < 0 or Step > 3), MOV instruction doesn't execute (rung-condition-out is false).

Example: Two dimension array

In this example, a single timer instruction times the duration of Step_1 and Step_2. Each pair of steps requires a different preset value.

Scope: Main Program

	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	⊞ Step1			SINT	0	Decimal	
	⊞ Step2			SINT	0	Decimal	
	⊞ Timer_1			TIMER	{...}		
	⊟ TimerPresets			DINT[4,2]	{...}		
	⊞ TimerPresets[0,0]			DINT	2000	Decimal	
	⊞ TimerPresets[0,1]			DINT	3000	Decimal	
	⊞ TimerPresets[1,0]			DINT	4000	Decimal	
	⊞ TimerPresets[1,1]			DINT	5000	Decimal	
	⊞ TimerPresets[2,0]			DINT	6000	Decimal	
	⊞ TimerPresets[2,1]			DINT	7000	Decimal	
	⊞ TimerPresets[3,0]			DINT	8000	Decimal	
	⊞ TimerPresets[3,1]			DINT	9000	Decimal	
*							



On MOV instruction Source operand indexes TimerPresets tag by Step1 and Step2.

When Step1 = 0 and Step2 = 0, TON instruction accumulate time to TimerPresets[0,0] = 2000 milliseconds. When Step1 = 0 and Step2 = 1, TON instruction accumulate time to TimerPresets[0,1] = 3000 milliseconds and vice versa.

When Step1 is out of TimerPresets first index range (Step1 < 0 or Step1 > 3) or Step2 is out of TimerPresets second index range (Step2 < 0 or Step2 > 1), MOV instruction doesn't execute (rung-condition-out is false).

2.4 Assign Alias Tags

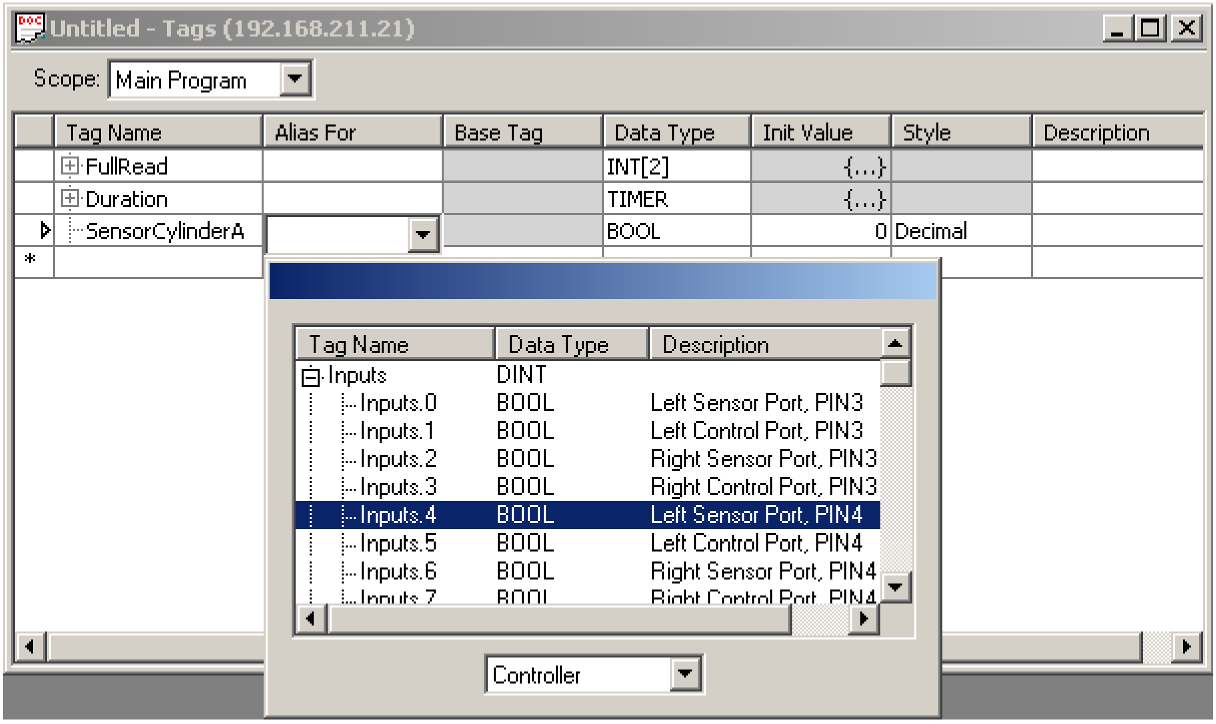
An alias tag lets you create one tag that represents another tag. Both tags share the same value. When the value of one of the tags changes, the other tag reflects the change as well.

Use aliases in the following situations:

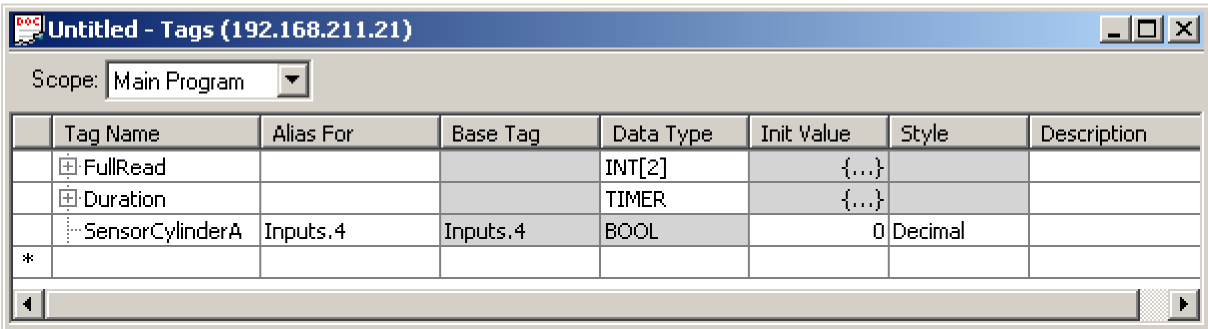
- Program logic in advance of wiring diagrams.
- Assign a descriptive name to controller I/O.
- Provide a simpler name for a complex tag.
- Use a descriptive name for an element of an array.

The tags window displays alias information. Aliases may be assigned only for Main Program tags.

To assign an alias, click on *Alias For* cell to desired tag. Combo-box will appear. Type tag name or open the combo-box to choose a tag from existing. For example, change the scope to Controller, click sign "+" on Inputs tag and select Inputs.4.



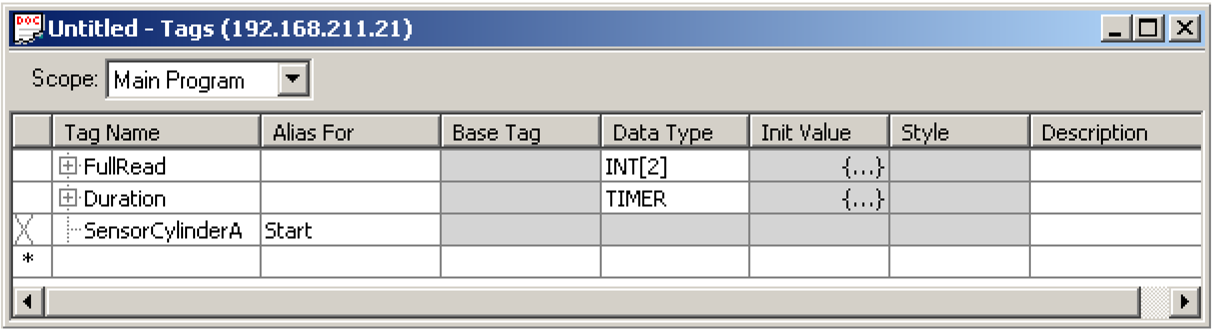
Double-click on Inputs.4 and then press Enter or click outside the combo-box.



Alias For shows the name of chosen tag. Base Tag shows the original tag. Data Type and Init Value are the values of Base Tag (in this example are on Input.4). If you change the Init Value of SensorCylinderA, you exactly change the Init Value of Input.4.

This example shows how to assign a descriptive name to controller I/O.

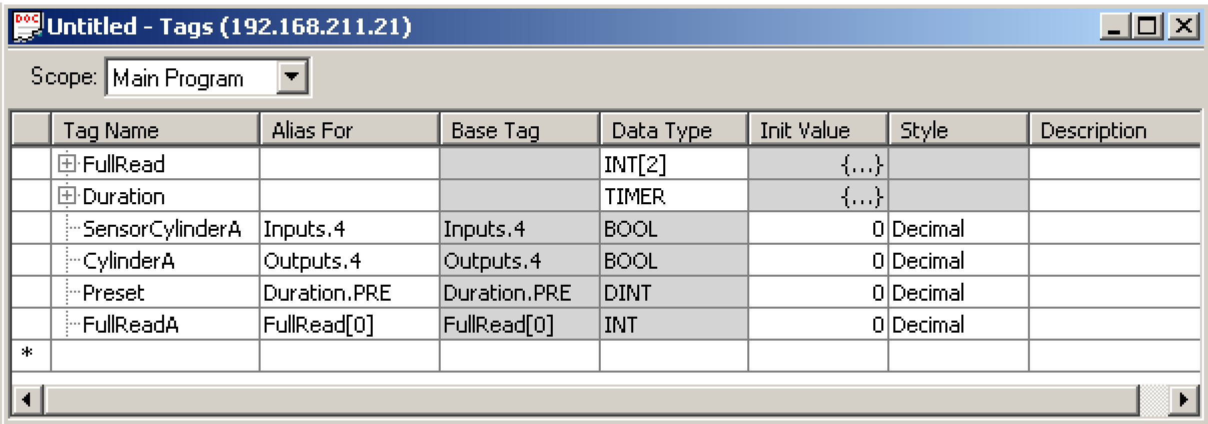
If you type an non-existent tag name for Alias For, the sign "X" will show in first column.


Untitled - Tags (192.168.211.21)

Scope: Main Program

	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	⊞ FullRead			INT[2]	{...}		
	⊞ Duration			TIMER	{...}		
	SensorCylinderA	Start					
*							

Use the steps above to assign the next tags aliases.


Untitled - Tags (192.168.211.21)

Scope: Main Program

	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	⊞ FullRead			INT[2]	{...}		
	⊞ Duration			TIMER	{...}		
	SensorCylinderA	Inputs.4	Inputs.4	BOOL	0	Decimal	
	CylinderA	Outputs.4	Outputs.4	BOOL	0	Decimal	
	Preset	Duration.PRE	Duration.PRE	DINT	0	Decimal	
	FullReadA	FullRead[0]	FullRead[0]	INT	0	Decimal	
*							

- CylinderA shows how to assign a descriptive name to controller I/O.
- Preset shows how to provide a simpler name for a complex tag.
- FullReadA is a descriptive name for an element of an array.

2.5 Non-Volatile Tag

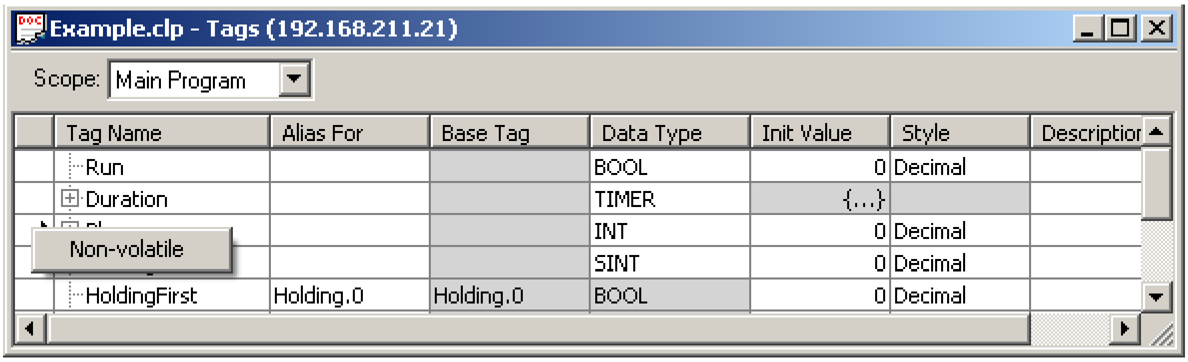
Non-volatile tags are power independent tags. They use the part of controller's Flash memory. After power-up controller cycle, the values of non-volatile tags remain unchanged.

Non-volatile tags are supported only for ConveyLinx controller.

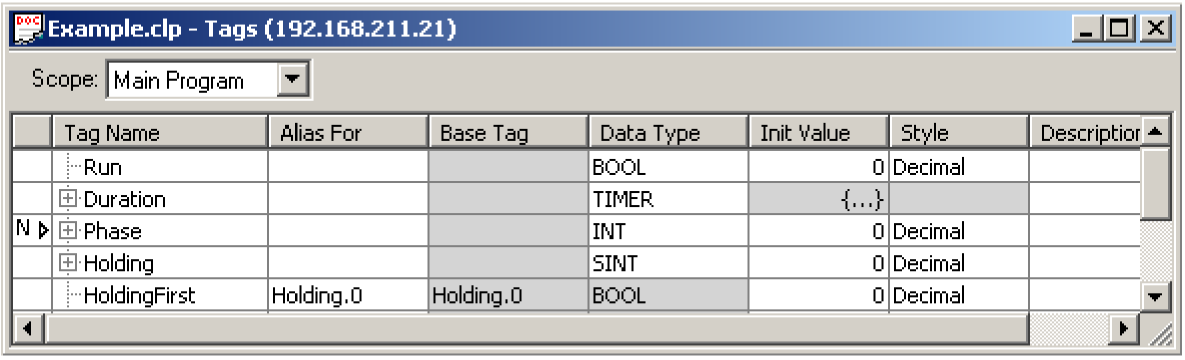
Size of all Non-volatile tags must not exceed 96 bytes.

Only Main Program tags may be non-volatile.

To make an existing tag as non-volatile, right-click on cell at the first column. The next menu appears:



Choose Non-volatile menu.



- Sign "N" in the first column shows that the tag is non-volatile.
- To make a non-volatile tag as ordinary, right-click on cell at the first column and select Non-volatile menu.

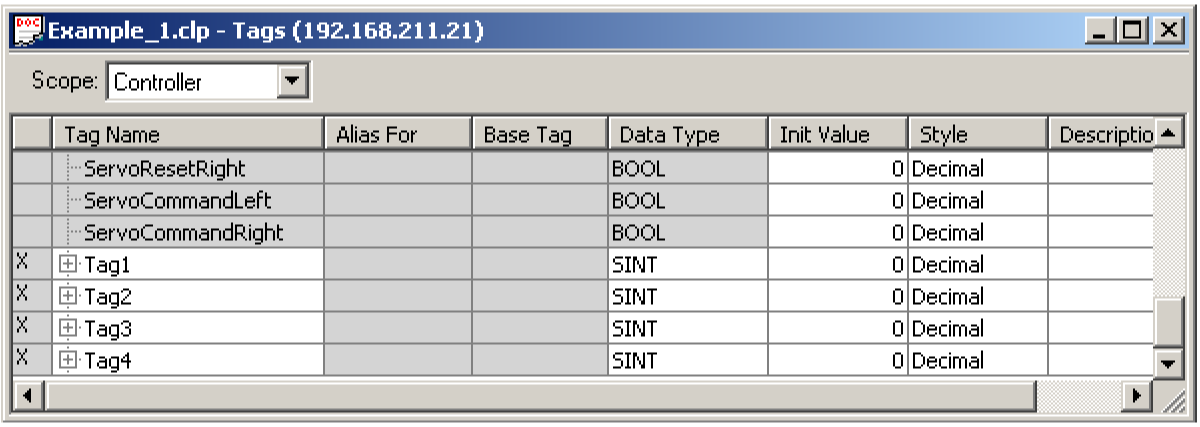
2.6 Produced and Consumed Tags

Produced and consumed tags are use to transfer data between controllers.

Produced tag sends data to another controller. Consumed tag receives data from another controller.

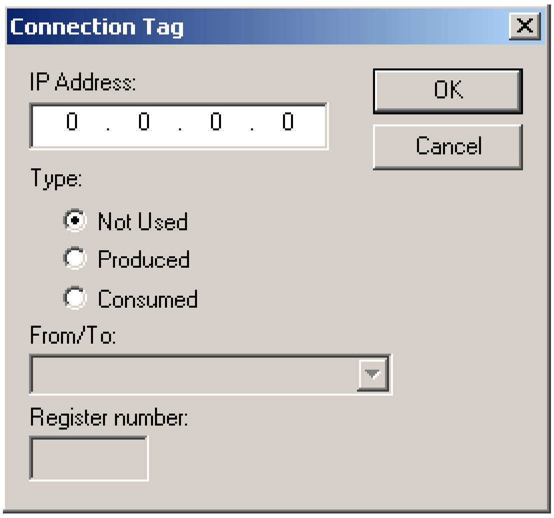
ConveyLogix Programmer supports up to four produced/consumed tags.

Information about produced/consumed tags is displayed in Tags View. To show it, change Scope to Controller.



	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Descriptio
	ServoResetRight			BOOL	0	Decimal	
	ServoCommandLeft			BOOL	0	Decimal	
	ServoCommandRight			BOOL	0	Decimal	
X	Tag1			SINT	0	Decimal	
X	Tag2			SINT	0	Decimal	
X	Tag3			SINT	0	Decimal	
X	Tag4			SINT	0	Decimal	

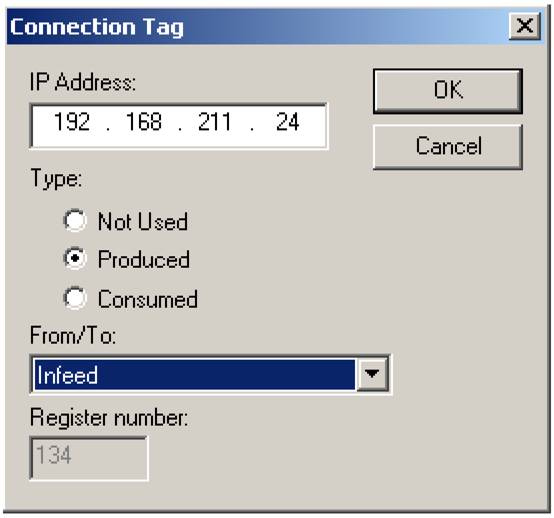
- When a produced/consumed tag is not used, the sign “X” is shown at the first column.
- To assign a produced/consumed tag, click on cell at the first column of Tag1. The next dialog box appears.



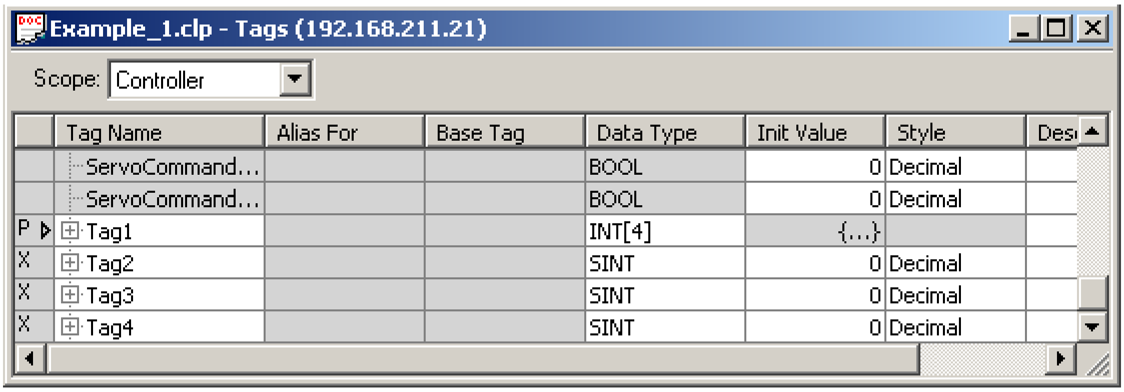
- IP Address – IP Address of the controller, which is received/sent the data.
- Type – type of the connection.
- From/To – packet of data, which is received/sent. This field is enabled when Type of the connection is Produced/Consumed.
- Register number – the first local Modbus register of the packet of data. This field is disabled and is only for information in all cases, except the last. In the last case (Register number) this field is enabled. Allowed Modbus register numbers are form 1 up to 299 or greater and equal then 1100.

2.6.1 Assign a Produced Tag

To assign a produced tag (for example Tag1) fill the above dialog with the next data.



Press OK button. Then click on Data Type cell on Tag1 and select type INT and array Dimension 1 to 4. Press OK button.



In this example, Tag1 is a block of data with size 8 bytes. Our controller will send these 8 bytes to controller with IP Address 192.168.211.24 into Modbus registers 134 to 137 (8 bytes).

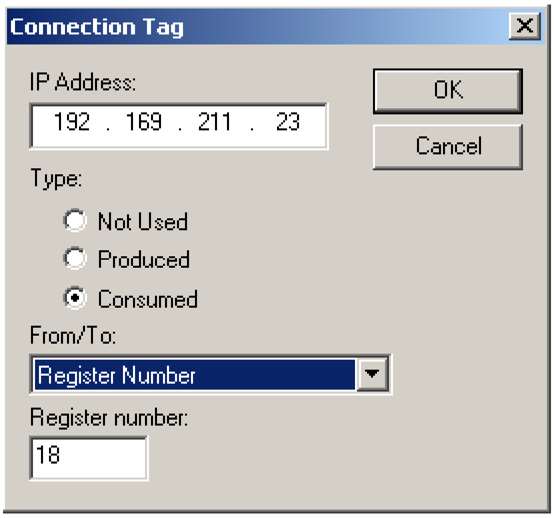
In the next table are shown starting Modbus registers of the controller, which will receive the data (in this example – 192.168.211.24). Size of the data depends of produced tag data type.

To	Starting Modbus register
Accumulate/Release Up	**104**
Accumulate/Release Down	**184**
Infeed	**134**
Discharge	**232**
Register Number	**User defined**

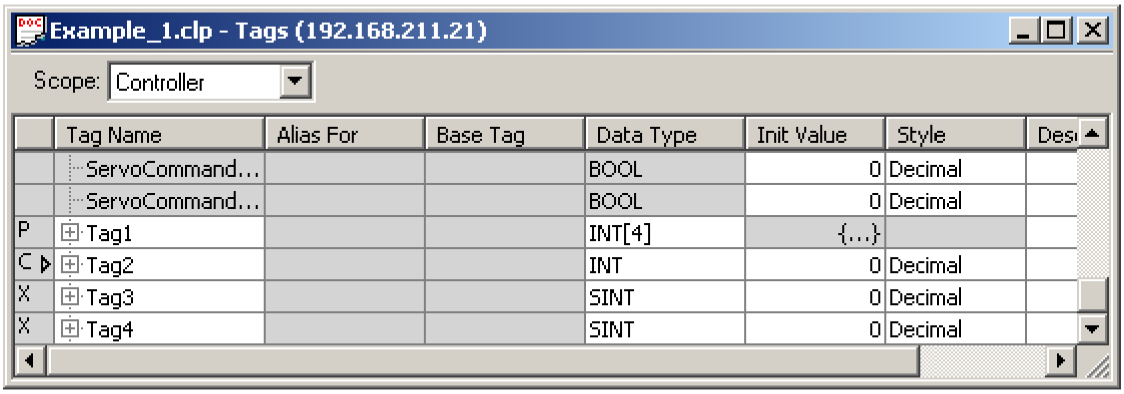
Size of the data of produced tag cannot exceed 32 bytes.

2.6.2 Assign a Consumed Tag

To assign a consumed tag (for example Tag2) fill the above dialog with the next data.



Press OK button. Then click on Data Type cell on Tag2 and select type INT.



In this example, Tag2 is a block of data with size 2 bytes. Our controller will receive these 2 bytes from controller with IP Address 192.168.211.23 from Modbus registers 18 (2 bytes).

In the next table are shown starting Modbus registers of the controller, which will send the data (in this example – 192.168.211.24). Size of the data depends of produced tag data type.

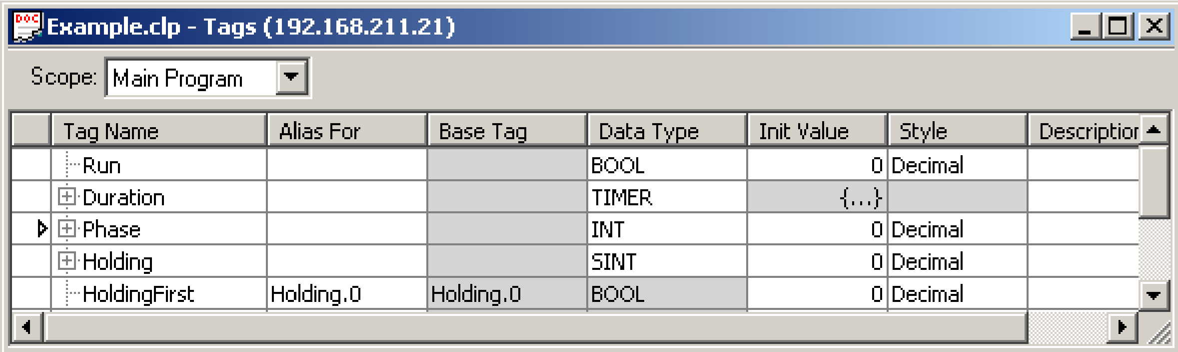
To	Starting Modbus register
Accumulate/Release Up	**106**
Accumulate/Release Down	**186**
Upstream Zone	**116**
Downstream Zone	**190**
Register Number	**User defined**

Size of the data of produced tag can not exceed 32 bytes.

You may change Tag Name, Data Type, Init Value, Style and Description in the same way as normal tags.

2.7 Delete a Tag

Click on cell at the first column of tag, which you want to delete. Sing " ▸" will appear.



This sign indicate that this tag is currently selected. To delete a selected tag, press Del key.

Conformation massage will appear.

3.0 Program Ladder Logic

3.1 Definitions

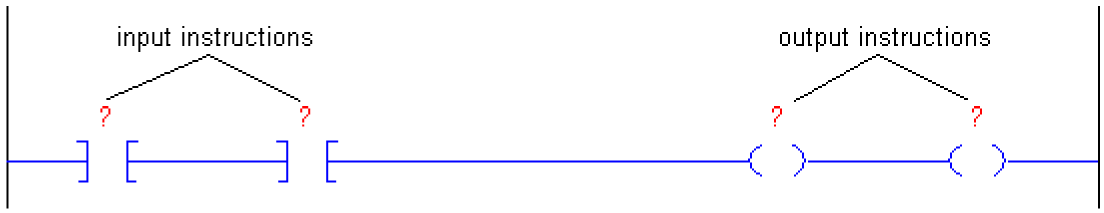
Before you write or enter ladder logic, review the following terms:

- Instruction

You organize ladder logic as rungs on a ladder and put instructions on each rung. There are two basic types of instructions:

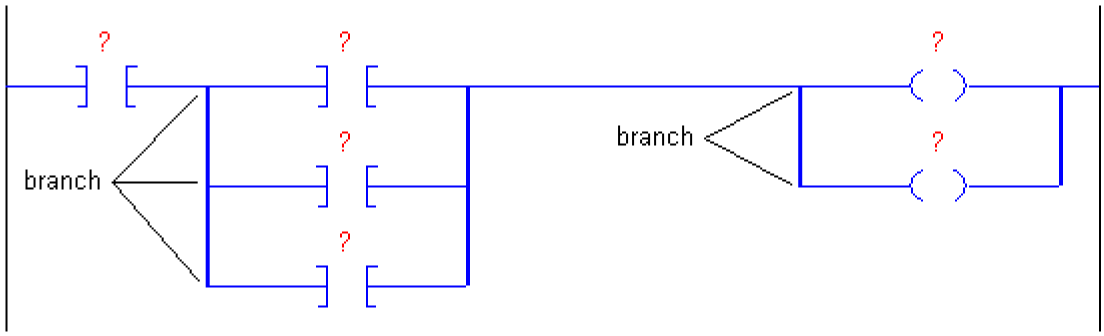
- Input instruction - An instruction that checks, compares, or examines specific conditions in your machine or process.

- Output instruction - An instruction that takes some action, such as turn on a device, turn off a device, copy data, or calculate a value.

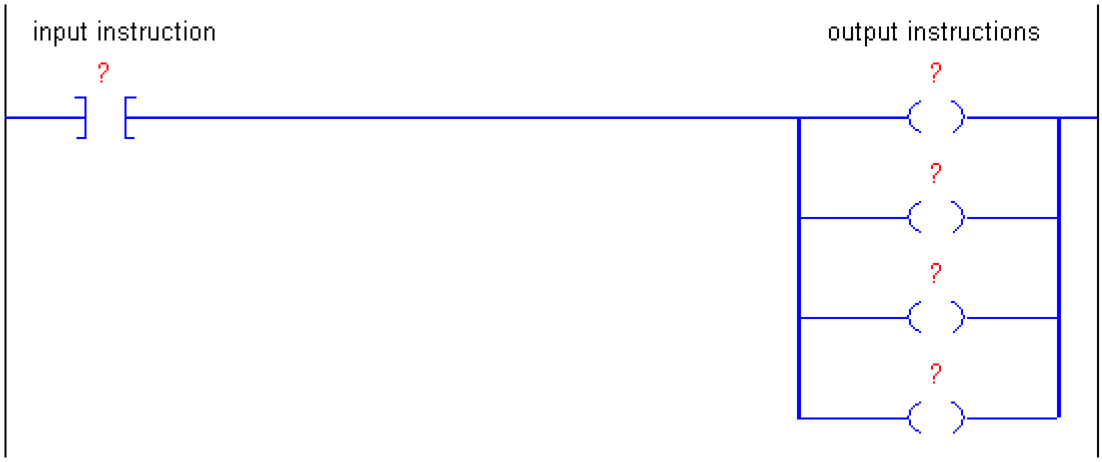


- Branch

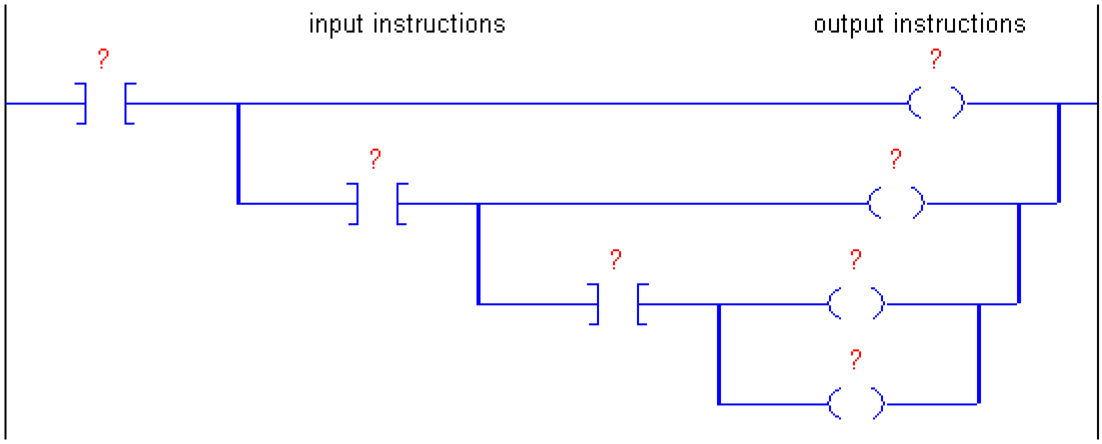
A branch is two or more instructions in parallel.



There is no limit to the number of parallel branch levels that you can enter. The following figure shows a parallel branch with four levels. The main rung is the first branch level, followed by three additional branches.

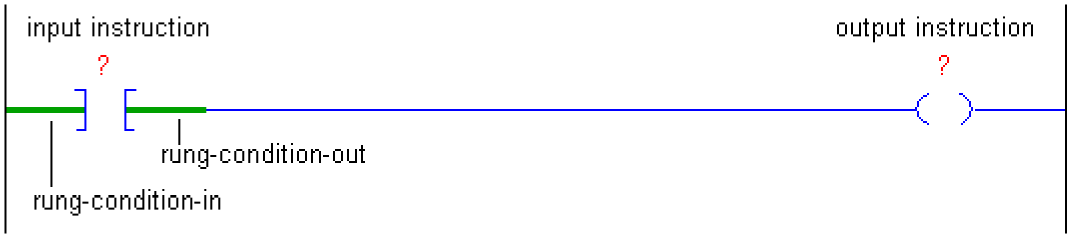


You can nest branches to levels. The following figure shows a nested branch. The bottom output instruction is on a nested branch that is three levels deep.



- Rung Condition

The controller evaluates ladder instructions based on the rung condition preceding the instruction (rung-condition-in). Based on the rung-condition-in and the instruction, the controller sets the rung condition following the instruction (rung-condition-out), which in turn, affects any subsequent instruction.



Only input instructions affect the rung-condition-in of subsequent instructions on the rung:

If the rung-condition-in to an input instruction is true, the controller evaluates the instruction and sets the rung-condition-out to match the results of the evaluation.

If the instruction evaluates to true, the rung-condition-out is true.

If the instruction evaluates to false, the rung-condition-out is false.

An output instruction does not change the rung-condition-out.

If the rung-condition-in to an output instruction is true, the rung-condition-out is set to true.

If the rung-condition-in to an output instruction is false, the rung-condition-out is set to false.

- Prescan

The controller also prescans instructions. Prescan is a special scan of all routines in the controller. The controller scans all main routines during prescan, but ignores jumps that could skip the execution of instructions. The controller uses prescan of relay ladder instructions to reset non-retentive I/O and internal values.

During prescan, input values are not current and outputs are not written. The following conditions generate prescan:

- Toggle from Program to Run mode.
- Automatically enter Run mode from a power-up condition.

Prescan does not occur for a program when:

- The program becomes scheduled while the controller is running.
- The program is unscheduled when the controller enters Run mode.

3.2 Write Ladder Logic

To develop your ladder logic, perform the following actions:

- Choose the Required Instructions;
- Arrange the Input Instructions;
- Arrange the Output Instructions;
- Choose a Tag Name for an Operand(s).

Separate the conditions to check from the action to take. Choose the appropriate input instruction for each condition and the appropriate output instruction for each action.

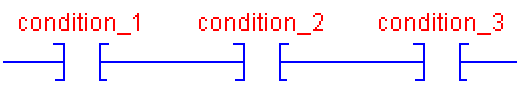
To choose specific instructions, see Chapter 4.

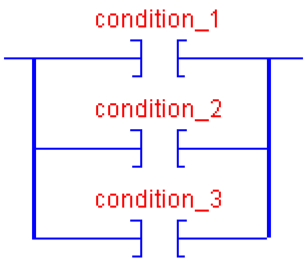
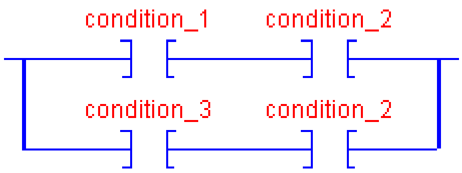
The examples in this chapter use two simple instructions to help you learn how to write ladder logic. The rules that you learn for these instructions apply to all other instructions.

Symbol	Name	Mnemonic	Description	
—] [—	Examine If Closed	XIC	An input instruction that looks at one bit of data.	
			If the bit is:	**Then the instruction (rung-condition-out) is:**
			on (1)	true
			off (0)	false
—( )—	Output Energize	OTE	An output instruction that controls one bit of data.	
			If the instructions to the left (rung-condition-in) are:	**Then the instruction turns the bit:**
			true	on (1)
			false	off (0)

3.2.1 Arrange the Input Instructions

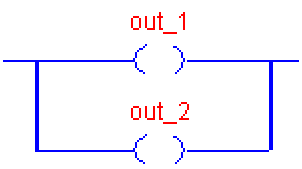
Arrange the input instructions on a rung using the following table.

To check multiple input conditions when:	Arrange the input instructions:
all conditions must be met in order to take action For example, If condition_1 AND condition_2 AND condition_3…	In series: 
any one of several conditions must be met in order	In parallel:

to take action For example, If condition_1 OR condition_2 OR condition_3…	
there is a combination of the above For example, If condition_1 AND condition_2… OR If condition_3 AND condition_2…	In combination: 

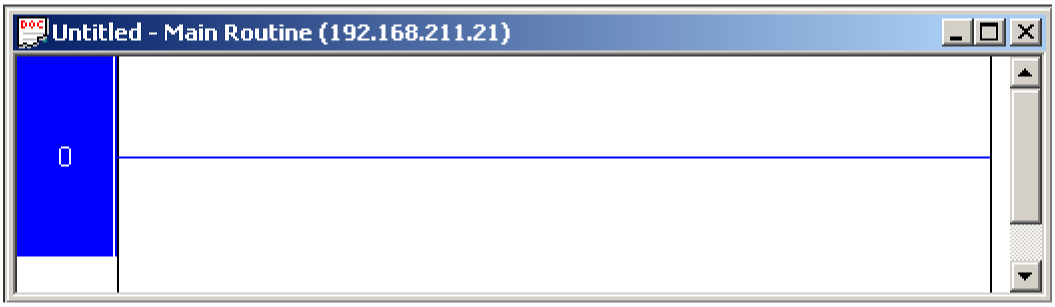
3.2.2 Arrange the Output Instructions

Place at least one output instruction to the right of the input instructions. You can enter multiple output instructions per rung of logic, as follows:

Option:	**Example:**
sequence on the rung (serial)	
branches (parallel)	
between input instructions, as long as the last instruction on the rung is an output instruction	

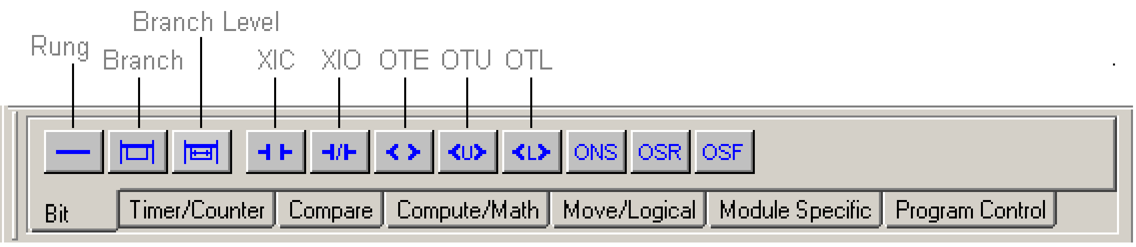
3.3 Enter Ladder Logic

A new routine contains a rung that is ready for instructions.



When rung is selected, the cursor is blue. When you add an instruction or branch, it appears to the right of the cursor.

Use the Instruction Bar to add a ladder logic element to your routine.



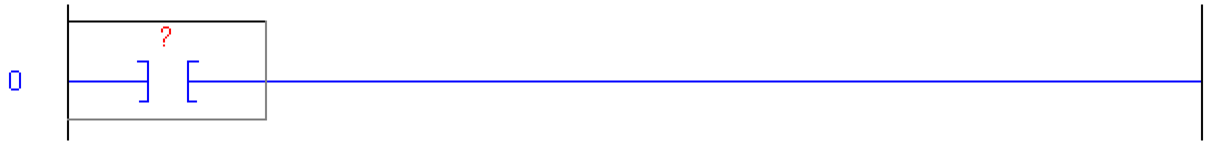
3.3.1 Append an Element

There is three ways to append an element:

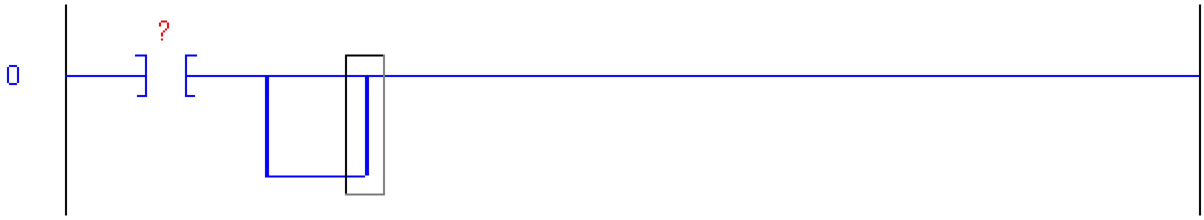
- using buttons from Instruction Bar;
- drag & drop an existing element;
- copy and paste an existing element.

Example: This example shows how to append elements, using methods above.

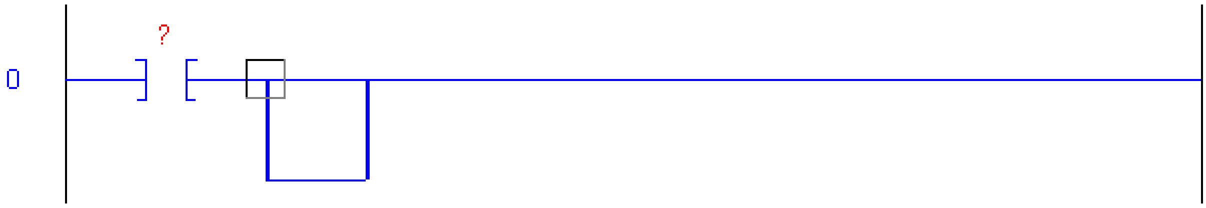
Click on XIC button from Instruction Bar.



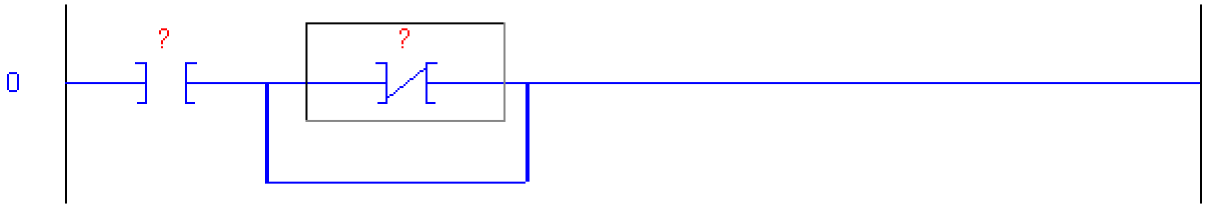
XIC element is appended and cursor is positioned around it. To add a parallel combination after selected XIC, click on Branch button.



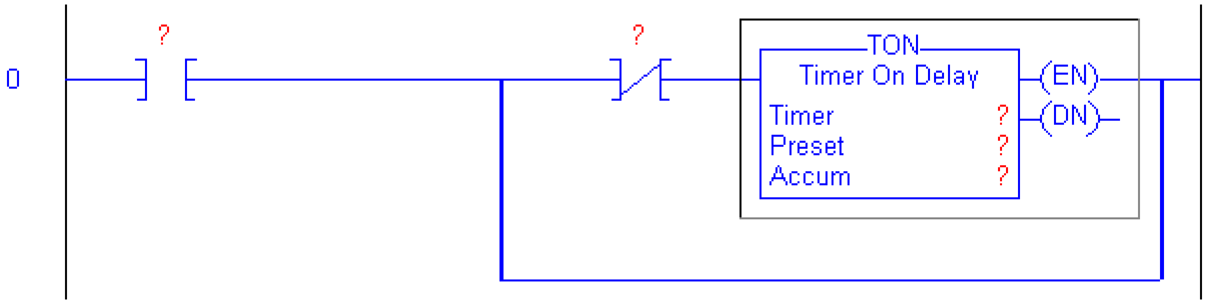
To append elements on first branch select on the beginning of the first branch.



Click on XIO button from Instruction Bar.

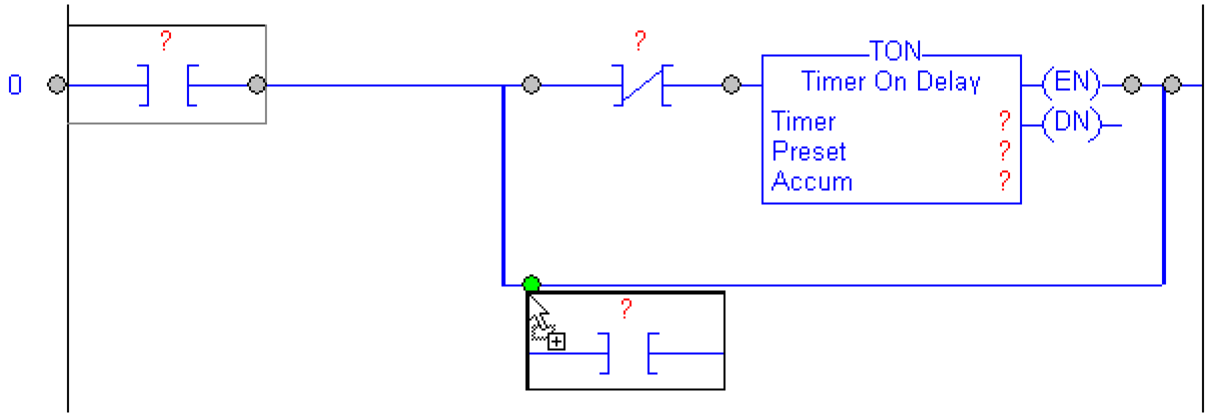


To append Timer On Delay element, select Timer/Counter tab from Instruction Bar and then click on TON button. Now parallel combination is on the left part on Ladder View because contains only input instruction.

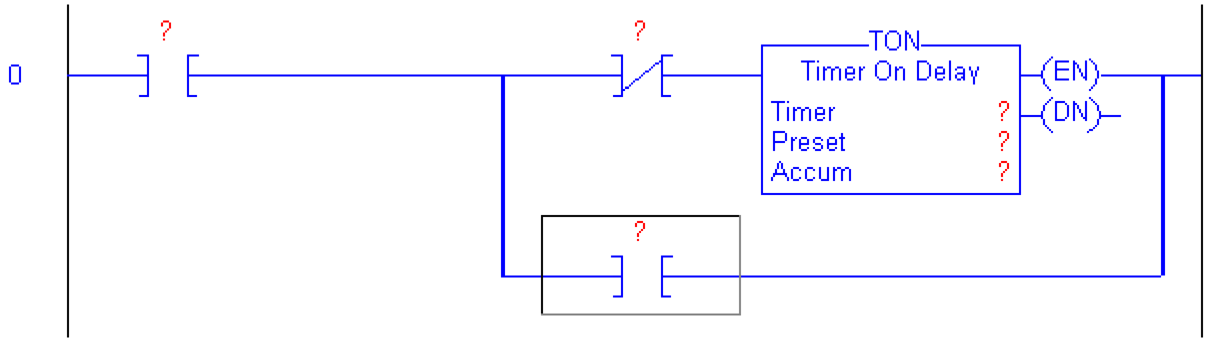


The last instruction in parallel combination is output instruction (TON) therefore parallel combination is placed on the left part on Ladder View.

Now XIC element will copy on the second branch by drag & drop operation. Select XIC element. Press left mouse button inside the selection, press CTRL key and then drag mouse. The cursor will change as on picture below.



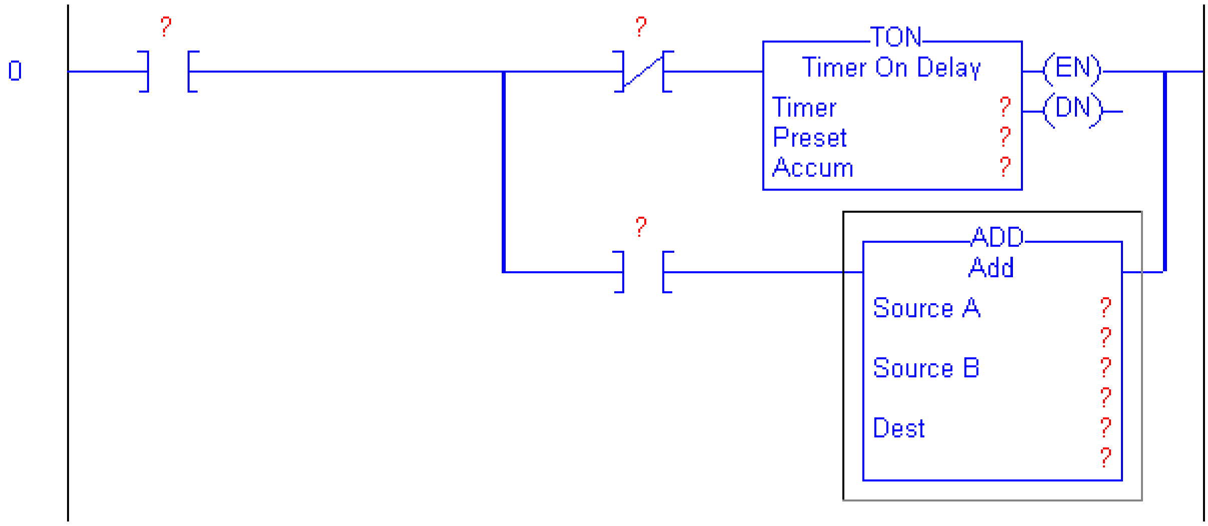
Grey circles show the possible places to copy the element. Grey circle is the chosen place. Release left mouse button on the beginning on the second branch.



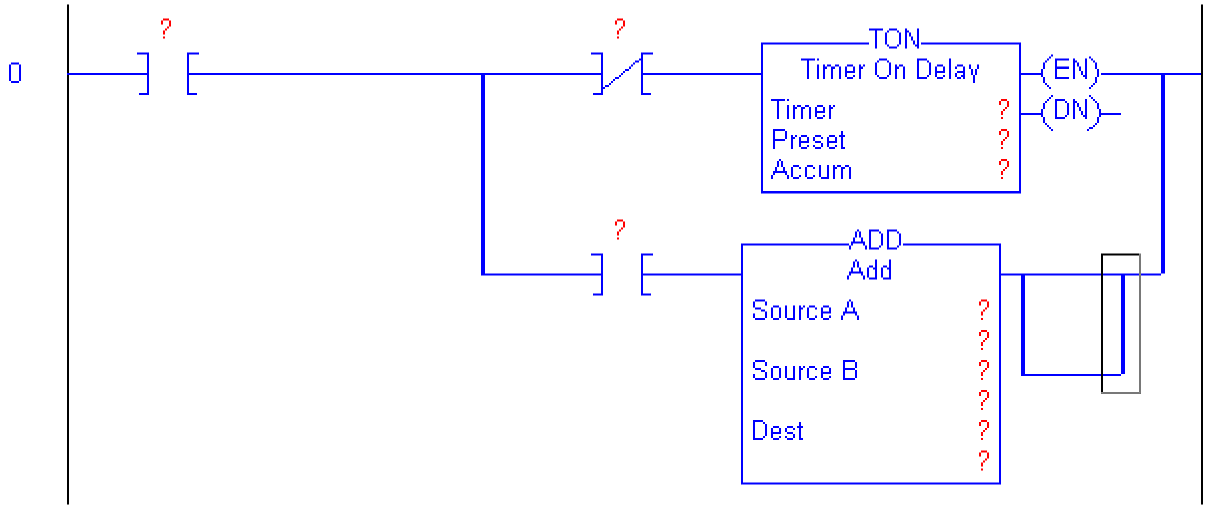
XIC element will copy on the beginning on the second branch.

If Ctrl key is not pressed, the selected element will move to chosen place.

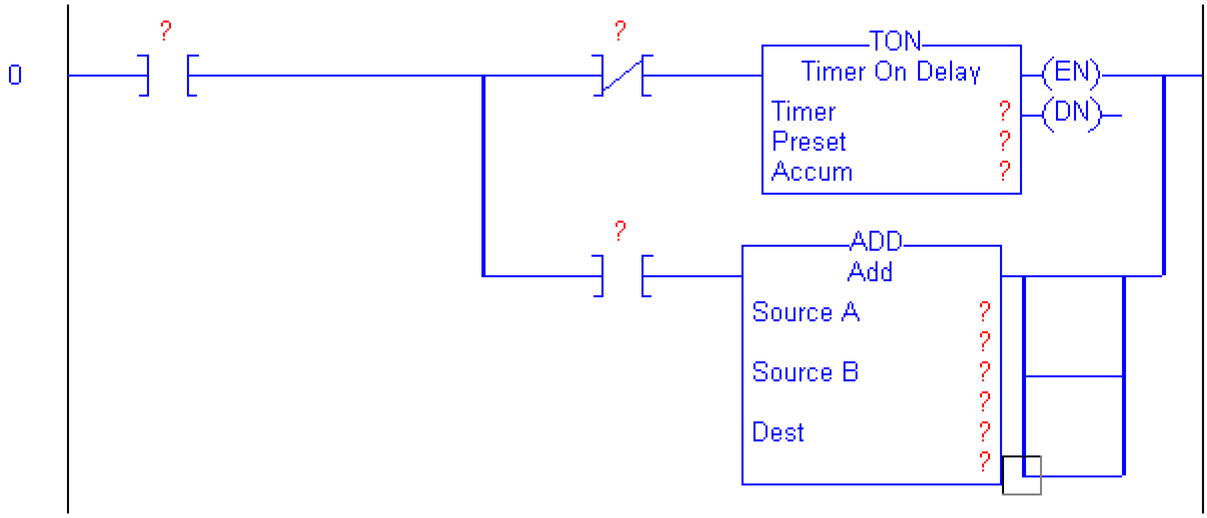
To append Add element, select Compute/Math tab from Instruction Bar and then click on ADD button.



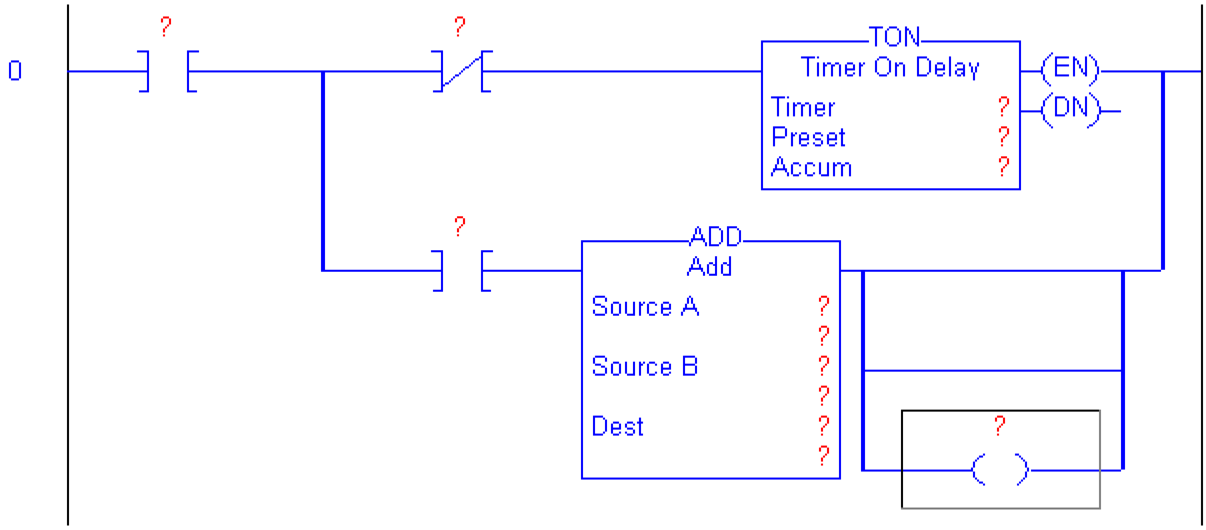
Click Branch button to add a parallel combinations after Add element.



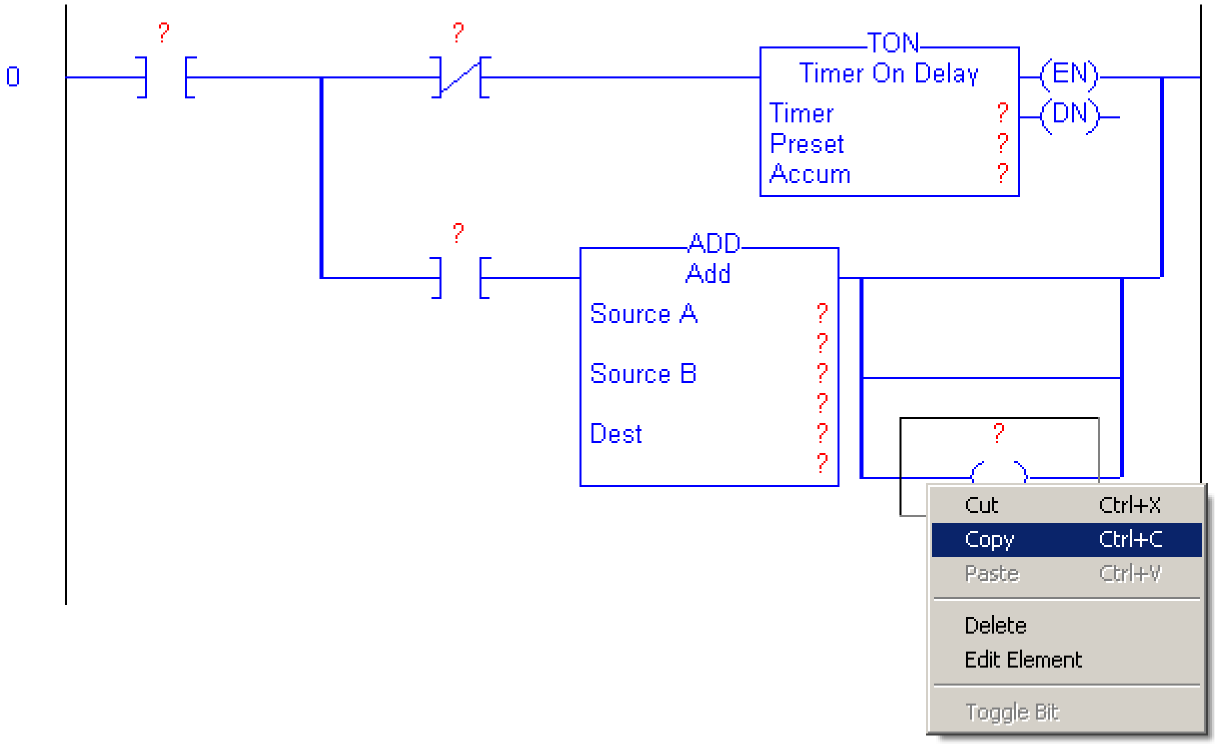
Then select on the beginning of any branch (for example of the second branch). Click on Branch Level button to append a branch. Branch is appended after the branch which element is selected.



Change Instruction Bar tab again to Bits and click on OTE button.

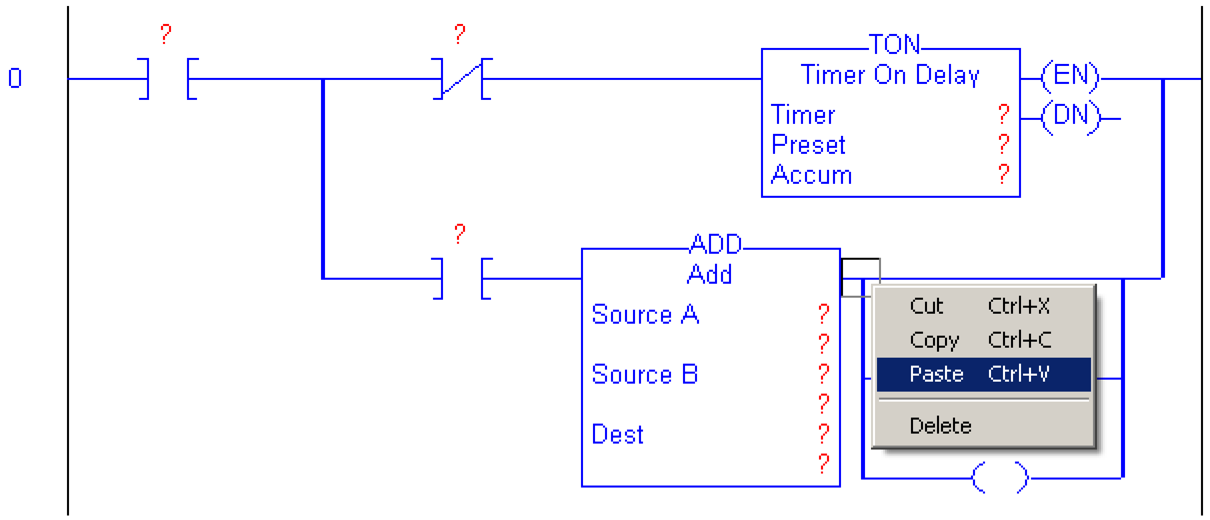


Now OTE elements will append by Copy/Paste operation. Right click on OTE element. The next menu will show:



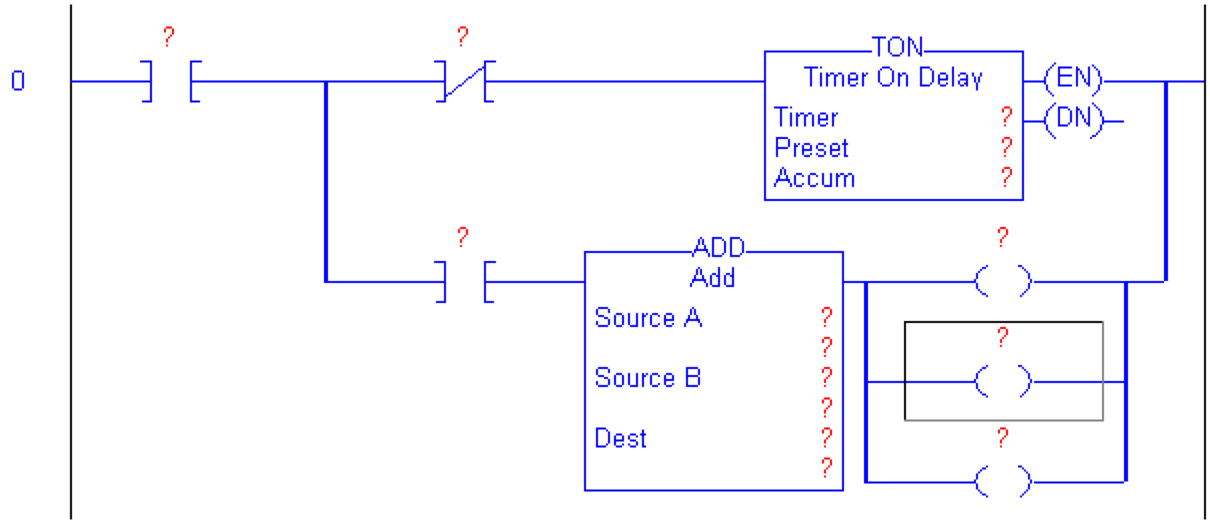
Click on menu Copy. OTE element copies into Clipboard.

Select the beginning of the first branch and right click in selection area.



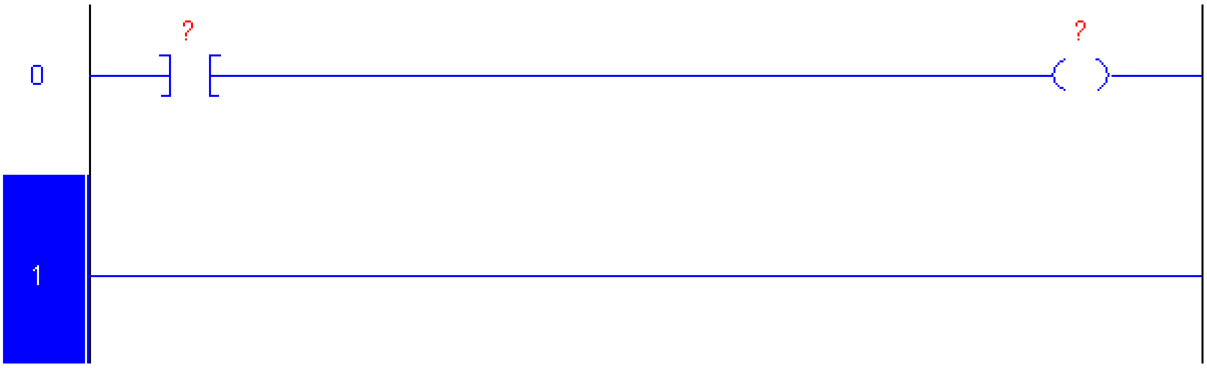
Click on menu Paste. OTE element copies from Clipboard.

Do the same to append OTE element on the second branch.



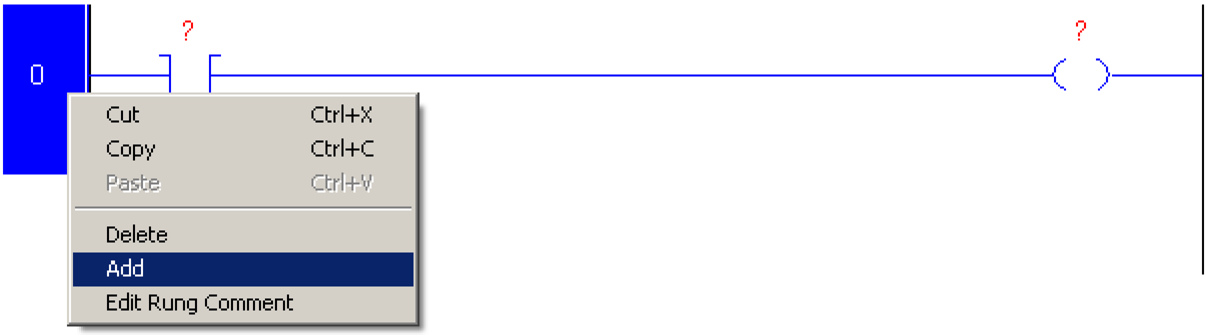
3.3.2 Append a Rung

To append a rung, click on button Rung from Instruction Bar.

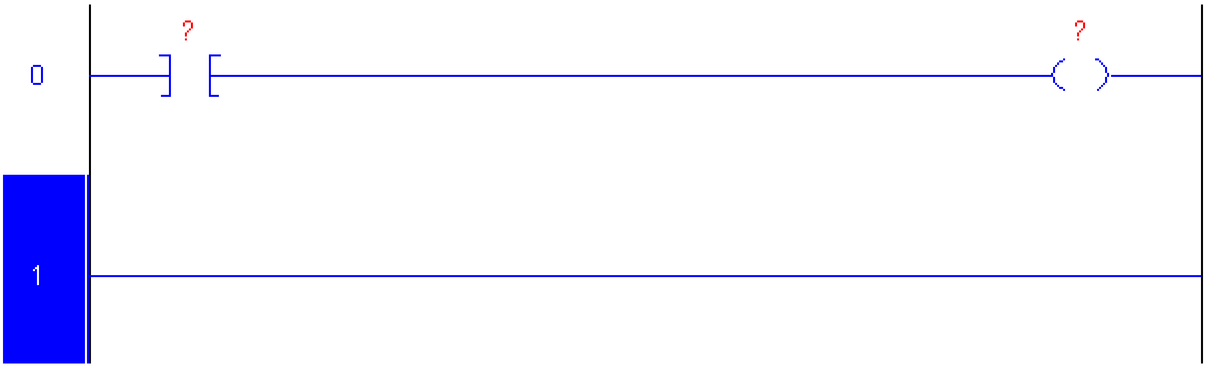


In this example rung will append on the end on ladder logic. Rung appends/insert after rung where the selected element is.

There is a second way to append a rung. Right-click on the rectangle before input power line of the desired rung and select Add menu.



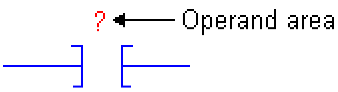
The new Rung (1) will append after the selected (0).



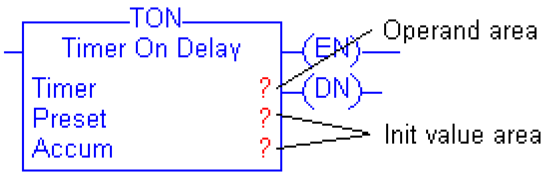
3.4 Assign Operands

Every element has one to three operands. Every operand has an operand area.

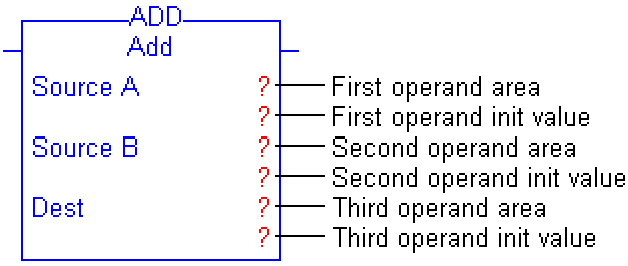
Most usable bit instructions (like XIC, XIO, OTE, OTU and OTL) have only one operand.



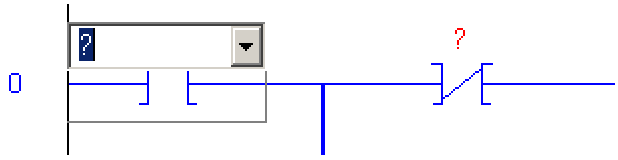
Timers and counters also have one operand.



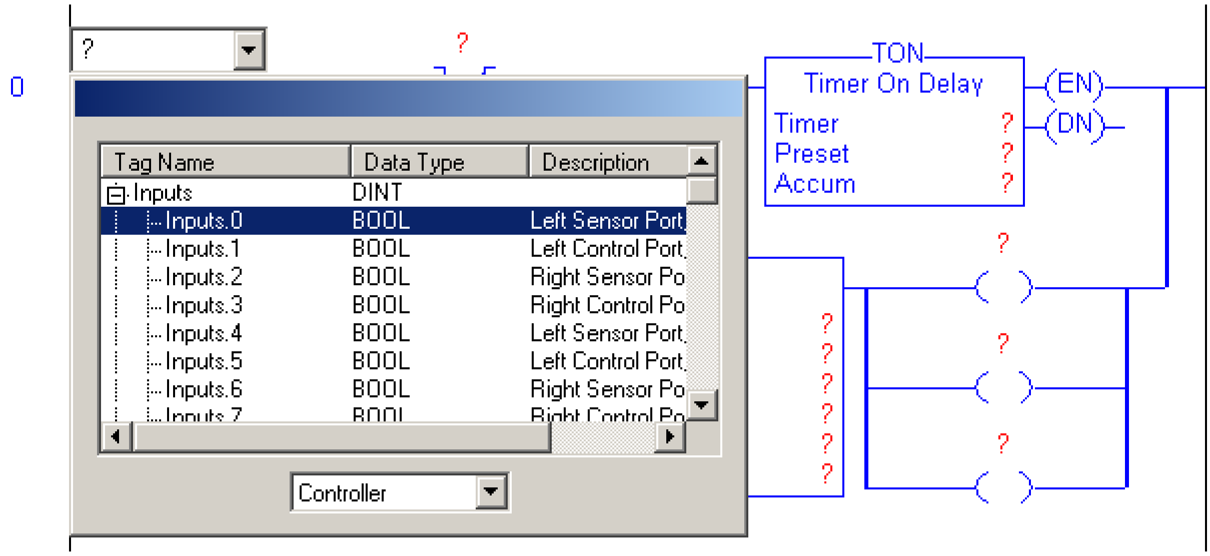
Mathematical elements (ADD, SUB, MUL and DIV) have three operands.



Unassigned operand is represents by red "?". To assign an operand double-click on operand area.

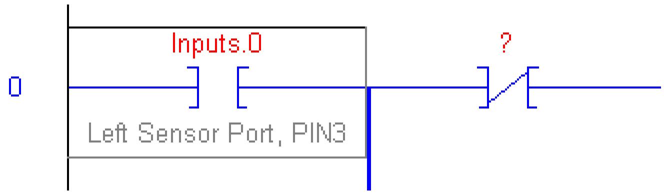


Type the operand name or open the combo-box to select the name from existing tags.



Because in example has no entering tags change the scope to Controller (combo-box at the bottom), open Inputs tag and select for example Inputs.0. Double-click on it or press Enter.

Tag name will put on the operand edit box. Click outside or press Enter to confirm operand name.



Inputs.0 is appeared in operand area. Tag's description is shown bellow the element (if any).

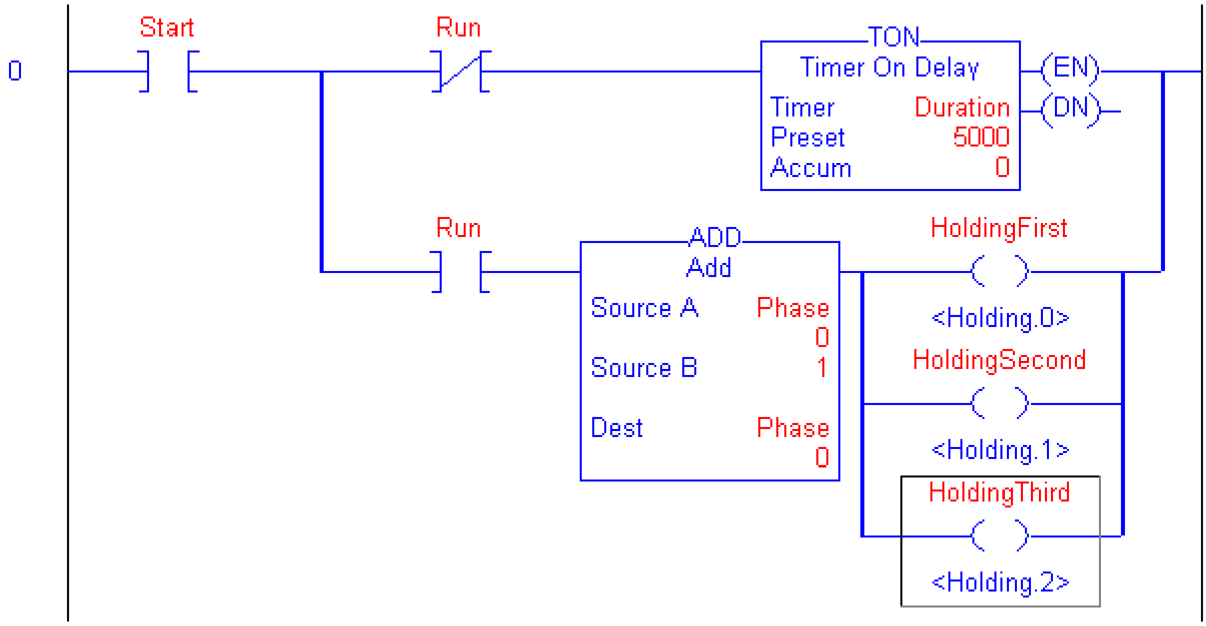
Now we will open Tags View and will created tag for this example usage.

Example.clp - Tags (192.168.211.21)

Scope: Main Program

	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	Run			BOOL	0	Decimal	
	⊟ Duration			TIMER	{...}		
	⊞ Duration.PRE			DINT	5000	Decimal	
	⊞ Duration.ACC			DINT	0	Decimal	
	Duration.EN			BOOL	0	Decimal	
	Duration.TT			BOOL	0	Decimal	
	Duration.DN			BOOL	0	Decimal	
	⊞ Phase			INT	0	Decimal	
	⊞ Holding			SINT	0	Decimal	
	HoldingFirst	Holding.0	Holding.0	BOOL	0	Decimal	
	HoldingSecond	Holding.1	Holding.1	BOOL	0	Decimal	
	HoldingThird	Holding.2	Holding.2	BOOL	0	Decimal	
▶	Start			BOOL	0	Decimal	
*							

Assign tags to element as a picture below.



Operands for XIC and XIO elements are tags with BOOL type.

TON has only one operand – Duration, which is a TIMER structure. Preset shows the init value of Duration.PRE element from Duration structure. Accum shows the Duration.ACC init value.

First operand (Source A) and Third operand (Dest) are tag Phase, which has INT type. For Second operand (Source B) is typed immediate (constant) value.

Tags for OTE elements in parallel are respectively HoldingFirst, HoldingSecond and HoldingThird. These tags are aliases and below the elements are shown base tag names.

If tag type is not supported to element operand, "?" symbol shows in init value area.

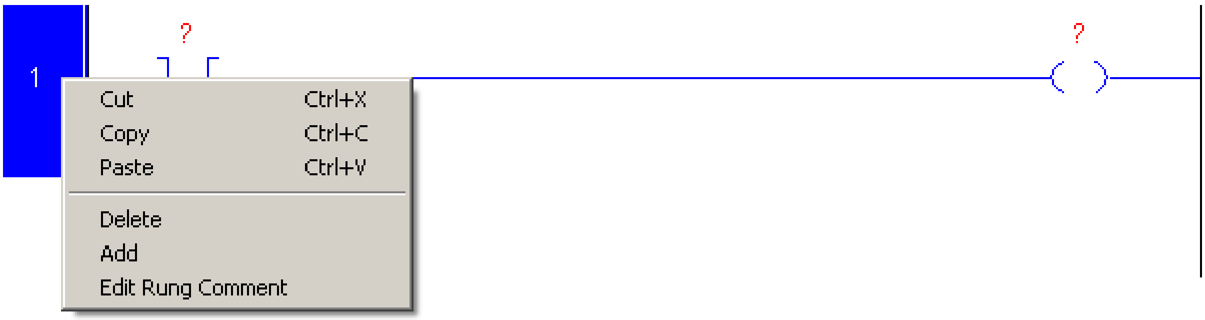
If operand is a constant, init value area below is hidden. If a constant is not in the type range, "?" symbol shows in init value area.

When init value for a tag is changed in Tags View, corresponding init values in Ladder View are refreshed immediately. Likewise, if init value in Ladder View is changed, it reflects to init value in Tags View.

3.5 Editing Ladder Logic

3.5.1 Edit a Rung

Right-click in the rectangle before input power line of the desired rung. The next menu appears.



Use Cut or Copy menu to put the selected rung into Clipboard. When use cut operation, the selected rung deletes from the ladder logic.

Paste menu is enabled only when rung is put to Clipboard.

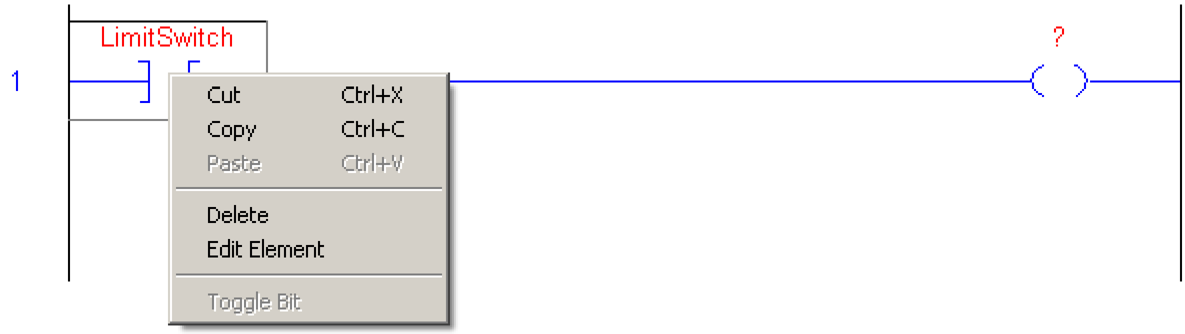
Cut, Copy and Paste menus are duplicated in Edit menu.

Select Delete menu to delete a rung.

There is a second way to delete a rung. Select a rung (right-click on the rectangle before input power line) and press Del key.

3.5.2 Edit an Element

To edit an element, simply right-click on it. The next menu appears.



Use Cut or Copy menu to put the selected element into Clipboard. When use cut operation, the selected element deletes from the ladder logic.

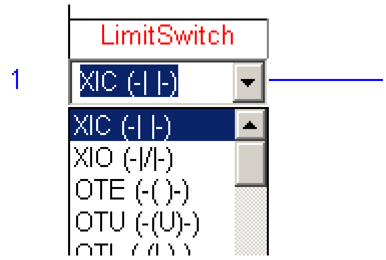
Paste menu is enabled only when element is put to Clipboard.

Cut, Copy and Paste menus are duplicated in Edit menu.

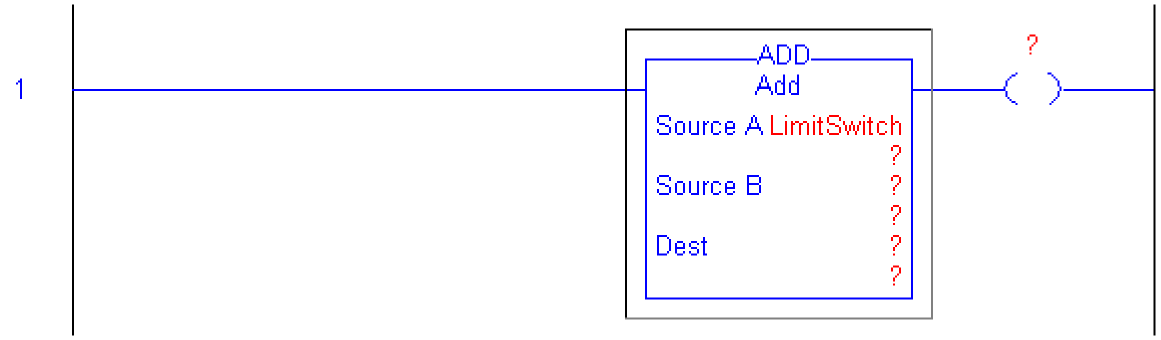
Select Delete menu to delete a rung.

The second way to delete an element is to select an element and press Del key.

To change an element instruction, select Edit Element menu. Combo-box with all supported instructions appears.

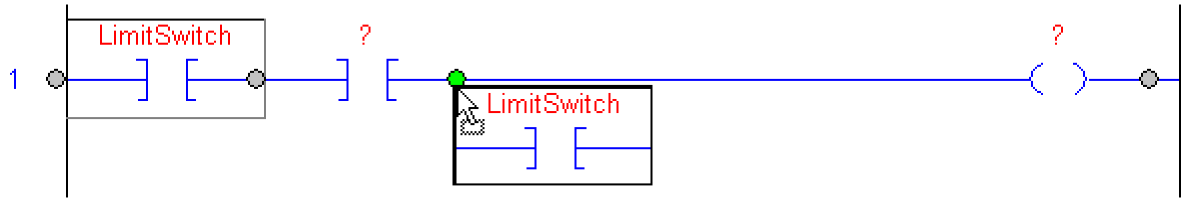


Select the desired instruction (for example ADD instruction) and click outside the combo-box or press Enter key. If you want to cancel the changing, press Esc key.

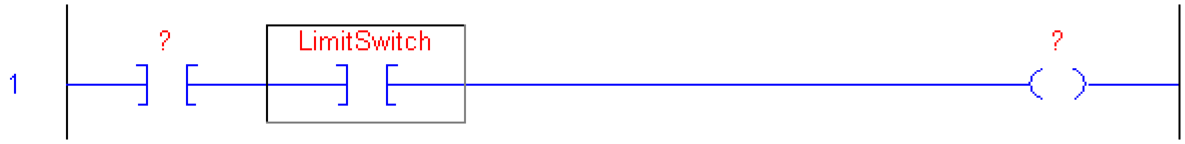


Operands from old instruction are copied to operands to the new instruction. Count of copied operands is equal of less count of operands of two instructions.

To move an element, click on it and drag over the ladder logic.

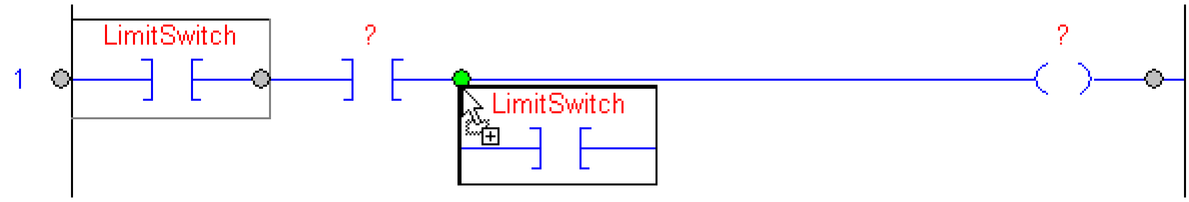


The grey circles show the possible places where you can move the dragged element. The current place is displayed in green circle. Drop the element by releasing the left mouse button.

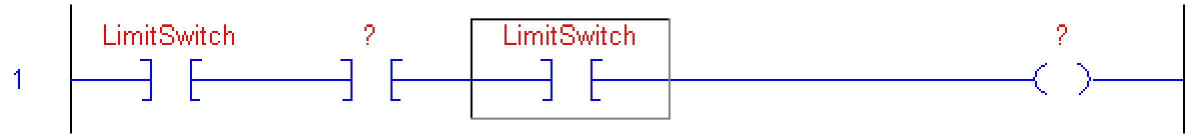


There is a way to copy an element by using drag & drop operation. In this way copied element doesn't put into Clipboard.

Press Ctrl key and then drag the element. Also, you may press Ctrl key during the drag operation (on the cursor displayed sign "+").



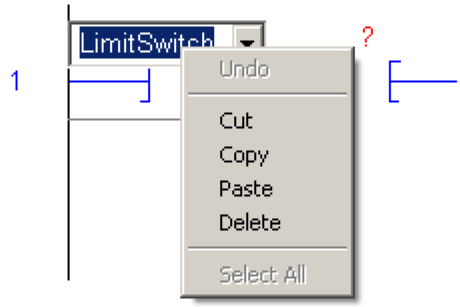
Drop the element.



3.5.3 Edit an Operand

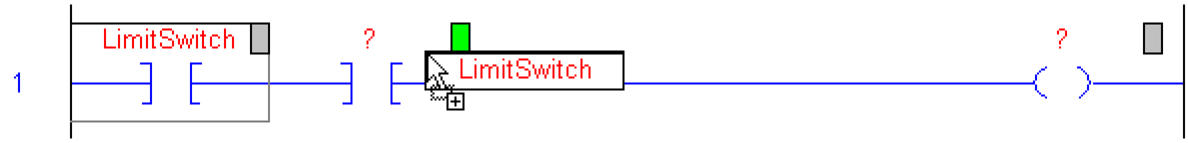
Editing an operand is performed by double-clicking on operand area as the same way, described in point 3.4.

You may cut, copy, paste and delete the text from/to operand edit-box using right-click menu commands.

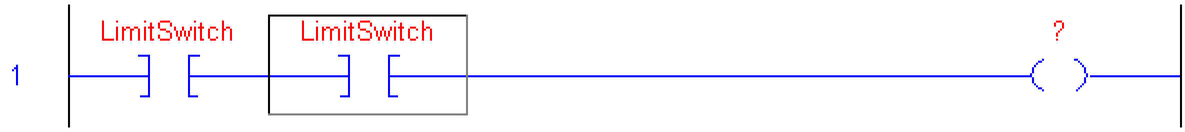


The second way to copy an operand is by using drag & drop operation.

Click on operand area and drag over the ladder logic.

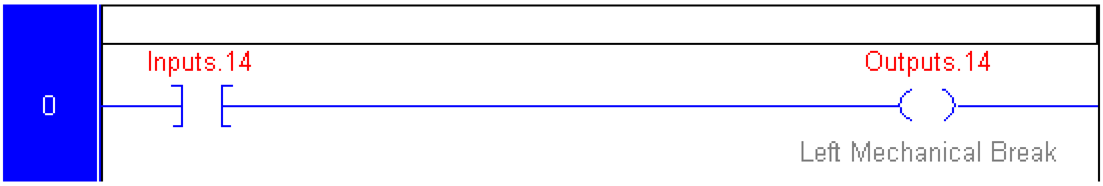


The grey rectangles show the possible places where you can move the dragged operand. The current place is displayed in green rectangle. Drop the operand by releasing the left mouse button.



3.6 Enter Rung Comment

To enter/edit rung comment double-click in marked rectangle (picture below) above the rung.



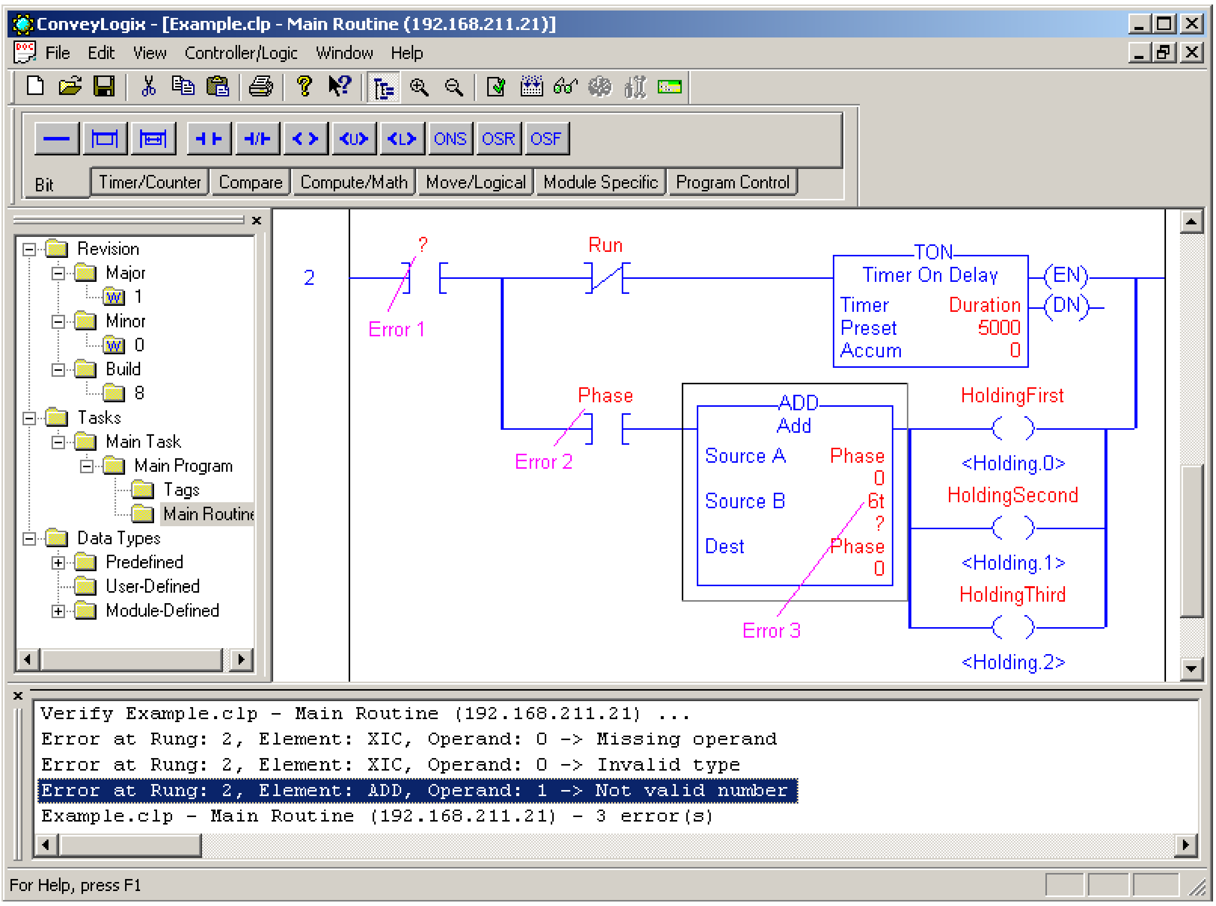
Type the comment text and then press Enter key or click outside.

3.7 Verify the Routine

As you program your routine, periodically you may check your work.

Choose Controller/Logic / Verify Program menu or click on icon. Your program will be check and the result will display in Output window.

On the picture below is shown program with 3 errors. For example errors are marked and enumerated in mangenta color.



Double-click on error in Output window to select an element where is the error. In this example the selected error is related to ADD element.

Every error line contains the next information of the error:

- Rung number;
- Element instruction;
- Number of operand – started at 0;
- Error description.

Here is the explanation of errors in this example:

Error 1 – there are no assigned tag to the XIC instruction operand.

Error 2 – the operand of XIC instruction allow BOOL tag, but type of tag Phase is INT.

Error 3 – it is expected for Source B operand to be entered a immediate (constant) value, but 6t is not a constant.

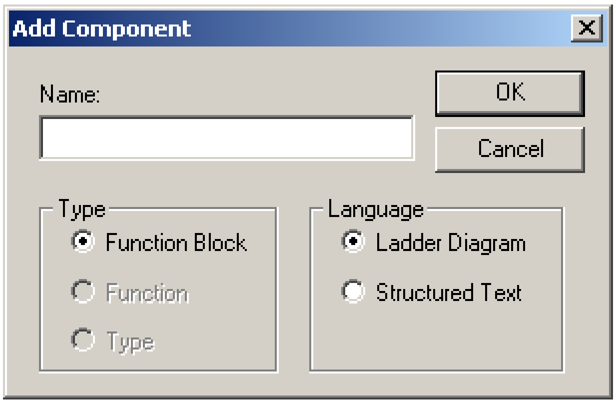
If the routine reports error, Download Program will break.

4.0 Function Blocks

Function block (FB) is a programmable organization unit which, when executed, yields one or more values. ConveyLogix Programmer uses two screens to represent FB definition. FB Routine contains your program instructions and FB Tags – FB parameters. Function block is called from Main Program or other FB by defined instance (tag) in the controller's memory.

4.1 Creating a Function Block

To create a Function block right click on Function blocks in Project Bar tree and select Add menu. The following dialog box appears:



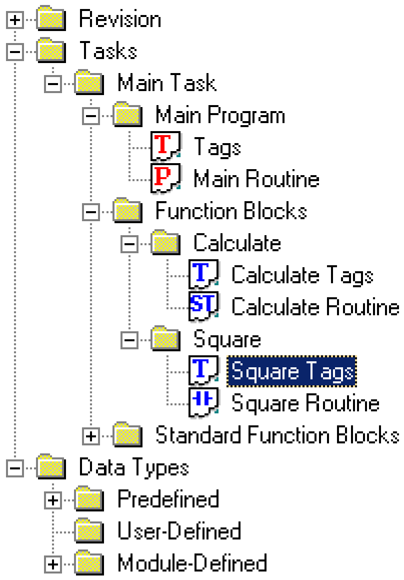
A FB is characterized with two elements:

- Name – unique name of Function block type;
- Language – program language of Function block instructions.

Press OK button to create the Function block type.

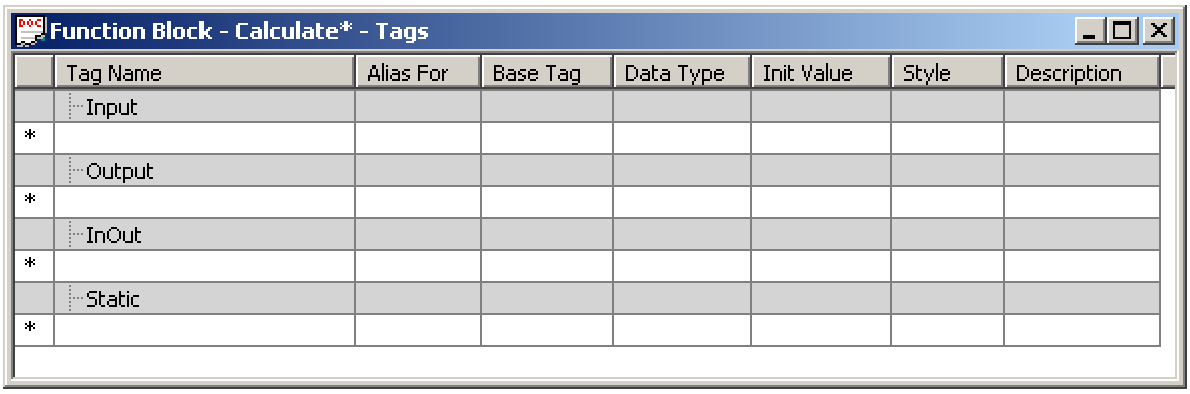
For example:

Create two function blocks named Calculate, used Structured Text and Square – used Ladder Diagram. They are added to Project Bar tree.



4.2 Function Block Parameters

To view and edit parameters double-click on created Function block Tags in Project Bar tree.



The block parameters are defined in the interface of the called block. These parameters are referred to as formal parameters. They are placeholders for the parameters that are transferred to the block when it is called. The parameters transferred to the block when it is called are referred to as actual parameters.

The following rules apply to the use of block parameters within the block:

- Input parameters may only be read.
- Output parameters may only be written.
- In/out parameters may be read and written.

Static parameters are accessible only inside of an instance of a function block.

Input, Output and InOut parameters are accessible outside of an instance of a function block.

For example:

Add parameters to FB Calculate as the picture below:

Function Block - Calculate - Tags

	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	⊟ Input						
	⊞ ParamA			INT	200	Decimal	First Param
	⊞ ParamB			INT	0	Decimal	Second Param
*							
	⊟ Output						
	⊞ Sum			DINT	0	Decimal	Sum
*							
	⊟ InOut						
	⊞ ParamC			INT	0	Decimal	Third Param
*							
	⊟ Static						
	⊞ Const			SINT	100	Decimal	Constant
*							

4.3 Function Block Program

Function block program represents a set of instructions, which are executed on function block instance.

ConveyLogix supports two languages for function block program:

- Ladder Diagram (LD) – enables the programmable controller to test and modify data by means of graphic symbols. These symbols are laid out in networks in a similar manner to a “rung” of a relay ladder logic diagram. LD networks are bounded on the left and right by power rails;
- Structured Text (ST) – a textural programming language, derived from Pascal.

For example:

Function Block - Calculate - Routine

IF ELSIF CASE FOR

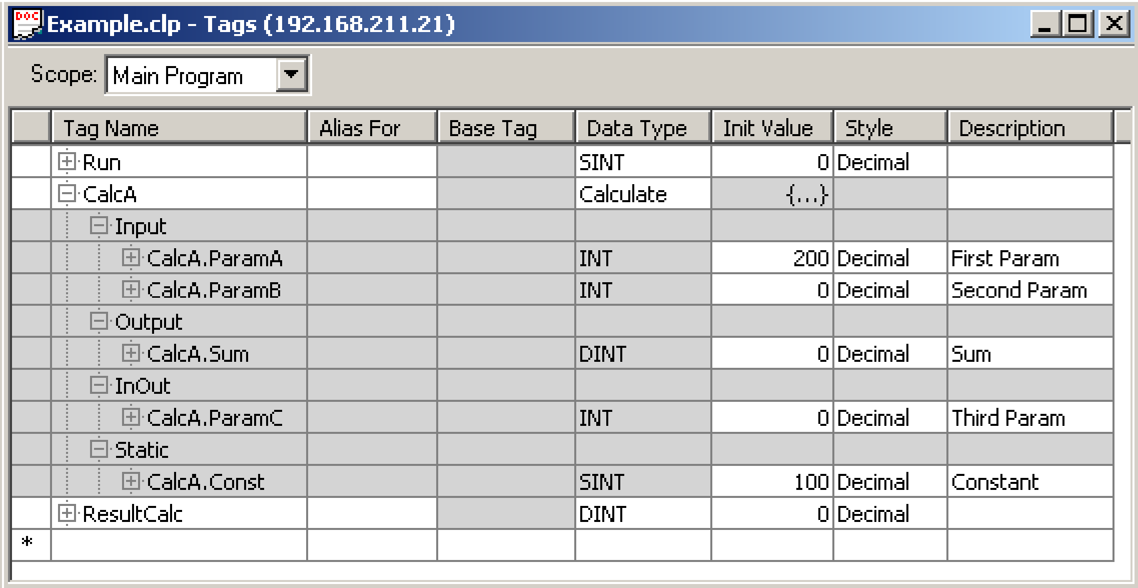
```
Sum := ParamA * ParamA + ParamB + ParamC + Const;

IF Sum > 5000 THEN
        Sum := 5000;
END_IF;

```

4.4 Instances of Function Blocks

A call of a function block is referred to as an instance. An instance of function block is a block in controller's memory (tag) which type is a function block name.

For example:

Add an instance of FB Calculate in Main Tags – first define a tag named CalcA and then change its type to Calculate.


Example.clp - Tags (192.168.211.21)

Scope: Main Program

Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
Run			SINT	0	Decimal	
CalcA			Calculate	{...}		
Input						
CalcA.ParamA			INT	200	Decimal	First Param
CalcA.ParamB			INT	0	Decimal	Second Param
Output						
CalcA.Sum			DINT	0	Decimal	Sum
InOut						
CalcA.ParamC			INT	0	Decimal	Third Param
Static						
CalcA.Const			SINT	100	Decimal	Constant
ResultCalc			DINT	0	Decimal	
*						

When you assign a FB type to a tag, FB parameters derive initial values of FB definition. Then if you change a parameter initial value for one instance, it is not changed to other instances and to FB definition.

4.5 Function Block Calls

When a block is called, you must assign values to the parameters in the block interface. By providing input parameters you specify the data with which the block is executed. By providing the output parameters you specify where the execution results are saved.

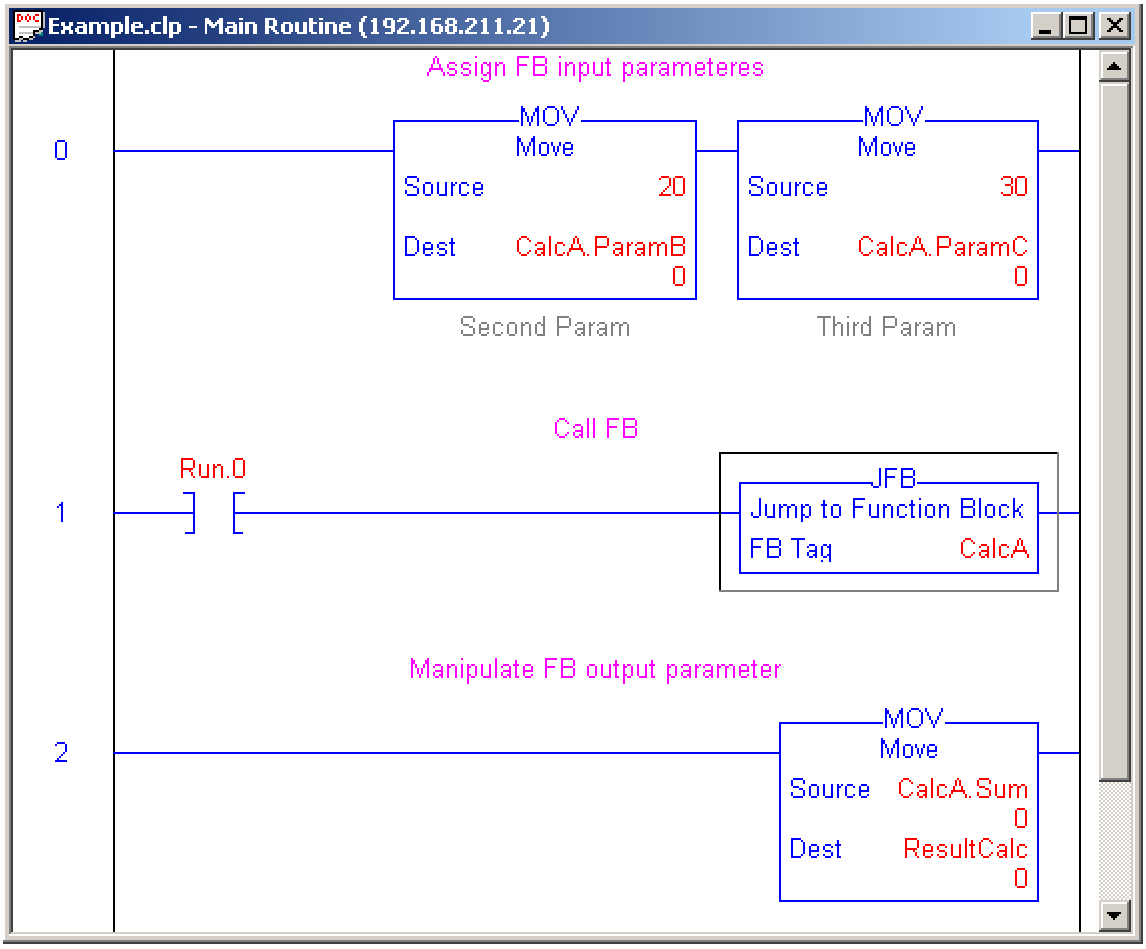
In your program (Main Routine or FB Routine) you can examine function block output parameters, but you can not assign a value to output parameter.

Also you can not use invoked function block static parameters.

From LD programs function block is called by JFB instruction with function block instance (FB Tag).

Example:

Calling function block from Main Program:



When Run.0 is false, FB is not executed and data in CalcA remain unchanged.

Rung2

When FB call is finished, you may check or assign output parameters. In this example main tag ResultCalc = CalcA.Sum.

When one block calls another block, the instructions of the called block are executed. Only when execution of the called block has been completed does execution of the calling block resume. The execution is continued with the instruction that follows on the block call.

When FB which calls another block is on LD language, calling performs in the same way as it is called from Main program.

When FB which calls another block is on ST language, calling performs by using called FB instance. In parentheses are assigned inputs parameters (by := sign) and refers outputs parameters (by => sign).

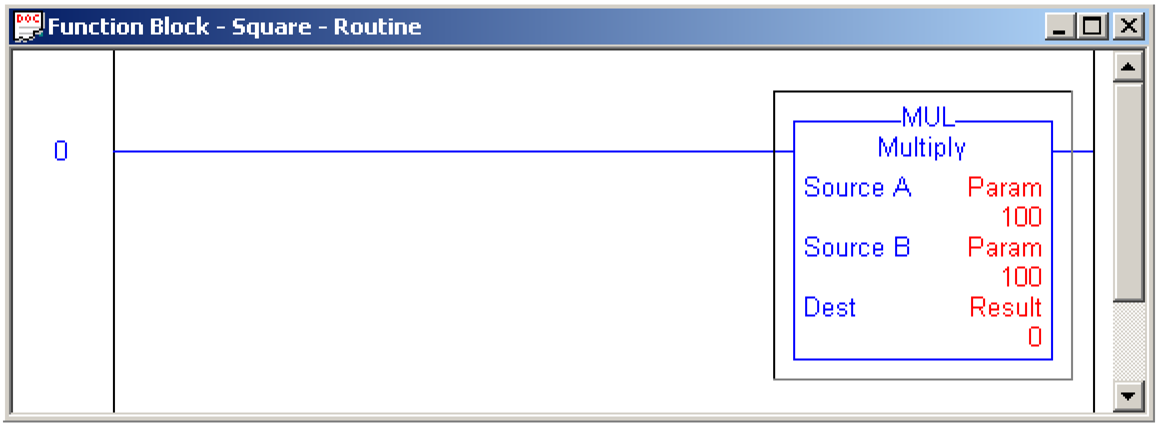
Example:

Calling instance SquareA from FB Square type from FB Calculate:

First create tag and routine of Square FB type.

Function Block - Square - Tags

	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	⊟ Input						
	⊞ Param			INT	100	Decimal	
*							
	⊟ Output						
	⊞ Result			DINT	0	Decimal	
*							
	InOut						
*							
	Static						
*							



Then in Calculate FB create a tag, named SquareA with Square data type.

Function Block - Calculate - Tags

	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	⊟ InOut						
	⊞ ParamC			INT	0	Decimal	Third Param
*							
	⊟ Static						
	⊞ Const			SINT	100	Decimal	Constant
	⊟ SquareA			Square	{...}		
	⊟ Input						
	⊞ SquareA.Param			INT	100	Decimal	
	⊟ Output						
	⊞ SquareA.Result			DINT	0	Decimal	Square
	InOut						
	Static						
▶	⊞ ResultSquare			DINT	0	Decimal	
*							

Call SquareA instance from FB Calculate:

Function Block - Calculate - Routine

IF ELSIF CASE FOR

```
1       SquareA(Param := ParamA, Result => ResultSquare);
2
3       Sum := ResultSquare + ParamB + ParamC + Const;
4
5       IF Sum > 5000 THEN
6               Sum := 5000;
7       END_IF;
8
```

When SquareA instance is called (line 1) first ParamA is copied to Param. Then Square routine executes. After that Result is copied to Result.

There is second way to call SquareA instance – first assign inputs parameters, then call FB and after that assign outputs parameters.

Function Block - Calculate - Routine

IF ELSIF CASE FOR

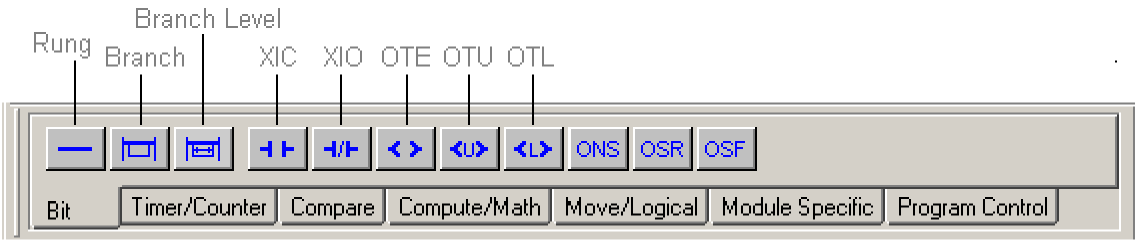
```
SquareA.Param := ParamA;
SquareA();
ResultSquare := SquareA.Result;

Sum := ResultSquare + ParamB + ParamC + Const;
```

5.0 Ladder Logic Instructions

5.1 Bit Instructions

Use the bit (relay-type) instructions to monitor and control the status of bits.

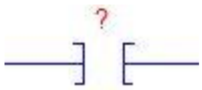
To enter a bit instructions use buttons form Bit tab of Instruction Bar.



Instruction	Description
XIC	enable outputs when a bit is set
XIO	enable outputs when a bit is cleared
OTE	set a bit
OTL	set a bit (retentive)
OTU	clear bit (retentive)
ONS	enable outputs for one scan each time a rung goes true
OSR	set a bit for one scan each time a rung goes true
OSF	set a bit for one scan each time the rung goes false

5.1.1 Examine If Closed (XIC)

The XIC instruction examines the data bit to see if it is set.



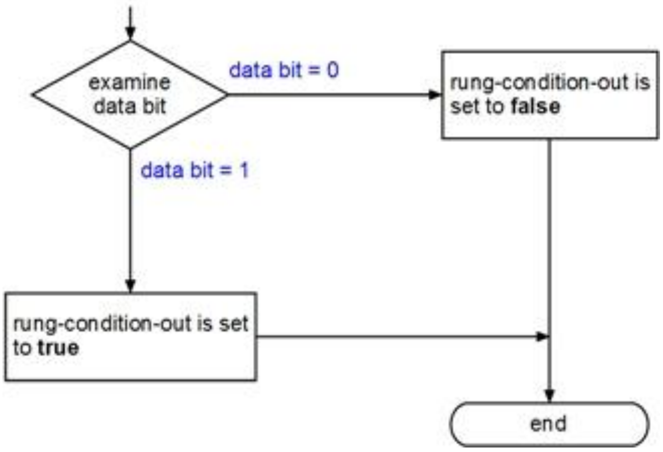
Operands:

Operand	Type	Format	Description
data bit	BOOL	tag	bit to be tested

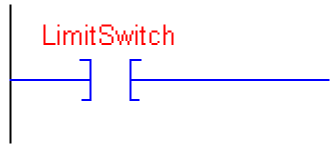
Description:

The XIC instruction examines the data bit to see if it is set.

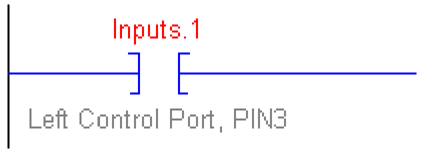
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	

Examples:



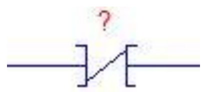
If LimitSwitch is set, this enables the next instruction (the rung-condition-out is true).



If Inputs.1 is set (indicates that an overflow has occurred), this enables the next instruction (the rung-condition-out is true).

5.1.2 Examine If Open (XIO)

The XIO instruction examines the data bit to see if it is cleared.



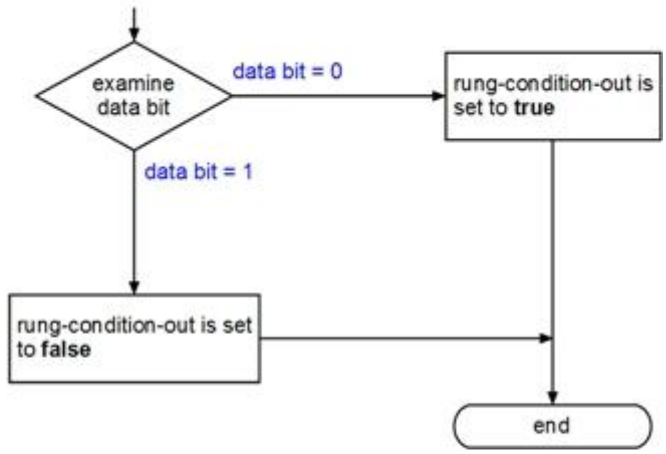
Operands:

Operand	Type	Format	Description
data bit	BOOL	tag	bit to be tested

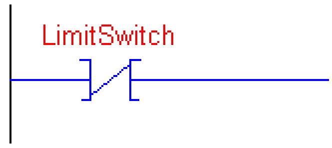
Description:

The XIO instruction examines the data bit to see if it is cleared.

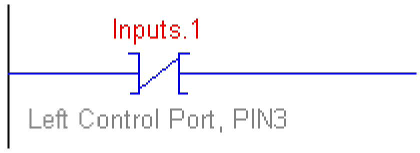
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	

Examples:



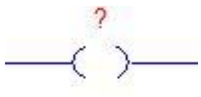
If LimitSwitch is cleared, this enables the next instruction (the rung-condition-out is true).



If Inputs.1 is cleared (indicates that no overflow has occurred), this enables the next instruction (the rung-condition-out is true).

5.1.3 Output Energize (OTE)

The OTE instruction sets or clears the data bit.



Operands:

Operand	Type	Format	Description
data bit	BOOL	tag	bit to be set or cleared

Description:

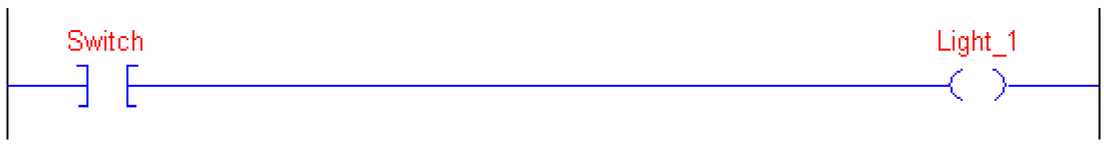
When the OTE instruction is enabled, the controller sets the data bit. When the OTE instruction is disabled, the controller clears the data bit.

Execution:

Condition	Action
prescan	The data bit is cleared. The rung-condition-out is set to false.
rung-condition-in is false	The data bit is cleared. The rung-condition-out is set to false.
rung-condition-in is true	The data bit is set. The rung-condition-out is set to true.

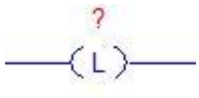
Example:

When Switch is set, the OTE instruction sets (turns on) Light_1. When Switch is cleared, the OTE instruction clears (turns off) Light_1.



5.1.4 Output Latch (OTL)

The OTL instruction sets (latches) the data bit.



Operands:

Operand	Type	Format	Description
data bit	BOOL	tag	bit to be set

Description:

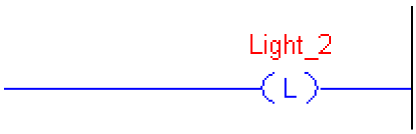
When enabled, the OTL instruction sets the data bit. The data bit remains set until it is cleared, typically by an OTU instruction. When disabled, the OTL instruction does not change the status of the data bit.

Execution:

Condition	Action
prescan	The data bit is not modified. The rung-condition-out is set to false.
rung-condition-in is false	The data bit is not modified. The rung-condition-out is set to false.
rung-condition-in is true	The data bit is set. The rung-condition-out is set to true.

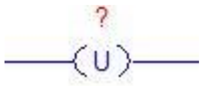
Example:

When enabled, the OTL instruction sets Light_2. This bit remains set until it is cleared, typically by an OTU instruction.



5.1.5 Output Unlatch (OTU)

The OTU instruction clears (unlatches) the data bit.



Operands:

Operand	Type	Format	Description
data bit	BOOL	tag	bit to be cleared

Description:

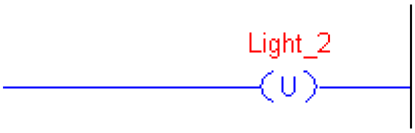
When enabled, the OTU instruction clears the data bit. When disabled, the OTU instruction does not change the status of the data bit.

Execution:

Condition	Action
prescan	The data bit is not modified. The rung-condition-out is set to false.
rung-condition-in is false	The data bit is not modified. The rung-condition-out is set to false.
rung-condition-in is true	The data bit is cleared. The rung-condition-out is set to true.

Example:

When enabled, the OTU instruction clears Light_2.



5.1.6 One Shot (ONS)

The ONS instruction enables or disables the remainder of the rung, depending on the status of the storage bit.

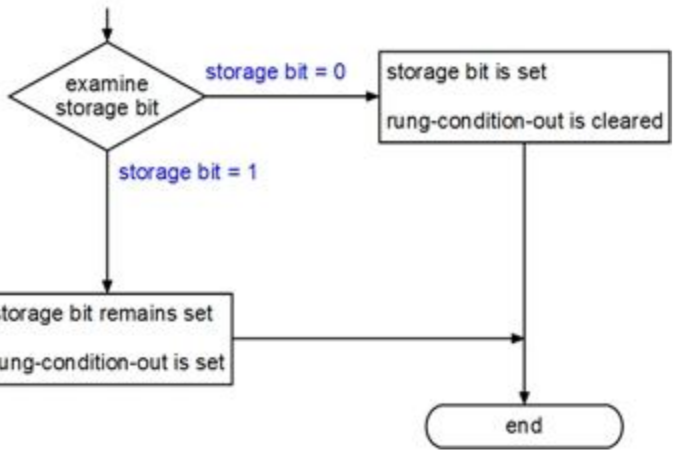
Operands:

Operand	Type	Format	Description
storage bit	BOOL	tag	internal storage bit stores the rung-condition-in from the last time the instruction was executed

Description:

When enabled and the storage bit is cleared, the ONS instruction enables the remainder of the rung. When disabled or when the storage bit is set, the ONS instruction disables the remainder of the rung.

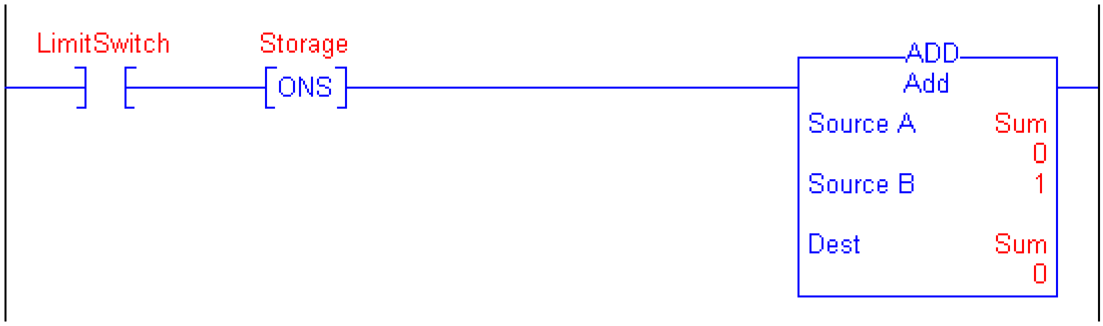
Execution:

Condition	Action
prescan	The storage bit is set to prevent an invalid trigger during the first scan. The rung-condition-out is set to false.
rung-condition-in is false	The storage bit is cleared. The rung-condition-out is set to false.
rung-condition-in is true	

Example:

You typically precede the ONS instruction with an input instruction because you scan the ONS instruction when it is enabled and when it is disabled for it to operate correctly. Once the ONS instruction is enabled, the rung-condition-in must go clear or the storage bit must be cleared for the ONS instruction to be enabled again.

On any scan for which LimitSwitch is cleared or Storage is set, this rung has no affect. On any scan for which LimitSwitch is set and Storage is cleared, the ONS instruction sets Storage and the ADD instruction increments Sum by 1. As long as LimitSwitch stays set, Sum stays the same value. The LimitSwitch must go from cleared to set again for Sum to be incremented again.



5.1.7 One Shot Rising (OSR)

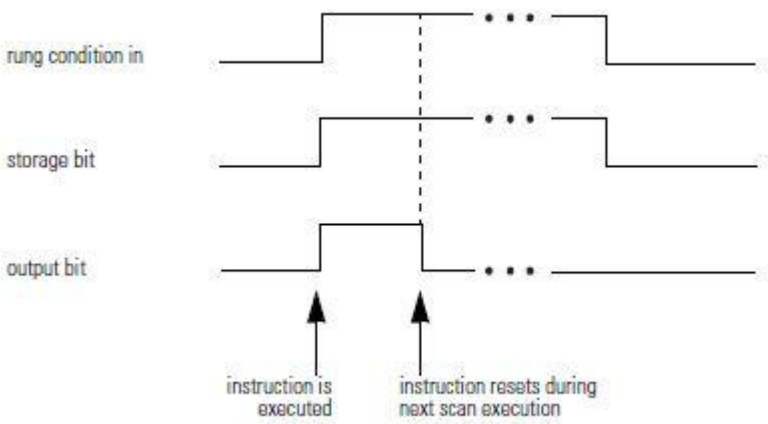
The OSR instruction sets or clears the output bit, depending on the status of the storage bit.

Operands:

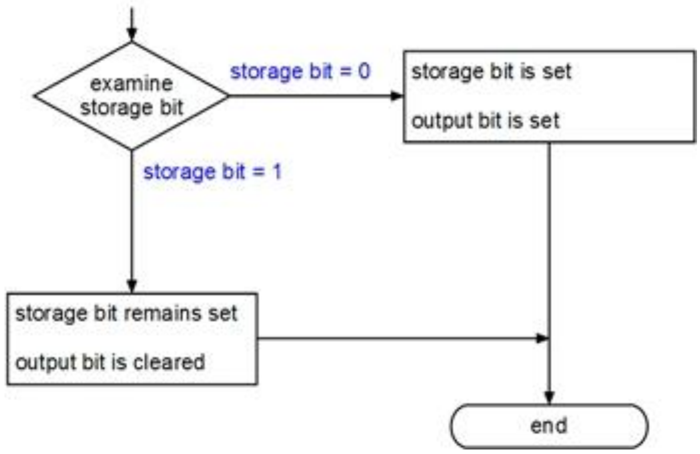
Operand	Type	Format	Description
storage bit	BOOL	tag	internal storage bit stores the rung-condition-in from the last time the instruction was executed
output bit	BOOL	tag	bit to be set

Description:

When enabled and the storage bit is cleared, the OSR instruction sets the output bit. When enabled and the storage bit is set or when disabled, the OSR instruction clears the output bit

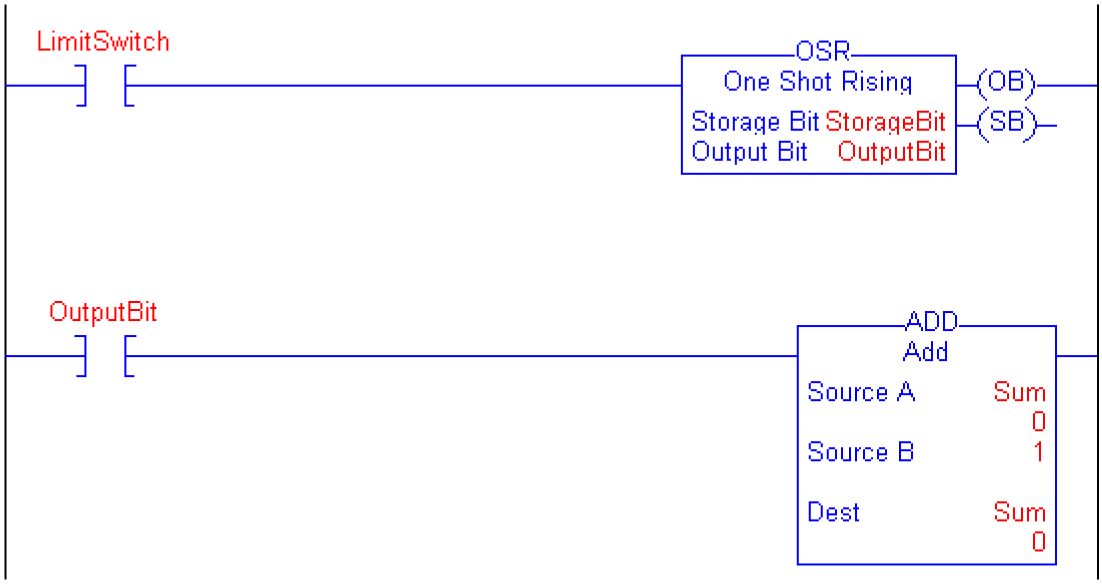


Execution:

Condition	Action
prescan	The storage bit is set to prevent an invalid trigger during the first scan. The rung-condition-out is set to false.
rung-condition-in is false	The storage bit is cleared. The output bit is not modified. The rung-condition-out is set to false.
rung-condition-in is true	

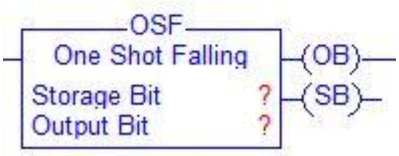
Example:

Each time LimitSwitch goes from cleared to set, the OSR instruction sets OutputBit and the ADD instruction increments sum by 1. As long as LimitSwitch stays set, Sum stays the same value. The LimitSwitch must go from cleared to set again for Sum to be incremented again. You can use OutputBit on multiple rungs to trigger other operations.



5.1.8 One Shot Falling (OSF)

The OSF instruction sets or clears the output bit depending on the status of the storage bit.

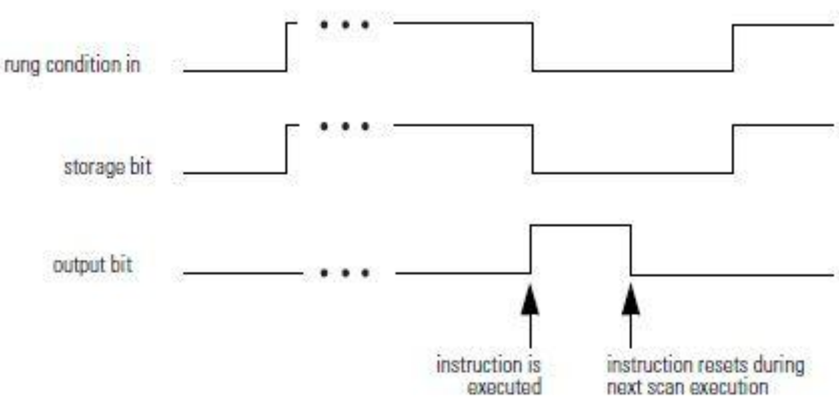


Operands:

Operand	Type	Format	Description
storage bit	BOOL	tag	internal storage bit stores the rung-condition-in from the last time the instruction was executed
output bit	BOOL	tag	bit to be set

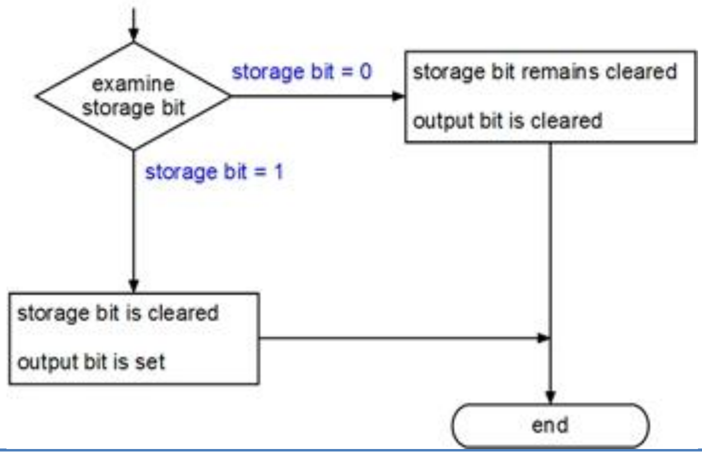
Description:

When disabled and the storage bit is set, the OSF instruction sets the output bit. When disabled and the storage bit is cleared, or when enabled, the OSF instruction clears the output bit.



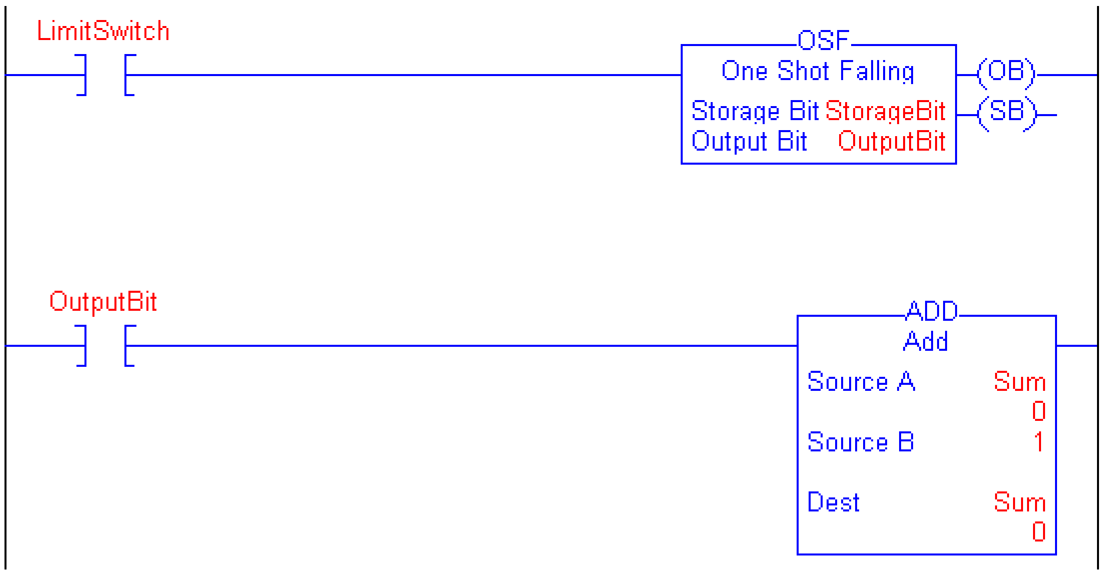
Execution:

Condition	Action
prescan	The storage bit is cleared to prevent an invalid trigger during the first scan. The output bit is cleared. The rung-condition-out is set to false.

Condition	Action
rung-condition-in is false	
rung-condition-in is true	The storage bit is set. The output bit is cleared. The rung-condition-out is set to true.

Example:

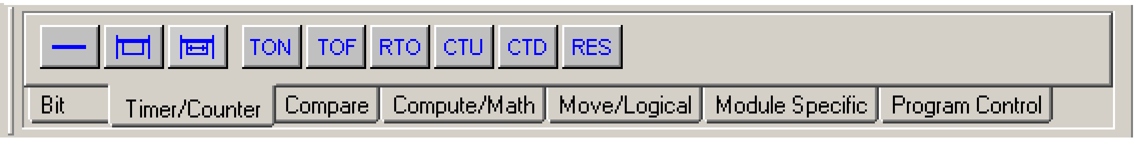
Each time LimitSwitch goes from set to cleared, the OSF instruction sets OutputBit and the ADD instruction increments Sum by 1. As long as LimitSwitch stays cleared, Sum stays the same value. The LimitSwitch must go from set to clear again for Sum to be incremented again. You can use OutputBit on multiple rungs to trigger other operations.



5.2 Timer and Counter Instructions

Timers and counters control operations based on time or the number of events.

To enter a timer/counter instruction use buttons form Timer/Counter tab of Instruction Bar.

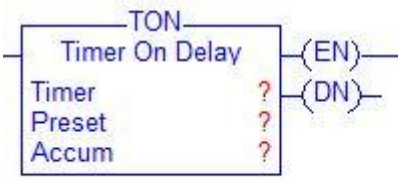


Instruction	Description
TON	time how long a timer is enabled
TOF	time how long a timer is disabled
RTO	accumulate time
CTU	count up
CTD	count down
RES	reset a timer or counter

The time base for all timers is 1 msec.

5.2.1 Timer On Delay (TON)

The TON instruction is a non-retentive timer that accumulates time when the instruction is enabled (rung-condition-in is true).



Operands:

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

TIMER Structure

Mnemonic	Data Type	Description
.EN	BOOL	The enable bit indicates that the TON instruction is enabled.
.TT	BOOL	The timing bit indicates that a timing operation is in process
.DN	BOOL	The done bit is set when .ACC ≥ .PRE.
.PRE	DINT	The preset value specifies the value (1 msec units) which the accumulated value must reach before the instruction sets the .DN bit.
.ACC	DINT	The accumulated value specifies the number of milliseconds that have elapsed since the TON instruction was enabled.

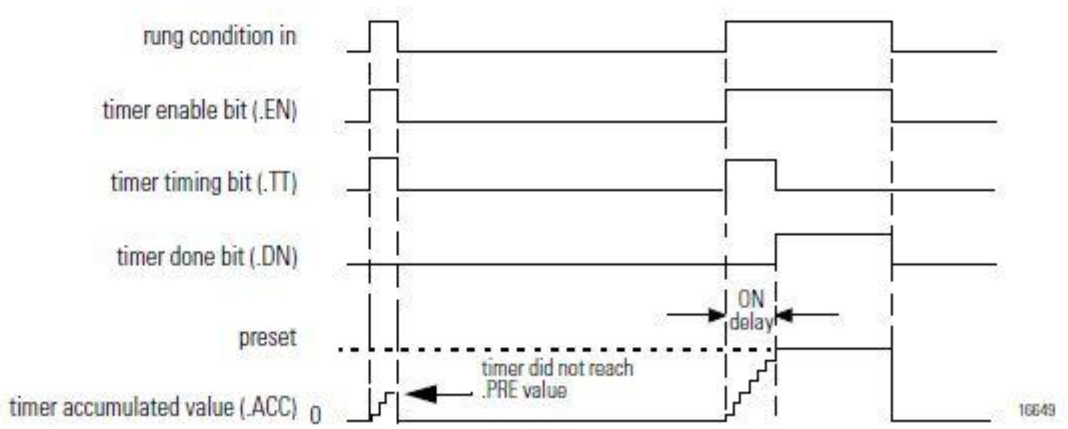
Description:

The TON instruction accumulates time until:

- the TON instruction is disabled
- the .ACC ≥ .PRE

The time base is always 1 msec. For example, for a 2-second timer, enter 2000 for the .PRE value.

When the TON instruction is disabled, the .ACC value is cleared.



A timer runs by subtracting the time of its last scan from the time now:

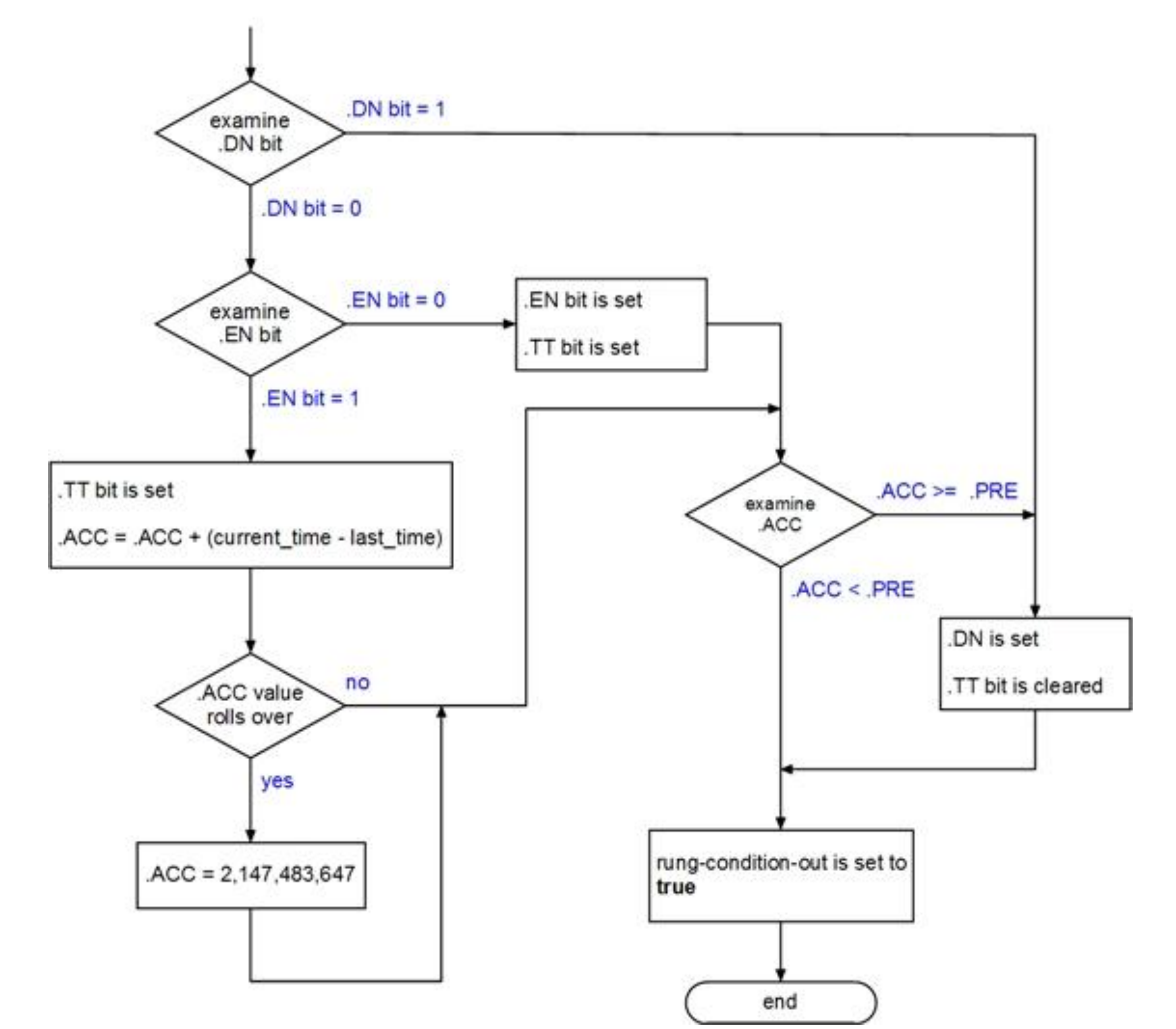
ACC = ACC + (current_time - last_time_scanned)

After it updates the ACC, the timer sets last_time_scanned = current_time. This gets the timer ready for the next scan.

Execution:

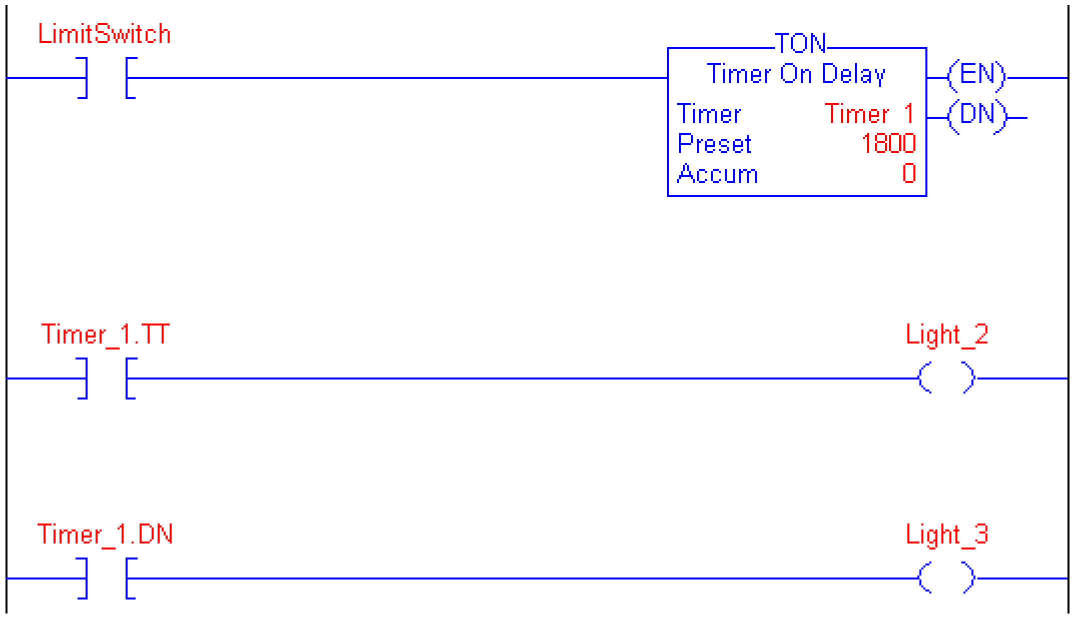
Condition	Action
prescan	The .EN, .TT, and .DN bits are cleared. The .ACC value is cleared. The rung-condition-out is set to false.
rung-condition-in is false	The .EN, .TT, and .DN bits are cleared. The .ACC value is cleared. The rung-condition-out is set to false.
rung-condition-in is true	

Condition	Action



Example:

When LimitSwitch is set, Light_2 is on for 1800 msec (Timer_1 is timing). When Timer _1.ACC reaches 1800, Light_2 goes off and Light_3 goes on. Light_3 remains on until the TON instruction is disabled. If LimitSwitch is cleared while Timer_1 is timing, Light_2 goes off.



5.2.2 Timer Off Delay (TOF)

The TOF instruction is a non-retentive timer that accumulates time when the instruction is enabled (rung-condition-in is false).

Operands:

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

TIMER Structure

Mnemonic	Data Type	Description
.EN	BOOL	The enable bit indicates that the TOF instruction is enabled.
.TT	BOOL	The timing bit indicates that a timing operation is in process
.DN	BOOL	The done bit is cleared when .ACC ≥ .PRE.
.PRE	DINT	The preset value specifies the value (1 msec units) which the accumulated value must reach before the instruction clears the .DN bit.
.ACC	DINT	The accumulated value specifies the number of milliseconds that have elapsed since the TOF instruction was enabled.

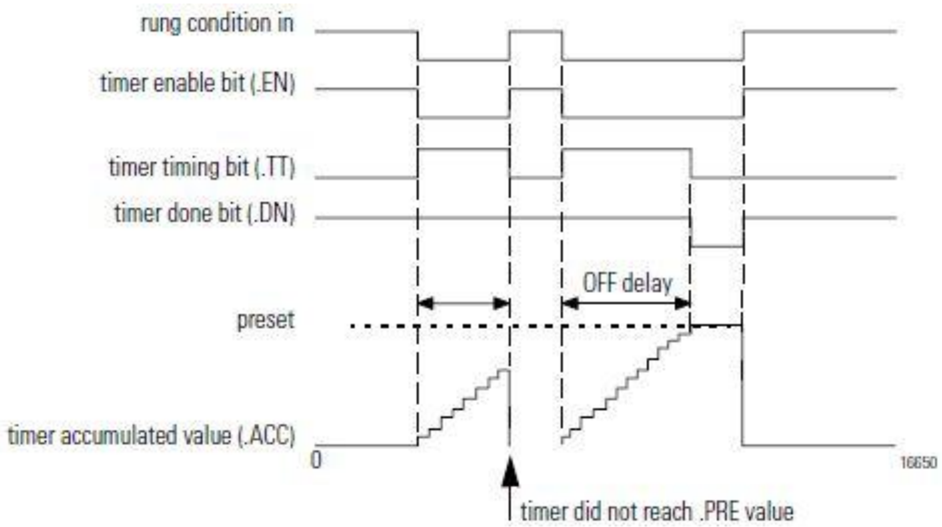
Description:

The TOF instruction accumulates time until:

- the TOF instruction is disabled
- the .ACC ≥ .PRE

The time base is always 1 msec. For example, for a 2-second timer, enter 2000 for the .PRE value.

When the TOF instruction is disabled, the .ACC value is cleared.

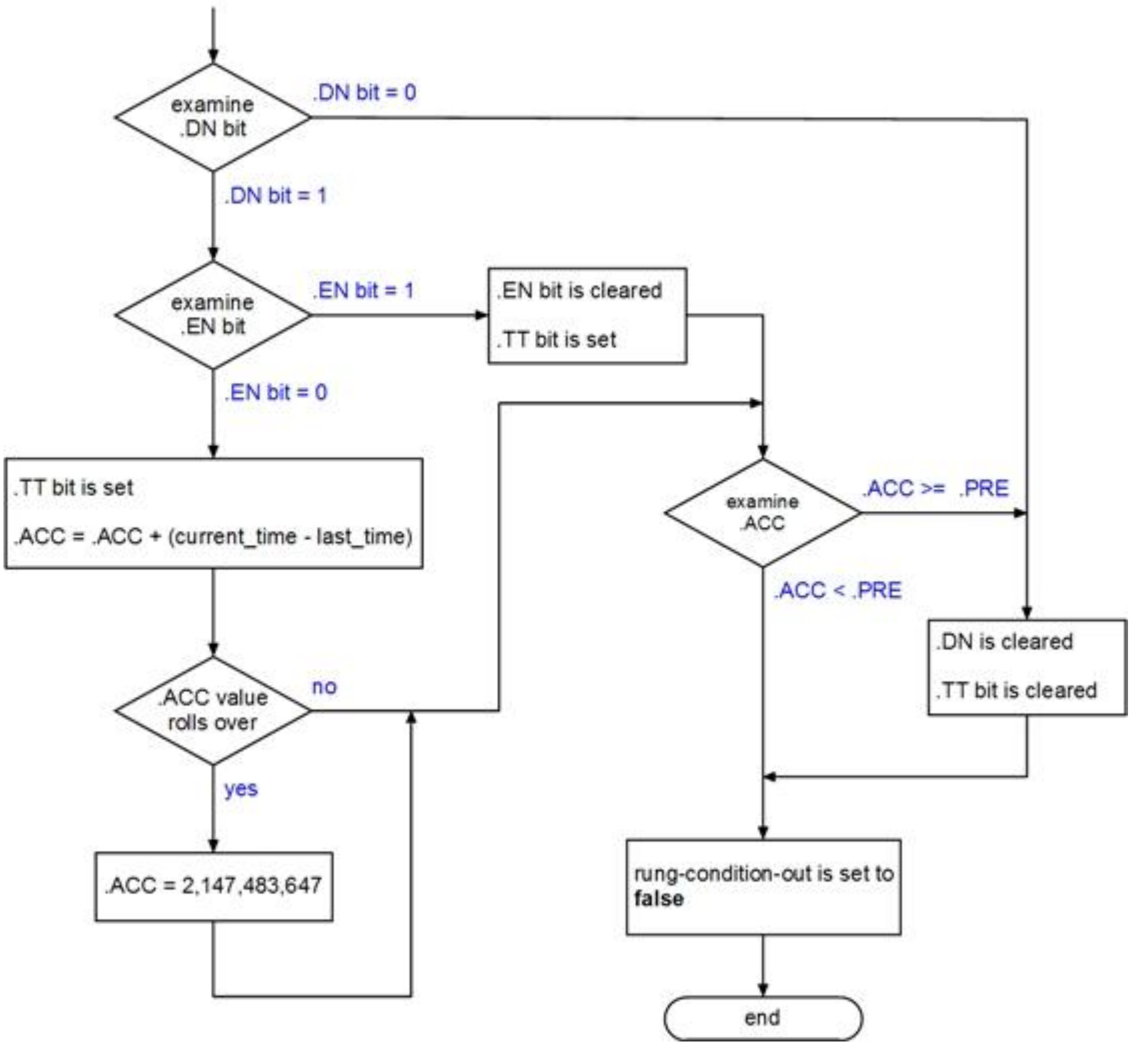


A timer runs by subtracting the time of its last scan from the time now:

ACC = ACC + (current_time - last_time_scanned)

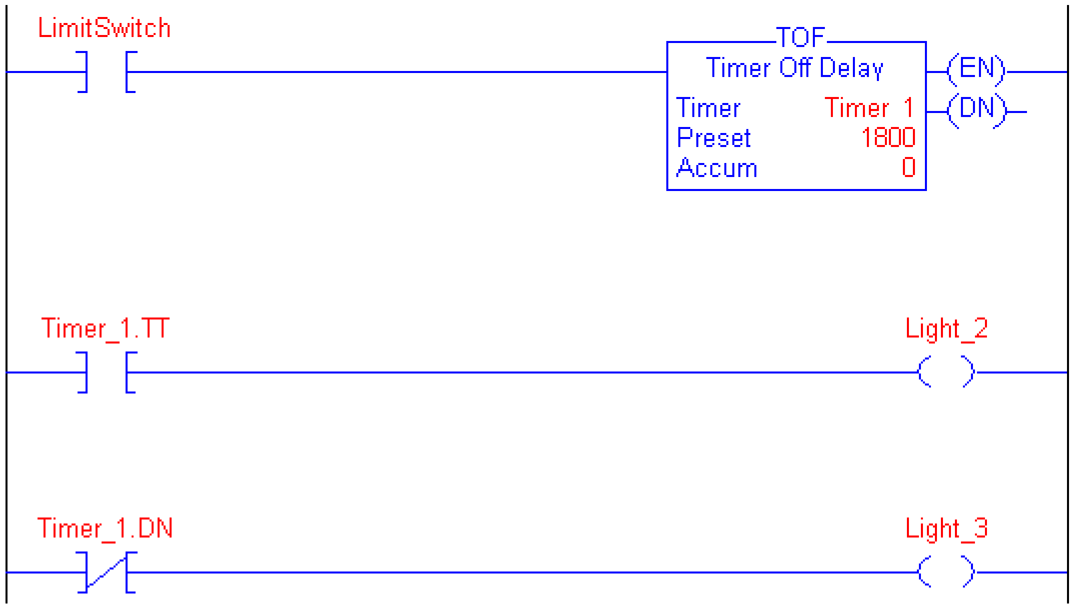
After it updates the ACC, the timer sets last_time_scanned = current_time. This gets the timer ready for the next scan.

Execution:

Condition	Action
prescan	The .EN, .TT, and .DN bits are cleared. The .ACC value is set to equal the .PRE value. The rung-condition-out is set to false.
rung-condition-in is false	
	
rung-condition-in is true	The .EN, .TT, and .DN bits are set. The .ACC value is cleared. The rung-condition-out is set to true.

Example:

When LimitSwitch is cleared, Light_2 is on for 1800 msec (Timer_1 is timing). When Timer_1.ACC reaches 1800, Light_2 goes off and Light_3 goes on. Light_3 remains on until the TOF instruction is enabled. If LimitSwitch is set while Timer_1 is timing, Light_2 goes off.



5.2.3 Retentive Timer On (RTO)

The RTO instruction is a retentive timer that accumulates time when the instruction is enabled.

Operands:

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

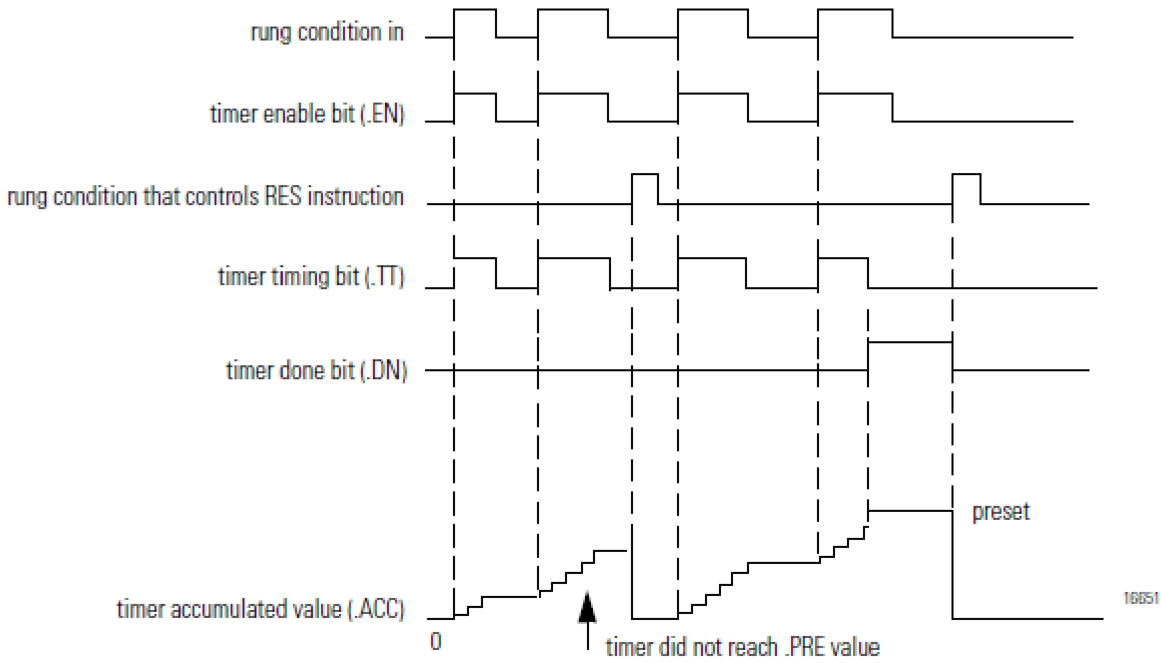
TIMER Structure

Mnemonic	Data Type	Description
.EN	BOOL	The enable bit indicates that the RTO instruction is enabled.
.TT	BOOL	The timing bit indicates that a timing operation is in process
.DN	BOOL	The done bit indicates that .ACC ≥ .PRE.
.PRE	DINT	The preset value specifies the value (1 msec units) which the accumulated value must reach before the instruction sets the .DN bit.
.ACC	DINT	The accumulated value specifies the number of milliseconds that have elapsed since the RTO instruction was enabled.

Description:

The RTO instruction accumulates time until it is disabled. When the RTO instruction is disabled, it retains its .ACC value. You must clear the .ACC value, typically with a RES instruction referencing the same TIMER structure.

The time base is always 1 msec. For example, for a 2-second timer, enter 2000 for the .PRE value.



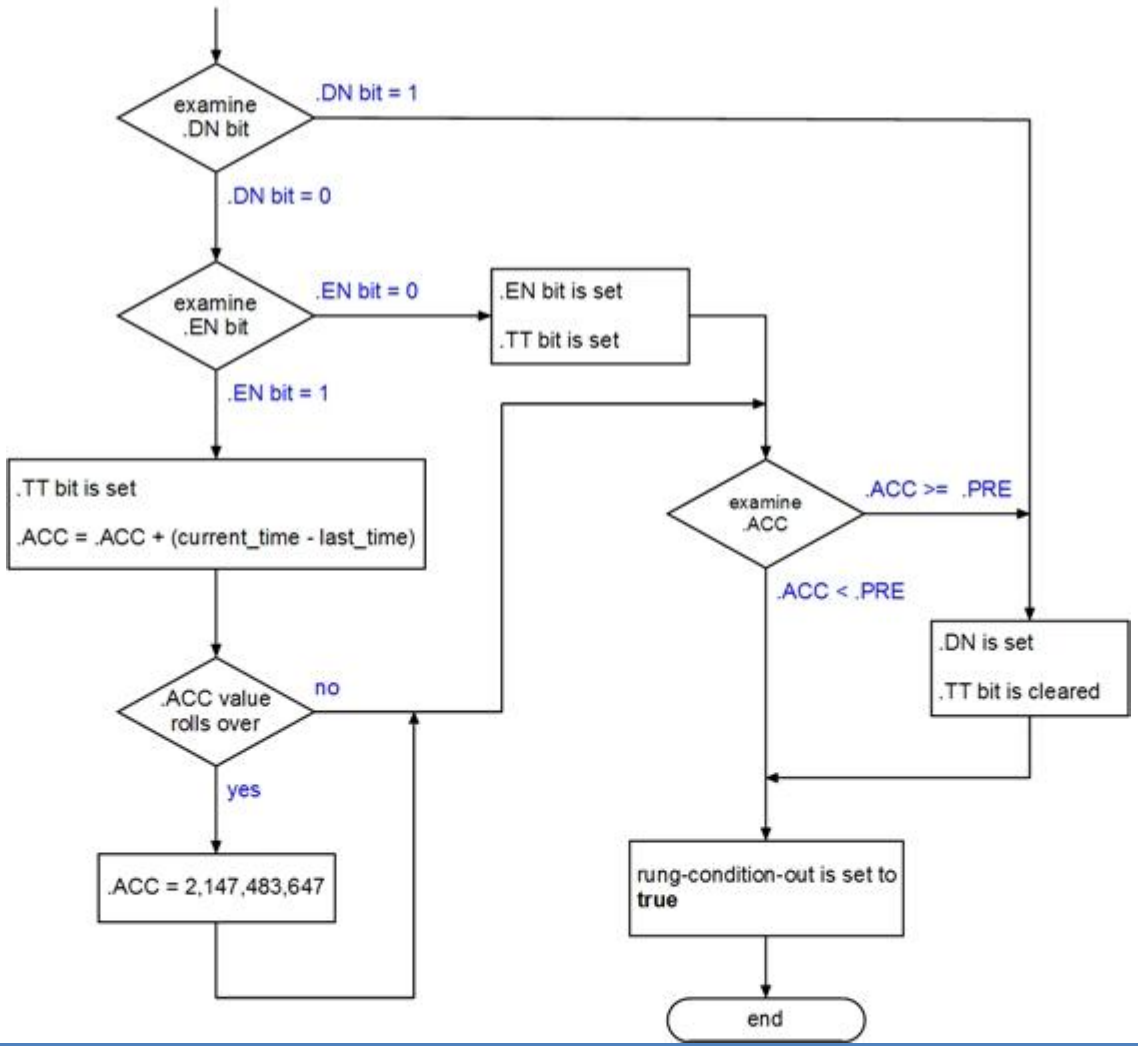
A timer runs by subtracting the time of its last scan from the time now:

ACC = ACC + (current_time - last_time_scanned)

After it updates the ACC, the timer sets last_time_scanned = current_time. This gets the timer ready for the next scan.

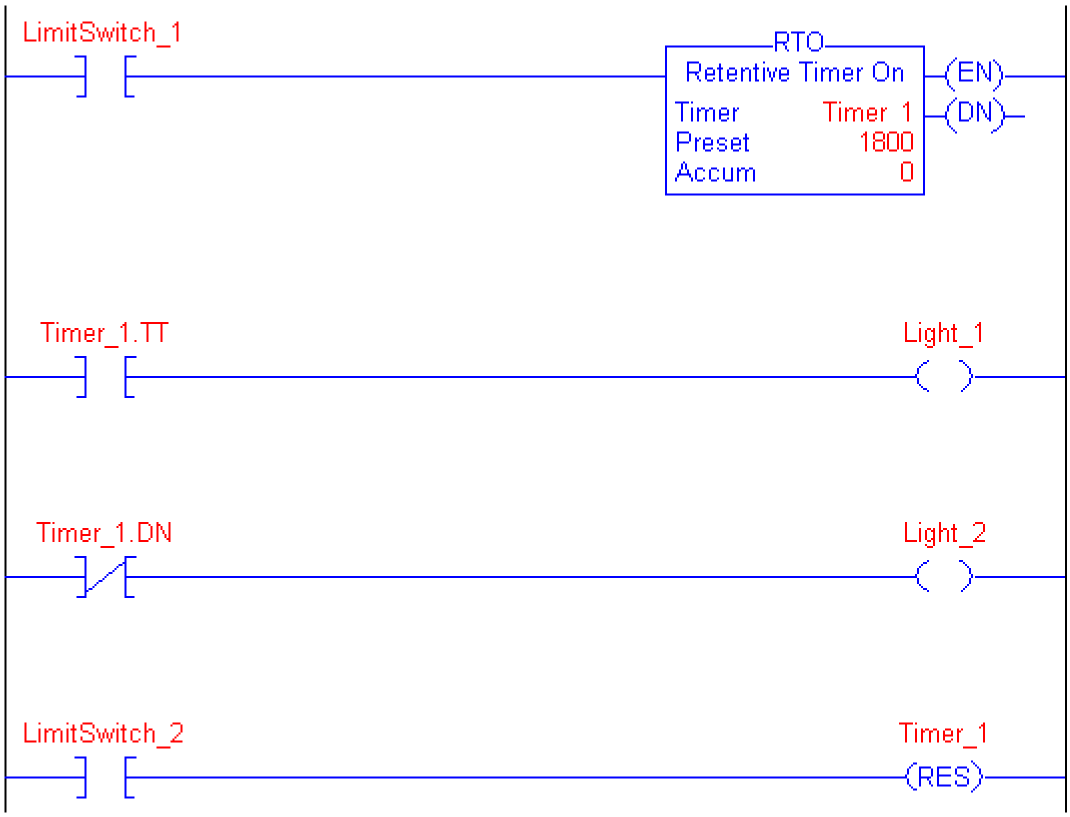
Execution:

Condition	Action
prescan	The .EN, .TT, and .DN bits are cleared. The .ACC value is not modified. The rung-condition-out is set to false.
rung-condition-in is false	The .EN and .TT bits are cleared. The .DN bit is not modified. The .ACC value is not modified. The rung-condition-out is set to false.
rung-condition-in is true	



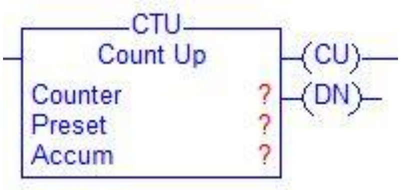
Example:

When LimitSwitch_1 is set, Light_1 is on for 1800 msec (Timer_1 is timing). When Timer_1.ACC reaches 1800, Light_1 goes off and Light_2 goes on. Light_2 remains until Timer_1 is reset. If LimitSwitch_2 is cleared while Timer_1 is timing, Light_1 remains on. When LimitSwitch_2 is set, the RES instruction resets Timer_1 (clears status bits and .ACC value).



5.2.4 Count Up (CTU)

The CTU instruction counts upward.



Operands:

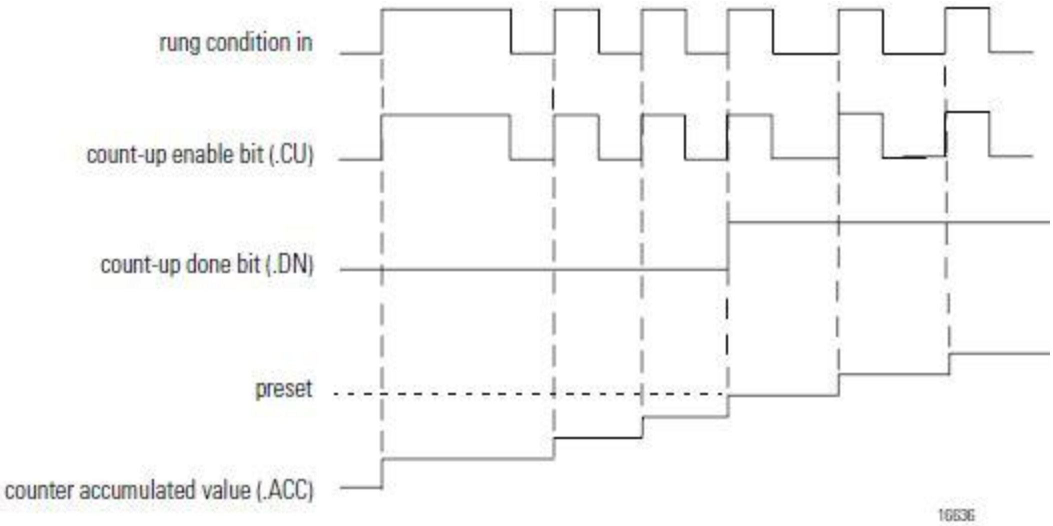
Operand	Type	Format	Description
Counter	COUNTER	tag	COUNTER structure
Preset	DINT	immediate	how high to count
Accum	DINT	immediate	number of times the counter has counted initial value is typically 0

COUNTER Structure

Mnemonic	Data Type	Description
.CU	BOOL	The count up enable bit indicates that the CTU instruction is enabled.
.DN	BOOL	The done bit indicates that .ACC ≥ .PRE.
.OV	BOOL	The overflow bit indicates that the counter exceeded the upper limit of 2,147,483,647. The counter then rolls over to -2,147,483,648 and begins counting up again.
.UN	BOOL	The underflow bit indicates that the counter exceeded the lower limit of -2,147,483,648. The counter then rolls over to 2,147,483,647 and begins counting down again.
.PRE	DINT	The preset value specifies the value which the accumulated value must reach before the instruction sets the .DN bit.
.ACC	DINT	The accumulated value specifies the number of transitions the instruction has counted.

Description:

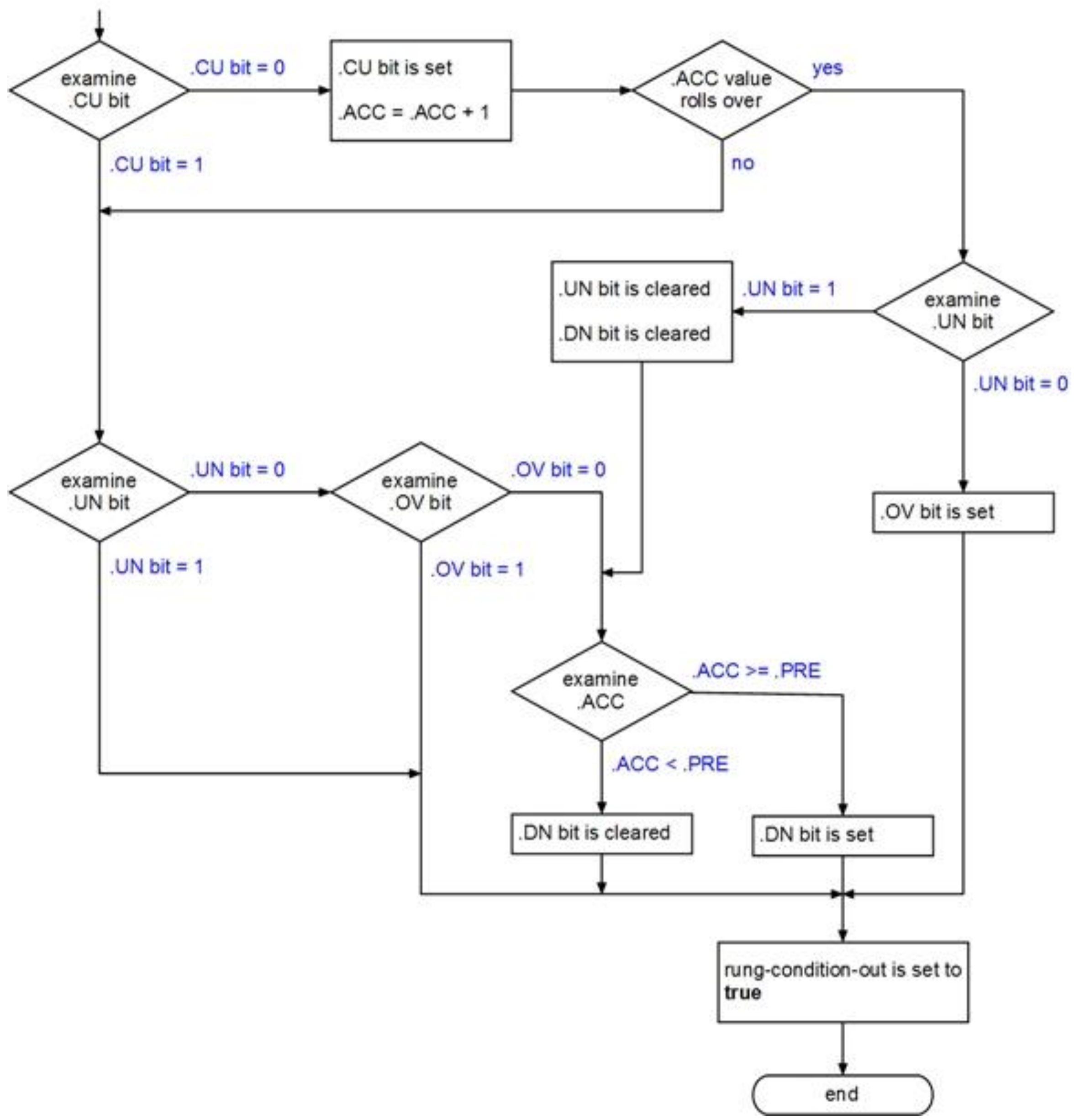
When enabled and the .CU bit is cleared, the CTU instruction increments the counter by one. When enabled and the .CU bit is set, or when disabled, the CTU instruction retains its .ACC value.



The accumulated value continues incrementing, even after the .DN bit is set. To clear the accumulated value, use a RES instruction that references the counter structure or write 0 to the accumulated value.

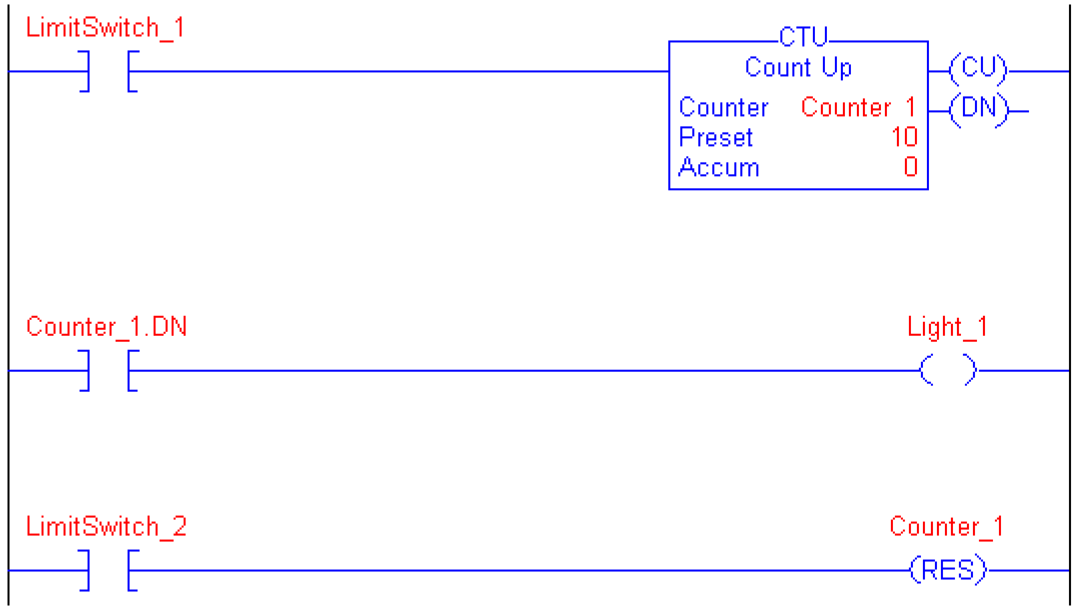
Execution:

Condition	Action
prescan	The .CU bit is set to prevent invalid increments during the first program scan. The rung-condition-out is set to false.
rung-condition-in is false	The .CU bit is cleared. The rung-condition-out is set to false.
rung-condition-in is true	



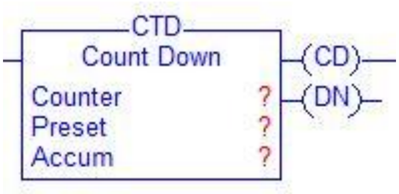
Example:

After LimitSwitch_1 goes from disabled to enabled 10 times, the .DN bit is set and Light_1 turns on. If LimitSwitch_1 continues to go from disabled to enabled, Counter_1 continues to increment its count and the .DN bit remains set. When LimitSwitch_2 is enabled, the RES instruction resets Counter_1 (clears the status bits and the .ACC value) and Light_1 turns off.



5.2.5 Count Down (CTD)

The CTD instruction counts downward.



Operands:

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

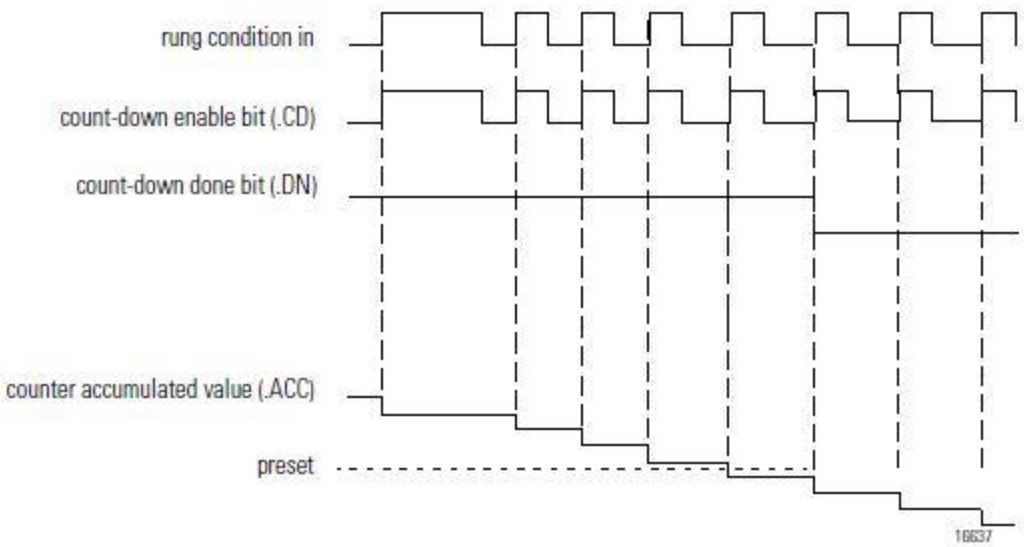
COUNTER Structure

Mnemonic	Data Type	Description
.CU	BOOL	The count down enable bit indicates that the CTD instruction is enabled.
.DN	BOOL	The done bit indicates that .ACC ≥ .PRE.
.OV	BOOL	The overflow bit indicates that the counter exceeded the upper limit of 2,147,483,647. The counter then rolls over to -2,147,483,648 and begins counting up again.
.UN	BOOL	The underflow bit indicates that the counter exceeded the lower limit of -2,147,483,648. The counter then rolls over to 2,147,483,647 and begins counting down again.
.PRE	DINT	The preset value specifies the value which the accumulated value must reach before the instruction sets the .DN bit.
.ACC	DINT	The accumulated value specifies the number of transitions the instruction has counted.

Description:

The CTD instruction is typically used with a CTU instruction that references the same counter structure.

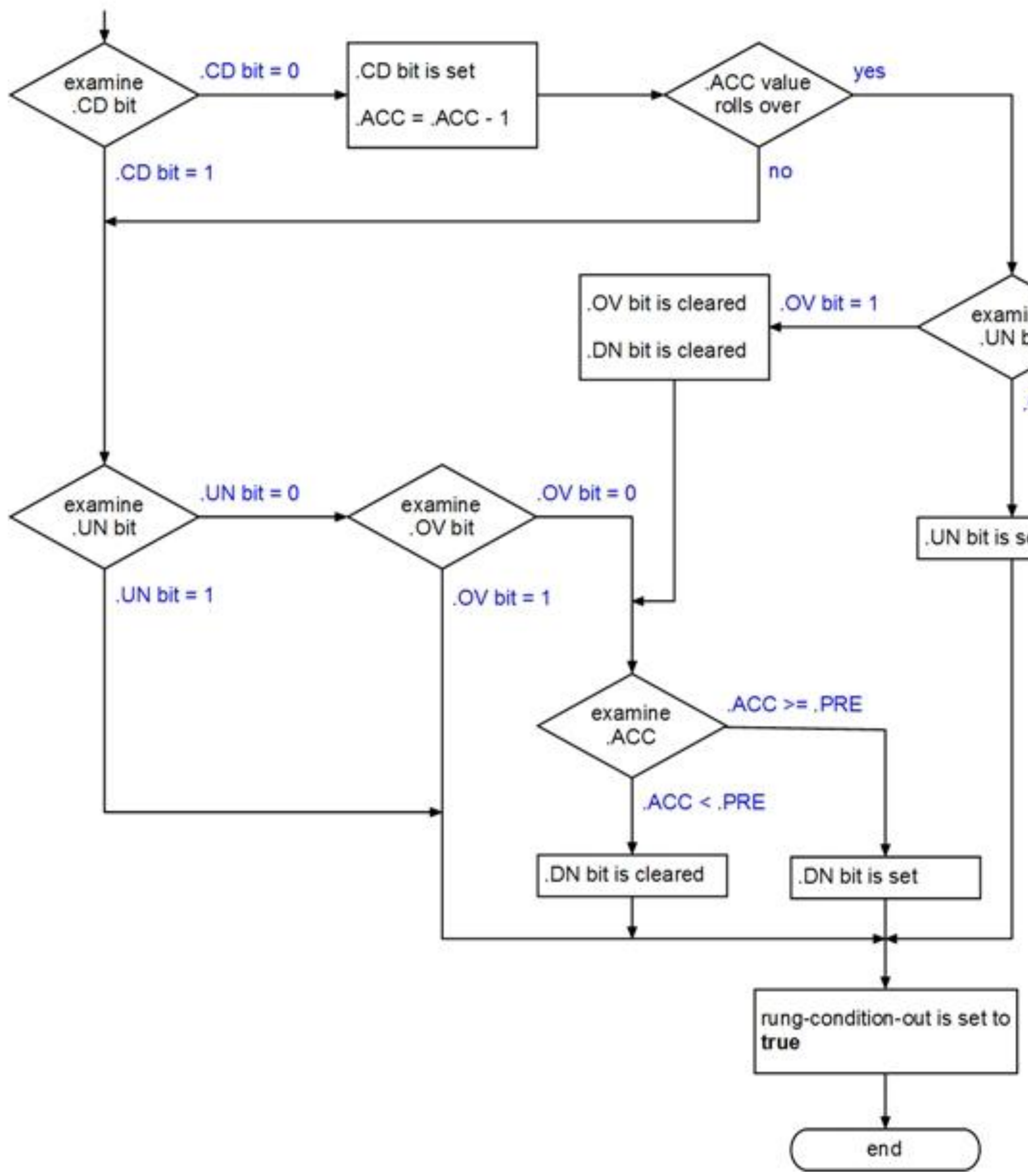
When enabled and the .CD bit is cleared, the CTD instruction decrements the counter by one. When enabled and the .CD bit is set, or when disabled, the CTD instruction retains its .ACC value.



The accumulated value continues decrementing, even after the .DN bit is set. To clear the accumulated value, use a RES instruction that references the counter structure or write 0 to the accumulated value.

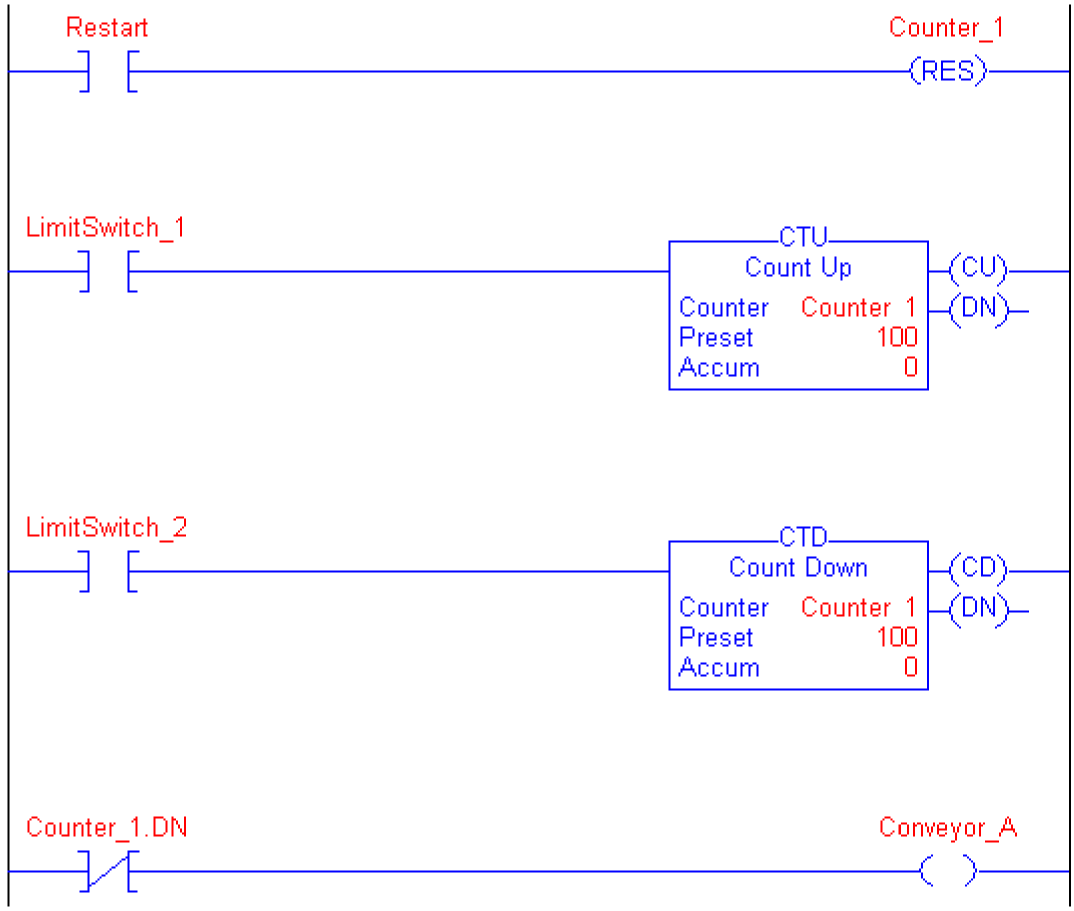
Execution:

Condition	Action
prescan	The .CD bit is set to prevent invalid decrements during the first program scan. The rung-condition-out is set to false.
rung-condition-in is false	The .CD bit is cleared. The rung-condition-out is set to false.
rung-condition-in is true	



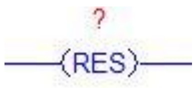
Example:

A conveyor brings parts into a buffer zone. Each time a part enters, LimitSwitch_1 is enabled and Counter_1 increments by 1. Each time a part leaves, LimitSwitch_2 is enabled and Counter_1 decrements by 1. If there are 100 parts in the buffer zone (Counter_1.DN is set), Conveyor_A turns on and stops the conveyor from bringing in any more parts until the buffer has room for more parts.



5.2.6 Reset (RES)

The RES instruction resets a TIMER or COUNTER structure.



Operands:

Operand	Type	Format	Description
structure	TIMER COUNTER	tag	structure to reset

Description:

When enabled the RES instruction clears these elements:

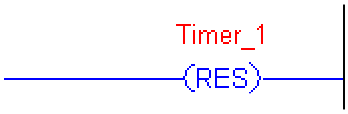
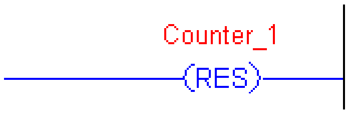
When Using a Res Instruction For a	The Instruction Clears
TIMER	.ACC value control status bits
COUNTER	.ACC value control status bits

ATTENTION Because the RES instruction clears the .ACC value, .DN bit and .TT bit, do not use the RES instruction to reset a TOF timer.

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	The RES instruction resets the specified structure. The rung-condition-out is set to true.

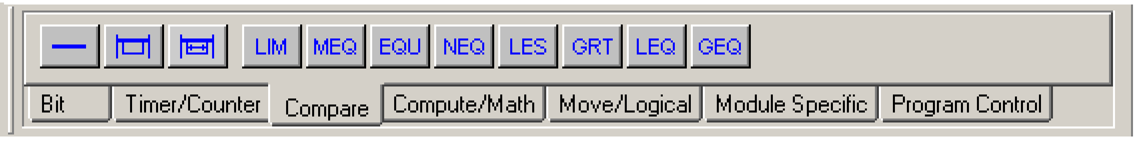
Example:

Example	**Description**
	When enabled, reset Timer_1.
	When enabled, reset Counter_1.

5.3 Compare Instructions

The compare instructions let you compare values by using an expression or a specific compare instruction.

To enter a timer/counter instruction use buttons form Timer/Counter tab of Instruction Bar.



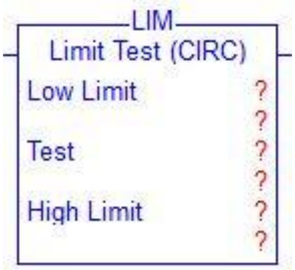
Instruction	Description
LIM	test whether one value is between two other values
MEQ	pass two values through a mask and test whether they are equal
EQU	test whether two values are equal
NEQ	test whether one value is not equal to a second value
LES	test whether one value is less than a second value
GRT	test whether one value is greater than a second value
LEQ	test whether one value is less than or equal to a second value
GEQ	test whether one value is greater than or equal to a second value

You can compare values of different data types, such as floating point and integer.

For relay ladder instructions, bold data types indicate optimal data types. An instruction executes faster and requires less memory if all the operands of the instruction use the same optimal data type, typically DINT.

5.3.1 Limit (LIM)

The LIM instruction tests whether the Test value is within the range of the Low Limit to the High Limit.



Operands:

Operand	Type	Format	Description
Low limit	SINT INT **DINT**	immediate tag	value of lower limit
Test	SINT INT **DINT**	immediate tag	value to test
High limit	SINT INT **DINT**	immediate tag	value of upper limit

If you enter a SINT or INT tag, the value converts to a DINT value by sign-extension.

Description:

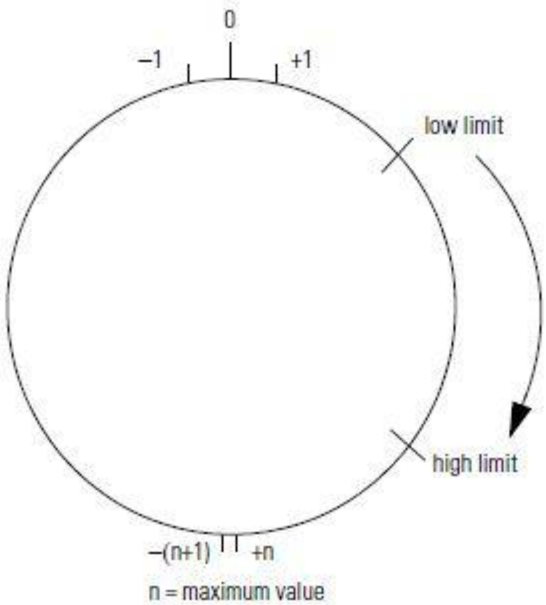
The LIM instruction tests whether the Test value is within the range of the Low Limit to the High Limit.

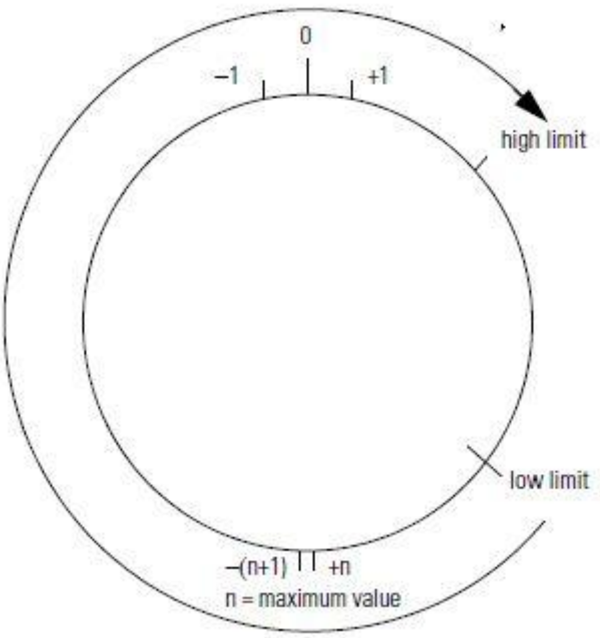
If Low Limit	And Test Value Is	The Rung-condition-out Is
≤ High Limit	equal to or between limits	true
	not equal to or outside limits	false
≥ High Limit	equal to or outside limits	true
	not equal to or inside limits	false

Signed integers “roll over” from the maximum positive number to the maximum negative number when the most significant bit is set. For example, in 16-bit integers (INT type), the maximum positive integer is 32767, which is represented in hexadecimal as 16#7FFF (bits 0 through 14 are all set). If you increment that number by one, the result is 16#8000 (bit 15 is set). For signed integers, hexadecimal 16#8000 is equal to -32768 decimal. Incrementing from this point on until all 16 bits are set ends up at 16#FFFF, which is equal to -1 decimal.

This can be shown as a circular number line (see the following diagrams). The LIM instruction starts at the Low Limit and increments clockwise until it reaches the High Limit. Any Test value in the clockwise range from the Low Limit to the High Limit sets the rung-condition-out to true. Any Test value in the clockwise range from the High Limit to the Low Limit sets the rung-condition-out to false.

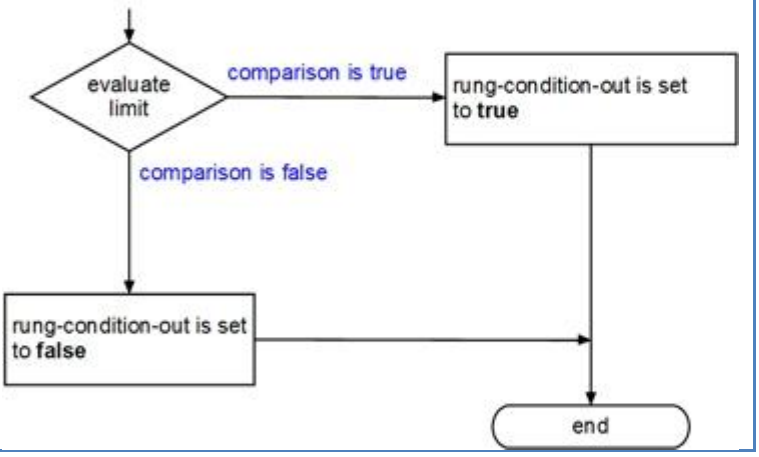
Low Limit ≤ High Limit	**Low Limit ≥ High Limit**
The instruction is true if the test value is equal to or between the low and high limit.	The instruction is true if the test value is equal to or outside the low and high limit.





Execution:

Condition	Action
prescan	The rung-condition-out is set to false.

Condition	Action
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	

Example 1:

Low Limit ≤ High Limit:

When 0 ≤ Value ≥ 100, set Light_1. If Value < 0 or Value >100, clear Light_1.

Example 2:

Low Limit ≥ High Limit:

When Value ≥ 0 or Value ≤ −100, set Light_1. If Value < 0 or Value >−100, clear Light_1.


LIM
Limit Test (CIRC)
Low Limit 0
Test Value
0
High Limit -100
Light_1

5.3.2 Mask Equal to (MEQ)

The MEQ instruction passes the Source and Compare values through a Mask and compares the results.

Operands:

Operand	Type	Format	Description
Source	SINT INT **DINT**	immediate tag	value to test against Compare
Mask	SINT INT **DINT**	immediate tag	defines which bits to block or pass
Compare	SINT INT **DINT**	immediate tag	value to test against Source

If you enter a SINT or INT tag, the value converts to a DINT value by zero-fill.

Description:

A "1" in the mask means the data bit is passed. A "0" in the mask means the data bit is blocked. Typically, the Source, Mask, and Compare values are all the same data type.

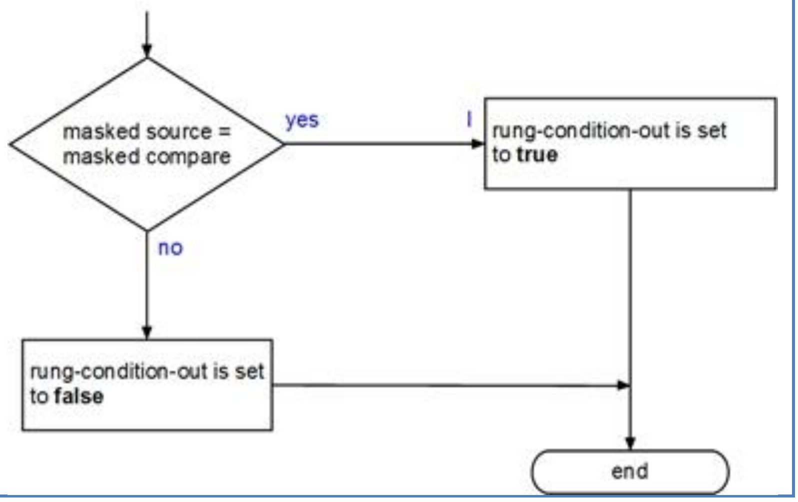
If you mix integer data types, the instruction fills the upper bits of the smaller integer data types with 0s so that they are the same size as the largest data type.

Entering an Immediate Mask Value:

When you enter a mask, the programming software defaults to decimal values. If you want to enter a mask using another format, precede the value with the correct prefix.

Prefix	Description	Example
2#	binary	2#00110011
8#	octal	8#16
16#	hexadecimal	16#0F0F

Execution:

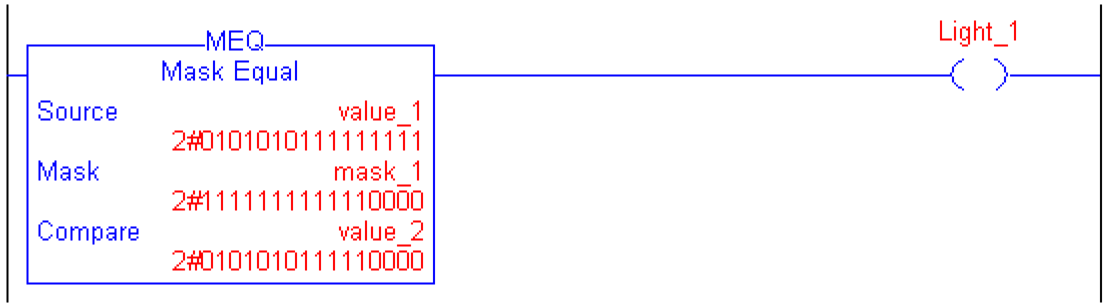
Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	

Example 1:

If the masked value_1 is equal to the masked value_2, set light_1. If the masked value_1 is not equal to the masked value_2, clear light_1. This example shows that the masked values are equal. A 0 in the mask restrains the instruction from comparing that bit (shown by x in the example).

value_1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1
mask_1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
Masked *value_1*	0	1	0	1	0	1	0	1	1	1	1	1	x	x	x	x

value_2	0	1	0	1	0	1	0	1	1	1	1	1	0	0	0	0
mask_1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
Masked *value_2*	0	1	0	1	0	1	0	1	1	1	1	1	x	x	x	x

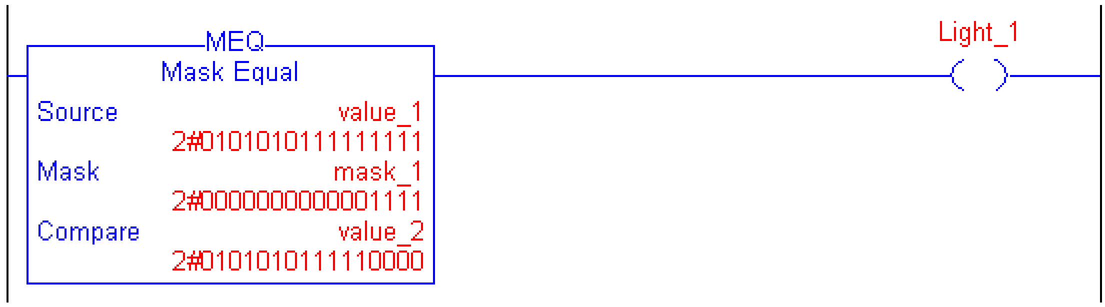


Example 2:

If the masked value_1 is equal to the masked value_2, set light_1. If the masked value_1 is not equal to the masked value_2, clear light_1. This example shows that the masked values are not equal. A 0 in the mask restrains the instruction from comparing that bit (shown by x in the example).

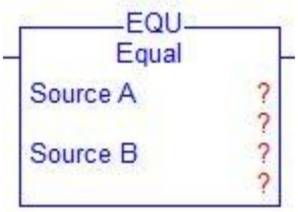
value_1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1
mask_1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
Masked *value_1*	x	x	x	x	x	x	x	x	x	x	x	x	1	1	1	1

value_2	0	1	0	1	0	1	0	1	1	1	1	1	0	0	0	0
mask_1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
Masked *value_2*	x	x	x	x	x	x	x	x	x	x	x	x	0	0	0	0



5.3.3 Equal to (EQU)

The EQU instruction tests whether Source A is equal to Source B.



Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value to test against Source B
Source B	SINT INT **DINT**	immediate tag	value to test against Source A

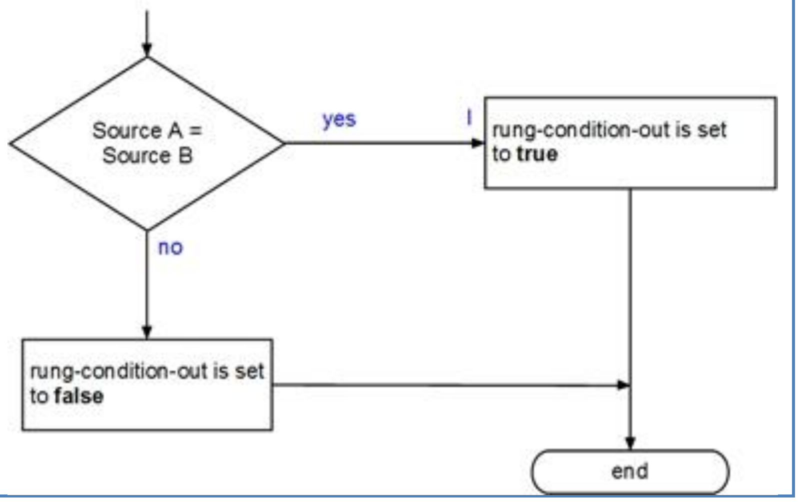
If you enter a SINT or INT tag, the value converts to a DINT value by sign-extension.

Description:

Use the EQU instruction to compare two numbers.

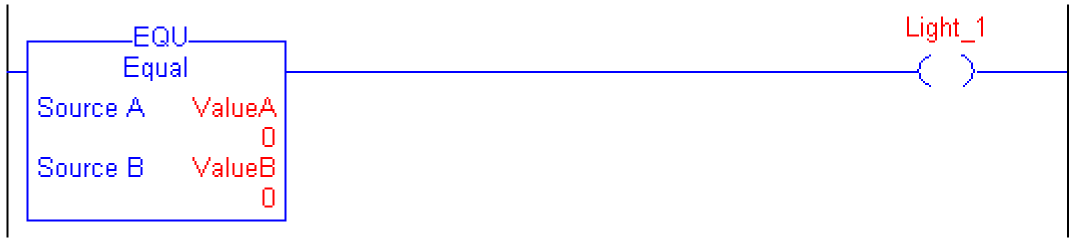
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.

Condition	Action
rung-condition-in is true	

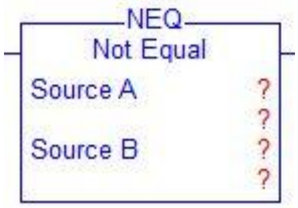
Example:

If ValueA is equal to ValueB, set Light_1. If ValueA is not equal to ValueB, clear Light_1.



5.3.4 Not Equal to (NEQ)

The NEQ instruction tests whether Source A is not equal to Source B.



Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value to test against Source B
Source B	SINT INT **DINT**	immediate tag	value to test against Source A

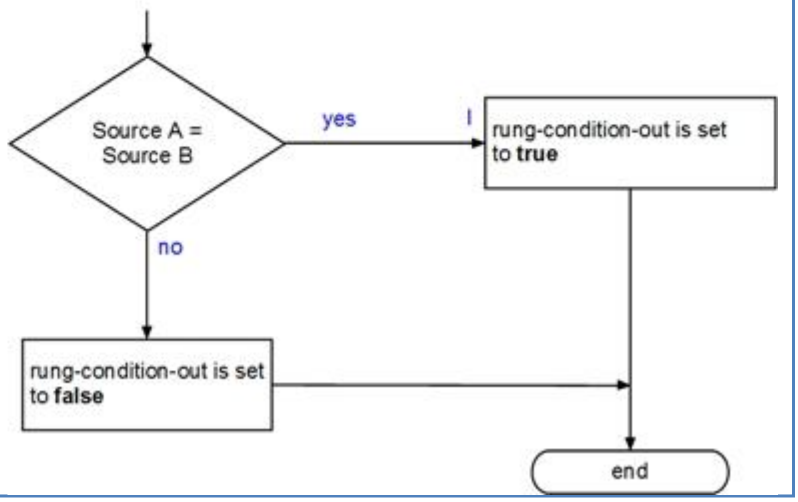
If you enter a SINT or INT tag, the value converts to a DINT value by sign-extension.

Description:

The NEQ instruction tests whether Source A is not equal to Source B.

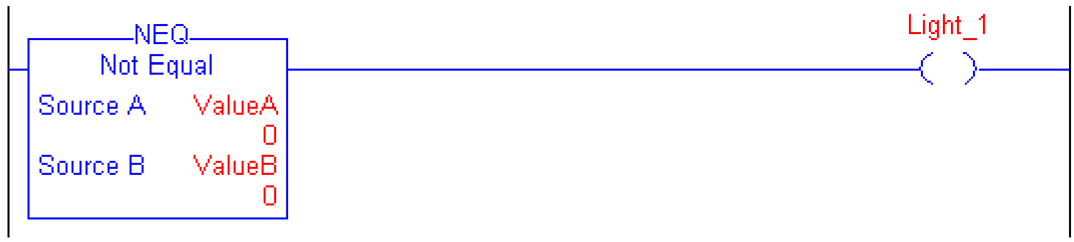
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.

<table>
<tr><th>Condition</th><th>Action</th></tr>
<tr><td>rung-condition-in is true</td><td>
</td></tr>
</table>

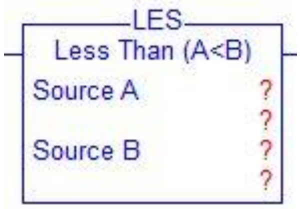
Example:

If ValueA is not equal to ValueB, set Light_1. If ValueA is equal to ValueB, clear Light_1.



5.3.5 Less Than (LES)

The LES instruction tests whether Source A is less than Source B.



Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value to test against Source B
Source B	SINT INT **DINT**	immediate tag	value to test against Source A

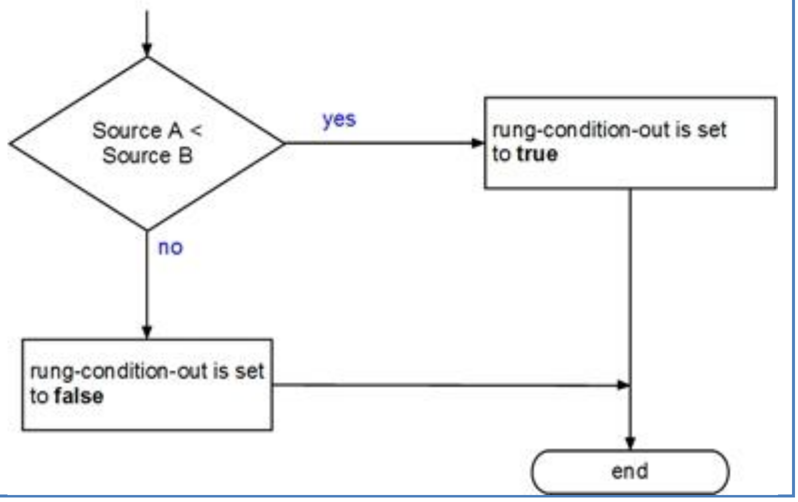
If you enter a SINT or INT tag, the value converts to a DINT value by sign-extension.

Description:

The LES instruction tests whether Source A is less than Source B.

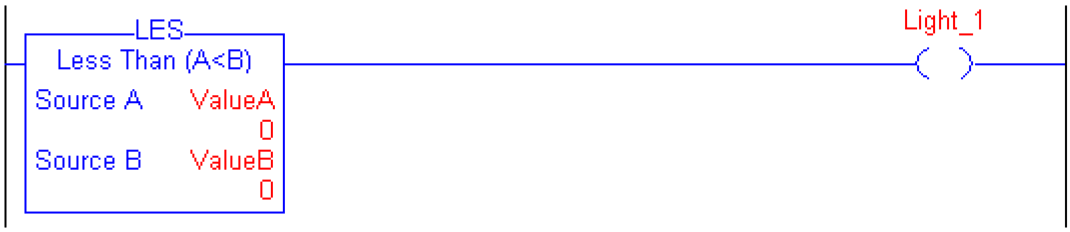
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.

Condition	Action
rung-condition-in is true	

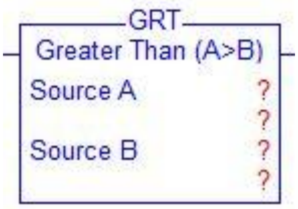
Example:

If ValueA is less than ValueB, set Light_1. If ValueA is greater than or equal to ValueB, clear Light_1.



5.3.6 Greater Than (GRT)

The GRT instruction tests whether Source A is greater than Source B.



Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value to test against Source B
Source B	SINT INT **DINT**	immediate tag	value to test against Source A

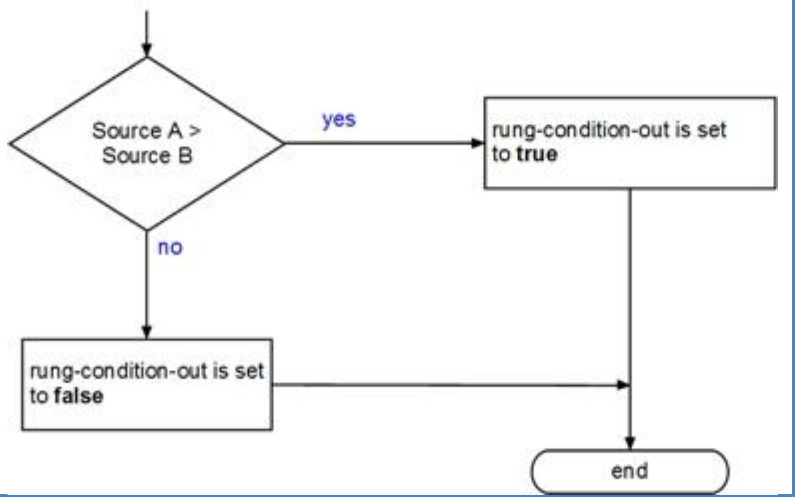
If you enter a SINT or INT tag, the value converts to a DINT value by sign-extension.

Description:

The GRT instruction tests whether Source A is greater than Source B.

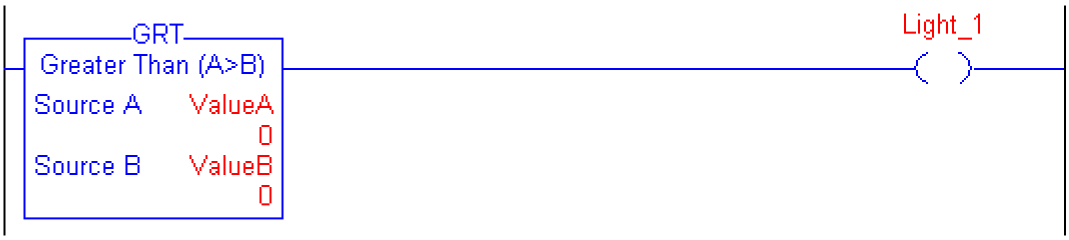
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.

Condition	Action
rung-condition-in is true	

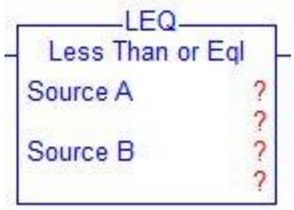
Example:

If ValueA is greater than ValueB, set Light_1. If ValueA is less than or equal to ValueB, clear Light_1.



5.3.7 Less Than or Equal to (LEQ)

The LEQ instruction tests whether Source A is less than or equal to Source B.



Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value to test against Source B
Source B	SINT INT **DINT**	immediate tag	value to test against Source A

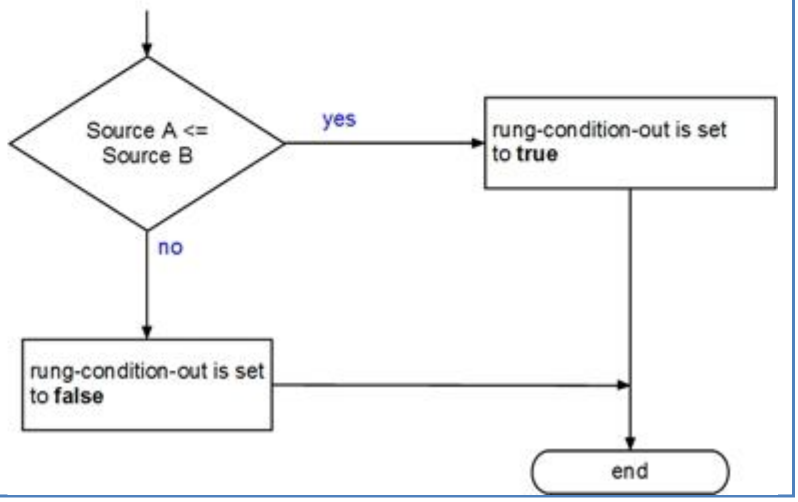
If you enter a SINT or INT tag, the value converts to a DINT value by sign-extension.

Description:

The LEQ instruction tests whether Source A is less than or equal to Source B.

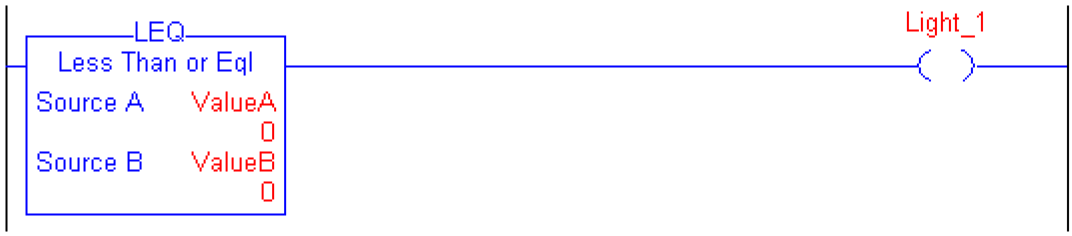
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.

Condition	Action
rung-condition-in is true	

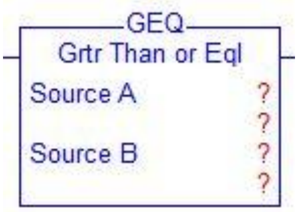
Example:

If ValueA is less than or equal to ValueB, set Light_1. If ValueA is greater than ValueB, clear Light_1.



5.3.8 Greater than or Equal to (GEQ)

The GEQ instruction tests whether Source A is greater than or equal to Source B.



Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value to test against Source B
Source B	SINT INT **DINT**	immediate tag	value to test against Source A

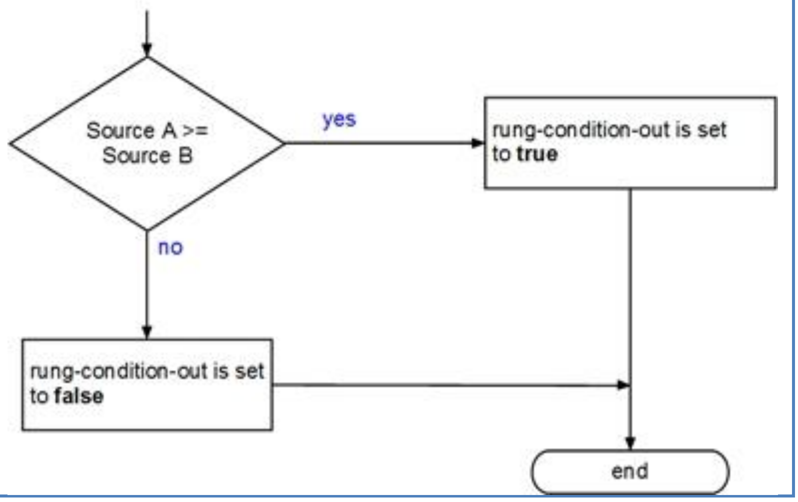
If you enter a SINT or INT tag, the value converts to a DINT value by sign-extension.

Description:

The LEQ instruction tests whether Source A is less than or equal to Source B.

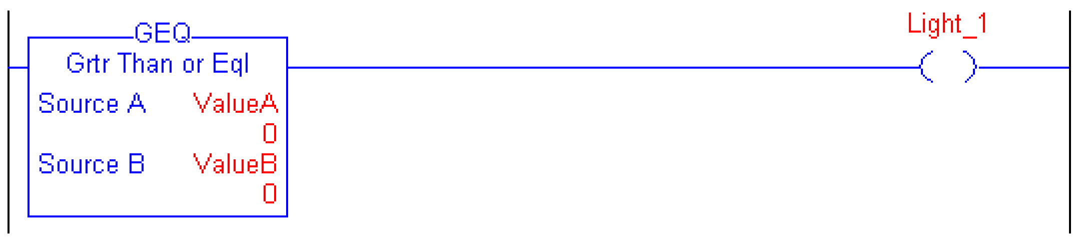
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.

Condition	Action
rung-condition-in is true	

Example:

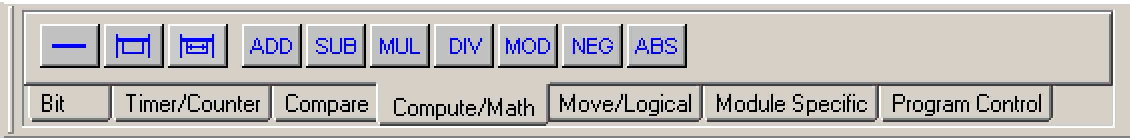
If ValueA is greater than or equal to ValueB, set Light_1. If ValueA is less than ValueB, clear Light_1.



5.4 Compute/Math Instructions

The compute/math instructions evaluate arithmetic operations using an expression or a specific arithmetic instruction.

To enter a compute/math instruction use buttons form Compute/Math tab of Instruction Bar.



Instruction	Description
ADD	add two values
SUB	subtract two values
MUL	multiply two values
DIV	divide two values
MOD	determine the remainder after one value is divided by another
NEG	take the opposite sign of a value
ABS	take the absolute value of a value

For relay ladder instructions, bold data types indicate optimal data types. An instruction executes faster and requires less memory if all the operands of the instruction use the same optimal data type, typically DINT.

5.4.1 Add (ADD)

The ADD instruction adds Source A to Source B and places the result in the Destination.

Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value to add to Source B
	A SINT or INT tag converts to a DINT value by sign-extension.		
Source B	SINT INT **DINT**	immediate tag	value to add to Source A
	A SINT or INT tag converts to a DINT value by sign-extension.		
Destination	SINT INT **DINT**	tag	tag to store the result

Description:

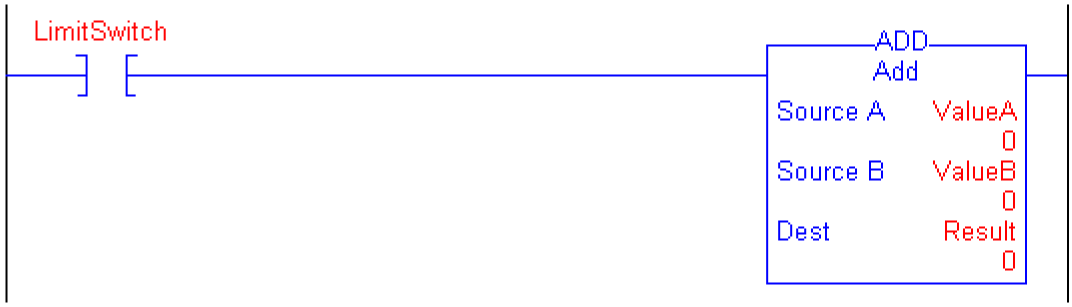
The ADD instruction adds Source A to Source B and places the result in the Destination.

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	Destination = Source A + Source B The rung-condition-out is set to true.

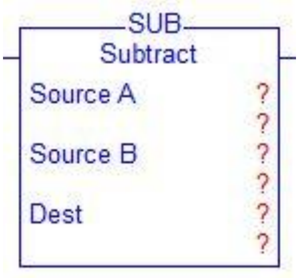
Example:

If LimitSwitch is set, add ValueA to ValueB and place the result in Result.



5.4.2 Subtract (SUB)

The SUB instruction subtracts Source B from Source A and places the result in the Destination.



Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value from which to subtract Source B
	A SINT or INT tag converts to a DINT value by sign-extension.		
Source B	SINT INT **DINT**	immediate tag	value to subtract from Source A
	A SINT or INT tag converts to a DINT value by sign-extension.		
Destination	SINT INT **DINT**	tag	tag to store the result

Description:

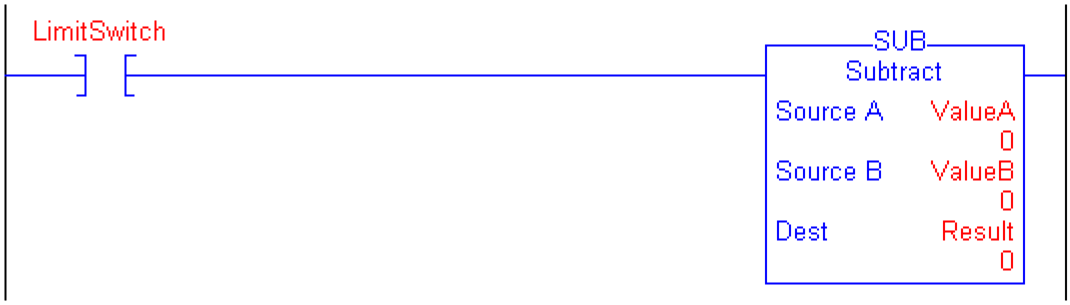
The SUB instruction subtracts Source B from Source A and places the result in the Destination.

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	Destination = Source B - Source A The rung-condition-out is set to true.

Example:

If LimitSwitch is set, subtract ValueB from ValueA and place the result in Result.



5.4.3 Multiply (MUL)

The MUL instruction multiplies Source A with Source B and places the result in the Destination.

Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value of the multiplicand
	A SINT or INT tag converts to a DINT value by sign-extension.		
Source B	SINT INT **DINT**	immediate tag	value of the multiplier
	A SINT or INT tag converts to a DINT value by sign-extension.		
Destination	SINT INT **DINT**	tag	tag to store the result

Description:

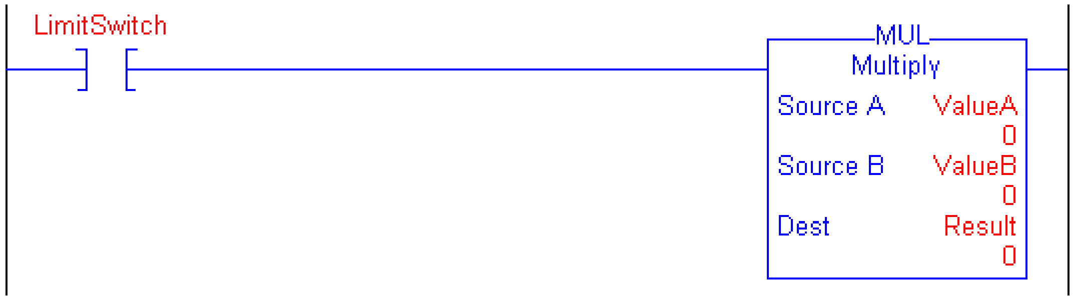
The MUL instruction multiplies Source A with Source B and places the result in the Destination.

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	Destination = Source B x Source A The rung-condition-out is set to true.

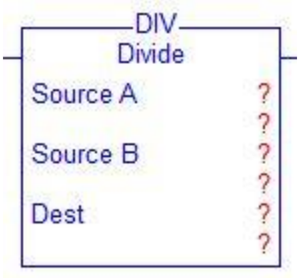
Example:

If LimitSwitch is set, multiply ValueA by ValueB and place the result in Result.



5.4.4 Divide (DIV)

The DIV instruction divides Source A by Source B and places the result in the Destination.



Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value of the dividend
	A SINT or INT tag converts to a DINT value by sign-extension.		
Source B	SINT INT **DINT**	immediate tag	value of the divisor
	A SINT or INT tag converts to a DINT value by sign-extension.		
Destination	SINT INT **DINT**	tag	tag to store the result

Description:

DIV instruction truncates the result.

Operand	Type	Value
Source A	DINT	5
Source B	DINT	3
Destination	DINT	1

If Source B (the divisor) is zero, DIV instruction doesn't evaluate and the next runtime error occurs:

#103 – Divide by Zero

If ConveyLogix Programmer is in Debug mode, runtime errors are shown in Output window.

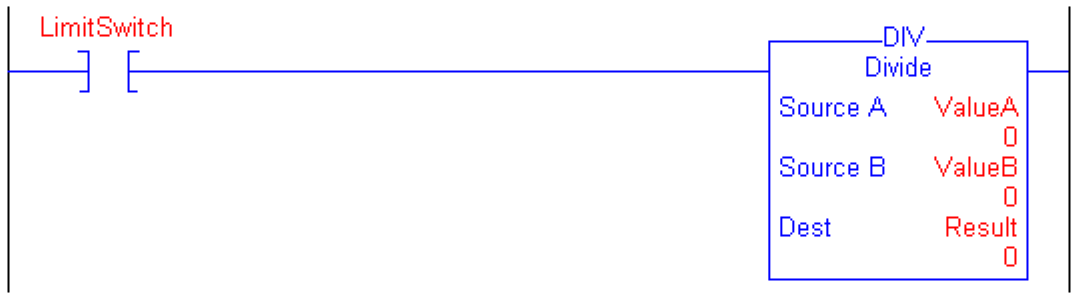
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	Destination = Source A / Source B The rung-condition-out is set to true.

Example:

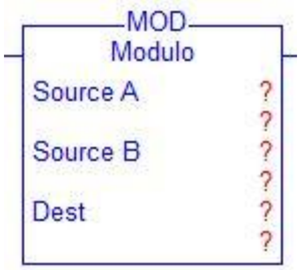
If LimitSwitch is set, divide ValueA by ValueB and place the result in Result.

If ValueB (the divisor) is zero, DIV instruction doesn't evaluate.



5.4.5 Modulo (MOD)

The MOD instruction divides Source A by Source B and places the remainder in the Destination.



Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value of the dividend
	A SINT or INT tag converts to a DINT value by sign-extension.		
Source B	SINT INT **DINT**	immediate tag	value of the divisor
	A SINT or INT tag converts to a DINT value by sign-extension.		
Destination	SINT INT **DINT**	tag	tag to store the result

Description:

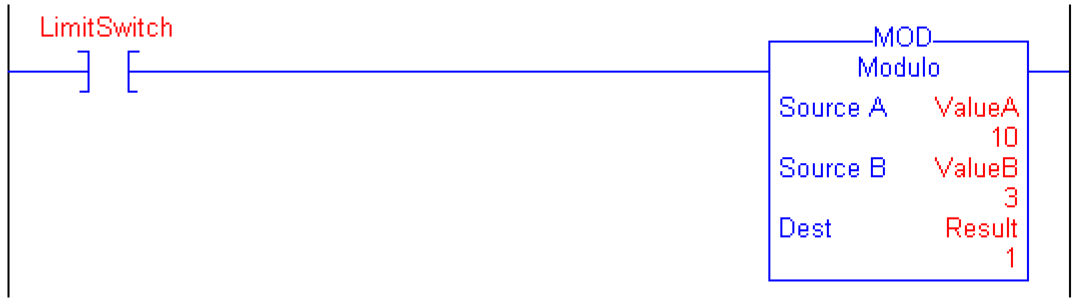
If Source B (the divisor) is zero, Source A is moved to Destination.

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	Destination = Source A – (TRN (Source A / Source B) * Source B) The rung-condition-out is set to true.

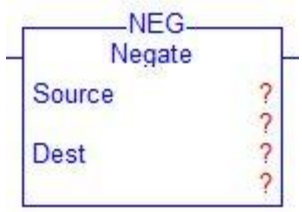
Example:

If LimitSwitch is set, divide ValueA by ValueB and place the remainder in Result. In this example, 3 goes into 10 three times, with a remainder of 1.



5.4.6 Negate (NEG)

The NEG instruction changes the sign of the Source and places the result in the Destination.



Operands:

Operand	Type	Format	Description
Source	SINT INT **DINT**	immediate tag	value to negate
	A SINT or INT tag converts to a DINT value by sign-extension.		
Destination	SINT INT **DINT**	tag	tag to store the result

Description:

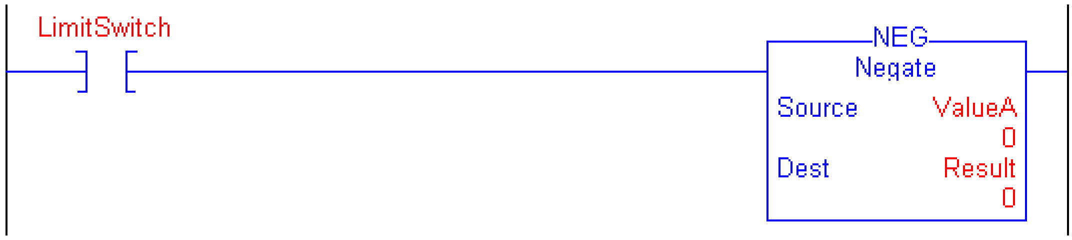
If you negate a negative value, the result is positive. If you negate a positive value, the result is negative.

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	Destination = 0 − Source The rung-condition-out is set to true.

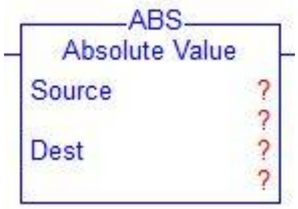
Example:

If LimitSwitch is set, change the sign of ValueA and place the result in Result.



5.4.7 Absolute Value (ABS)

The ABS instruction takes the absolute value of the Source and places the result in the Destination.



Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value of which to take the absolute value
	A SINT or INT tag converts to a DINT value by sign-extension.		
Destination	SINT INT **DINT**	tag	tag to store the result

Description:

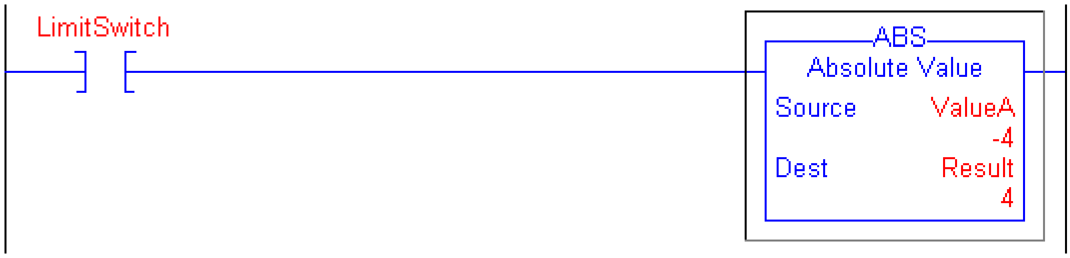
The ABS instruction takes the absolute value of the Source and places the result in the Destination.

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	Destination = |Source| The rung-condition-out is set to true.

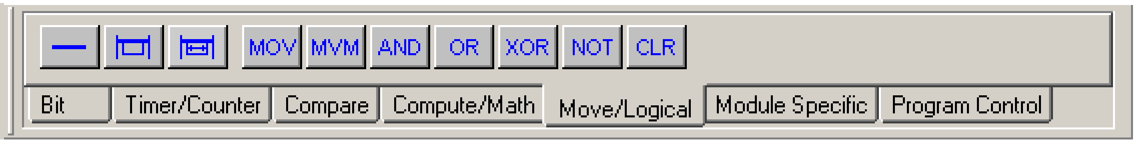
Example:

If LimitSwitch is set, place the absolute value of ValueA into Result. In this example, the absolute value of negative four is positive four.



5.5 Move/Logical Instructions

To enter a move/logical instruction use buttons form Move/Logical tab of Instruction Bar.



The move instructions modify and move bits.

Instruction	Description
MOV	copy a value
MVM	copy a specific part of an integer
CLR	clear a value

The logical instructions perform operations on bits.

Instruction	Description
Bitwise AND	bitwise AND operation
Bitwise OR	bitwise OR operation
Bitwise XOR	bitwise, exclusive OR operation
Bitwise NOT	bitwise NOT operation

You can mix data types, but loss of accuracy and the instruction takes more time to execute.

Bold data types indicate optimal data types. An instruction executes faster if all the operands of the instruction use the same optimal data type, typically DINT.

5.5.1 Move (MOV)

The MOV instruction copies the Source to the Destination. The Source remains unchanged.

Operands:

Operand	Type	Format	Description
Source	SINT INT **DINT**	immediate tag	value to move (copy)
	A SINT or INT tag converts to a DINT value by sign-extension.		
Destination	SINT INT **DINT**	tag	tag to store the result

Description:

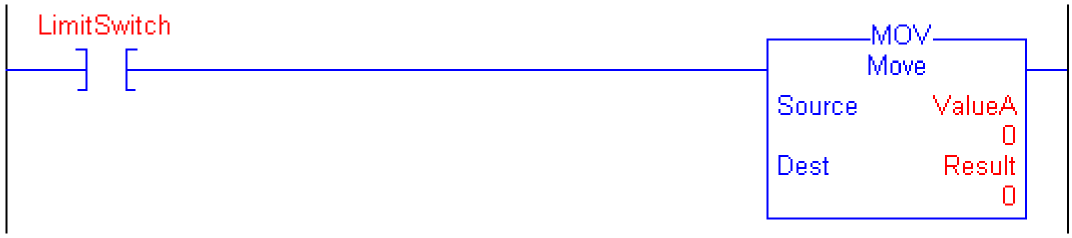
The MOV instruction copies the Source to the Destination. The Source remains unchanged.

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	The instruction copies the Source into the Destination. The rung-condition-out is set to true.

Example:

If LimitSwitch is set, move the data in ValueA to Result.



5.5.2 Masked Move (MVM)

The MVM instruction copies the Source to a Destination and allows portions of the data to be masked.

Operands:

Operand	Type	Format	Description
Source	SINT INT **DINT**	immediate tag	value to move
	A SINT or INT tag converts to a DINT value by zero-fill.		
Mask	SINT INT **DINT**	immediate tag	which bits to block or pass
	A SINT or INT tag converts to a DINT value by zero-fill.		
Destination	SINT INT **DINT**	tag	tag to store the result

Description:

The MVM instruction uses a Mask to either pass or block Source data bits. A “1” in the mask means the data bit is passed. A “0” in the mask means the data bit is blocked.

If you mix integer data types, the instruction fills the upper bits of the smaller integer data types with 0s so that they are the same size as the largest data type.

Entering an Immediate Mask Value:

When you enter a mask; the programming software defaults to decimal values. If you want to enter a mask using another format, precede the value with the correct prefix.

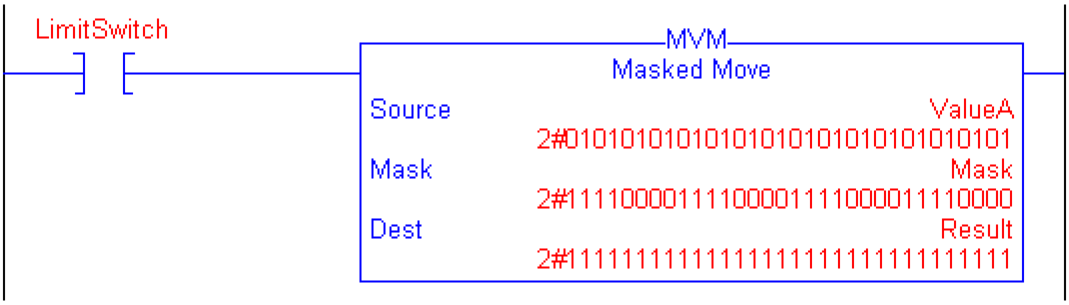
Prefix	Description	Example
2#	binary	2#00110011
8#	octal	8#16
16#	hexadecimal	16#0F0F

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	The instruction passes the Source through the Mask and copies the result into the Destination. Unmasked bits in the Destination remain unchanged. The rung-condition-out is set to true.

Example:

If LimitSwitch is set, copy data from ValueA to Result, while allowing data to be masked (a 0 masks the data in ValueA).



1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
0	1	0	1	1	1	1	1	0	1	0	1	1	1	1	1	0	1	0	1	1	1	1	1	0	1	0	1	1	1	1	1

The shaded boxes show the bits that changed in Result.

5.5.3 Bitwise AND (AND)

The AND instruction performs a bitwise AND operation using the bits in Source A and Source B and places the result in the Destination.

Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value to AND with Source B
	A SINT or INT tag converts to a DINT value by zero-fill.		
Source B	SINT INT **DINT**	immediate tag	value to AND with Source A
	A SINT or INT tag converts to a DINT value by zero-fill.		
Destination	SINT INT **DINT**	tag	tag to store the result

Description:

When enabled, the instruction evaluates the AND operation:

If the Bit In Source A Is	And the Bit In Source B Is:	The Bit In the Destination Is:
0	0	0
0	1	0
1	0	0
1	1	1

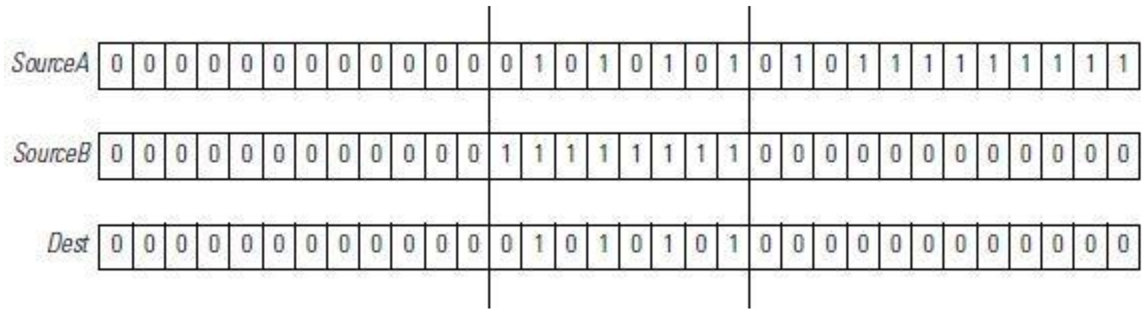
If you mix integer data types, the instruction fills the upper bits of the smaller integer data types with 0s so that they are the same size as the largest data type.

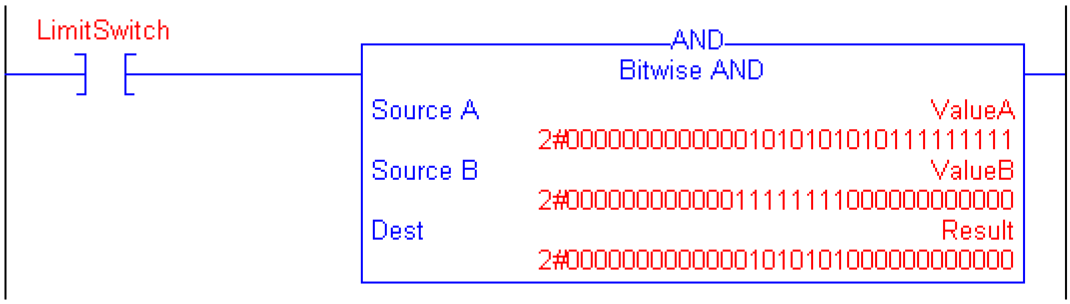
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	The instruction performs a bitwise AND operation. The rung-condition-out is set to true.

Example:

When enabled, the AND instruction performs a bitwise AND operation on ValueA and ValueB and places the result in the Result.





5.5.4 Bitwise OR (OR)

The OR instruction performs a bitwise OR operation using the bits in Source A and Source B and places the result in the Destination.

Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value to OR with Source B
	A SINT or INT tag converts to a DINT value by zero-fill.		
Source B	SINT INT **DINT**	immediate tag	value to OR with Source A
	A SINT or INT tag converts to a DINT value by zero-fill.		
Destination	SINT INT **DINT**	tag	tag to store the result

Description:

When enabled, the instruction evaluates the OR operation:

If the Bit In Source A Is	And the Bit In Source B Is:	The Bit In the Destination Is:
0	0	0
0	1	1
1	0	1
1	1	1

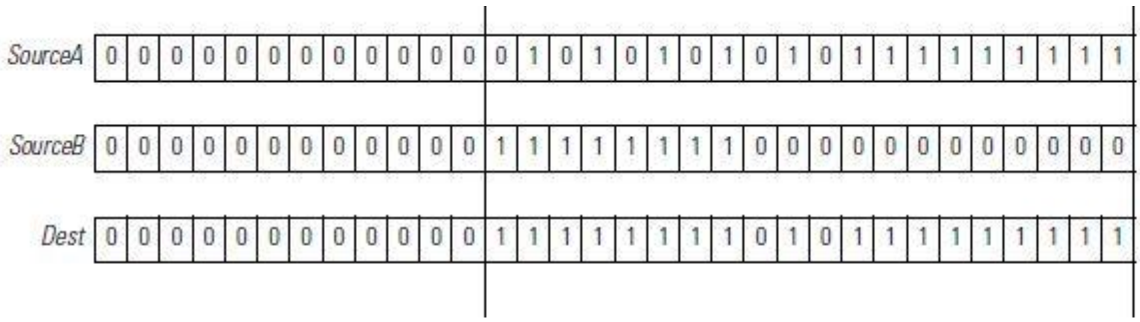
If you mix integer data types, the instruction fills the upper bits of the smaller integer data types with 0s so that they are the same size as the largest data type.

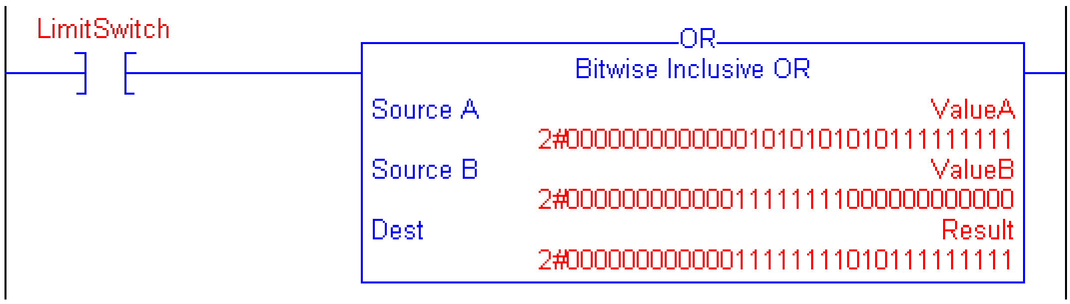
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	The instruction performs a bitwise OR operation. The rung-condition-out is set to true.

Example:

When enabled, the OR instruction performs a bitwise OR operation on ValueA and ValueB and places the result in Result.





5.5.5 Bitwise Exclusive OR (XOR)

The XOR instruction performs a bitwise XOR operation using the bits in Source A and Source B and places the result in the Destination.

Operands:

Operand	Type	Format	Description
Source A	SINT INT **DINT**	immediate tag	value to XOR with Source B
	A SINT or INT tag converts to a DINT value by zero-fill.		
Source B	SINT INT **DINT**	immediate tag	value to XOR with Source A
	A SINT or INT tag converts to a DINT value by zero-fill.		
Destination	SINT INT **DINT**	tag	tag to store the result

Description:

When enabled, the instruction evaluates the XOR operation:

If the Bit In Source A Is	And the Bit In Source B Is:	The Bit In the Destination Is:
0	0	0
0	1	1
1	0	1
1	1	0

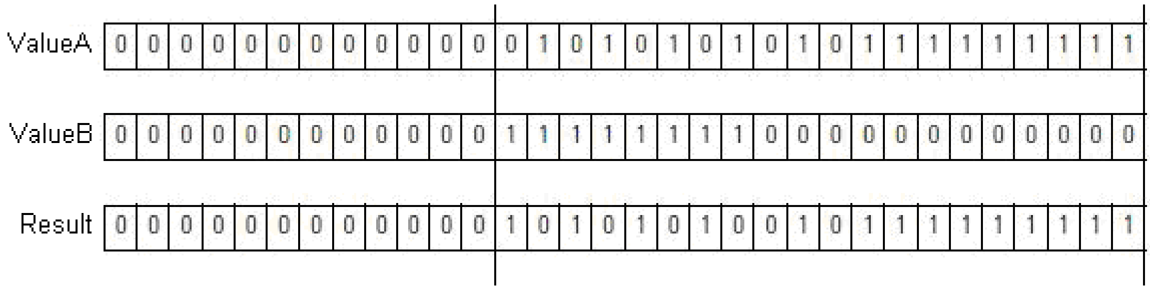
If you mix integer data types, the instruction fills the upper bits of the smaller integer data types with 0s so that they are the same size as the largest data type.

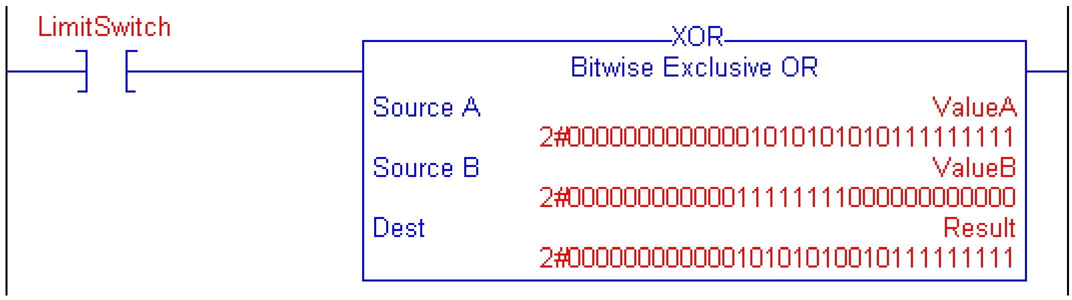
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	The instruction performs a bitwise OR operation. The rung-condition-out is set to true.

Example:

When enabled, the XOR instruction performs a bitwise XOR operation on ValueA and ValueB and places the result in the Result tag.





5.5.6 Bitwise NOT (NOT)

The NOT instruction performs a bitwise NOT operation using the bits in the Source and places the result in the Destination.

Operands:

Operand	Type	Format	Description
Source	SINT INT **DINT**	immediate tag	value to NOT
	A SINT or INT tag converts to a DINT value by sign-extension.		
Destination	SINT INT **DINT**	tag	tag to store the result

Description:

When enabled, the instruction evaluates the NOT operation:

If the Bit In Source Is:	The Bit In theDestination Is:
0	1
1	0

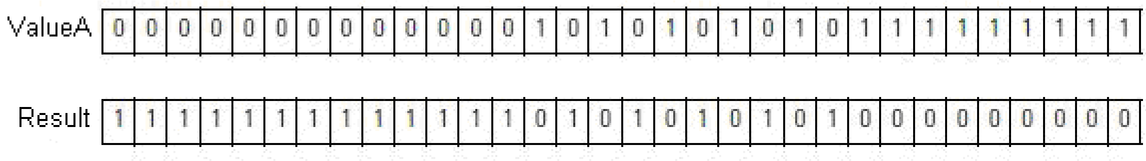
If you mix integer data types, the instruction fills the upper bits of the smaller integer data types with 0s so that they are the same size as the largest data type.

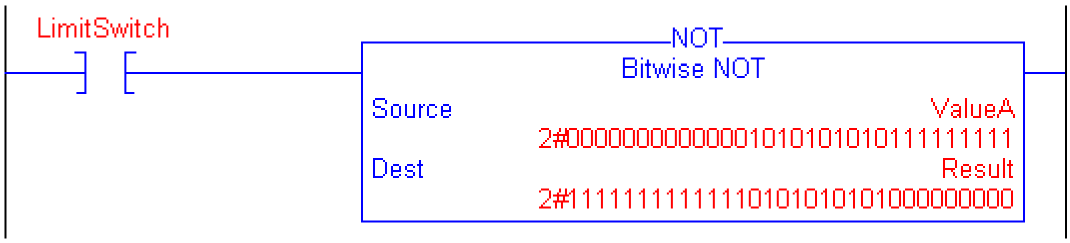
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	The instruction performs a bitwise NOT operation. The rung-condition-out is set to true.

Example:

When enabled, the NOT instruction performs a bitwise NOT operation on ValueA and places the result in Result tag.





5.5.7 Clear (CLR)

The CLR instruction clears all the bits of the Destination.

Operands:

Operand	Type	Format	Description
Destination	SINT INT **DINT**	tag	tag to clear

Description:

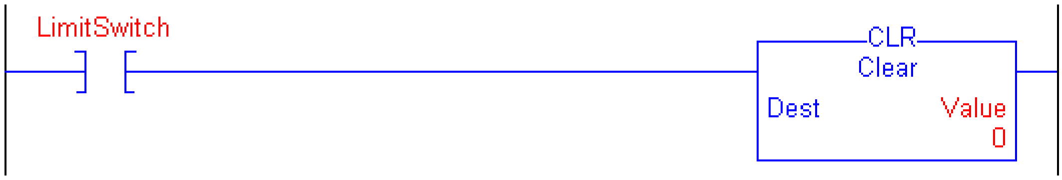
The CLR instruction clears all the bits of the Destination.

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	The instruction clears the Destination. The rung-condition-out is set to true.

Example:

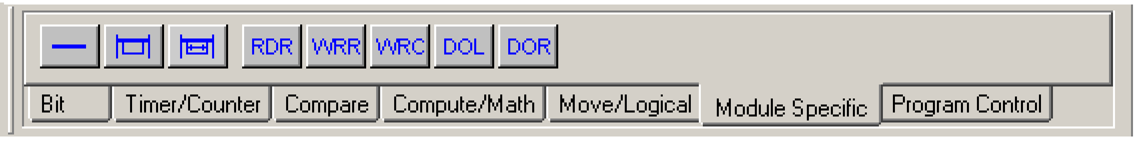
Let Value is equal to 9999. When enabled, clear all the bits of Value to 0.



5.6 Module Specific Instructions

The module specific instructions perform controller-specific operations.

To enter a module specific instruction use buttons form Module Specific tab of Instruction Bar.



Instruction	Description
RDR	read local Modbus register
WRR	write local Modbus register
WRC	write local Modbus register and send via communication
DOL	count pulses of the left motor when enabled
DOR	count pulses of the right motor when enabled

DOL and DOR instructions are available only for ConveyLinx controller type.

5.6.1 Read Register (RDR)

The RDR instruction copies the value of local Modbus register, referred to Reg No, to the Destination.

Operands:

Operand	Type	Format	Description
Reg No	Modbus Register	immediate	Modbus register number. Must be from 1 to 512.
Destination	SINT INT DINT	tag	tag to store the result

Description:

The RDR instruction copies the value of local Modbus register, referred to Reg No, to the Destination. The Modbus register value remains unchanged.

Destination Type	Action
SINT	Low BYTE of the Modbus register is copied to the Destination.
INT	The Modbus register is copied to the Destination.
DINT	Two consecutive Modbus registers are copied to the Destination. The first register is copied to Low WORD and the second – to High WORD of the Destination.

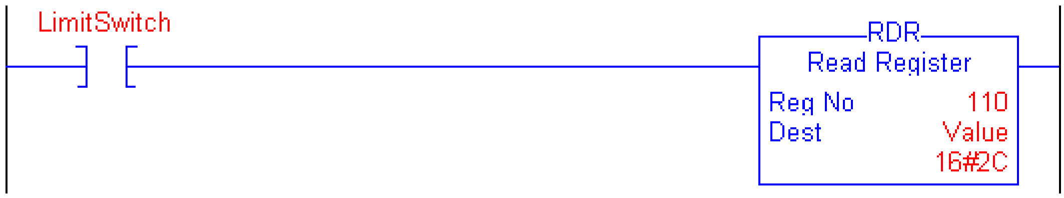
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	The instruction copies the value, referred to Reg No, into the Destination. The rung-condition-out is set to true.

Example 1:

Type of Value is SINT. Let the value of local Modbus register 110 is 300 (16#012C).

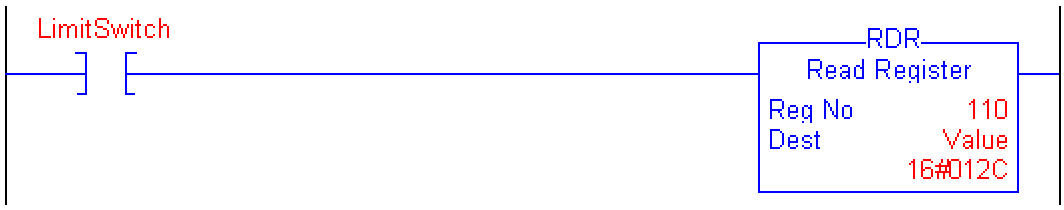
When enabled, read register 110 and put low BYTE (16#2C) of the value to Value tag. The high BYTE is truncated.



Example 2:

Type of Value is INT. Again let the value of local Modbus register 110 is 300 (16#012C).

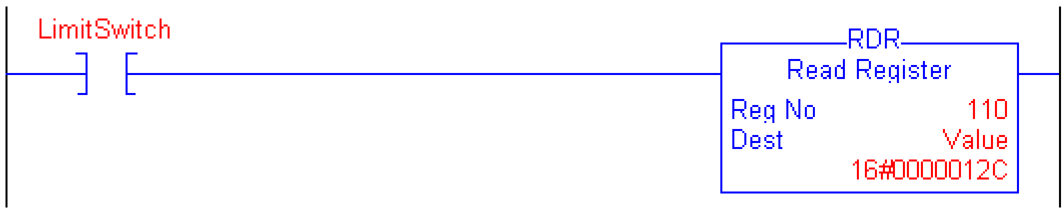
When enabled, read register 110 and put the value to Value tag.



Example 3:

Type of Value is DINT. Let the value of local Modbus register 110 is 300 (16#012C) and value of local Modbus register 111 is 0 (16#0000).

When enabled, read registers 110 and 111 and put the value of register 110 to low WORD (16#012C) of Value tag and the value of registers 111 to high WORD (16#0000) of Value tag.



5.6.2 Write Register (WRR)

The WRR instruction copies the value of Source to local Modbus register, referred to Reg No.

Operands:

Operand	Type	Format	Description
Source	SINT INT DINT	tag	value to write
Reg No	Modbus Register	immediate	Modbus register number. Must be from 1 to 512.

Description:

The WRR instruction copies the value of Source to local Modbus register, referred to Reg No. The Source value remains unchanged.

Source Type	Action
SINT	The Source is copied to the Low BYTE of the Modbus register. The High BYTE of the Modbus register remains unchanged.
INT	The Source is copied to the Modbus register.
DINT	The Source is copied to two consecutive Modbus registers. The Low WORD of Source is copied to the first Modbus register and the High WORD – to the second Modbus registers.

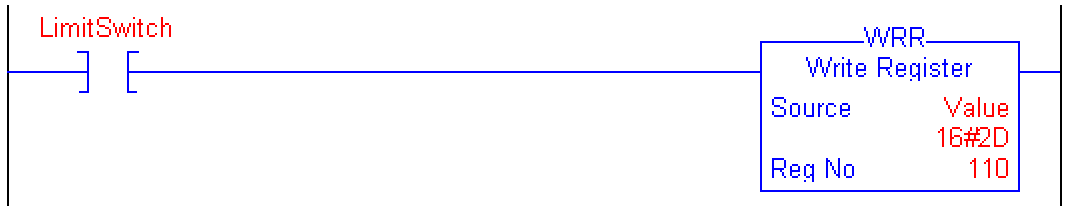
Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	The instruction copies the value of Source to Modbus register, referred to Reg No. The rung-condition-out is set to true.

Example 1:

Let type of Value is SINT and Value is equal to 45 (16#2D).

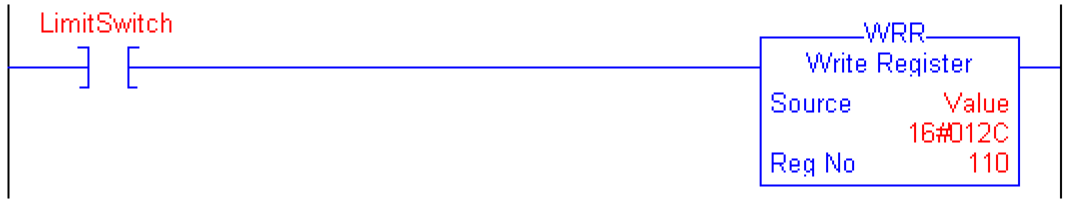
When enabled, copies the value of Value tag to the Low BYTE of the Modbus register 110. The High BYTE of the Modbus register 110 remains unchanged.



Example 2:

Let type of Value is INT and Value is equal to 300 (16#012C).

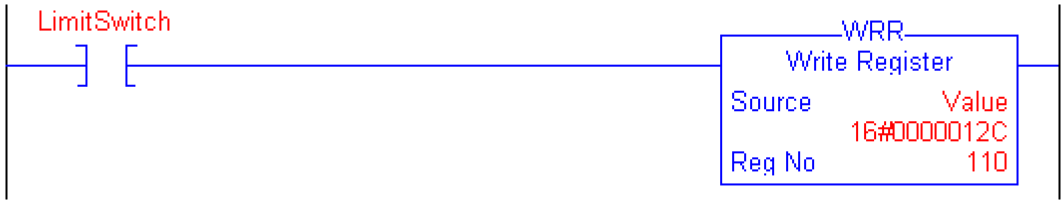
When enabled, copies the value of Value tag to Modbus register 110.



Example 3:

Let type of Value is DINT and Value is equal to 300 (16#0000012C).

When enabled, copies the low WORD of Value tag (16#012C) to Modbus register 110 and the high WORD of Value tag (16#0000) to Modbus register 111.



5.6.3 Write Register Comm (WRC)

The WRC instruction copies the value of Source to local Modbus register, referred to Reg No and send via communication.

Operands:

Operand	Type	Format	Description
Source	SINT INT DINT	tag	value to write
Reg No	Modbus Register	immediate	Modbus register number. Must be from 1 to 512.

Description:

The WRC instruction copies the value of Source to local Modbus register, referred to Reg No and send via communication. The Source value remains unchanged.

ConveyLinx and ConveyNet controllers are organized by events. When using the WRC instruction; it may cause interrupts to awaken idle tasks. Frequent use of the WRC instruction in certain cases may affect processor loading and performance such that communications and/or motor commutation tasks may delay or cause unexpected results.

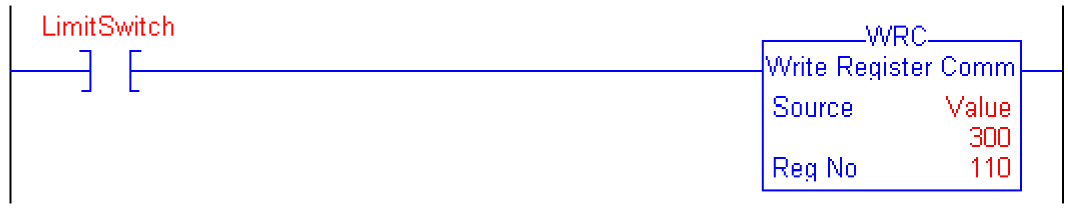
Source Type	Action
SINT	The Source is copied to the Low BYTE of the Modbus register. The High BYTE of the Modbus register remains unchanged.
INT	The Source is copied to the Modbus register.
DINT	The Source is copied to two consecutive Modbus registers. The Low WORD of Source is copied to the first Modbus register and the High WORD – to the second Modbus registers.

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	The instruction copies the value of Source to Modbus register, referred to Reg No. The rung-condition-out is set to true.

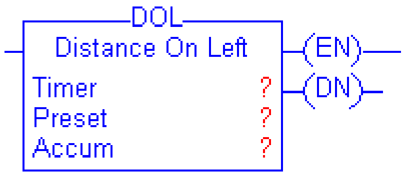
Example:

When enabled, copies the value of Value tag to local Modbus register 110. If register 110 participates to any of the controller events, sends update to the other controller(s).



5.6.4 Distance On Left (DOL)

The DOL instruction counts evaluated pulses of the left motor when the instruction is enabled.



Operands:

Operand	Type	Format	Description
Timer	TIMER	tag	TIMER structure
Preset	DINT	immediate	how high to count
Accum	DINT	immediate	evaluated pulses of the left motor initial value is typically 0

TIMER Structure

Mnemonic	Data Type	Description
.EN	BOOL	The enable bit indicates that the DOL instruction is enabled.
.TT	BOOL	The timing bit indicates that a counting operation is in process
.DN	BOOL	The done bit is set when .ACC ≥ .PRE.
.PRE	DINT	The preset value specifies the value which the accumulated value must reach before the instruction sets the .DN bit.
.ACC	DINT	The accumulated value specifies the number of pulses, evaluated from the left motor, the instruction has counted.

Description:

When enabled, the DOL instruction counts the pulses, evaluated of left motor.

The DOL instruction accumulates pulses until:

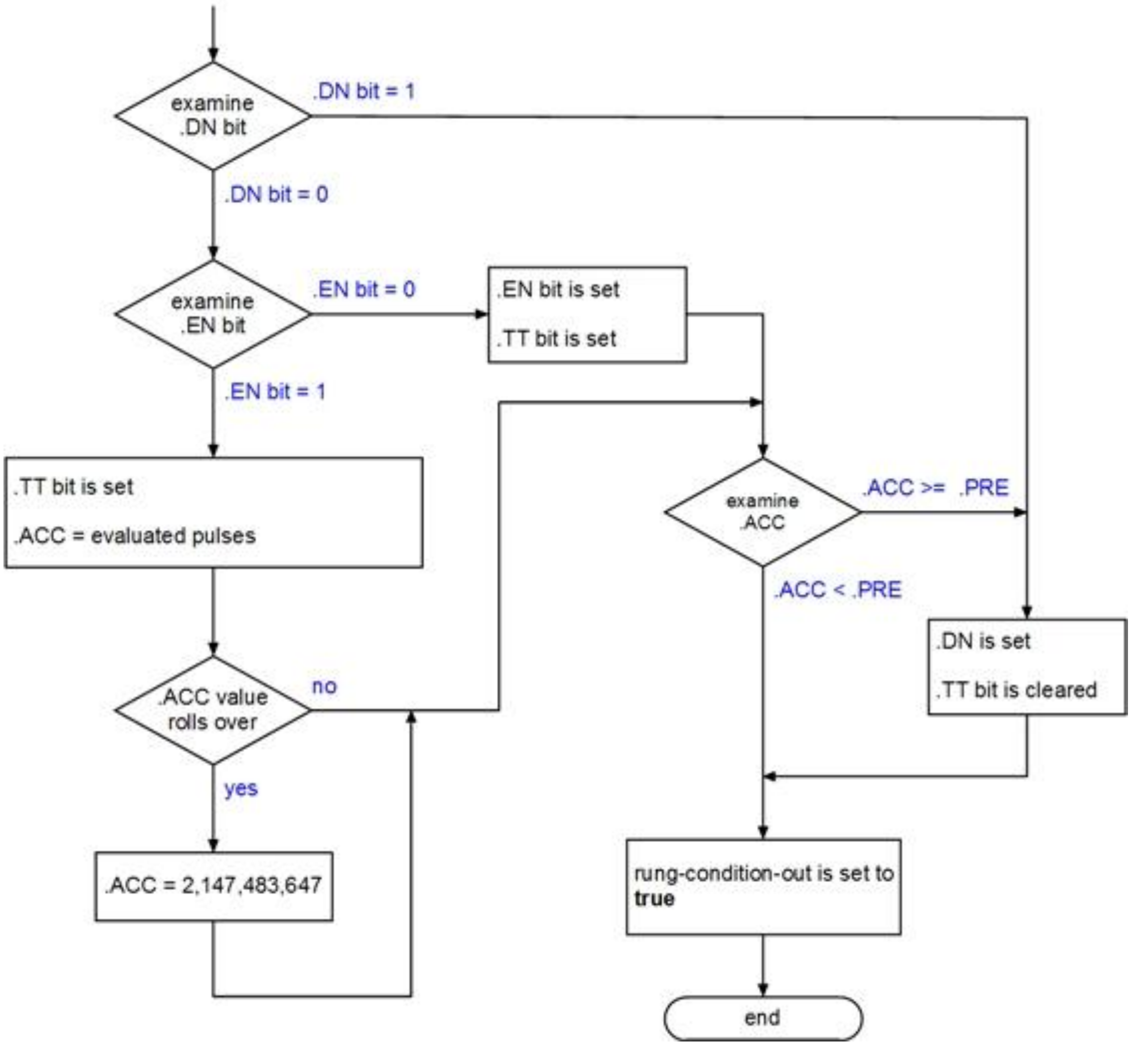
- the DOL instruction is disabled
- the .ACC ≥ .PRE

When the DOL instruction is disabled, the .ACC value is cleared.

DOL instruction is available only for ConveyLinx controller type.

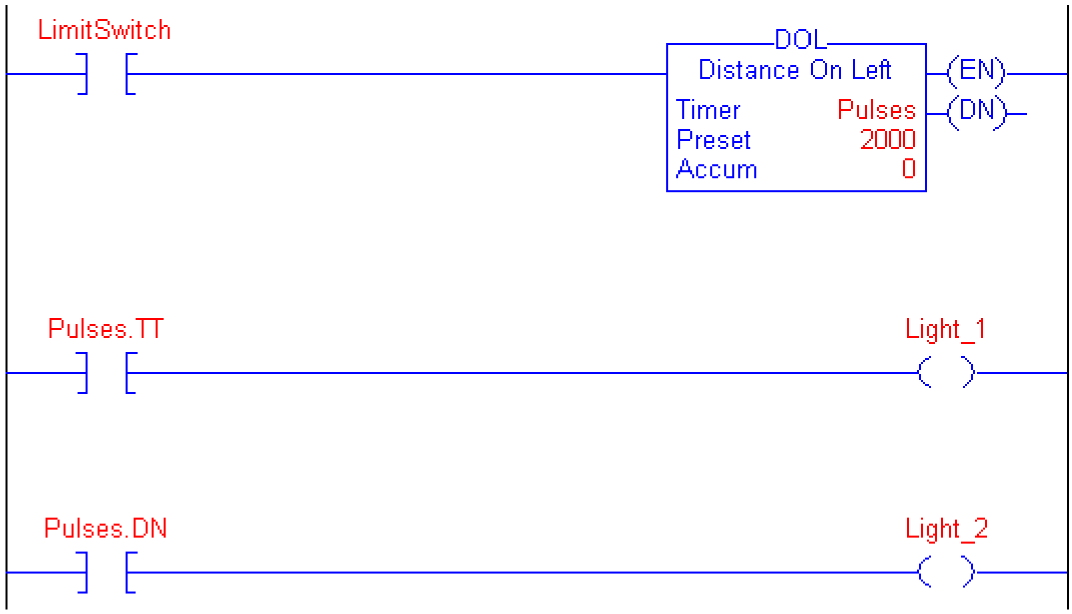
Execution:

Condition	Action
prescan	The .EN, .TT, and .DN bits are cleared. The .ACC value is cleared. The rung-condition-out is set to false.
rung-condition-in is false	The .EN, .TT, and .DN bits are cleared. The .ACC value is cleared. The rung-condition-out is set to false.
rung-condition-in is true	



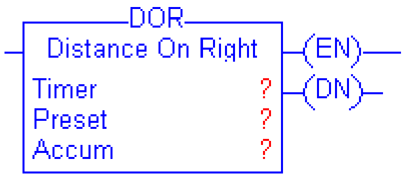
Example:

When LimitSwitch is set, Light_1 is on for 2000 pulses. When Pulses.ACC reaches 2000, Light_1 goes off and Light_2 goes on. Light_2 remains on until the DOL instruction is disabled. If LimitSwitch is cleared while DOL is counting, Light_1 goes off.



5.6.5 Distance On Right (DOR)

The DOR instruction counts evaluated pulses of the right motor when the instruction is enabled.



Operands:

Operand	Type	Format	Description
Timer	TIMER	tag	TIMER structure
Preset	DINT	immediate	how high to count
Accum	DINT	immediate	evaluated pulses of the right motor initial value is typically 0

TIMER Structure

Mnemonic	Data Type	Description
.EN	BOOL	The enable bit indicates that the DOR instruction is enabled.
.TT	BOOL	The timing bit indicates that a counting operation is in process
.DN	BOOL	The done bit is set when .ACC ≥ .PRE.
.PRE	DINT	The preset value specifies the value which the accumulated value must reach before the instruction sets the .DN bit.
.ACC	DINT	The accumulated value specifies the number of pulses, evaluated from the right motor, the instruction has counted.

Description:

When enabled, the DOR instruction counts the pulses, evaluated of right motor.

The DOR instruction accumulates pulses until:

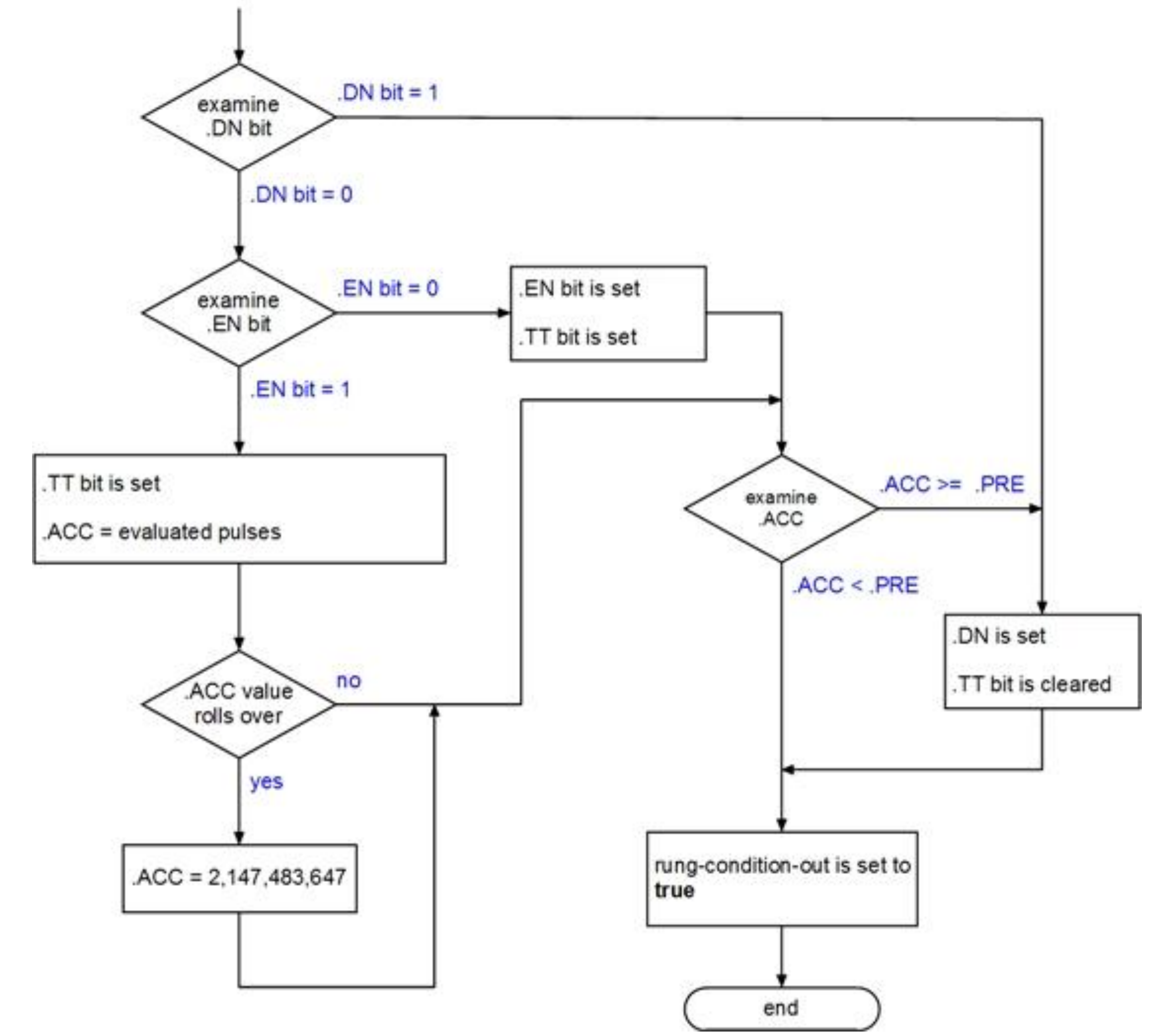
- the DOR instruction is disabled
- the .ACC ≥ .PRE

When the DOR instruction is disabled, the .ACC value is cleared.

DOL instruction is available only for ConveyLinx controller type.

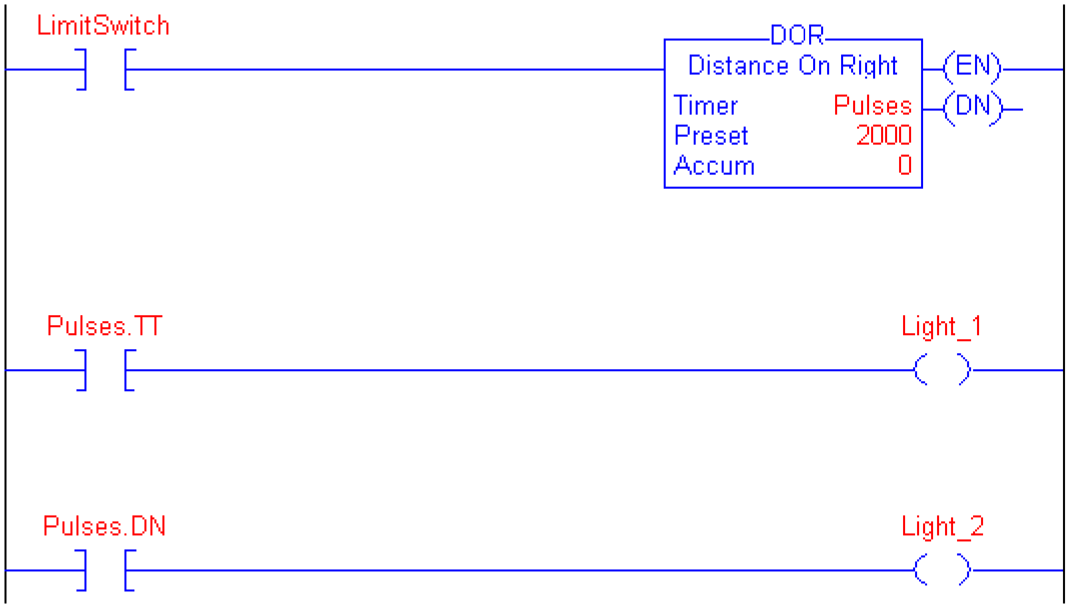
Execution:

Condition	Action
prescan	The .EN, .TT, and .DN bits are cleared. The .ACC value is cleared. The rung-condition-out is set to false.
rung-condition-in is false	The .EN, .TT, and .DN bits are cleared. The .ACC value is cleared. The rung-condition-out is set to false.
rung-condition-in is true	



Example:

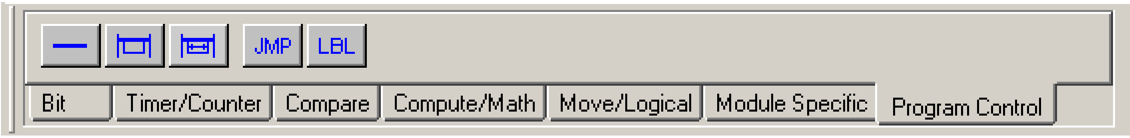
When LimitSwitch is set, Light_1 is on for 2000 pulses. When Pulses.ACC reaches 2000, Light_1 goes off and Light_2 goes on. Light_2 remains on until the DOR instruction is disabled. If LimitSwitch is cleared while DOR is counting, Light_1 goes off.



5.7 Program Control Instructions

Use the program control instructions to change the flow of logic.

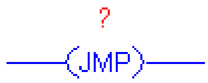
To enter a program control instruction use buttons form Program Control tab of Instruction Bar.



Instruction	Description
JMP	skip portions of ladder logic
LBL	the target of the JMP instruction

5.7.1 Jump (JMP)

The JMP instruction skips portions of ladder logic.



Operands:

Operand	Type	Format	Description
label name	LABEL	LABEL name	name for associated LBL instruction

Description:

When enabled, the JMP instruction skips to the referenced LBL instruction and the controller continues executing from there. When disabled, the JMP instruction does not affect ladder execution.

The JMP instruction can move ladder execution only forward.

Jumping to a label saves program scan time by omitting a logic segment until it's needed.

JMP conditions are scanned and it is not allowed to jump forward ladder logic. If it occurs, controller doesn't run and the next error reports:

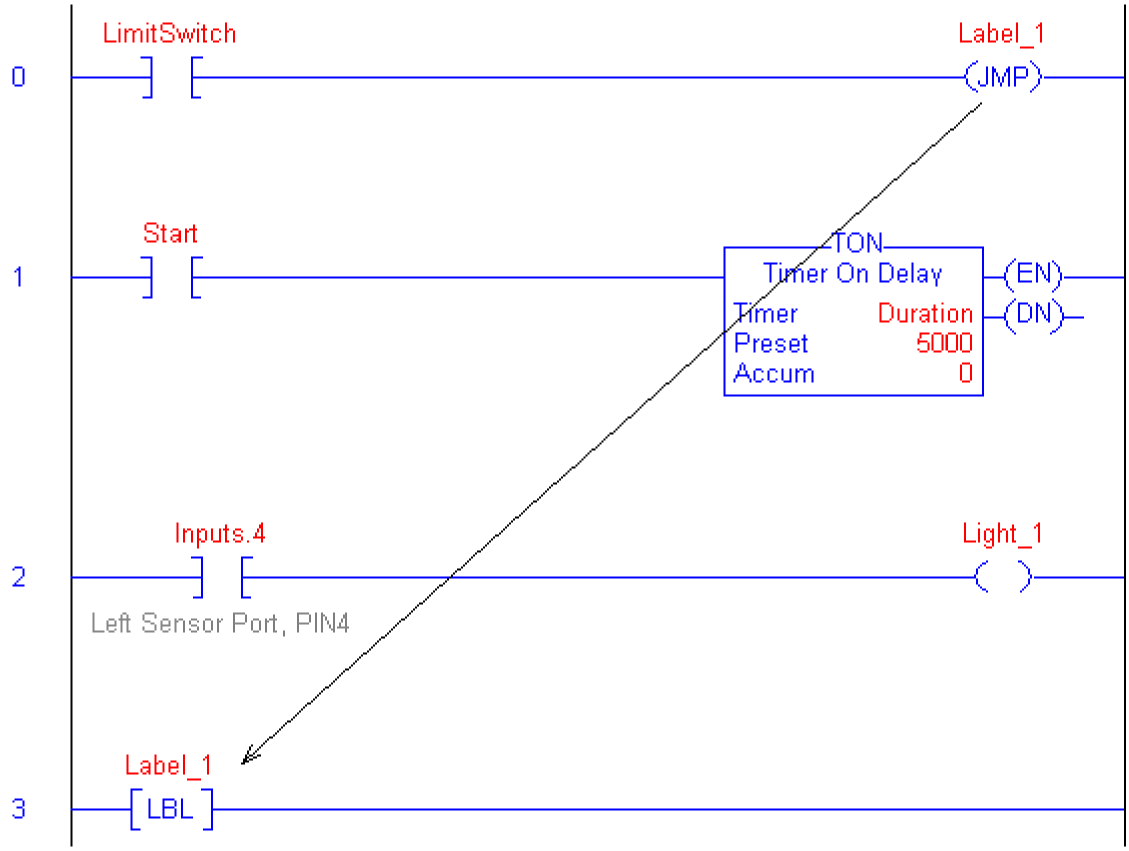
#11 – Wrong Jump

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	Ladder logic execution jumps to the rung that contains the LBL instruction with the referenced label name. The rung-condition-out is set to true.

Example:

When the JMP instruction is enabled, execution jumps over successive rungs of logic until it reaches the rung that started with LBL instruction with name Label_1.

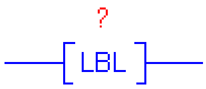


When the JMP instruction is executed, instructions between JMP and LBL instructions are not executed (in this example - instructions of Rung 1 and Rung 2).

In this example TON instruction will not be executed.

5.7.2 Label (LBL)

The LBL instruction is the target of the JMP instruction that has the same label name.



Operands:

Operand	Type	Format	Description
label name	LABEL	LABEL name	name for LBL instruction

Description:

The LBL instruction marks the rung where the logic will continue after execution of JMP instruction with the same name.

Make sure the LBL instruction is the first instruction on its rung.

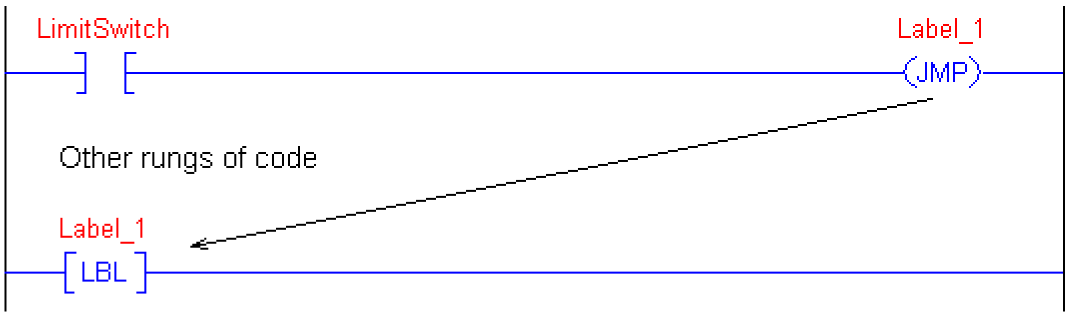
A label name must be unique within a routine. The name can contain letters, numbers, and underscores (_).

Execution:

The LBL instruction is a blank instruction. It is not executed.

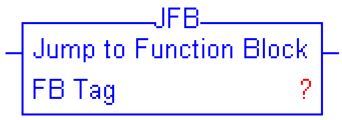
Example:

When the JMP instruction is enabled, “Other rungs of code” are jumped, and logic continues the rung that started with LBL instruction with name Label_1.



5.7.3 Jump to Function Block (JFB)

The JFB instruction calls function block.



Operands:

Operand	Type	Format	Description
FB Tag	FB type	tag	name of function block instance

Description:

When enabled, the JFB instruction executes function block routine.

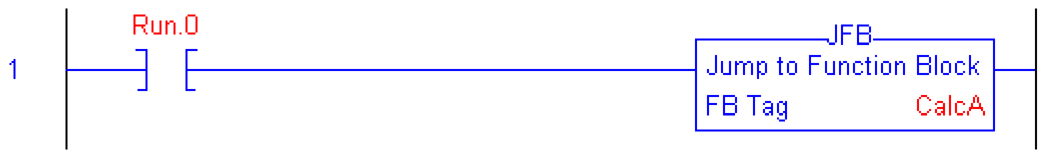
⚠ The JFB instruction is complete when all function block routine instructions are executed.

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	Executes all function block routine instruction. The rung-condition-out is set to true.

Example:

When Run.0 is set, routine of function block Calculate is executed, using CalcA instance data.



5.7.4 Return from Function Block (RFB)

The RFB instruction breaks the execution of current function block routine.

Operands:

The RFB instruction has no operands.

Description:

When enabled, the RFB instruction breaks the execution of current function block routine.

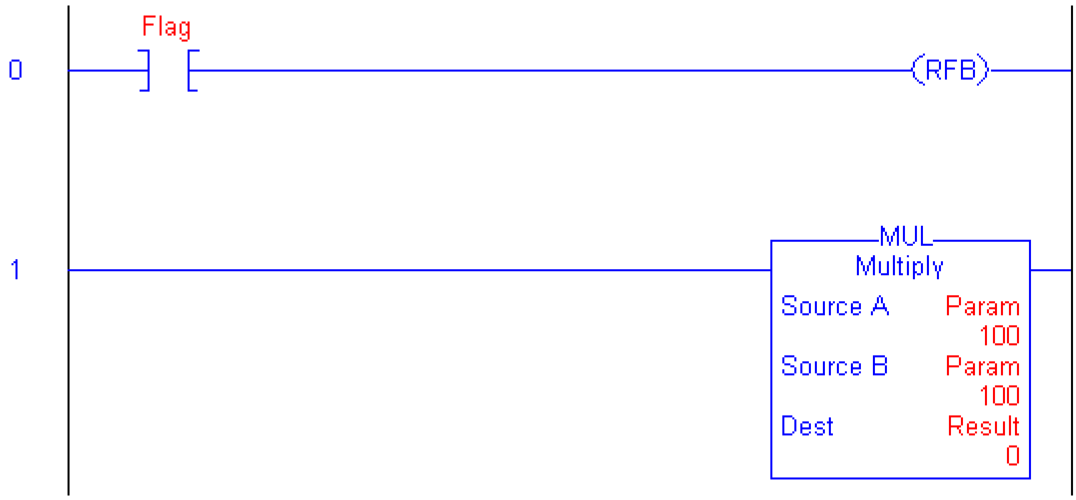
All instructions after RFB are not executed.

Execution:

Condition	Action
prescan	The rung-condition-out is set to false.
rung-condition-in is false	The rung-condition-out is set to false.
rung-condition-in is true	Breaks the execution of current function block routine. The rung-condition-out is set to true.

Example:

When Flag is set, all instructions after RFB are not executed (instruction MUL is not executed).



6.0 Program Structured Text

Structured text is a textual programming language that uses statements to define what to execute.

- Structured text is case sensitive.
- Use tabs and carriage returns (separate lines) to make your structured text easier to read. They have no effect on the execution of the structured text.

Structured text can contain these components:

<table>
<tr><th>Term</th><th colspan="2">Definition</th><th>Examples</th></tr>
<tr><td>Assignment</td><td colspan="2">Use an assignment statement to assign values to tags.
The “:=” operator is the assignment operator.
Terminate the assignment with a semi colon “;”.</td><td>value2 := value1;</td></tr>
<tr><td rowspan="5">Expression</td><td colspan="2">An expression is a part of a complete assignment or construct statement.
An expression evaluates to a numerical expression (number) or to a BOOL expression (true or false).
An expression contains:</td><td></td></tr>
<tr><td>Tag</td><td>A named area of the memory where data is stored (BOOL, SINT, INT, DINT).</td><td>value1</td></tr>
<tr><td>Immediate</td><td>A constant value.</td><td>4</td></tr>
<tr><td>Operator</td><td>A symbol or mnemonic that specifies an operation within an expression.</td><td>value1 + value2
value2 >= value1</td></tr>
<tr><td>Function</td><td>When executed, a function yields one value. Use parentheses to contain the operand of a function.
Functions can be used in expressions.</td><td>function(value1)</td></tr>
<tr><td>Function Block</td><td colspan="2">A function block call is a standalone statement and cannot be used in expressions.
A function block call uses parenthesis to contain its input or/and output parameters.
Depending on the function block type and call, there can be zero, one, or multiple parameters.
When executed, a function block yields one or more values that are part of a data structure.
Terminate the instruction with a semi colon “;”.</td><td>FB_instance();
FB_instance(In1 := value1);
FB_instance(In1 := value1, In2 := value2, Out => value3);</td></tr>
</table>

Term	Definition	Examples
Construct	A conditional statement used to trigger structured text code (other statements). Terminate the construct with a semi colon “;”.	IF...THEN CASE FOR...DO EXIT CONTINUE RETURN
Comment	Text that explains or clarifies what a section of structured text does. Use comments to make it easier to interpret the structured text. Comments do not affect the execution of the structured text. Comments can appear anywhere in structured text.	//comment (*start of comment . . . end of comment*) /*start of comment . . . end of comment*/

6.1 Assignment

Use an assignment to change the value stored within a tag. An assignment has this syntax:

tag := expression;

<table>
<tr><th>Component</th><th colspan="2">Description</th></tr>
<tr><td>tag</td><td colspan="2">Represents the tag that is getting the new value.
The tag must be a BOOL, SINT, INT, or DINT.</td></tr>
<tr><td>:=</td><td colspan="2">Is the assignment symbol.</td></tr>
<tr><td rowspan="6">expression</td><td colspan="2">Represents the new value to assign to the tag.</td></tr>
<tr><td>If tag is this data type</td><td>Use this type of expression</td></tr>
<tr><td>BOOL</td><td>BOOL expression</td></tr>
<tr><td>SINT</td><td rowspan="3">Numeric expression</td></tr>
<tr><td>INT</td></tr>
<tr><td>DINT</td></tr>
<tr><td>;</td><td colspan="2">Ends the assignment.</td></tr>
</table>

The tag retains the assigned value until another assignment changes the value.

The expression can be simple, such as an immediate value or another tag name, or the expression can be complex and include several operators and/or functions.

6.2 Expression

An expression is a tag name, equation, or comparison. To write an expression, use any of these elements:

- Tag name that stores the value (variable).
- Number that you enter directly into the expression (immediate value).
- Functions, such as: MOD.
- Operators, such as: +, -, <, >, AND, OR.

For more complex requirements, use parentheses to group expressions within expressions. This makes the whole expression easier to read and ensures that the expression executes in the desired sequence.

You may add user comments inline. Therefore, local language switching does not apply to your programming language.

In structured text, you use two types of expressions:

- BOOL expression – an expression that produces either the BOOL value of TRUE (1) or FALSE (0).

A BOOL expression uses BOOL tags, relational operators, and logical operators to compare values or check if conditions are true or false.

For example, tag1 > 65

A simple BOOL expression can be a single BOOL tag.

Typically, you use BOOL expressions to condition the execution of other logic.

- Numeric expression – an expression that calculates an integer value.

A numeric expression uses arithmetic operators, arithmetic functions, and bitwise operators.

For example, tag1 + 5

Often, you nest a numeric expression within a BOOL expression.

For example, (tag1 + 5) > 65

Use the following table to choose operators for your expressions:

If you want to	Then
Calculate an arithmetic value	Use Arithmetic Operators and Functions
Compare two values	Use Relational Operators
Check if conditions are true or false	Use Logical Operators
Compare the bits within values	Use Bitwise Logical Operators
Read/write Modbus Register	Use Modbus Register Operators

6.2.1 Arithmetic Operators and Functions

Arithmetic operators calculate new values. You can combine multiple operators and functions in arithmetic expressions.

To	Use this operator	Optimal data type
Add	+	DINT
Subtract	-	DINT
Multiply	*	DINT
Divide	/	DINT
Absolute value	ABS	DINT
Negate	NEG	DINT

Arithmetic functions perform math operations. Specify a constant, a non-boolean tag, or an numeric expression for the function.

For	Use this operator	Optimal data type
Modulo-divide	MOD(num_exp1, num_exp2)	DINT
Absolute value	ABS(num_exp)	DINT

For example:

Use this format	Example	
	For this situation	**You'd write**
value1 operator value2	If gain and gain_adj are DINT tags and your specification says: "Add 15 to gain and store the result in gain_adj."	gain_adj := gain + 15;
operator value1	If alarm and high_alarm are DINT tags and your specification says: "Negate high_alarm and store the result in alarm."	alarm:= -high_alarm;
function(numeric_expression)	If overtravel and overtravel_POS are DINT tags and your specification says: "Calculate the absolute value of overtravel and store the result in overtravel_POS."	overtravel_POS := ABS(overtravel);
value1 operator (function((value2+value3)/2)	If adjustment and position are DINT tags and sensor1 and sensor2 are REAL tags and your specification says: "Find the absolute value of the average of sensor1 and sensor2,	position := adjustment + ABS((sensor1 + sensor2)/2);

Use this format	Example
	add the adjustment, and store the result in position.”

6.2.2 Relational Operators

Relational operators compare two values or strings to provide a true or false result. The result of a relational operation is a BOOL value.

If the comparison is	The result is
true	1
false	0

Use these relational operators.

For this comparison:	Use this operator:	Optimal Data Type:
Equal	=	DINT
Less than	<	DINT
Less than or equal	<=	DINT
Greater than	>	DINT
Greater than or equal	>=	DINT
Not equal	<>	DINT

For example:

Use this format	Example	
	For this situation	**You'd write**
value1 operator value2	If temp is a DINT tag and your specification says: "If temp is less than 100·then …"	IF temp < 100 THEN ...
bool_tag := *bool_expression*	If count and length are DINT tags, done is a BOOL tag, and your specification says "If count is greater than or equal to length, you are done counting."	done := (count >= length);

6.2.3 Logical Operators

Logical operators let you check if multiple conditions are true or false. The result of a logical operation is a BOOL value:

If the comparison is	The result is
true	1
false	0

Use these logical operators:

For	Use this operator	Data Type
Logical AND	&, AND	BOOL
Logical OR	OR	BOOL
Logical exclusive OR	XOR	BOOL
Logical complement	NOT	BOOL

For example:

Use this format	Example	
	For this situation	**You'd write**
BOOLtag	If photoeye is a BOOL tag and your specification says: "If photoeye_1 is on then…"	IF photoeye THEN...
NOT *BOOLtag*	If photoeye is a BOOL tag and your specification says: "If photoeye is off then…"	IF NOT photoeye THEN...
expression1 & *expression2*	If photoeye is a BOOL tag, temp is a DINT tag, and your specification says: "If photoeye is on and temp is less than 100·then…".	IF photoeye & (temp < 100) THEN...
expression1 OR *expression2*	If photoeye is a BOOL tag, temp is a DINT tag, and your specification says: "If photoeye is on or temp is less than 100·then…".	IF photoeye OR (temp < 100) THEN...
expression1 XOR *expression2*	If photoeye1 and photoeye2 are BOOL tags and your specification says: "If: • photoeye1 is on while photoeye2 is off or • photoeye1 is off while photoeye2 is on	IF photoeye1 XOR photoeye2 THEN...

Use this format	Example	
	then…"	
BOOLtag := *expression1* & *expression2*	If photoeye1 and photoeye2 are BOOL tags, open is a BOOL tag, and your specification says: "If photoeye1 and photoeye2 are both on, set open to true".	open := photoeye1 & photoeye2;

6.2.4 Bitwise Operators

Bitwise operators manipulate the bits within a value based on two values.

Operator	Use this operator	Optimal Data Type
Bitwise AND	&, AND	DINT
Bitwise OR	OR	DINT
Bitwise exclusive OR	XOR	DINT
Bitwise complement	NOT	DINT
Bitwise clear	CLR	DINT
Shift left	<<	DINT
Shift right	>>	DINT

For example:

Use this format	**Example**	
	For this situation	**You'd write**
value1 operator value2	If input1, input2, and result1 are DINT tags and your specification says: "Calculate the bitwise result of input1 and input2. Store the result in result1."	result1 := input1 AND input2;
value1 << 2	If input1 and result1 are DINT tags and your specification says: "Shift left input1 two times and store the result in result1."	result1 := input1 << 2;

6.2.5 Modbus Register Operators

Modbus register operators allow read from or write to the controller's Modbus registers.

%R*reg_number*

Operands:

Operand	Type	Format	Description
reg_number	Modbus Register	immediate	Modbus register number. Must be from 1 to 512.

Description:

To read a Modbus register use the next syntax:

tag := %R*reg_number*;

The value of Modbus register is 2 byte. If *tag* type is SINT, only Low BYTE of the Modbus register is copied to *tag*.

To write into Modbus register use the next syntax:

%R*reg_number* := *tag*;

The value of Modbus register is 2 byte.

If *tag* type is DINT, only the Low WORD of *tag* value is copied to Modbus register.

For example:

Use this format	Example	
	For this situation	**You'd write**
tag := %Rreg_number	The value of Modbus register 110 will be put to Value.	Value := %R110;
%Rreg_number := tag	The value of Value will be put to Modbus register 110.	%R110 := Value;
tag := %Rreg_number1 + %Rreg_number2	The sum of Modbus registers 110 and 112 values will be put to Value.	Value := %R110 + %R112;

6.2.6 Order of Execution

The operations you write into an expression are performed in a prescribed order, not necessarily from left to right.

- Operations of equal order are performed from left to right.

- If an expression contains multiple operators or functions, group the conditions in parenthesis “( )”. This ensures the correct order of execution and makes it easier to read the expression.

Order	Operation
1	( )
2	function(…)
3	%R
4	NOT, NEG, ABS, CLR
5	*, /, MOD
6	+, -
7	<<, >>
8	<, <=, >, >=
9	=, <>
10	&, AND
11	XOR
12	OR

6.3 Constructs

Constructs can be programmed singly or nested within other constructs.

If you want to	Use this construct
Do something if or when specific conditions occur	IF...THEN
Select what to do based on a numerical value	CASE...OF
Do something a specific number of times before doing anything else	FOR...DO
Continue the loop	CONTINUE
Exit the loop	EXIT
Exit the function block	RETURN

6.3.1 IF...THEN

Use IF…THEN construct to do something if or when specific conditions occur.

```
IF bool_expression THEN
        <statement>;
END_IF;
```

Operands:

Operand	Type	Format	Description
bool_expression	BOOL	tag expression	BOOL tag or expression that evaluates to a BOOL value (BOOL expression)

Syntax:

```
IF bool_expression1 THEN
        <statement>; //Statements to execute when bool_expression1 is true
        ...
//Optional
ELSIF bool_expression2 THEN
        <statement>; //Statements to execute when bool_expression2 is true
        ...
//Optional
ELSE
        <statement>; //Statements to execute when both expressions are false
        ...
END_IF;
```

To use ELSIF or ELSE, follow these guidelines:

- To select from several possible groups of statements, add one or more ELSIF statements.
  - Each ELSIF represents an alternative path.
  - Specify as many ELSIF paths as you need.
  - The controller executes the first true IF or ELSIF and skips the rest of the ELSIFs and the ELSE.

- To do something when all of the IF or ELSIF conditions are false, add an ELSE statement.

This table summarizes combinations of IF, THEN, ELSIF, and ELSE.

If you want to	And	Then use this construct
Do something if or when conditions are true	Do nothing if conditions are false	IF…THEN
	Do something else if conditions are false	IF…THEN…ESLE
Choose from alternative statements (or groups of statements) based on input conditions	Do nothing if conditions are false	IF…THEN…ELSIF
	Assign default statements if all conditions are false	IF…THEN…ELSIF…ELSE

Example 1:

IF…THEN

If you want this	Enter this structured text

If rejects > 3 then
conveyor = off (0)
alarm = on (1)
	IF rejects > 3 THEN
	conveyor := 0;
	alarm := 1;
	END_IF;

Example 2:

IF…THEN…ELSE

If you want this	Enter this structured text

If conveyor direction contact = forward (1) then
light = off
Otherwise light = on
	IF conveyor_direction THEN
	light := 0;
	ELSE
	light := 1;
	END_IF;

Example 3:

IF…THEN…ELSIF

If you want this	Enter this structured text
If sugar low limit switch = low (on) and sugar high limit switch = not high (on) then inlet valve = open (on) Until sugar high limit switch = high (off)	IF Sugar.Low & Sugar.High THEN Sugar.Inlet := 1; ELSIF NOT(Sugar.High) THEN Sugar.Inlet := 0; END_IF;

Example 4:

IF…THEN…ELSIF…ELSE

If you want this	Enter this structured text

If tank temperature > 100
then pump = slow
If tank temperature > 200
then pump = fast
otherwise pump = off
	IF tank.temp > 200 THEN
	pump.fast :=1; pump.slow :=0; pump.off :=0;
	ELSIF tank.temp > 100 THEN
	pump.fast :=0; pump.slow :=1; pump.off :=0;
	ELSE
	pump.fast :=0; pump.slow :=0; pump.off :=1;
	END_IF;

6.3.2 CASE...OF

Use CASE…OF construct to select what to do based on a numerical value.

```
CASE numeric_expression OF
        selector1: <statement>;
        selectorN: <statement>;
ELSE
        <statement>;
END_CASE;
```

Operands:

Operand	Type	Format	Description
numeric_expression	SINT INT DINT	tag expression	tag or expression that evaluates to a number (numeric expression)
selector	SINT INT DINT	immediate	same type as numeric_expression

Syntax:

```
CASE numeric_expression OF
//specify as many alternative selector values (paths) as you need
selector1:
        <statement>; //statements to execute when numeric_expression = selector1
        …
selector2:
        <statement>; //statements to execute when numeric_expression = selector2
        …
selector3 :
        <statement>; //statements to execute when numeric_expression = selector3
        ...
optional
```

```
ELSE //statements to execute when numeric_expression ≠ any selector
        <statement>;
        ...
END_CASE;
```

The syntax for entering the selector values is:

When selector is:	Enter:
one value	*value*: *statement*
multiple, distinct values	*value1*, *value2*, *valueN* : *<statement>* Use a comma (,) to separate each value.
a range of values	*value1*..*valueN* : *<statement>* Use two periods (..) to identify the range.
distinct values plus a range of values	*valuea, valueb, value1*..*valueN* : *<statement>*

The CASE construct is similar to a switch statement in the C or C++ programming languages. However, with the CASE construct the controller executes *only* the statements that are associated with the *first matching* selector value. Execution *always breaks after the statements of that selector* and goes to the END_CASE statement.

Example:

If you want this	Enter this structured text
If recipe number = 1 then	CASE recipe_number OF
Ingredient A outlet 1 = open (1)	1: Ingredient_A.Outlet_1 :=1;
Ingredient B outlet 4 = open (1)	Ingredient_B.Outlet_4 :=1;
If recipe number = 2 or 3 then	2,3: Ingredient_A.Outlet_4 :=1;
Ingredient A outlet 4 = open (1)	Ingredient_B.Outlet_2 :=1;
Ingredient B outlet 2 = open (1)	4..7: Ingredient_A.Outlet_4 :=1;
If recipe number = 4, 5, 6, or 7 then	Ingredient_B.Outlet_2 :=1;
Ingredient A outlet 4 = open (1)	8,11..13: Ingredient_A.Outlet_1 :=1;
Ingredient B outlet 2 = open (1)	Ingredient_B.Outlet_4 :=1;
If recipe number = 8, 11, 12, or 13 then	ELSE
Ingredient A outlet 1 = open (1)	Ingredient_A.Outlet_1 :=0;

Ingredient B outlet 4 = open (1)	Ingredient_A.Outlet_4 :=0;
Otherwise all outlets = closed (0)	Ingredient_B.Outlet_2 :=0;
	Ingredient_B.Outlet_4 :=0;
	END_CASE;

6.3.3 FOR…DO

Use the FOR…DO loop to do something a specific number of times before doing anything else.

```
FOR count:= initial_value TO final_value BY increment DO
        <statement>;
END_FOR;
```

Operands:

Operand	Type	Format	Description
count	SINT INT DINT	tag	tag to store count position as the FOR…DO executes
initial_value	SINT INT DINT	tag expression immediate	must evaluate to a number specifies initial value for count
final_value	SINT INT DINT	tag expression immediate	specifies final value for count, which determines when to exit the loop
increment	SINT INT DINT	tag expression immediate	(optional) amount to increment count each time through the loop If you don't specify an increment, the count increments by 1.

Syntax:

```
FOR count := initial_value
        TO final_value
        //optional
        BY increment //If you don't specify an increment, the loop increments by 1.
DO
        <statement>;
        //optional
        IF bool_expression1 THEN
```

EXIT; //If there are conditions when you want to exit the loop early, use other statements, such as an IF...THEN construct, to condition an EXIT statement.

END_IF;

//optional

IF bool_expression2 THEN

CONTINUE; //If there are conditions when you want to continue the loop, use other statements, such as an IF...THEN construct, to condition a CONTINUE statement.

END_IF;

END_FOR;

Make sure that you *do not* iterate within the loop too many times in a single scan.

The controller *does not* execute any other statements in the routine until it completes the loop.

Consider using a different construct, such as IF...THEN.

Example 1:

If you want this	Enter this structured text

Clear bits 0 - 31 in an array of BOOLs:
1. Initialize the subscript tag to 0.
2. Clear array[subscript]. For example, when subscript = 5, clear array[5].
3. Add 1 to subscript.
4. If subscript is ≤ to 31, repeat 2. and 3.
Otherwise, stop.
	FOR subscript: = 0 TO 31 BY 1 DO
	array[subscript] := 0;
	END_FOR;

Example 2:

If you want this	Enter this structured text

Copy elements from one array to another until the position not exceeds the number of valid elements.
Both arrays are from DINT type and contain 10 elements.
1. Initialize the position tag to 0.
	FOR position := 0 TO 10 BY 1 DO
	IF position <= valid_count THEN
	Quantity[position] := Inventory[position];
	ELSIF
	EXIT;

2. If valid_count not exceeds current position the value of position copies from Inventory array to Quantity. Otherwise, stop.	END_IF; END_FOR;
3. Add 1 to position.	
4. If position is ≤ to 10, repeat 2 and 3. Otherwise, stop.	

6.3.4 RETURN

Use the RETURN statement if you want to exit the program directly.

```
RETURN;
```

Description:

RETURN statement exits the program directly, without executing any code.

RETURN statement may be used anywhere in program code.

Example:

If you want this	Enter this structured text

If rejects > 3 then
conveyor = off (0)
alarm = on (1)
return program
	IF rejects > 3 THEN
	conveyor := 0;
	alarm := 1;
	RETURN;
	END_IF;

6.4 Function Block

Function block statements consist of the mechanisms for invoking a function block and for returning control. Function block is invoked by a statement consisting of the name of the function block instance followed by a parenthesized list of input or/and output parameters assignment.

```
FB_instance(In1 := TRUE, In2 := 44, Out => bDone);
```

<table>
<tr><th>Component</th><th colspan="2">Description</th></tr>
<tr><td>FB_instance</td><td colspan="2">tag name of the function block instance</td></tr>
<tr><td rowspan="4">()</td><td colspan="2">Optional consist function block input or/and output parameters assignment.</td></tr>
<tr><td>Symbol</td><td>Description</td></tr>
<tr><td>:=</td><td>Assign tag, immediate or expression to input or in-out parameter.</td></tr>
<tr><td>=></td><td>Assign output or in-out parameter value to tag.</td></tr>
<tr><td>;</td><td colspan="2">Ends the function block call.</td></tr>
</table>

Description:

A function block call is a standalone statement and cannot be used in expressions.

A function block call uses parenthesis to contain its input or/and output parameters.

Depending on the function block type and call, there can be zero, one, or multiple parameters.

When executed, a function block yields one or more values that are part of a data structure.

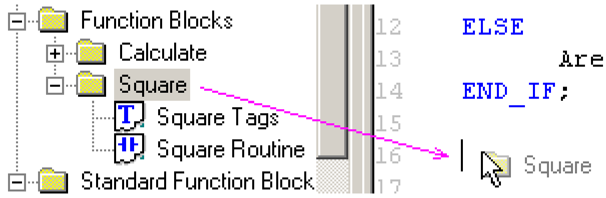
Terminate the instruction with a semi colon “;”.

The order in which parameters are listed in a function block invocation shall not be significant. It is not required that all parameters be assigned in every invocation of a function block. If a particular input parameter is not assigned a value in a function block invocation, the previously assigned value (or initial value, if no previous assignment has been made) shall apply.

There is two ways for entering function block call:

- by Drag&Drop operation.

Click on function block name in Project Bar and drag it to ST Routine View:



The cursor displays the place where instance will be inserted. Leave the mouse button. Create New Tag dialog appears. Write a desired name and select OK.

If tag squareN from type Square does not exist, it is created

- by typing the symbol "(" after existing function block tag name.

6.4.1 Standard Function Blocks

Standard function blocks are involved in ConveyLogix Programmer.

Unlike ladder logic, in structured text there is no rung-condition-in that trigger execution and rung-condition-out to state transition. For some standard function block (for example IEC_TON) input parameter EN is used for rung-condition-in and output parameter Q – for rung-condition-out.

IEC_TON

IEC_TON function block is a non-retentive timer that accumulates time when an instance is called and enabled (EN operand is true).

Syntax

Declaration of an instance of IEC_TON is performed in “Static” section of the function block tags (for example: myIEC_TON).

To call IEC_TON use the following syntax:

myIEC_TON(EN := <Operand>, PT := <Operand>, Q => <Operand>, ET => <Operand>)

Operands

Operand	Declaration	Type	Description
EN	Input	BOOL	Enable input
PT	Input	DINT	Duration of the on delay in milliseconds. The value of the PT parameter must be positive.
Q	Output	BOOL	Operand that is set when the time PT expires
ET	Output	DINT	Current time value

Description

IEC_TON instruction is used to delay the setting of the Q parameter for the programmed duration PT. The instruction starts when EN parameter changes from "0" to "1" (positive signal edge). The programmed time PT begins when the instruction starts. When the PT duration has expired, the Q parameter returns signal state "1". The parameter Q remains set as long as the start input is still "1". If the signal state of the EN parameter changes from "1" to "0", the parameter Q will be reset. The timer function is started again when a new rising edge is detected at the parameter EN.

The current time value is stored in the ET parameter. The time value starts at “0” and ends when the value of the time duration PT is reached. The ET parameter is reset as soon as the signal state of the parameter EN changes to "0".

Example:

Lets TimerA is a tag from standard function block IEC_TON type. When Switch is set, Light will be set after 1800 ms. Then when Switch is cleared, Light goes off.

TimerA(EN := Switch, PT := 1800, Q => Light);

There is second way to write this example – first assign inputs parameters, then call function block and after that assign outputs parameters.

Timer.EN := Switch;

TimerA.PT := 1800;

TimerA();

Light := TimerA.Q;

IEC_TOF

IEC_TOF function block is a non-retentive timer that accumulates time when an instance is called and enabled (EN operand is false).

Syntax

Declaration of an instance of IEC_TOF is performed in "Static" section of the function block tags (for example: myIEC_TOF).

To call IEC_TOF use the following syntax:

myIEC_TOF(EN := <Operand>, PT := <Operand>, Q => <Operand>, ET => <Operand>)

Operands

Operand	Declaration	Type	Description
EN	Input	BOOL	Enable input
PT	Input	DINT	Duration of the on delay in milliseconds. The value of the PT parameter must be positive.
Q	Output	BOOL	Operand that is reset when the time PT expires
ET	Output	DINT	Current time value

Description

IEC_TOF instruction is used to delay the resetting of the Q parameter for the programmed duration PT. The Q parameter is set when EN parameter changes from "0" to "1" (positive signal edge). When the signal state of the EN parameter changes back to "0", the programmed time PT starts. The parameter Q remains set as long as the time duration PT is running. When the time duration PT expires, the Q parameter is reset. If the signal state of the IN parameter changes to "1" before the time duration PT has expired, the timer is reset. The signal state of the Q parameter remains set to "1".

The current time value is stored in the ET parameter. The current time value starts at 0 and ends when the value of the time duration PT is reached. When the time duration PT expires, the ET parameter remains set to the current value until the EN parameter changes back to "1". If the EN parameter changes to "1" before the time duration PT has expired, the ET parameter is reset to the value 0.

Example:

Lets myTOF is a tag from standard function block IEC_TOF type. To call IEC_TON use the following syntax:

```
myTOF(EN := Tag_Start, PT := Tag_PresetTime,
        Q => Tag_Status, ET => Tag_ElapsedTime);
```

There is second way to write this example – first assign inputs parameters, then call function block and after that assign outputs parameters.

```
myTOF.EN := Tag_Start;

myTOF.PT := Tag_PresetTime;

myTOF ();

Tag_Status := myTOF.Q;

Tag_ElapsedTime := myTOF.ET;
```

With a change in the signal state of the "Tag_Start" operand from "0" to "1", the "Tag_Status" operand is set. When the signal state of the "Tag_Start" operand changes from "1" to "0", the time programmed for the “Tag_PresetTime” parameter is started. As long as the time is running, the "Tag_Status" operand remains set. When the time has expired, the Tag_Status operand is reset. The current time value is stored in the "Tag_ElapsedTime" operand.

IEC_RTO

IEC_RTO function block is a retentive timer that accumulates time when an instance is called and enabled (EN operand is true).

The syntax and operands of IEC_RTO are the same as IEC_TON function block. IEC_RTO accumulates the time until it is disabled.

IEC_DOL

IEC_DOL function block counts evaluated pulses of the left motor when an instance is called and enabled (EN operand is true).

The syntax and operands of IEC_DOL are the same as IEC_TON function block.

IEC_DOR

IEC_DOR function block counts evaluated pulses of the right motor when an instance is called and enabled (EN operand is true).

The syntax and operands of IEC_DOR are the same as IEC_TON function block.

IEC_CTU

IEC_CTU function block counts upward when an instance is called and enabled (CU operand is true).

Syntax

Declaration of an instance of IEC_CTU is performed in "Static" section of the function block tags (for example: myIEC_CTU).

To call IEC_CTU use the following syntax:

myIEC_CTU(CU := <Operand>, R := <Operand>, PV := <Operand>,

Q => <Operand>, CV => <Operand>)

Operands

Operand	Declaration	Type	Description
CU	Input	BOOL	Count up input
R	Input	BOOL	Reset input
PV	Input	DINT	Value at which the output Q is set.
Q	Output	BOOL	Counter status
CV	Output	DINT	Current counter value

Description

IEC_CTU instruction is used to increment the value at the CV parameter. When the signal state of the parameter CU changes from "0" to "1" (positive signal edge), the instruction is executed and the current counter value of the parameter CV is incremented by one. When the instruction is executed for the first time the current count of the CV parameter is set to zero. The counter value is increased each time a positive signal edge is detected, until it reaches the value of the parameter CV. When the CV value is reached, the signal state of the CU parameter no longer has an effect on the instruction.

The signal state of the Q parameter is determined by the PV parameter. When the current counter value is greater than or equal to the value of the PV parameter, the Q parameter is set to signal state "1". In all other cases, the signal state of the Q parameter is "0". You can also specify a constant for the PV parameter.

The value of the CV parameter is reset to zero when the signal state at the R parameter changes to "1". As long as the signal state of the R parameter is "1", the signal state of the CU parameter has no effect on the instruction.

Example:

Lets myCTU is a tag from standard function block IEC_CTU type. To call IEC_CTU use the following syntax:

myCTU(CU := Tag_Count, R := Tag_Reset, PV := Tag_PresetValue

Q => Tag_Status, CV => Tag_CounterValue)

There is second way to write this example – first assign inputs parameters, then call function block and after that assign outputs parameters.

myCTU.CU := Tag_Count;

myCTU.R := Tag_Reset;

myCTU.PV := Tag_PresetValue;

myCTU();

Tag_Status := myCTU.Q;

Tag_CounterValue := myCTU.CV;

When the signal state of the "Tag_Count" operand changes from "0" to "1", the IEC_CTU instruction executes and the current counter value of the "Tag_CounterValue" operand is incremented by one. With each additional positive signal edge, the counter value is incremented until it reaches the “Tag_PresetValue” value.

The "Tag_Status" output has signal state "1" as long as the current counter value is greater than or equal to the value of the "Tag_PresetValue" operand. In all other cases, the "Tag_Status" output has signal state "0". The current counter value is stored in the "Tag_CounterValue" operand.

IEC_CTD

IEC_CTD function block counts downward when an instance is called and enabled (CD operand is true).

Syntax

Declaration of an instance of IEC_CTD is performed in “Static” section of the function block tags (for example: myIEC_CTD).

To call IEC_CTD use the following syntax:

myIEC_CTD(CD := <Operand>, LD := <Operand>, PV := <Operand>,

Q => <Operand>, CV => <Operand>)

Operands

Operand	Declaration	Type	Description
CD	Input	BOOL	Count down input
LD	Input	BOOL	Load input
PV	Input	DINT	Value at which the output Q is set.
Q	Output	BOOL	Counter status
CV	Output	DINT	Current counter value

Description

IEC_CTU instruction is used to decrement the value at the parameter CV. When the signal state of the CD parameter changes from "0" to "1" (positive signal edge), the instruction is executed and the current counter value of the CV parameter is decremented by one. When the instruction is executed for the first time, the counter value of the CV parameter will be set to the value of the PV parameter. Each time a positive signal edge is detected, the counter is decremented until it reaches the zero. When the zero is reached, the signal state of the CD parameter no longer has an effect on the instruction.

If the current counter value is less than or equal to zero, the Q parameter is set to signal state "1". In all other cases, the signal state of the Q parameter is "0".

The value of the CV parameter is set to the value of the PV parameter when the signal state of the LD parameter changes to "1". As long as the signal state of the LD parameter is "1", the signal state of the CD parameter has no effect on the instruction.

Example:

Lets myCTD is a tag from standard function block IEC_CTD type. To call IEC_CTD use the following syntax:

```
myCTD(CD := Tag_Count, LD := Tag_Load, PV := Tag_PresetValue
        Q => Tag_Status, CV => Tag_CounterValue)
```

There is second way to write this example – first assign inputs parameters, then call function block and after that assign outputs parameters.

```
myCTD.CD := Tag_Count;
myCTD.LD := Tag_Load;
myCTD.PV := Tag_PresetValue;
myCTD();
Tag_Status := myCTD.Q;
```

Tag_CounterValue := myCTD.CV;

When the signal state of the "Tag_Count" changes from "0" to "1", the IEC_CTD instruction executes and the value of the "Tag_CounterValue" operand is decremented by one. With each additional positive signal edge, the counter value will be decremented until it reaches the zero.

The operand "Tag_Status" has the signal state "1" as long as the current counter value is less than or equal to zero. In all other cases, the "Tag_Status" output has signal state "0". The current counter value is stored in the "Tag_CounterValue" operand.

6.4.2 User-defined Function Blocks

User-defined function blocks are created by the user (see point 4.1).

Example:

Call CalcA instance of Calculate function block (the same example from point 4.5) on ST from your custom function block.

```
//Assign input parameters
CalcA.ParamB := 20;
CalcA.ParamC := 30;
/*Call FB*/
CalcA();
(*Assign output parameter*)
ResultCalc := CalcA.Sum;
```

This part of can be written also in one line:

```
CalcA(ParamB := 20, ParamC := 30, Sum => ResultCalc);
```

You can examine function block output parameters:

IF CalcA.Sum > 500 THEN … END_IF;

But you cannot assign a value to output parameter:

~~CalcA.Sum := 500~~;

Also you cannot use called function block static parameters:

IF ~~CalcA.Const > 500~~ THEN

    ~~CalcA.Const := 500~~;

END_IF;

6.5 Comments

To make your structured text easier to interpret, add comments to it.

- Comments let you use plain language to describe how your structured text works.
- Comments do not affect the execution of the structured text.

To add comments to your structured text:

To add a comment:	Use one of these formats:
on a single line	*//comment*
at the end of a line of structured text	(*comment*) /**comment**/
within a line of structured text	(*comment*) /**comment**/
that spans more than one line	(**start of comment . . . end of comment**) /**start of comment . . . end of comment**/

For example:

Format:	Example:

//comment	**At the beginning of a line**
	//Check conveyor belt direction
	IF conveyor_direction THEN...
	At the end of a line
	ELSE //If conveyor isn't moving, set alarm light
	light := 1;
	END_IF;

(*comment*)	Sugar.Inlet[:=]1;(*open the inlet*) IF Sugar.Low (*low level LS*)& Sugar.High (*high level LS*)THEN... (*Controls the speed of the recirculation pump. The speed depends on the temperature in the tank.*) IF tank.temp > 200 THEN...

Format:	Example:
*/*comment*/*	Sugar.Inlet := 0;/*close the inlet*/ IF bar_code = 65 /*A*/ THEN... /*Gets the number of elements in the Inventory array and stores the value in the Inventory_Items tag*/ END_IF;

7.0 Download a Project into Controller

To download the project into controller, select Controller/Logic / Download Program menu or click on  icon.

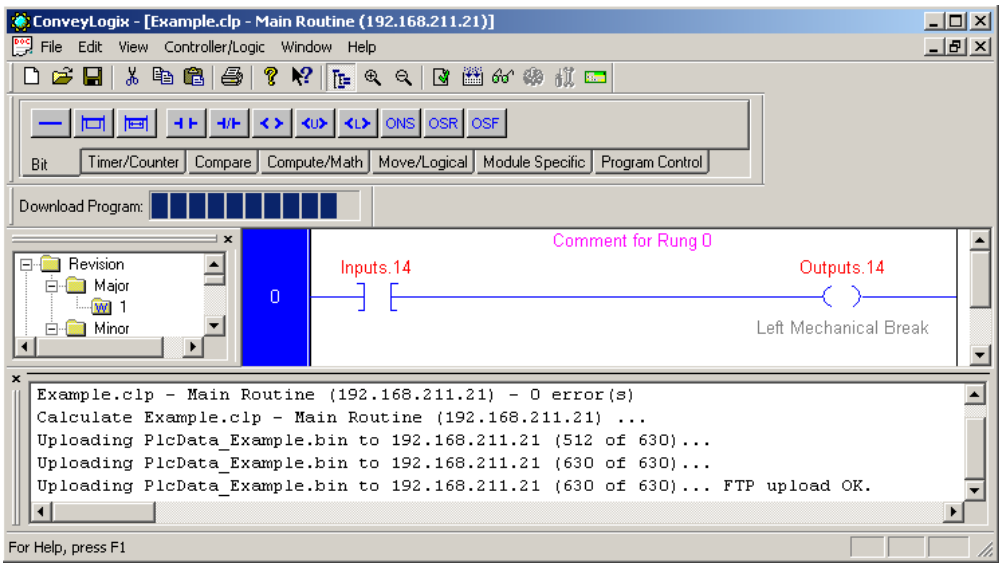
Download procedure requires to be fulfilled the next conditions:

- controller IP Address must be set;
- project must be saved on the disc;
- The controller must be ONLINE. If the controller is ConveyLinx, it have to be in PLC mode;
- No errors in the routine;
- All JMP/LBL instructions are correct.

Download procedure passes the next points:

- Verifies the routine;
- Calculates tags and instructions addresses;
- Verifies and calculates JMP/LBL conditions;
- Creates PLCDATA_xxx.bin file on the same folder, where is situated the project file. Xxx is the project name;
- Downloads PLCDATA_xxx.bin into the controller;
- Waits to give time the controller to start new program execution.

During Download procedure all features are disabled and progress bar is shown to indicate the process.



If some error occurs Download operation is interrupted. The result of Download operation is shown in Output bar.

8.0 Debug Mode

Debug mode is used to test and debug the ladder logic. ConveyLogix Progammer's Debug mode doesn't interfere with the controller's function.

In Debug mode ConveyLogix Progammer send requests for controller's header and for needed tags values.

In Debug mode tags values are displayed in green color.

8.1 Enter the Debug Mode

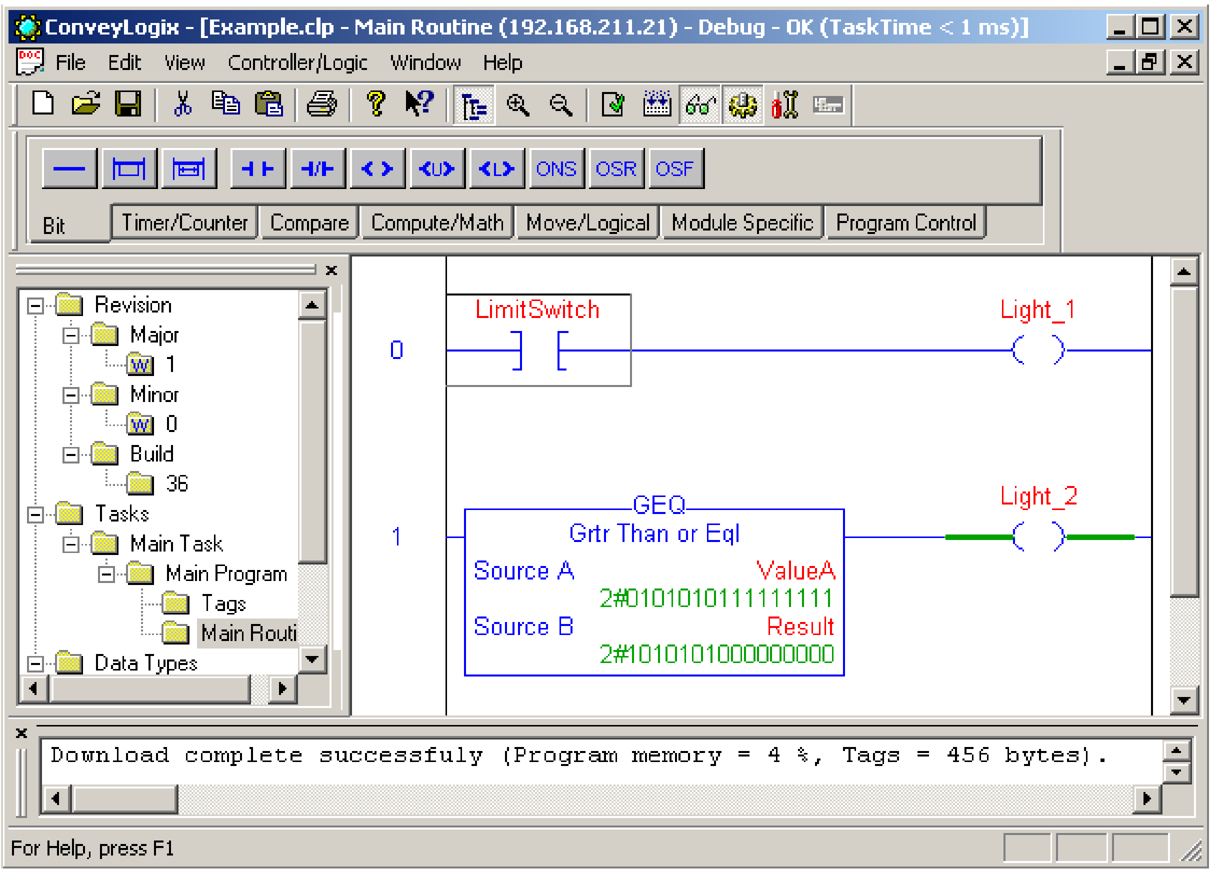
To enter the Debug mode, select Controller/Logic / Debug menu or click on  icon.

ConveyLogix Progammer checks the next conditions:

- The project is saved on the disc;
- The controller is ONLINE;
- There is a ladder program into controller;
- Ladder program into controller is the same as the project;
- Reading of controller service information is successful;
- The controller doesn't report critical errors.

If any of conditions are not fulfilled, the message is reported. The error descriptions are given on Appendix 1.

If Debug mode runs successful, debug icon is checked – .



On the Title bar is displayed Debug mode and time of ladder program execution.

8.2 Change the Controller Mode

The controller has two modes:

- Program Mode – the controller doesn’t execute ladder the logic program.
- Run Mode – the controller runs the ladder program.

You may see the controller’s mode only in ConveyLogix Progammer Debug mode.

When the controller is in Run Mode the Controller/Logic / Run Mode menu is checked and corresponding icon is chosen.

To change the controller’s mode to Program, select Controller/Logic / Program Mode menu or click on  icon.

When the controller is in Program Mode the Controller/Logic / Program Mode menu is checked and corresponding icon is chosen.

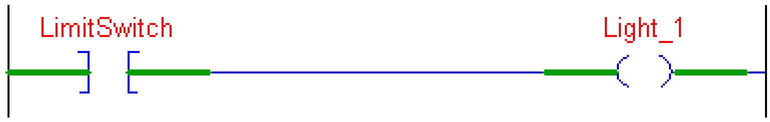
To change the controller's mode to Run, select Controller/Logic / Run Mode menu or click on

 icon.

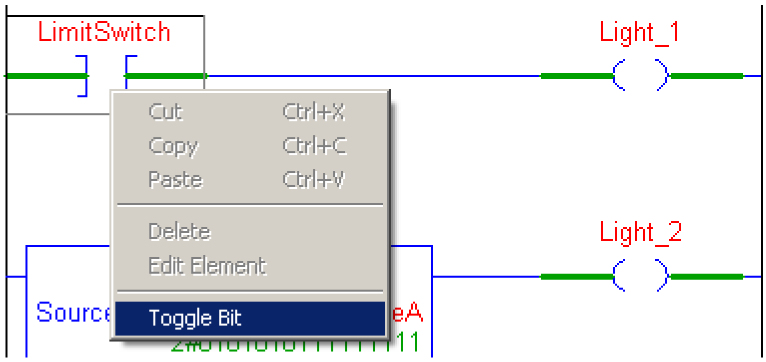
The controller's modes are mutually exclusive.

8.3 Watch and Change Boolean Tags

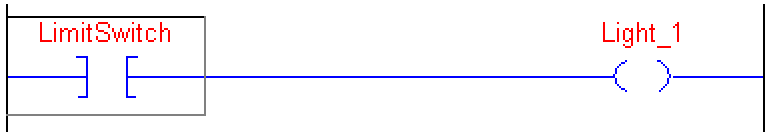
When the operand of boolean instructions is 1 (TRUE), rung-condition-in and rung-condition-out of the element are displayed in green colour.



To change the value of the operand of boolean instruction right-click on the element and select Toggle Bit menu (or press Ctrl + T keys).



If the value of the operand was 1 (TRUE), it is changed to 0 (FALSE).



Now in this example, LimitSwitch is cleared, and Light_1 is cleared.

If the value of the operand was 0 (FALSE), it is changed to 1 (TRUE).

You also may watch and change Boolean values in Tags view.

	Tag Name	Alias For	Base Tag	Data Type	Debug Value	Style
	⊞ ValueB			INT	2#1010010111111111	Binary
	Light_1			BOOL	0	Decimal
	LimitSwitch			BOOL	0	Decimal
	Light_2			BOOL	1	Decimal
*						

Current tags values are displayed in Debug Value column in green colour. To change Boolean value, click on Debug Value cell for the corresponding tag.

	Tag Name	Alias For	Base Tag	Data Type	Debug Value	Style
	⊞ ValueB			INT	2#1010010111111111	Binary
	Light_1			BOOL	0	Decimal
▶	LimitSwitch			BOOL	0	Decimal
	Light_2			BOOL	1	Decimal
*						

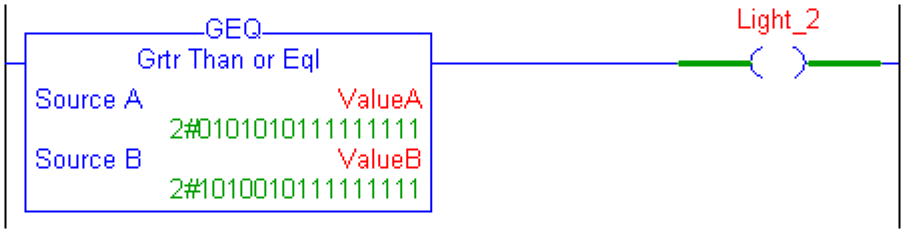
Type the new value (0 or 1) and click outside the rectangle or press Enter key.

	Tag Name	Alias For	Base Tag	Data Type	Debug Value	Style
	⊞ ValueB			INT	2#1010010111111111	Binary
	Light_1			BOOL	1	Decimal
▶	LimitSwitch			BOOL	1	Decimal
	Light_2			BOOL	1	Decimal
*						

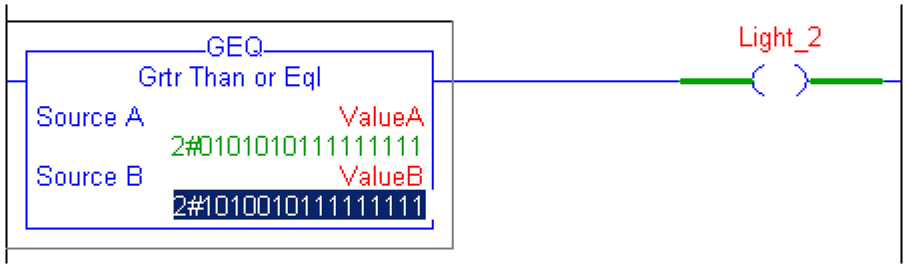
When you change a bit value on Tags View, changing is reflected all occurrences on on Ladder View. And backwards, when you change a bit value on Ladder View, changing is reflected on Tags View.

8.4 Watch and Change Non-boolean Tags

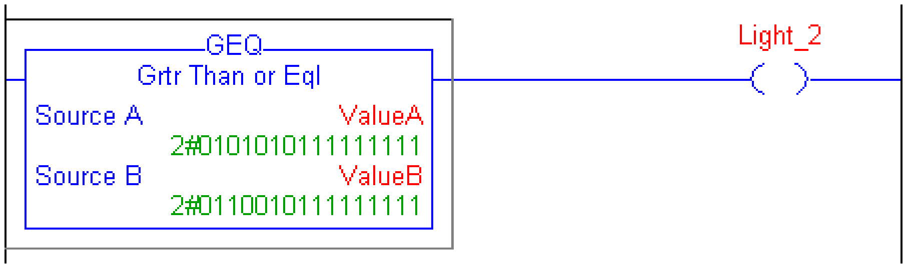
In Debug mode non-boolean operands are displayed below tag name in style, defined in Tags View.



To change the tag value, double-click on it. Edit box will appear.



Type the new value and click outside the edit box or press Enter key.



Now in this example, ValueA is not greater than or equal to ValueB, and Light_2 is cleared.

You also may watch and change non-boolean values in Tags View in the same way as boolean tags.

	Tag Name	Alias For	Base Tag	Data Type	Debug Value	Style
	⊞ ValueB			INT	2#0110010111111111	Binary
▶	Light_1			BOOL	0	Decimal
	LimitSwitch			BOOL	0	Decimal
	Light_2			BOOL	0	Decimal

To change non-boolean value, click on Debug Value cell for the corresponding tag.

	Tag Name	Alias For	Base Tag	Data Type	Debug Value	Style
▶	⊞ ValueB			INT	2#0110010111111111	Binary
	Light_1			BOOL	0	Decimal
	LimitSwitch			BOOL	0	Decimal
	Light_2			BOOL	0	Decimal

Type the new value and click outside the rectangle or press Enter key.

	Tag Name	Alias For	Base Tag	Data Type	Debug Value	Style
▶	⊞ ValueB			INT	2#0010010111111111	Binary
	Light_1			BOOL	0	Decimal
	LimitSwitch			BOOL	0	Decimal
	Light_2			BOOL	1	Decimal

When you change the tag value on Tags View, changing is reflected to all occurrences on Ladder View. And backwards, when you change the tag value on Ladder View, changing is reflected on Tags View.

8.5 Leave the Debug mode

To leave the Debug mode, select Controller/Logic / Stop Debugging menu or click on checked  icon.

Appendix A – Controller Tags

ConveyLinx Controller Tags

Controller Tag Name	Type	Modbus Register(s)
Input Controller Tags		
Inputs	DINT	See ConveyLinx Inputs Tag
FromUpstreamState	INT	134
FromUpstreamTracking	DINT	139, 140
FromDownstreamState	INT	232
FromPLC	DINT	266, 267
FromPLCArray	INT[16]	13200 – 13215
MyIPAddress	DINT	26, 27
ServoReadyLeft	BOOL	11 – bit 0
ServoReadyRight	BOOL	16 – bit 0
FirstLadderExec	BOOL	---
Output Controller Tags		
Outputs	DINT	See ConveyLinx Outputs Tag
ToUpstreamState	INT	116
ToDownstreamState	INT	196
ToDownstreamTracking	DINT	201, 202
ToPLC	DINT	268, 269
ToPLCArray	INT[16]	13100 - 13115
SensorPolarity	INT	34
SpeedLeftMTR	INT	40
SpeedRightMTR	INT	64
ServoControllLeft	INT	8
ServoControllRight	INT	13
ServoResetLeft	BOOL	9 – bit 0

Controller Tag Name	Type	Modbus Register(s)
ServoResetRight	BOOL	14 – bit 0
ServoCommandLeft	BOOL	9 – bit 1
ServoCommandRight	BOOL	14 – bit 1

ConveyLinx Inputs Tag

Tag Bit	Description	Modbus Register	Register Bit
0	PIN3, Left Sensor Port	35	0
1	PIN3, Left Control Port	35	1
2	PIN3, Right Sensor Port	35	2
3	PIN3, Right Control Port	35	3
4	PIN4, Left Sensor Port	35	4
5	PIN4, Left Control Port	35	5
6	PIN4, Right Sensor Port	35	6
7	PIN4, Right Control Port	35	7
16	Right Sensor Detect	36	0
17	Left Sensor Detect	36	1

ConveyLinx Outputs Tag

Tag Bit	Description	Modbus Register	Register Bit
0	Left MDR RUN	260	0
1	Left MDR Direction	260	8
2	Right MDR RUN	270	0
3	Right MDR Direction	270	8
4	Left Control Digital Out	37	1
5	Right Control Digital Out	37	3
6	Left MDR Dig. Mode Enable	60	15
7	Right MDR Dig. Mode Enable	84	15
8	Left MDR Low MOSFET 1	60	0

Tag Bit	Description	Modbus Register	Register Bit
9	Left MDR Low MOSFET 2	60	1
10	Left MDR Low MOSFET 3	60	2
11	Right MDR Low MOSFET 1	84	0
12	Right MDR Low MOSFET 2	84	1
13	Right MDR Low MOSFET 3	84	2
14	Left Mechanical Break	60	6
15	Right Mechanical Break	84	6
16	Left Mechanical Break Control	60	7
17	Right Mechanical Break Control	84	7

ConveyLinx-Ai and ConveyLinx-Ai2 Controller Tags

ConveyLinx-Ai and ConveyLinx-Ai2 Controller Tags are the same as ConveyLinx Controller Tags except Inputs, Outputs and the following two tags:

Controller Tag Name	Type	Modbus Register(s)
ServoPositionLeft	INT	62
ServoPositionRight	INT	86

ConveyLinx-Ai and ConveyLinx-Ai2 Inputs Tag

Tag Bit	Description	Modbus Register	Register Bit
0	Left Input, PIN2	35	0
2	Right Input, PIN2	35	2
4	Left Sensor Port, PIN4	35	4
6	Right Sensor Port, PIN4	35	6
16	Right Sensor Detect	36	0
17	Left Sensor Detect	36	1

ConveyLinx-Ai and ConveyLinx-Ai2 Outputs Tag

Tag Bit	Description	Modbus Register	Register Bit
0	Left MDR RUN	260	0
1	Left MDR Direction	260	8
2	Right MDR RUN	270	0
3	Right MDR Direction	270	8
4	Left Control Digital Out	37	0
5	Right Control Digital Out	37	1
6	Left MDR Dig. Mode Enable	60	15
7	Right MDR Dig. Mode Enable	84	15
8	Left MDR Low MOSFET 1	60	0
9	Left MDR Low MOSFET 2	60	1
10	Left MDR Low MOSFET 3	60	2
11	Right MDR Low MOSFET 1	84	0
12	Right MDR Low MOSFET 2	84	1
13	Right MDR Low MOSFET 3	84	2
18	Left Set Pin2 As Output	37	5
19	Left Set Pin2 As Output	37	6

ConveyNet I/P (CNIP) Controller Tags

Controller Tag Name	Type	Modbus Register(s)
Input Controller Tags		
Inputs	DINT	Physical Digital Inputs
FromUpstreamState	INT	134
FromUpstreamTracking	DINT	139, 140
FromDownstreamState	INT	232
FromPLC	DINT	266, 267
MyIPAddress	DINT	26, 27

Controller Tag Name	Type	Modbus Register(s)
RS485 InData	INT[4]	40, 41, 42, 43
RS485 Errors	INT	79
FirstLadderExec	BOOL	---
Output Controller Tags		
Outputs	DINT	Physical Digital Outputs
ToUpstreamState	INT	116
ToDownstreamState	INT	196
ToDownstreamTracking	DINT	201, 202
ToPLC	DINT	268, 269
RS485 OutData	INT[4]	50, 51, 52, 53
RS485 Default	INT[4]	60, 61, 62, 63
SlaveID	INT	70
StartRead	INT	71
NumToRead	INT	72
Start Write	INT	73
NumToWrite	INT	74
Baudrate	INT	75
RS485 Setings	INT	76
Scanrate	INT	77
RS485 Timeout	INT	78

Appendix B – Data Type Conversion

Data conversions occur when you mix data types for the parameters within one instruction.

Instructions execute faster and require less memory if all the operands of the instruction use:

- The same data type.
- An optimal data type:

– In the "Operands" section of each instruction in this manual, a **bold** data type indicates an optimal data type.

– The DINT data type is typically the optimal data types.

If you mix data types and use tags that are not the optimal data type, the controller converts the data according to these rules

- If any of the operands is not a DINT value, then input operands convert to DINT.
- After instruction execution, the result (a DINT value) converts to the destination data type, if necessary.

You cannot specify a BOOL tag in an instruction that operates on integer data types.

Because the conversion of data takes additional time and memory, you can increase the efficiency of your programs by:

- Using the same data type throughout the instruction.
- Minimizing the use of the SINT or INT data types.

In other words, use all DINT tags, along with immediate values, in your instructions.

The following sections explain how the data is converted when you use SINT or INT tags or when you mix data types.

SINT or INT to DINT

For those instructions that convert SINT or INT values to DINT values, the

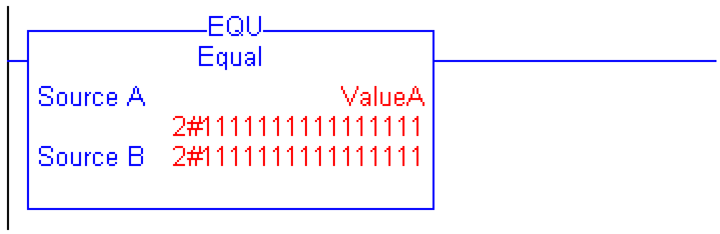
"Operands" sections in this manual identify the conversion method.

Conversion Method	Converts Data By Placing
Sign-extension	the value of the left-most bit (the sign of the value) into each bit position to the left of the existing bits until there are 32 bits.
Zero-fill	zeroes to the left of the existing bits until there are 32 bits.

The following example shows the results of converting a value using sign-extension and zero-fill.

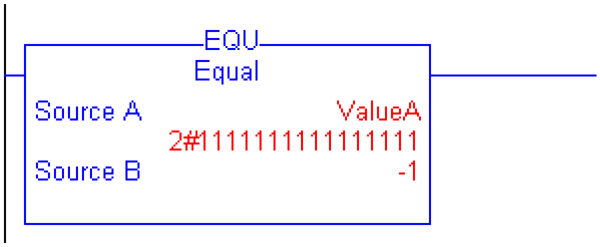
Value	2#1111_1111_1111_1111	(-1)
Converts by sign-extension	2#1111_1111_1111_1111_1111_1111_1111_1111	(-1)
Converts by zero-fill	2#0000_0000_0000_0000_1111_1111_1111_1111	(65535)

Because immediate values are always zero-filled, the conversion of a SINT or INT value may produce unexpected results. In the following example, the comparison is false because Source A, an INT, converts by sign-extension; while Source B, an immediate value, is zero-filled.

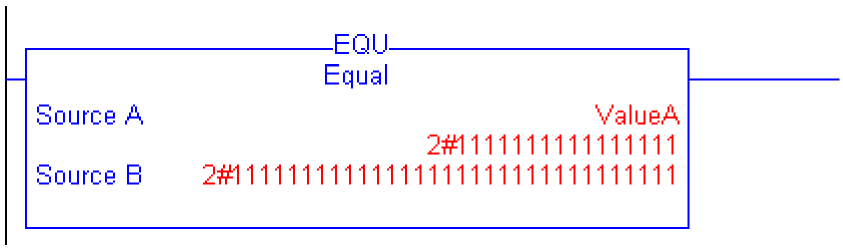


If you use a SINT or INT tag and an immediate value in an instruction that converts data by sign-extension, use one of these methods to handle immediate values:

- Specify any immediate value in the decimal radix.



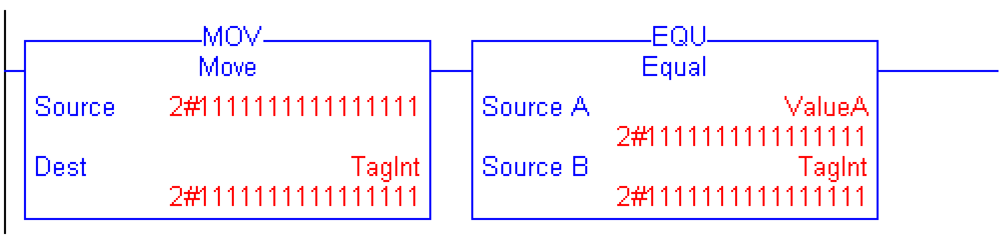
- If you are entering the value in a radix other than decimal, specify all 32 bits of the immediate value. To do so, enter the value of the left-most bit into each bit position to its left until there are 32 bits.



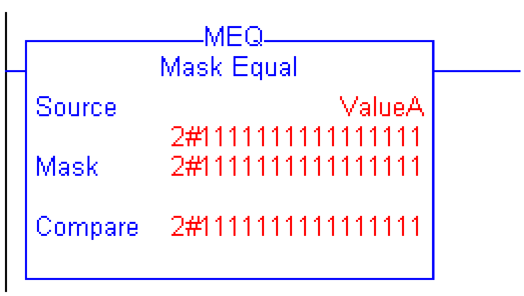
- Create a tag for each operand and use the same data type throughout the instruction. To assign a constant value, either:

  – Enter it into one of the tags.

  – Add a MOV instruction that moves the value into one of the tags.



- Use a MEQ instruction to check only the required bits.



DINT to SINT or INT

To convert a DINT value to a SINT or INT value, the controller truncates the upper portion of the DINT, if necessary. The following example shows the result of a DINT to SINT or INT conversion.

DINT Value	Converts To This Smaller Value	
16#00010081 (65665)	INT	16#0081 (129)
	SINT	16#81 (-127)

Appendix C – Errors description

Critical errors description

Error #	Description	Type
1	Type is not ConveyLinx or ConveyNet	Header
2	PLC program size is greater then PLC file	Header
3	Wrong Ladder Program size	Header
4	"?"	Header
5	Wrong Tags size	Header
6	"?"	Header
7	Ladder Program Start, Ladder Program End or Tags Start in not a DWORD address	Header
8	Allocating RAM for Tags Error	Header
9	Wrong Non Volatile Tags size	Header
10	Non Volatile Tags size is greater then 96 bytes	Header
100	Connection Tags Error	Header
1	First instruction is not RUNG or missing RUNG or RND	Prescan
2	Invalid Instruction Code	Prescan
3	BST number is different then BND number in one Rung	Prescan
4	BST number is different then NXB number in one Rung	Prescan
5	Too low stack for BST/BND instructions	Prescan
6	Bit Operand exceed 31	Prescan

7	Bit Operand Address >= Tags Size	Prescan
8	Timer Operand Address >= Tags Size	Prescan
9	Operand Address >= Tags Size	Prescan
10	Operand Address must be Tag Address	Prescan
11	Wrong JMP or JSR instructions	Prescan
12	Wrong MCR (must be even count)	Prescan
13	Ladder Program length error or missing two DWORDs after END	Prescan
14	Missing END of Ladder Program	Prescan
15	Missing RUNG or RND (must be equal)	Prescan
16	LBL is not first instruction of Rung	Prescan
17	Operand Address is not aligned to WORD/DWORD	Prescan
18	Wrong Operand Type (must be 0, 1, 2, 4 or 8)	Prescan
20	Subroutine parameters exceed 31	Prescan
21	Wrong Address of JSR or FOR instructions	Prescan
22	SBR must be first instruction in Rung	Prescan
23	JSR parameters (inputs and outputs) are different	Prescan
24	SBR parameters must be Tags	Prescan
25	Only one SBR must be in routine	Prescan
26	Each routine must finish with RET, RND or END	Prescan
27	Shouldn't have SBR in Main routine	Prescan
28	Before FOR(code 69) must be FOR(code 63) init	Prescan
29	Routine address in FOR must start first Rung	Prescan
30	FOR parameters must be Zero (4 DWORDs)	Prescan

31	BRK or RET instructions can't be use in Main routine	Prescan
32	Order Type (0, 1, 2) exceed 2	Prescan
33	In SWP if Source Operand is DWORD then Dest Operand must be DWORD	Prescan
34	Wrong Operand Type in SWP	Prescan

Runtime errors description

Error #	Description	Type
100	The End of Stack	Runtime
101	The numbers of JSR out parameters is different then in parameters	Runtime
102	FOR instruction Step Size is Zero	Runtime
103	Divide by Zero	Runtime
111	Incorrect Instruction - Online	Runtime

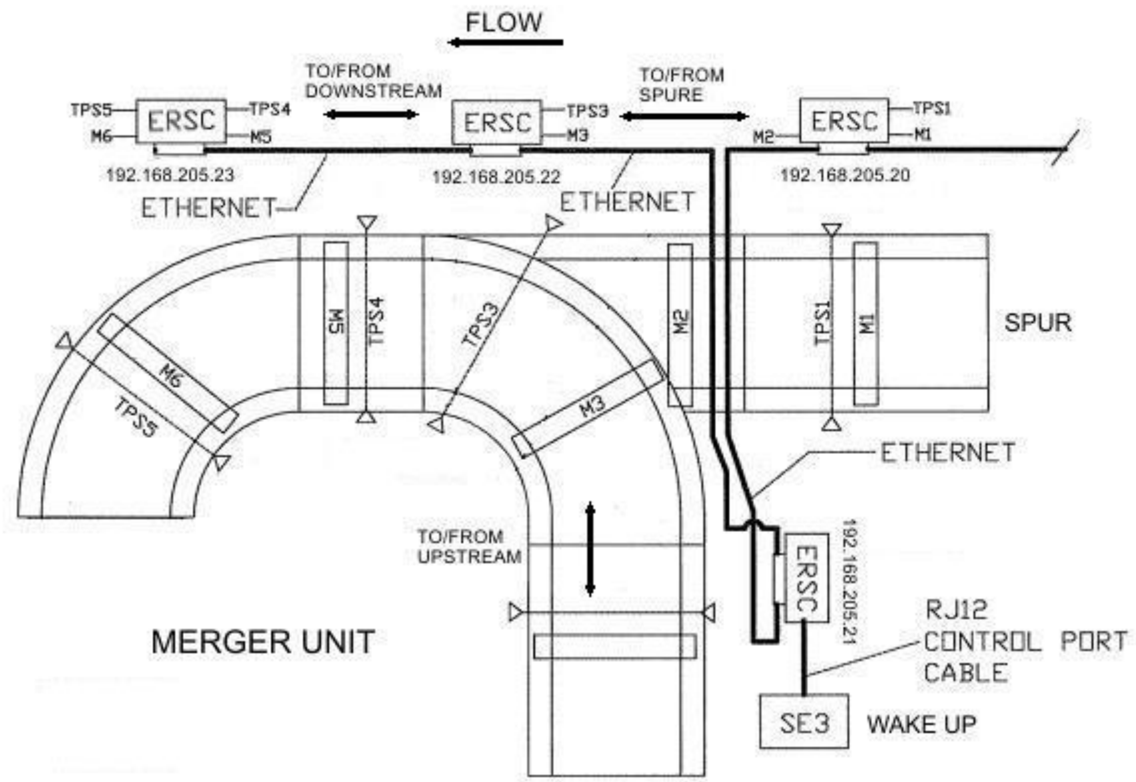
Appendix D – Module-Defined Structures

Zone Structure

Mnemonic	Data Type	Description
.NU1	SINT	Not used.
.NU2	SINT	Not used.
.State	SINT	
.ReverseState	SINT	
.NU3	SINT	Not used.
.NU4	SINT	Not used.
.Sensors	SINT	
.Motors	SINT	
.ZoneTracking	DINT	
.ToNextTracking	DINT	

Appendix E – Merger Unit Example

In this example is shown how to make a Merger Unit on picture below, using four ConveyLinx modules in 192.168.205.XX subnet.



Step 1

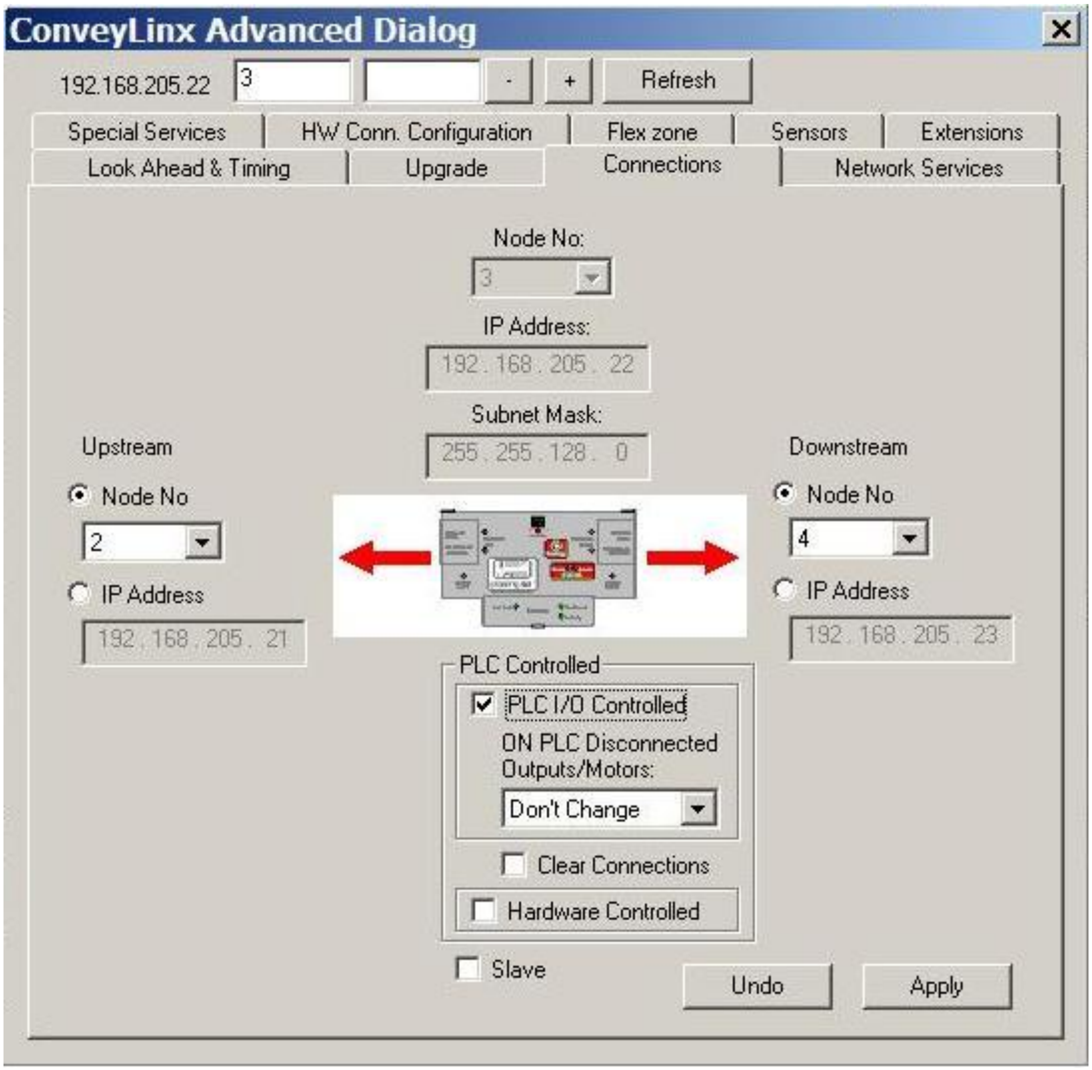
Wire the ConveyLinx modules how is shown on the picture. Press Install button of the first module (marked with 192.168.205.20 IP Address) and hold it pressed about 20 seconds. Install procedure starts. When the install procedure is finished the ConveyLinx modules will be with IP Addresses from 192.168.205.20 to 192.168.205.23.

Step 2

Put the ConveyLinx modules to corresponding mode depending of their purpose.

IP Address	Purpose	Mode
192.168.205.20	Spur control	ZPA mode
192.168.205.21	Upstream to Merge zone control	ZPA mode
192.168.205.22	Controls the Merge zone	PLC I/O Controlled mode
192.168.205.23	Downstream to Merge zone control	ZPA mode

Use EasyRoll, "Advanced Dialog" (F2)/"Connection" Tab to remove connection from 192.168.205.20 to 192.168.205.21 and vice versa.
Again use EasyRoll, "Connection" Tab to put 192.168.205.22 in PLC I/O Controlled mode, but LEAVE CONNECTIONS to Upstream and Downstream module.



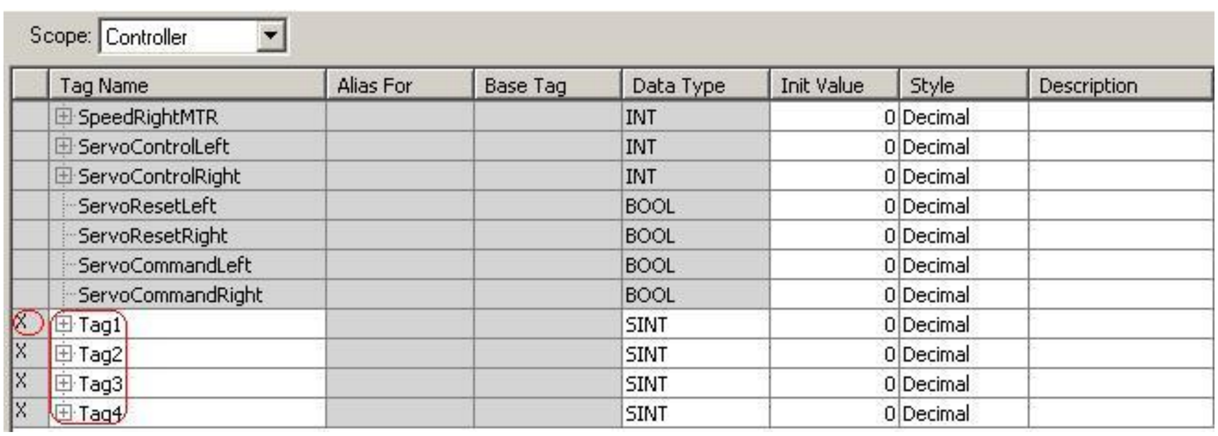
Because you left the connections to Upstream and Downstream modules, in your PLC program you may use the following Controller tags for:

Controller Tag	Purpose	
ToUpstreamState	Automatically propagated over connection to Upstream module. Use the next states to control Upstream module.	
	Value	**State**

	1 EMPTY 2 SENDING/ACCEPTING 4 FULL_RUNNING 5 FULL_STOPPED 6 BUSY
ToDownstreamState	Automatically propagated over connection to Downstream module. Same states values as above.
ToDownstreamTracking	Automatically propagated over connection to Downstream module.

Step 3

To communicate with other modules (different from Upstream and Downstream) you may use four special purpose tags in the Controller Tags. By default they are named Tag1, Tag2, Tag3, Tag 4, but you may change their names and data type.

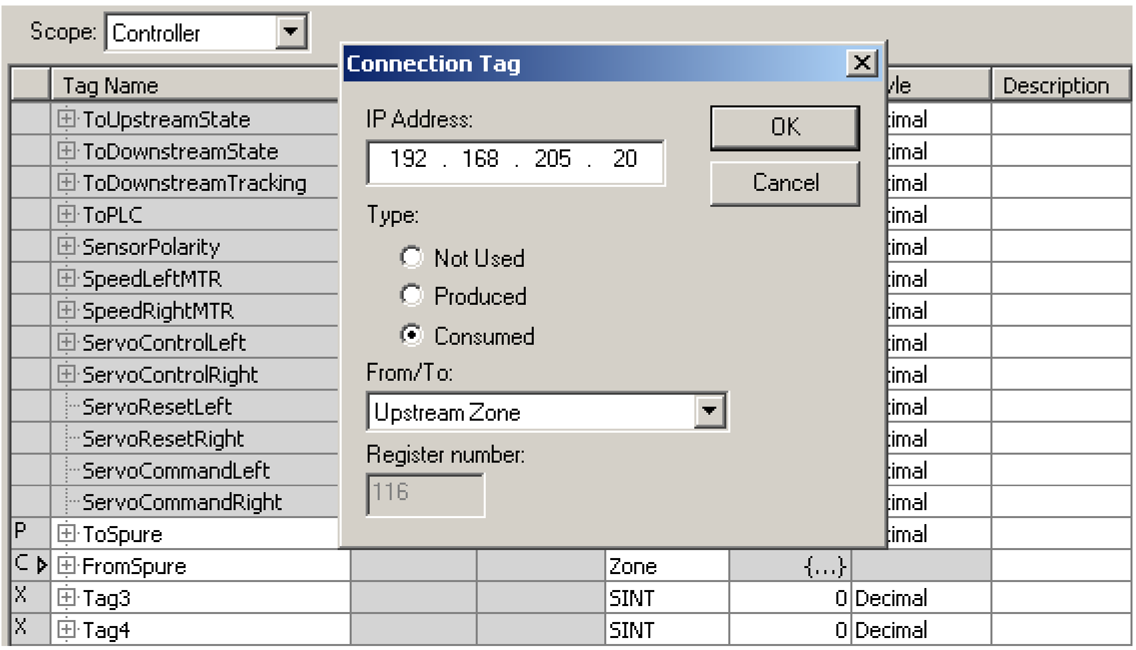


Scope: Controller

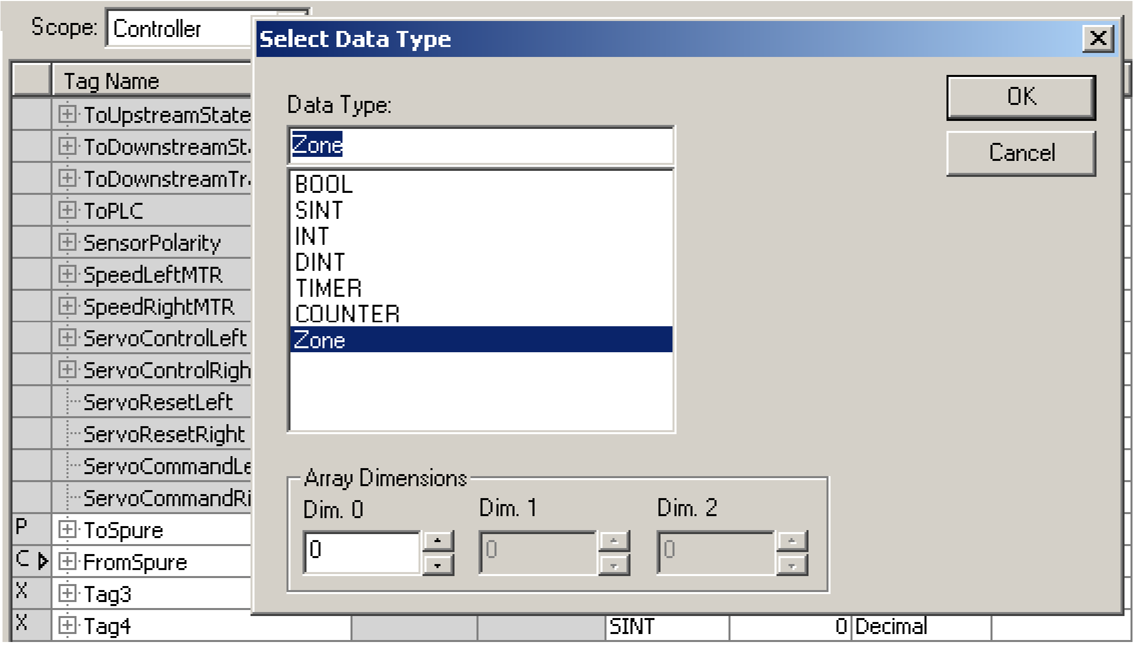
	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	SpeedRightMTR			INT	0	Decimal	
	ServoControlLeft			INT	0	Decimal	
	ServoControlRight			INT	0	Decimal	
	ServoResetLeft			BOOL	0	Decimal	
	ServoResetRight			BOOL	0	Decimal	
	ServoCommandLeft			BOOL	0	Decimal	
	ServoCommandRight			BOOL	0	Decimal	
X	Tag1			SINT	0	Decimal	
X	Tag2			SINT	0	Decimal	
X	Tag3			SINT	0	Decimal	
X	Tag4			SINT	0	Decimal	

To configure communication properties of these tags, click with mouse on the left most box (where X shows unused, C shows Consumed tag and P shows Produced tag).

For receiving data from the Spur module rename “Tag2” to “FromSpur” and configured it as consumed from 192.168.205.20. Select From UpstreamZone to receive data from the spur’s Upstream zone (as this module have only one zone, which is always upstream).



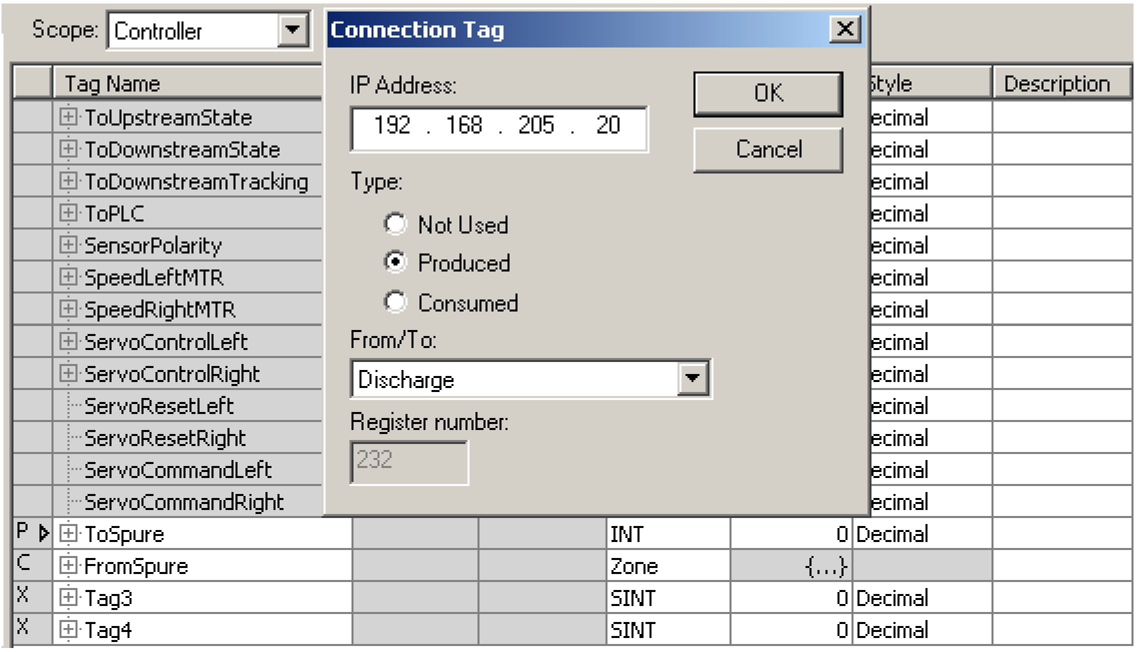
From the Spur module you need to receive both the state of the zone and the tracking. To do this you’ll have to change the Data Type of this tag to “Zone” (Module-Defined structure data type).



To control the Spur, rename "Tag1" to "ToSpur", Configure it as Produced to 192.168.205.20.

As you want to control the Discharge side of this module, select "To Discharge".

Leave data type of this tag SINT or INT.



Few details on the example:

All sensor and control port inputs are packed in controller tag Inputs. You may see description for each bit in Description field.

You may use SensorPolarity tag to inverse polarity of each Sensor/Control input.

Setting ON on any of SensorPolarity bit inverts the appropriate Input bit.

In this example on Merge zone is used only one sensor, attached to right sensor port.

It’s with a retro reflector, so it is needed to inverse Right Sensor Pin4 bit. Sensors also have ON on sensor error pin when there is no error, so it is also needed to invert Right Sensor Pin3.

Using SensorPolarity tag helps you in 2 ways:

- You may use positive logic in your program (ON when there is product on the sensor and OFF when there is no product, ON when there is gain error and OFF when there is no error).
- LEDs on the module will show the correct state – Green ON when there is product, Red ON when there is gain error.

You may see sensor polarity change in rung 0.

On the Merge zone it is used only one sensor and one MDR (connected to the right motor port).

You may add second motor control and Jam sensor logic.

There is no JAM or error control logic in the example.

Take special care on tracking manipulation. You should take tracking from Spur/Upstream on raising edge of the Merger sensor and place it in an internal tag.

You should prepare tracking for downstream module at the time you report to it that you are in the EMPTY/SENDING state. At the same time you should clear your internal tag to avoid tracking duplication if somebody throw product on the merge zone.

In the FromUpstreamState/FromDownstreamState tags you should always mask out the highest 8 bits (they are used in bi-directional operation and are not part of the tracking). In this example it is done by simply copying these tags in SINT tags.

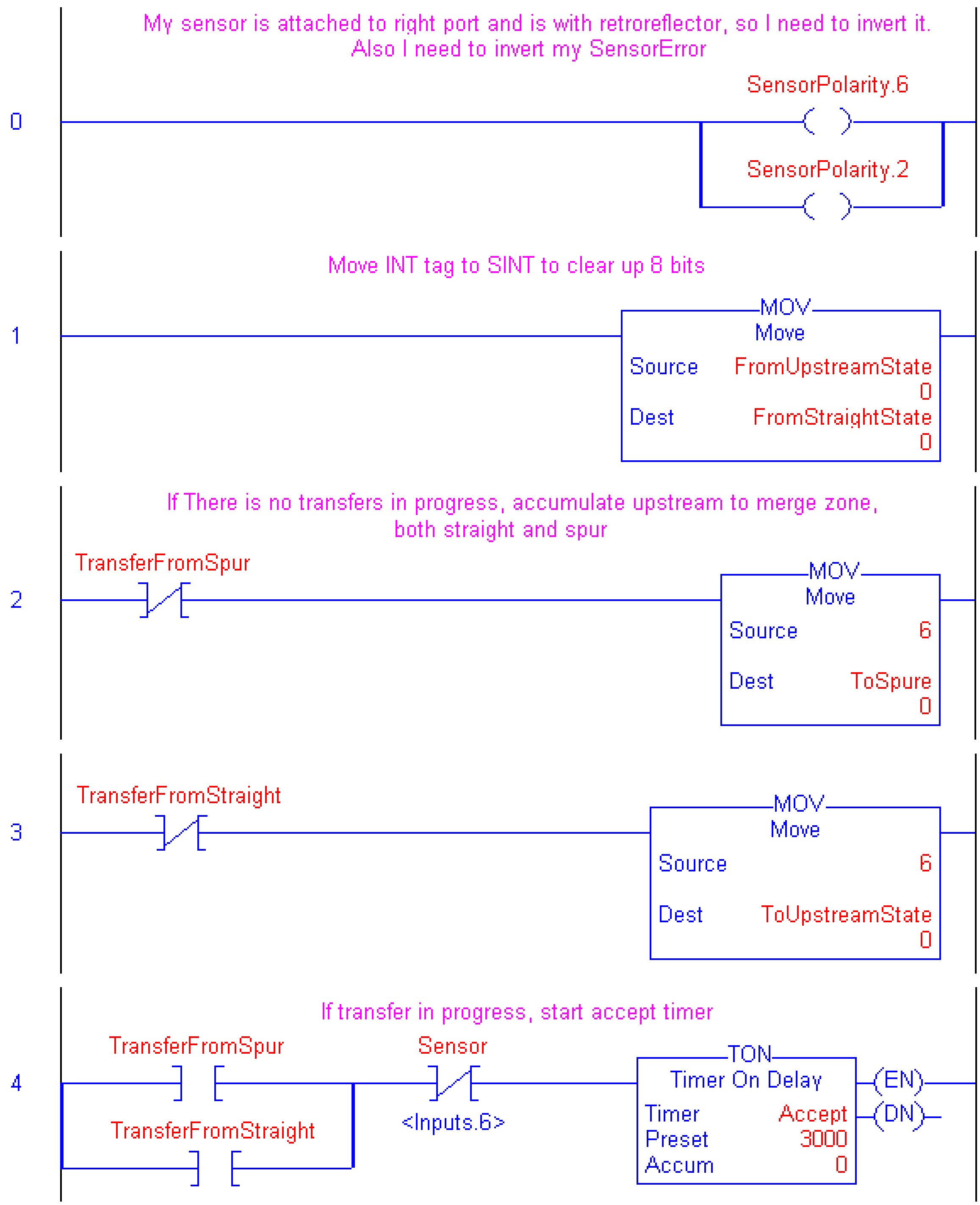
Main Program Tags

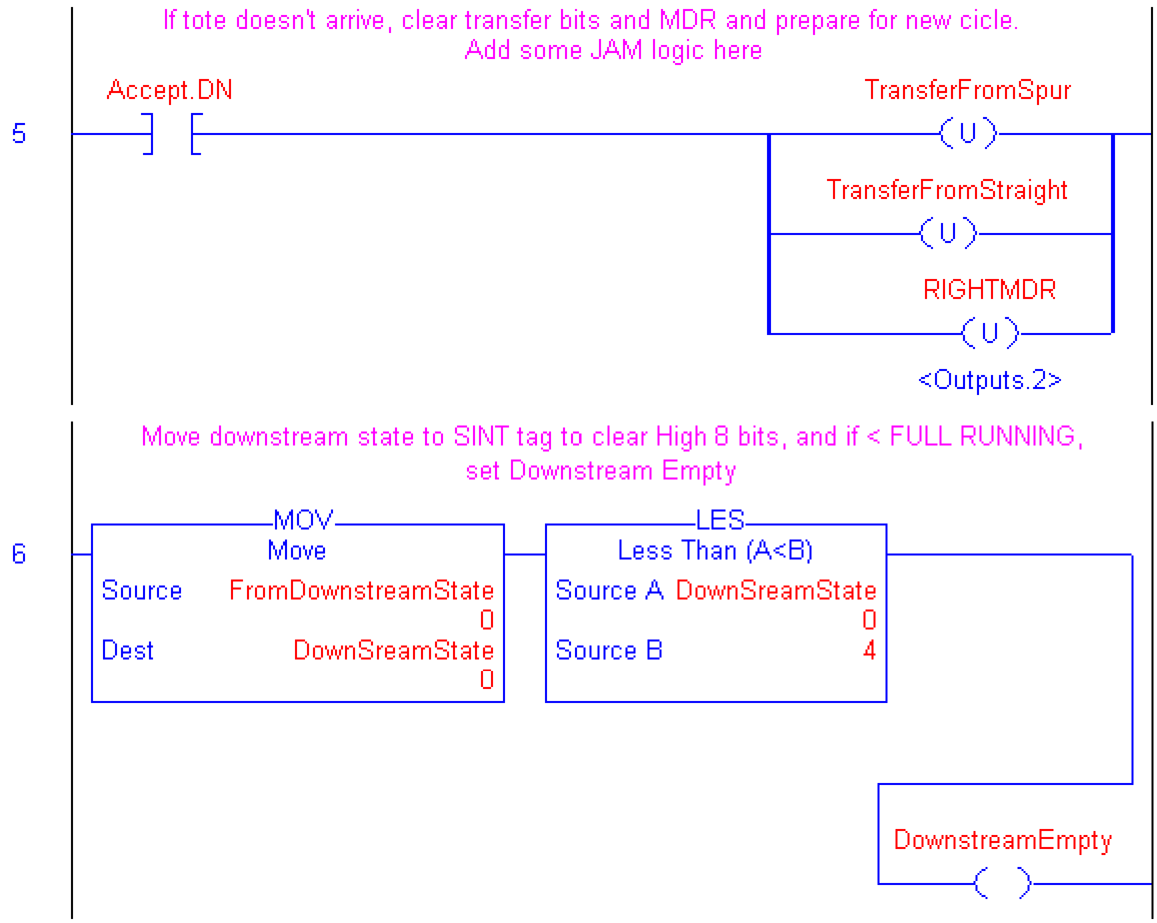
Scope: Main Program

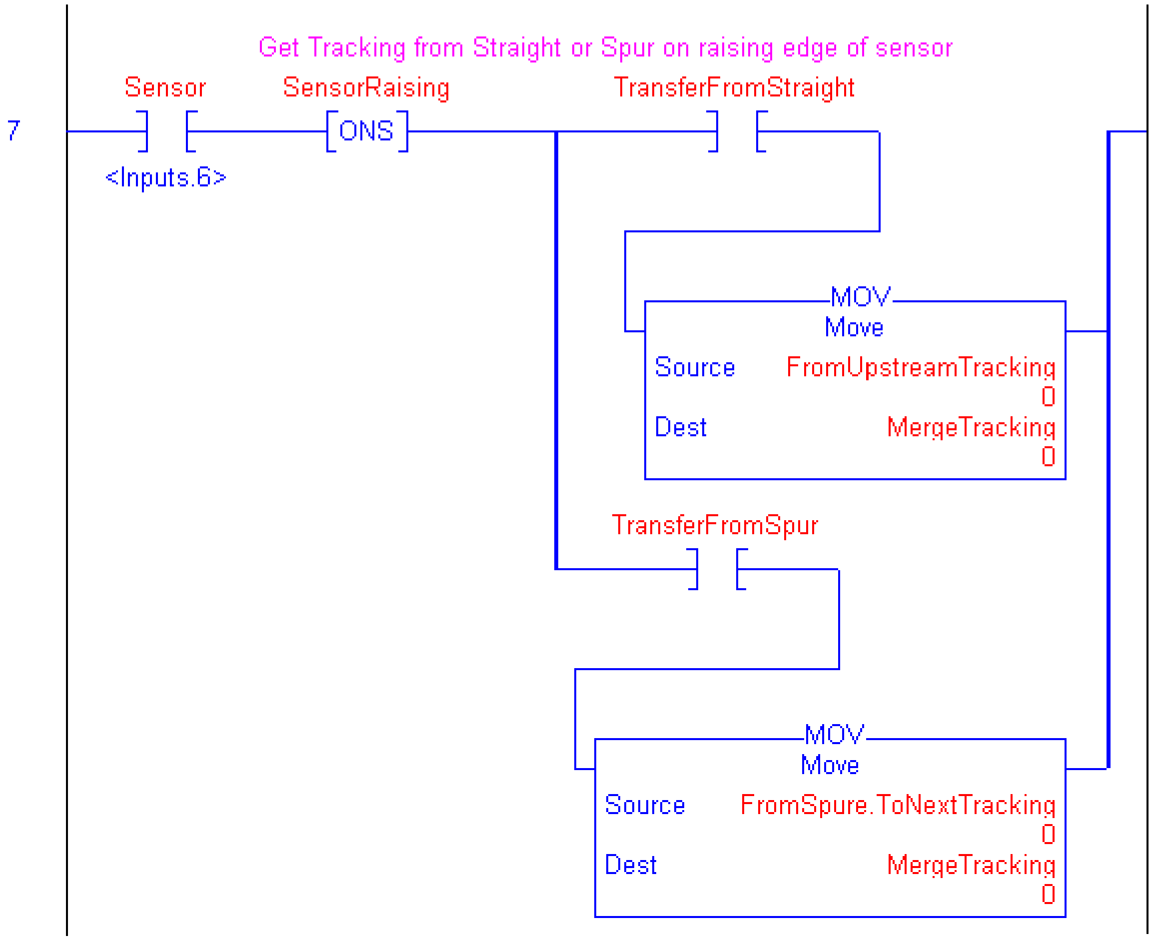
	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	TransferFromStraight			BOOL	0	Decimal	
	TransferFromSpur			BOOL	0	Decimal	
	+ Accept			TIMER	{...}		
	+ Transfer			TIMER	{...}		
	+ State			INT	0	Decimal	
	+ MergeTracking			DINT	0	Decimal	
	Sensor	Inputs.6	Inputs.6	BOOL	0	Decimal	
	DownstreamEmpty			BOOL	0	Decimal	
	+ DownSreamState			SINT	0	Decimal	
	TransferInProgress			BOOL	0	Decimal	
	SensorTrailing			BOOL	0	Decimal	
	RIGHTMDR	Outputs.2	Outputs.2	BOOL	0	Decimal	
	SensorRaising			BOOL	0	Decimal	
	+ FromStraightState			SINT	0	Decimal	
*							

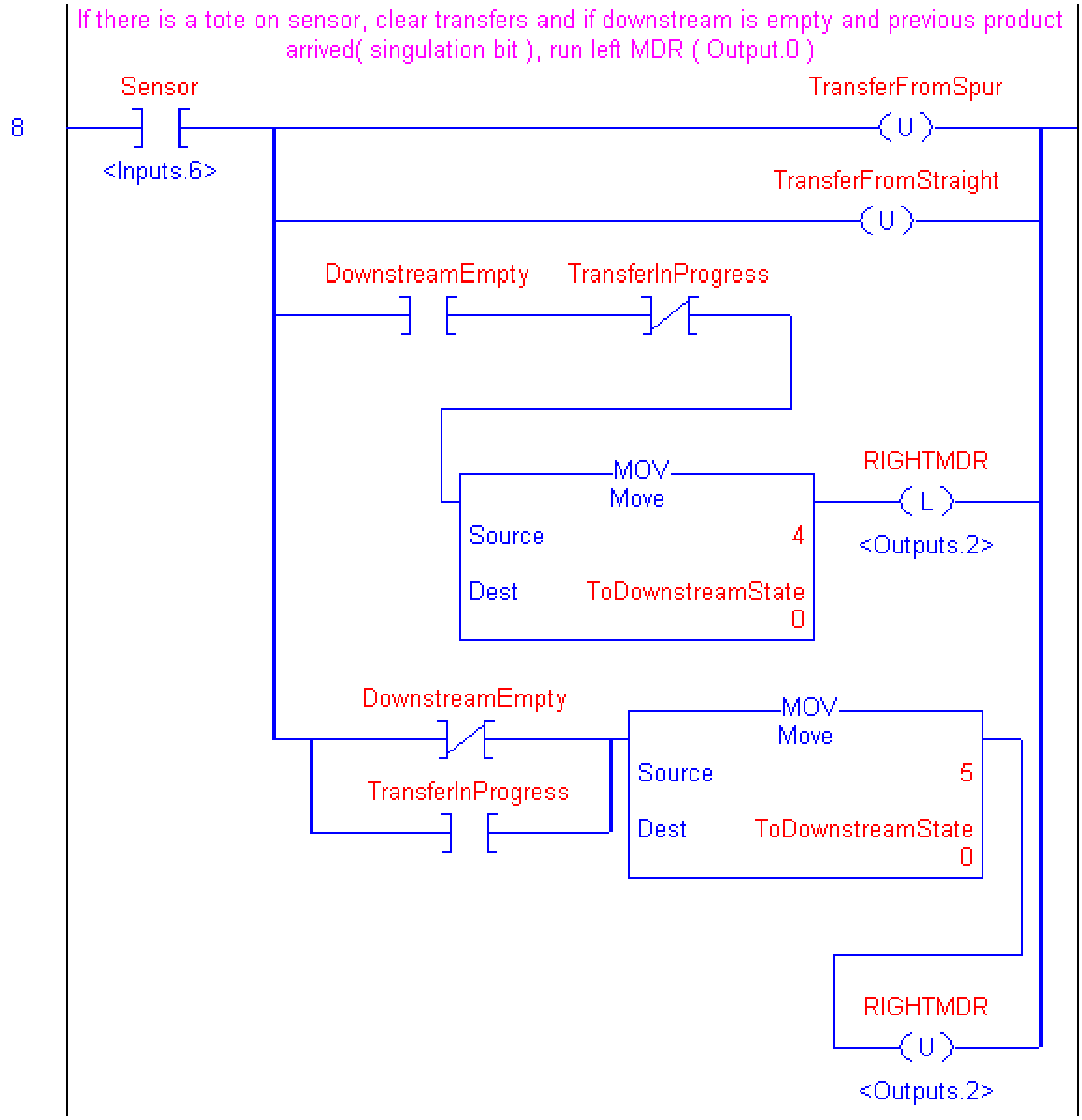
The values of Accept.PRE and Transfer.PRE are equal to 3000.

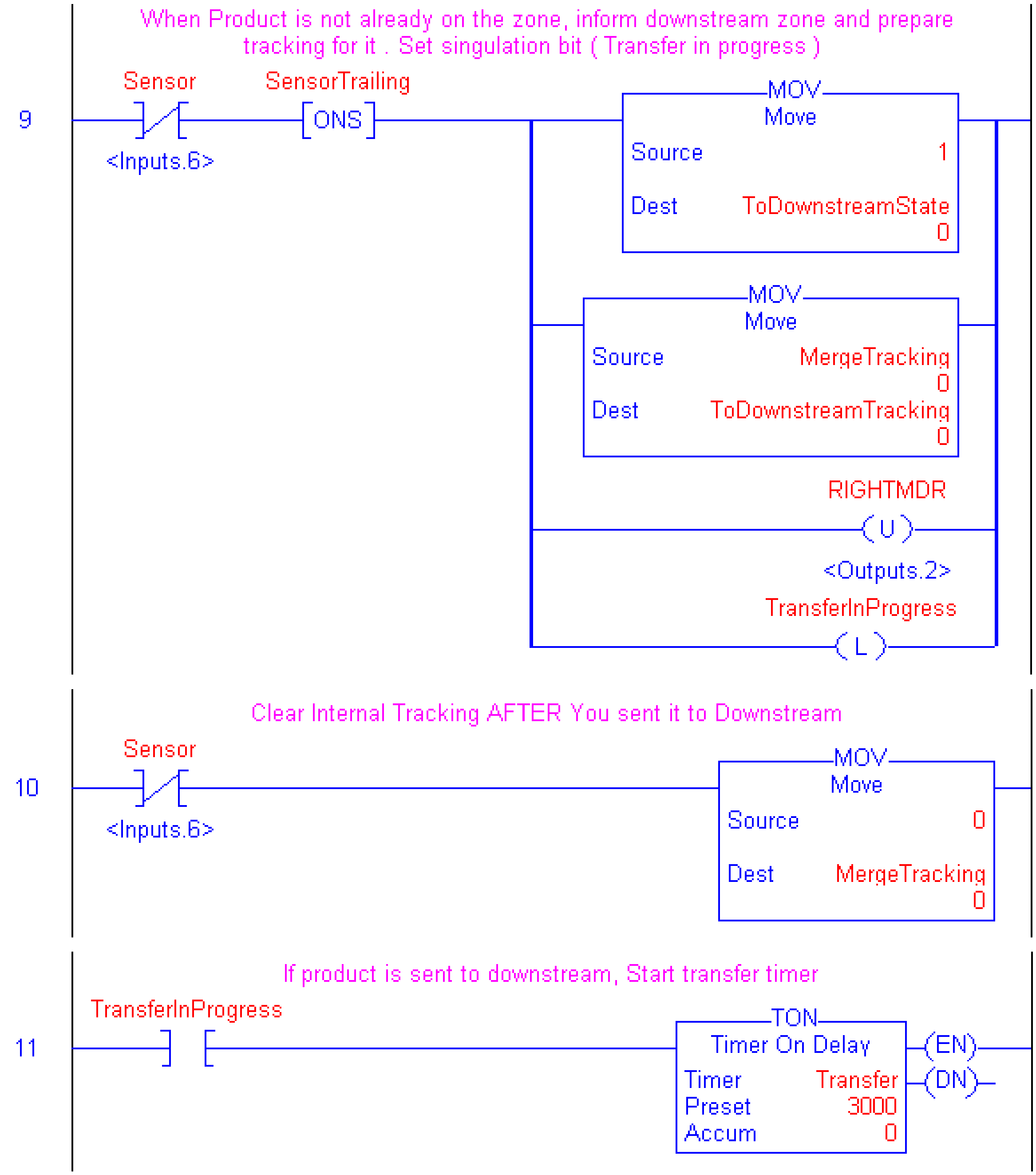
Ladder Logic Program

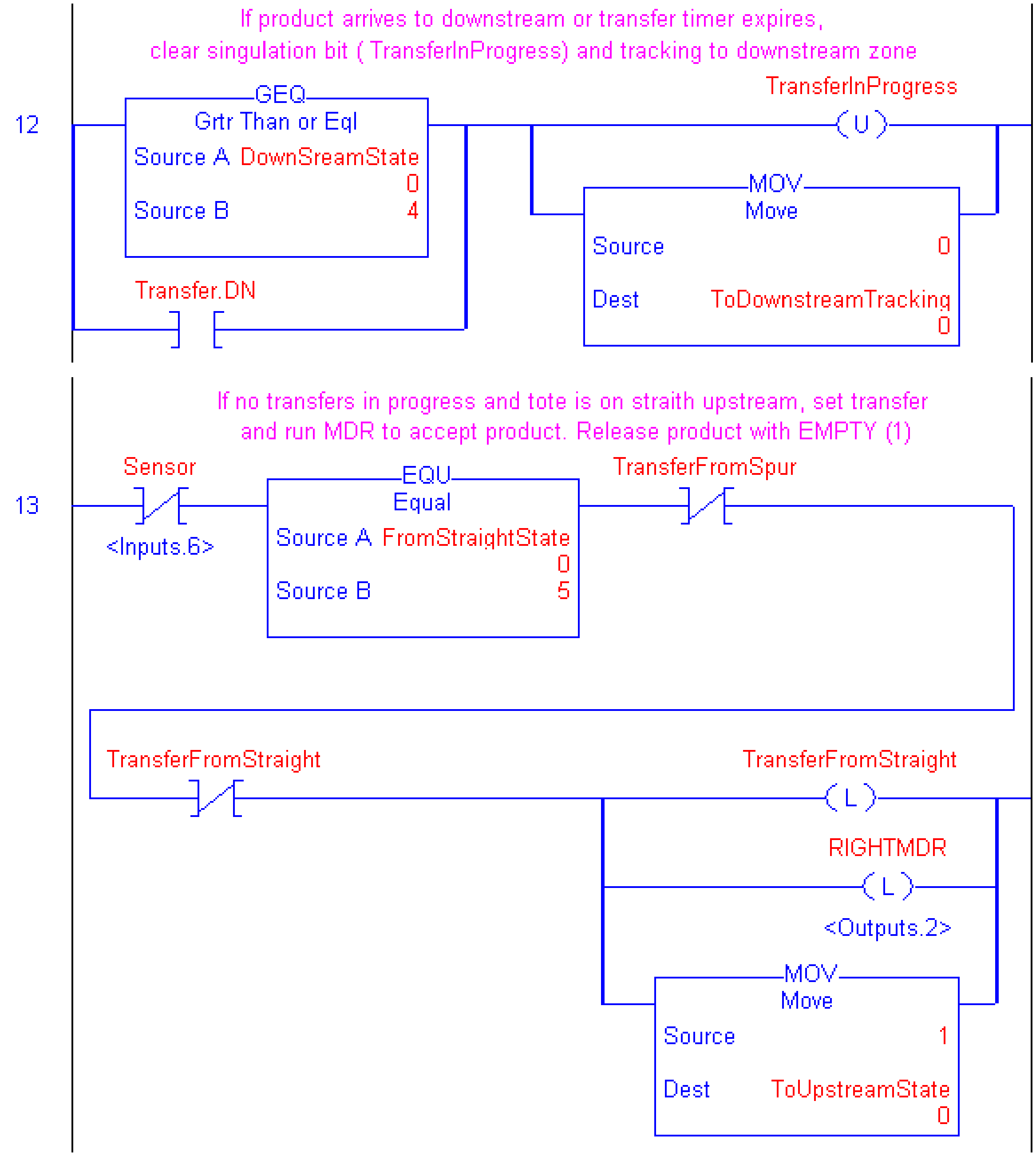



If tote doesn't arrive, clear transfer bits and MDR and prepare for new cicle.
Add some JAM logic here
5
Accept.DN
TransferFromSpur
U
TransferFromStraight
U
RIGHTMDR
U
<Outputs.2>
Move downstream state to SINT tag to clear High 8 bits, and if < FULL RUNNING,
set Downstream Empty
6
MOV
Move
Source FromDownstreamState
0
Dest DownSreamState
0
LES
Less Than (A<B)
Source A DownSreamState
0
Source B
4
DownstreamEmpty

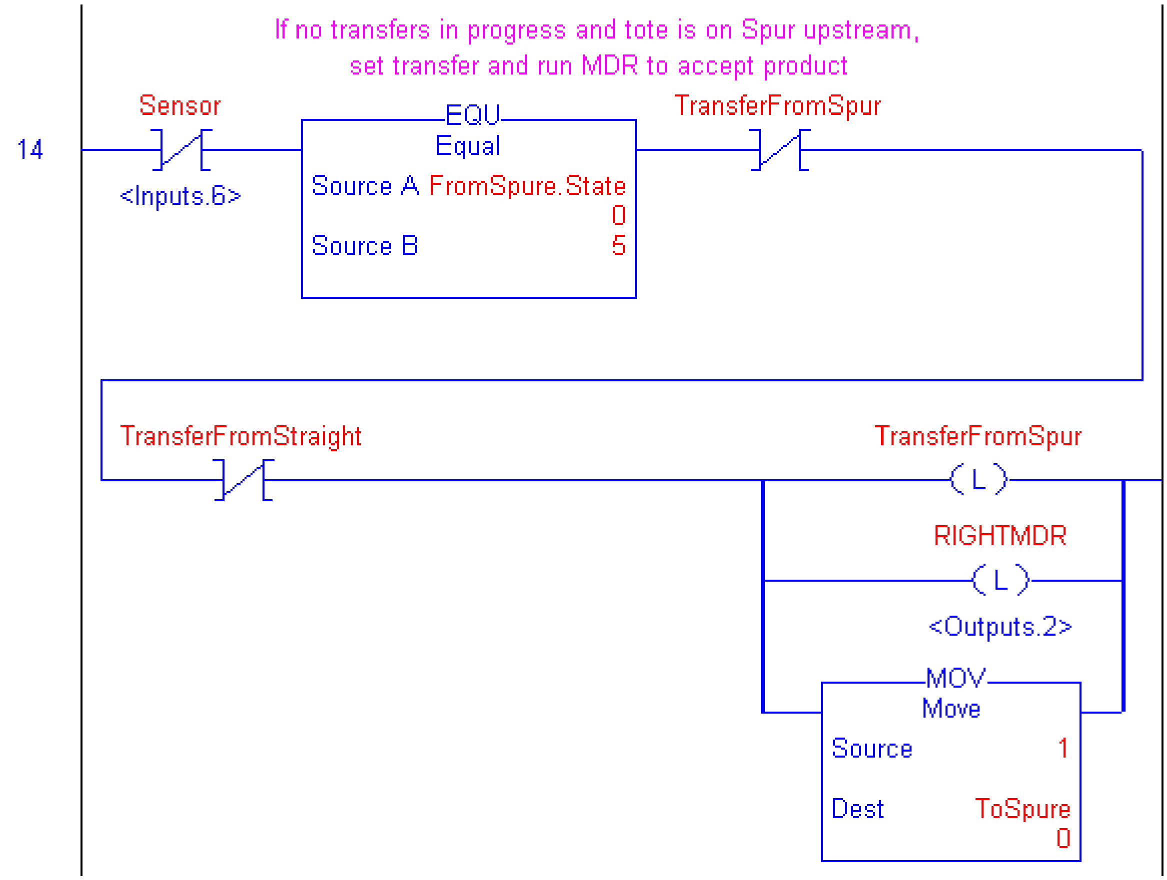

Get Tracking from Straight or Spur on raising edge of sensor
7
Sensor
SensorRaising
TransferFromStraight
ONS
<Inputs.6>
MOV
Move
Source
FromUpstreamTracking
0
Dest
MergeTracking
0
TransferFromSpur
MOV
Move
Source
FromSpure.ToNextTracking
0
Dest
MergeTracking
0


If there is a tote on sensor, clear transfers and if downstream is empty and previous product arrived(singulation bit), run left MDR (Output.0)
8
Sensor
<Inputs.6>
TransferFromSpur
U
TransferFromStraight
U
DownstreamEmpty
TransferInProgress
MOV
Move
Source 4
Dest ToDownstreamState
0
RIGHTMDR
L
<Outputs.2>
DownstreamEmpty
TransferInProgress
MOV
Move
Source 5
Dest ToDownstreamState
0
RIGHTMDR
U
<Outputs.2>


When Product is not already on the zone, inform downstream zone and prepare tracking for it . Set singulation bit (Transfer in progress)
9
Sensor
SensorTrailing
ONS
<Inputs.6>
MOV
Move
Source 1
Dest ToDownstreamState
0
MOV
Move
Source MergeTracking
0
Dest ToDownstreamTracking
0
RIGHTMDR
U
<Outputs.2>
TransferInProgress
L
Clear Internal Tracking AFTER You sent it to Downstream
10
Sensor
<Inputs.6>
MOV
Move
Source 0
Dest MergeTracking
0
If product is sent to downstream, Start transfer timer
11
TransferInProgress
TON
Timer On Delay
Timer Transfer
Preset 3000
Accum 0
EN
DN

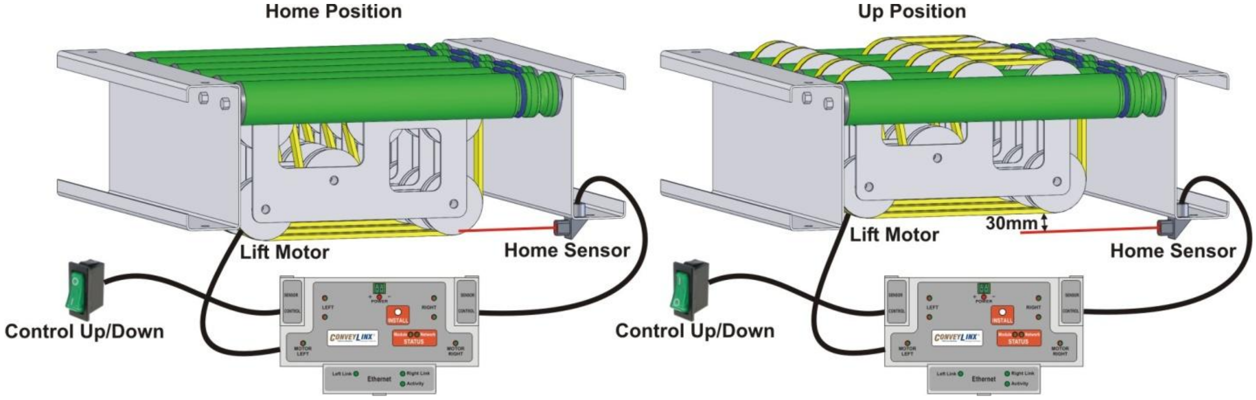


If product arrives to downstream or transfer timer expires,
clear singulation bit (TransferInProgress) and tracking to downstream zone
12
GEQ
Grtr Than or Eql
Source A DownSreamState
0
Source B 4
TransferInProgress
U
MOV
Move
Source 0
Dest ToDownstreamTracking
0
Transfer.DN
If no transfers in progress and tote is on straith upstream, set transfer
and run MDR to accept product. Release product with EMPTY (1)
13
Sensor
<Inputs.6>
EQU
Equal
Source A FromStraightState
0
Source B 5
TransferFromSpur
TransferFromStraight
TransferFromStraight
L
RIGHTMDR
L
<Outputs.2>
MOV
Move
Source 1
Dest ToUpstreamState
0


If no transfers in progress and tote is on Spur upstream,
set transfer and run MDR to accept product
14
Sensor
<Inputs.6>
EQU
Equal
Source A FromSpure.State
0
Source B
5
TransferFromSpur
TransferFromStraight
TransferFromSpur
L
RIGHTMDR
L
<Outputs.2>
MOV
Move
Source
1
Dest
ToSpure
0

Appendix F – Simple Motor Control Example with Servo Commands

In this example is shown how to make a Right Angle Transfer (RAT), using simple motor control.



There is one sensor, named Home Sensor and one switch – Control Up/Down.

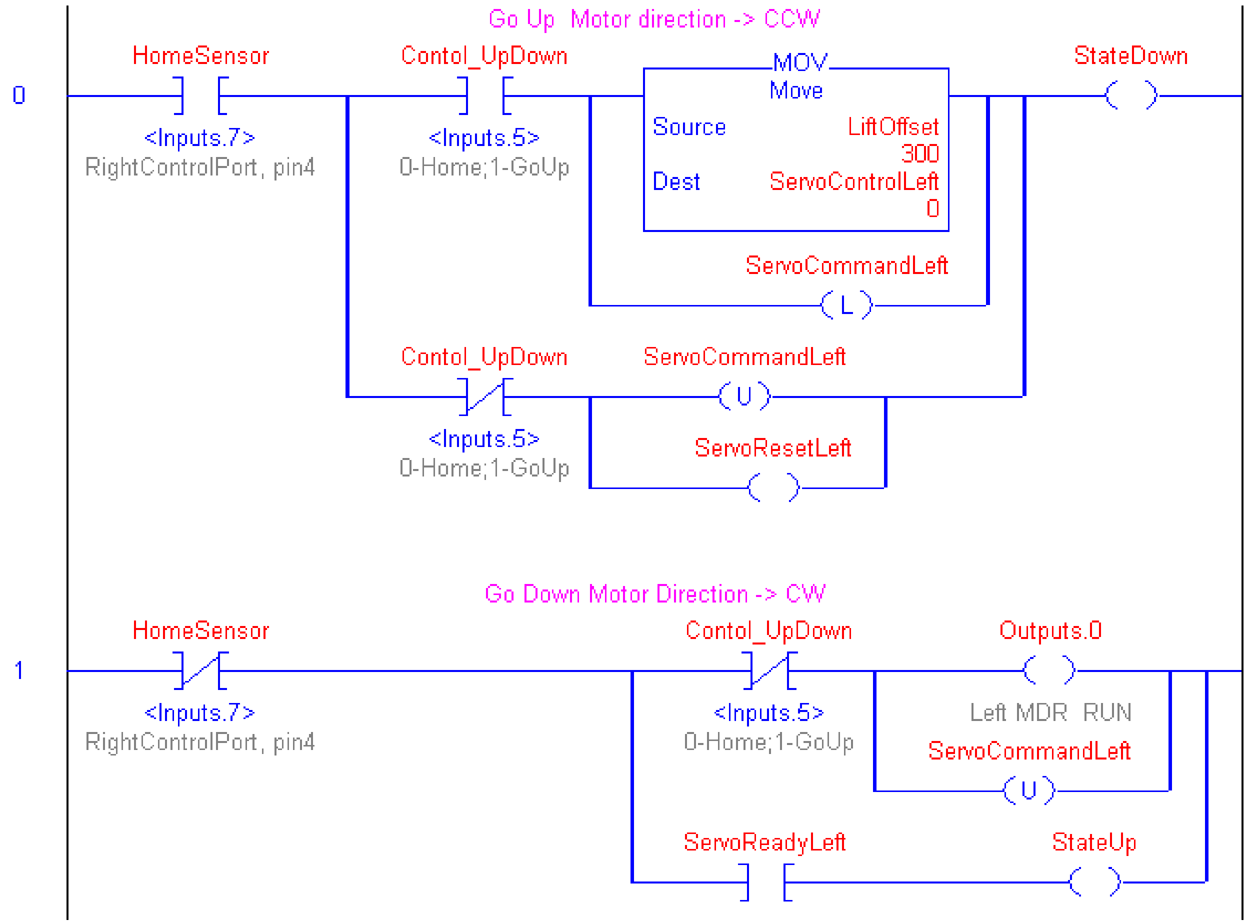
Home Sensor is connected to Right Control Port, PIN4 which corresponds to “Input.7” controller tag. In ConveyLogix program we create the tag “HomeSensor”, which is an alias of “Input.7”.
Control Up/Down switch is connected to Left Control Port, PIN4 which corresponds to Input.5 controller tag. “Control_UpDown” tag is an alias of Input.7.
Tags “StateUp” and “StateDown” indicate the end position of the RAT lift.
“LiftOffset” tag is the distance, which RAT lift has to move to reach the up position. In this example the Lift mechanism travels 30 mm that corresponds to 300 pulses.

In this example, the following Controller Tags are used:
“ServoCommandLeft” – when set, Lift motor starts to move upward (counter-clockwise) to the position which is set in “ServoControlLeft”.
“ServoControlLeft” – contains the pulses that the left motor has to process.
“ServoResetLeft” – clears the pulses that the left motor has to process.
“ServoReadyLeft” – indicates that the pulses are reached.

Main program tags and routine are the following:

Scope: Main Program

	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	LiftOffset			INT	300	Decimal	300 pulses = 30 mm
	StateUp			BOOL	0	Decimal	
	StateDown			BOOL	0	Decimal	
	HomeSensor	Inputs.7	Inputs.7	BOOL	0	Decimal	RightControlPort, pin4
	Contol_UpDown	Inputs.5	Inputs.5	BOOL	0	Decimal	0-Home;1-GoUp
*							



When Lift is in Home position ("HomeSensor" is true) and Control switch is off ("Control_UpDown" is false) the next operations are processed:

"ServoCommandLeft" is unlatched – left motor stops its movement.

"ServoResetLeft" is set – the pulses in "ServoControlLeft" are reset.

"StateDown" is set – Lift is in Home position.

When Lift is in Home position ("HomeSensor" is true) and Control switch is turned on ("Control_UpDown" is set) the next operations are processed:

To "ServoControlLeft" 300 pulses are set.

"ServoCommandLeft" is latched – the left motor starts to move upward

"StateDown" is set – Lift is still in Home position.

When Lift leaves Home Sensor ("HomeSensor" is changed to false) and Control switch is still on the motor continues to run counter-clockwise (upward) until it reaches the pulses. When Lift motor reaches the pulses, "ServoResetLeft" is reset and "StateUp" is set.

When Lift is in Up position ("HomeSensor" is false) and Control switch is turned off ("Control_UpDown" is reset) the next operations are processed:

"Output.0" is true – left motor starts to move downward.

"ServoCommandLeft" is unlatched – left motor servo command is cleared.

When Lift reaches Home Sensor, left motor stops its movement.

During the motor movement "StateUp" and "StateDown" are false.

The following is the same example written in Structured Text:

First you have to create a function block in Structured Text (in this example it is name "RAT").

RAT tags:

	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	⊟ Input						
	HomeSensor			BOOL	0	Decimal	
	Contol_UpDown			BOOL	0	Decimal	
	ServoReady			BOOL	0	Decimal	
*							
	⊟ Output						
	State_Down			BOOL	0	Decimal	
	State_Up			BOOL	0	Decimal	
	RunMotor			BOOL	0	Decimal	
	⊞ ServoControl			INT	0	Decimal	
	ServoCommand			BOOL	0	Decimal	
	ServoReset			BOOL	0	Decimal	
*							
	InOut						
*							
	⊟ Static						
	⊞ LiftOffset			INT	300	Decimal	300 pulses = 30 mm
*							

RAT Routine:

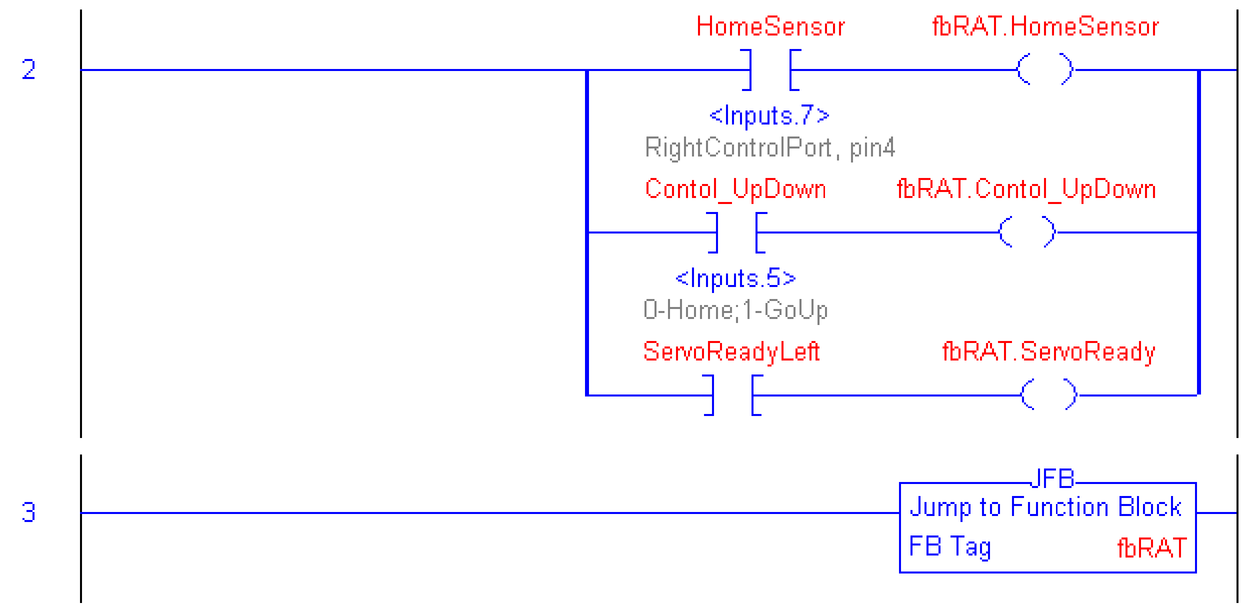
```
ServoReset := 0;
State_Down := 0;
State_Up := 0;
RunMotor := 0;
IF HomeSensor = 1 THEN
        State_Down := 1;
        IF Contol_UpDown = 1 THEN
                ServoCommand := 1;
                ServoControl := LiftOffset;
        ELSE
                ServoCommand := 0;
                ServoReset := 1;
        END_IF;
ELSE
        IF Contol_UpDown = 1 THEN
                IF ServoReady = 1 THEN
                        State_Up := 1;
                END_IF;
        ELSE
                RunMotor := 1;
                ServoCommand := 0;
        END_IF;
END_IF;
```

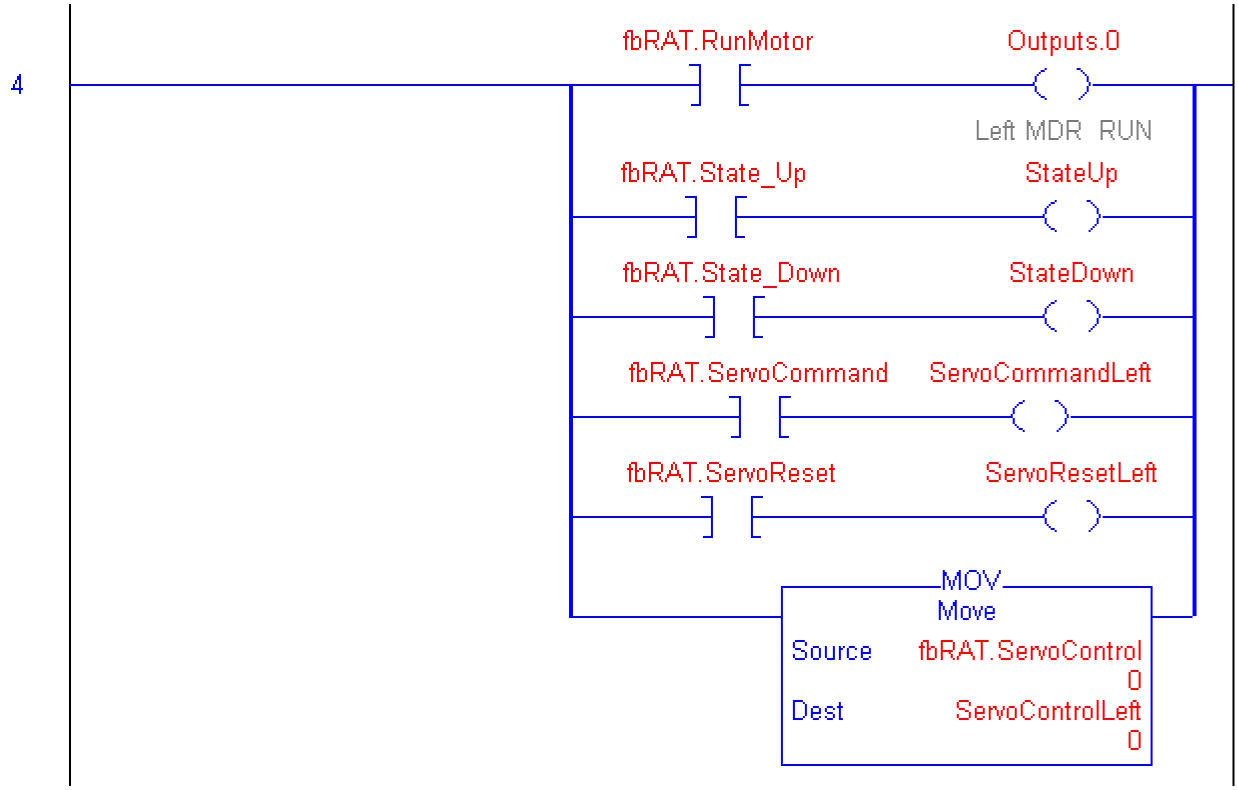
Second, you have to create an instance of “RAT” function block (named “fbRAT”) in Main Tags:

Scope: Main Program

	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	StateUp			BOOL	0	Decimal	
	StateDown			BOOL	0	Decimal	
	HomeSensor	Inputs.7	Inputs.7	BOOL	0	Decimal	RightControlPort, pin4
	Contol_UpDown	Inputs.5	Inputs.5	BOOL	0	Decimal	0-Home;1-GoUp
	⊞ fbRAT			RAT	{...}		
*							

And third, you have to initialize “Input” tags of “RAT” function block, call an instance and then return the values of “Output” tags in Main Program.

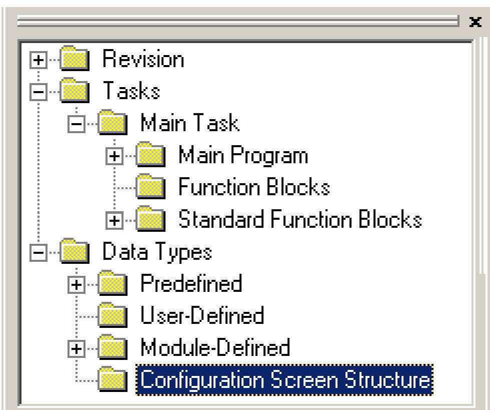



4
fbRAT.RunMotor
Outputs.0
Left MDR RUN
fbRAT.State_Up
StateUp
fbRAT.State_Down
StateDown
fbRAT.ServoCommand
ServoCommandLeft
fbRAT.ServoReset
ServoResetLeft
MOV
Move
Source fbRAT.ServoControl
0
Dest ServoControlLeft
0

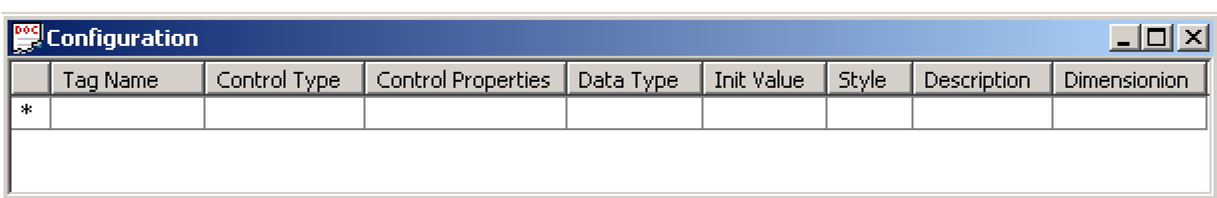
Appendix G – Configuration Screen Structure

Configuration Screen Structure is designed to provide an easy way to configure a module via EasyRoll software.

To open the view of the configuration structure, double-click on "Configuration Screen Structure" in Project Tree:



Blank tags view is opened. Configuration Structure view is similar to User-Defined type tags view:



Tags are added, edited and deleted as the same way as User-Defined type tags.

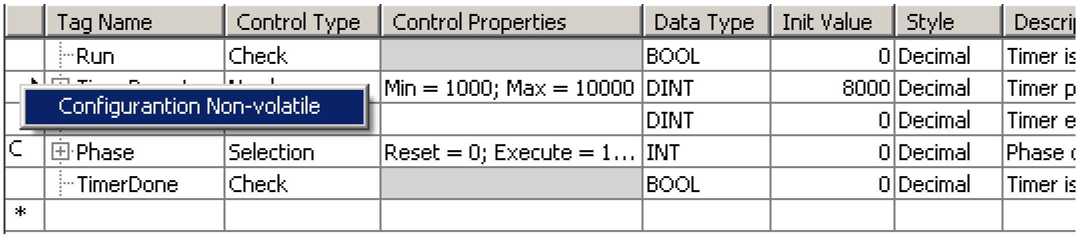
Data types of tags in Configuration Screen Structure can only be simple data types (BOOL, SINT, INT and DINT).

Tags may be divided into two categories, according to EasyRoll software usage:

- Normal – for monitoring only.

- Configuration Non-volatile – for configuration and monitoring. These tags are power independent and use the non-volatile memory of the module. After power cycle, their values remain unchanged.

To make a tag Configuration Non-volatile, right-click on the cell at the first column and select "Configuration Non-volatile" menu:



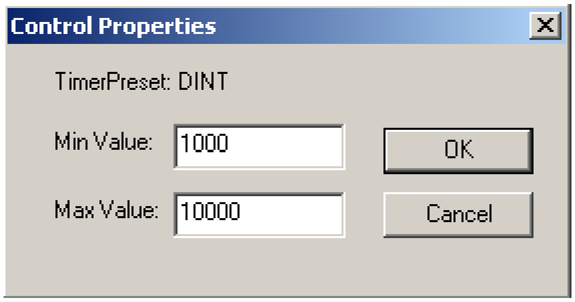
Configuration Non-volatile tags are marked with sign "C" in the first column.

"Control Type" field defines how the tag will appears in EasyRoll software. For some Control Types you may add Control Properties.

When you click on this field, a combo-box with three options appears:

- Check – check-box. It is suitable for BOOL data types.
- Number – edit-box. It is suitable for SINT, INT and DINT data types.

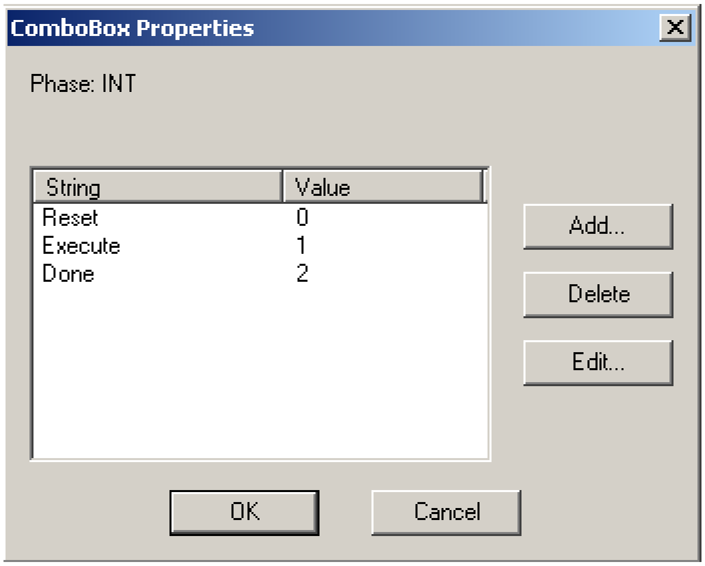
With left-click on "Control Properties" cell you may add minimum and/or maximum values for the tag.



When a tag is Configuration Non-volatile in EasyRoll software "Set" button is displayed next to the edit-box. "Set" button is used for changing tag values. When the value is not in defined range a message box appears.

- Selection – combo-box. It is suitable for SINT, INT and DINT data types.

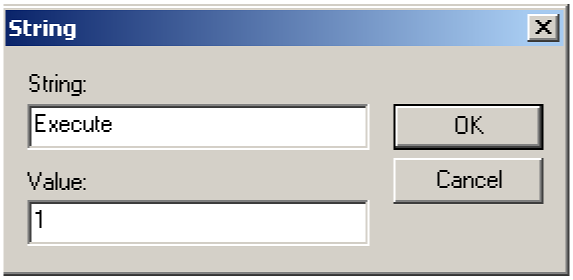
With left-click on "Control Properties" cell you may add, edit and delete selection strings of the tag.



To add a new selection string, click on Add button.

To change the string name or value, select the desired string and then click on Edit button.

To delete the string, select the desired string and then click on Delete button.



When the value of the tag does not correspond to any string, the selection in EasyRoll software is empty.

The text written in “Description” field will appear in the control tool tip in EasyRoll software.

The text written in “Dimension” field will appear after the control in EasyRoll software.

To work with Configuration Structure you have to add a tag in Main Tags with “Configuration” data type. You may create only one instance of “Configuration” data type.

When you change a field of Configuration Structure from Main Tags view, it is will be reflected in the corresponding field in Configuration Screen Structure. And vice versa, if a field from

Configuration Screen Structure tag is changed, it is reflected in the Configuration Structure instance in Main Tags.

Example:

In this example a timer will be started, monitored and reset, using Configuration Structure.

The following tags in Configuration Screen Structure are added:

	Tag Name	Control Type	Control Properties	Data Type	Init Value	Style	Description	Dimensionion
	Run	Check		BOOL	0	Decimal	Timer is running	
C	TimerPreset	Number	Min = 1000; Max …	DINT	8000	Decimal	Timer preset va…	ms
	TimerValue	Number		DINT	0	Decimal	Timer evaluate…	ms
C	Phase	Selection	Reset = 0; Execu…	INT	0	Decimal	Phase of the timer	
	TimerDone	Check		BOOL	0	Decimal	Timer is done	
*								

"Run", "TimerValue" and "TimerDone" are used for monitoring only.

"Run" tag indicates whether the timer is running.

"TimerValue" tag displays timer evaluated value in milliseconds.

"TimerDone" tag is set when timer expiries.

"TimerPreset" and "Phase" tags are configurable from EasyRoll software. They are power independent and use controller's Flash memory.

"TimerPreset" tag contains the timer preset value in milliseconds.

"Phase" tag controls the timer execution. When tag value is "0", the timer will reset. When tag value is "1", the timer will start. When timer expiries, "Phase" tag value will set to "2".

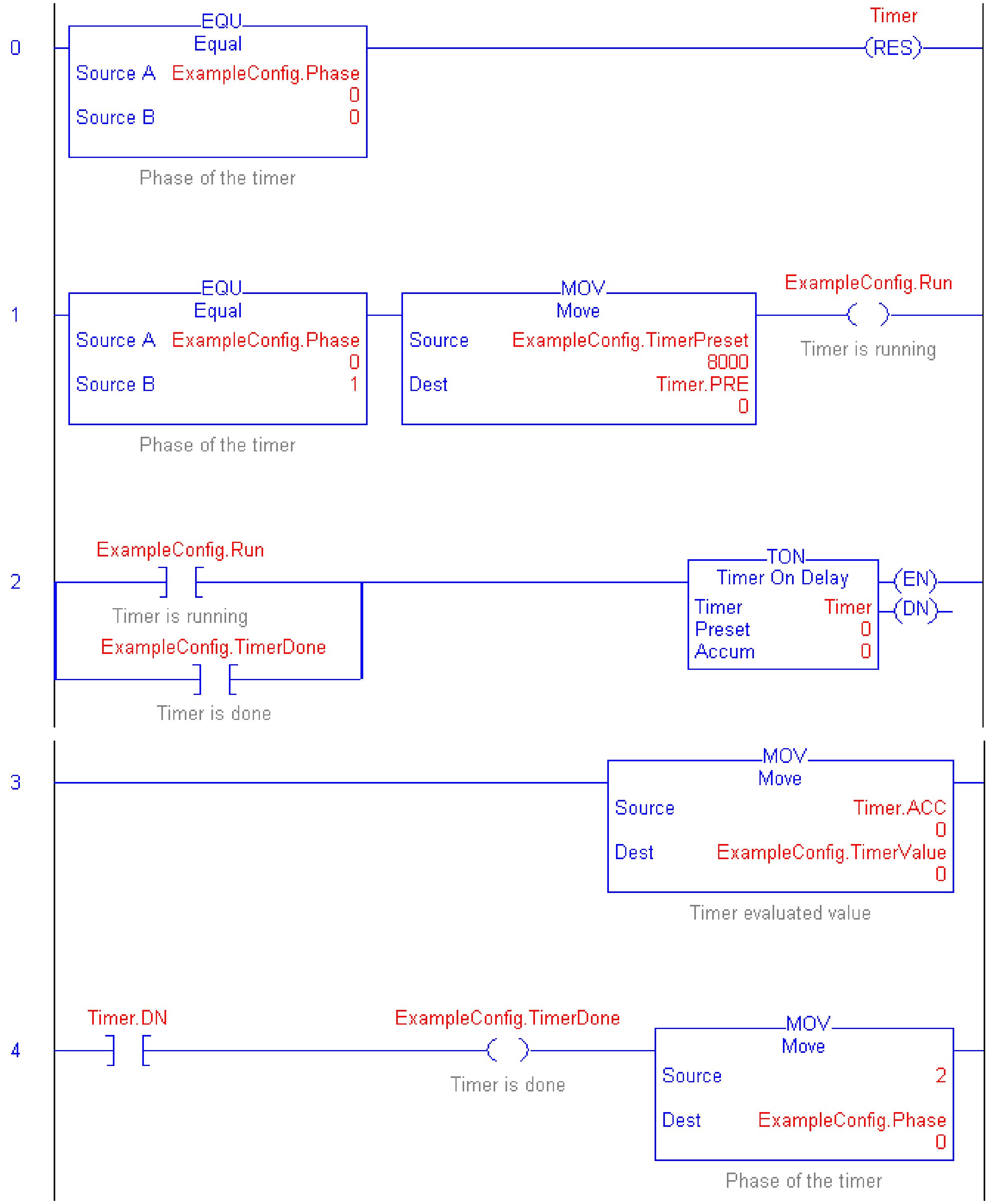
A tag named "ExampleConfig" with "Configuration" data type is added in Main tags.

Scope: Main Program

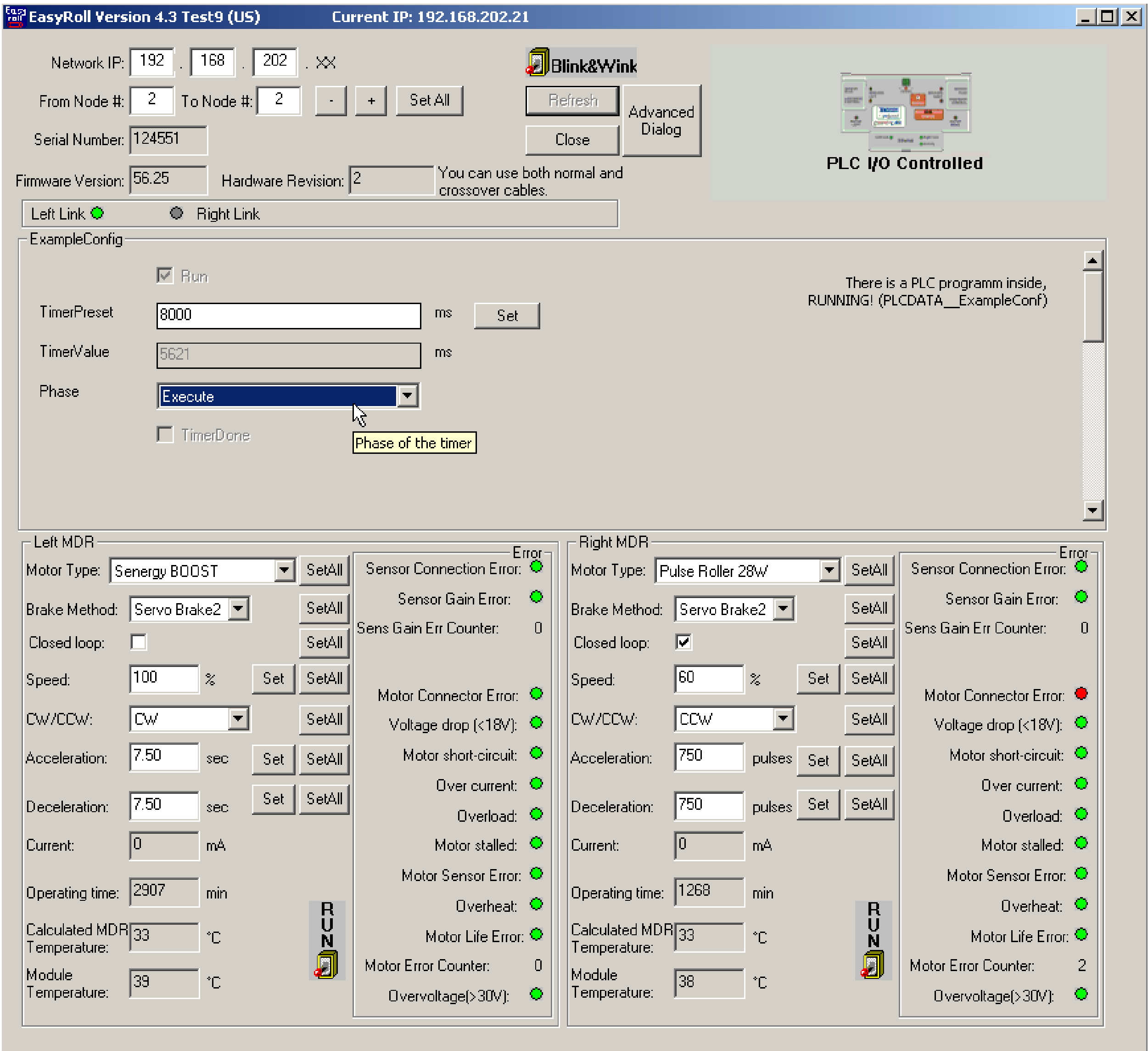
	Tag Name	Alias For	Base Tag	Data Type	Init Value	Style	Description
	ExampleConfig			Configuration	{...}		
	ExampleConfig.Run			BOOL	0	Decimal	Timer is running
N	ExampleConfig.TimerPreset			DINT	8000	Decimal	Timer preset v…
	ExampleConfig.TimerValue			DINT	0	Decimal	Timer evaluate…
N	ExampleConfig.Phase			INT	0	Decimal	Phase of the ti…
	ExampleConfig.TimerDone			BOOL	0	Decimal	Timer is done
	Timer			TIMER	{...}		
*							

The sign "N" shows that the tags are Non-volatile.

The next picture shows the example Main Program:



In EasyRoll software the controls of Configuration Structure are shown as follows:



Notes:

PULSEROLLER

WWW.PULSEROLLER.COM
SALES@PULSEROLLER.COM
SUPPORT@PULSEROLLER.COM

Publication ERSC-1200 Rev 2.2 – July 2016