

15

ControlLogix Controllers

Programmable logic controllers continue to evolve as new technologies are added to their capabilities. The PLC started out as a replacement for banks of relays used to turn outputs on and off as well as for timing and counting functions. Gradually, various math and logic manipulation functions were added. In order to serve today's expanding industrial control system needs, leading automation companies have created a new class of industrial controllers called *programmable automation controllers* or *PACs* (Figure 15-1). They look like PLCs in their physical appearance but incorporate advanced control of communication, data logging, and signal processing, motion, process control, and machine vision in a single programming environment.

The Allen-Bradley programmable automation controller family includes the ControlLogix

system, CompactLogix system, FlexLogix system, SoftLogix 5800 controller, and DriveLogix system. *Software* is the essential difference between PACs and PLCs. Basically, the ladder logic configuration does not change but the addressing of the instructions changes. Application of the software that pertains to the Logix control platform of controllers will be covered in the various sections of this chapter. Knowledge of basic ladder logic instructions and functions (bit, timer, counter, etc.) covered in previous chapters of the text is assumed and is thus not repeated in this chapter.



Figure 15-1 Programmable automation controllers (PACs).
Source: Image Used with Permission of Rockwell Automation, Inc.

Part 1 Memory and Project Organization

Memory Layout

ControlLogix processors provide a flexible memory structure. There are no fixed areas of memory allocated for specific types of data or for I/O. The internal memory organization of a ControlLogix controller is configured by the user when creating a project with RSLogix 5000 software (Figure 15-2). This feature allows the program data to be constructed to meet the needs of your applications rather than requiring your application to fit a particular memory structure. A ControlLogix (CLX) system can consist of anything from a stand-alone controller and I/O modules in a single chassis, to a highly distributed system consisting of multiple chassis and networks working together.

Configuration

Configuration of a modular CLX system involves establishing a communications link between the controller and the process. The programming software needs to know what CLX hardware is being used in order to be able to send or receive data. Configuration information includes information about the type of processor and I/O modules used.

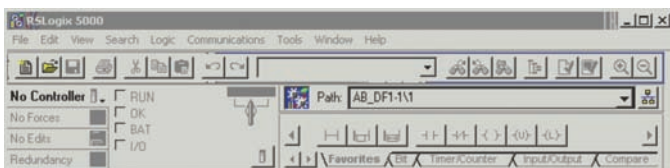


Figure 15-2 RSLogix 5000 screen.

Part Objectives

After completing this part, you will be able to:

- Outline project organization
- Define tasks, programs, and routines
- Identify data file types
- Organize and apply the various data file types

RSLogix 5000 programming software is used to set up or *configure* the memory organization of an Allen-Bradley ControlLogix controller. *RSLink* communication software is used to set up a communications link between RSLogix 5000 programming software and the ControlLogix hardware as illustrated in Figure 15-3. To establish communications with a controller, a driver must be created in RSLink software. This driver functions as the software interface to a hardware device. The *RSWho* is the network browse interface that provides a single window to view all configured network drivers.

Figure 15-4 shows an example of the ControlLogix's *controllers properties* and *modules properties* dialog boxes used as part of the configuration process. The parameters shown are typical of what general information is required. After first configuring the controller,

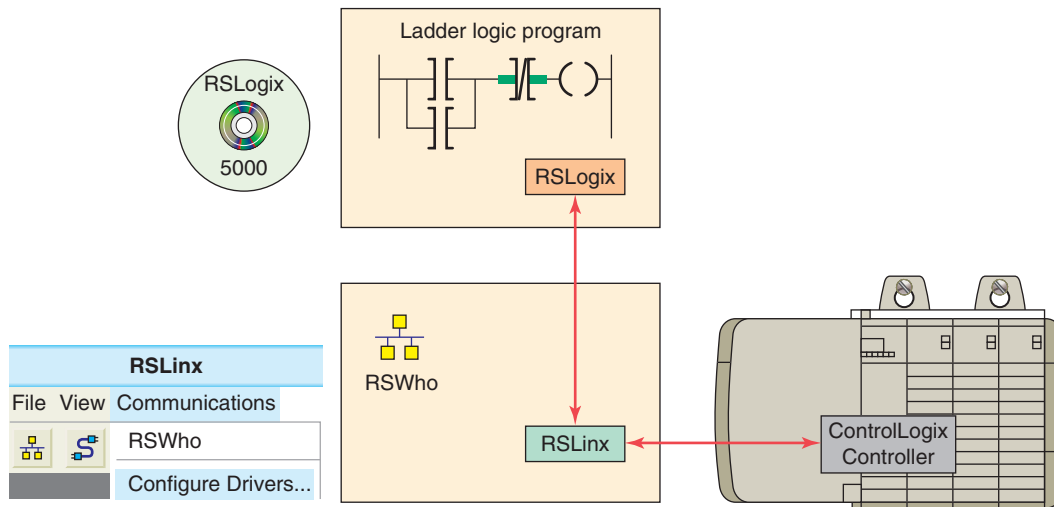


Figure 15-3 RSLinx and RSLogix software.

General		Controllers properties	
Vendor:	Allen-Bradley	Type:	1756-L55ControlLogix5555Controller
Revision:	10.24	Name:	Controller 1
Description:	Prime Controller	Chassis Type:	1756-A7 7-Slot Chassis
Slot:	1		

General		Modules properties	
Type:	1756-IB16 16 Point 10V-31.2V DC Input	Vendor:	Allen-Bradley
Parent:	Local	Slot:	0
Name:	Digital_Input_16pt	Description:	Optional
Comm Form:	Input Data	Revision:	1
Electronic Keying:	Exact Match		

Figure 15-4 Controllers properties and modules properties dialog boxes.

the I/O modules are configured using RSLogix 5000 software. Modules will not work unless they have been properly configured. The software contains all the hardware information needed to configure any ControlLogix module.

Project

RSLogix software stores a controller's programming and configuration information in a file called a *project*. The block diagram of the processor's project file is shown in Figure 15-5. A project file contains all information relating to the project. The main components of the project file are tasks, programs, and routines. A controller can hold and execute only one project at a time.

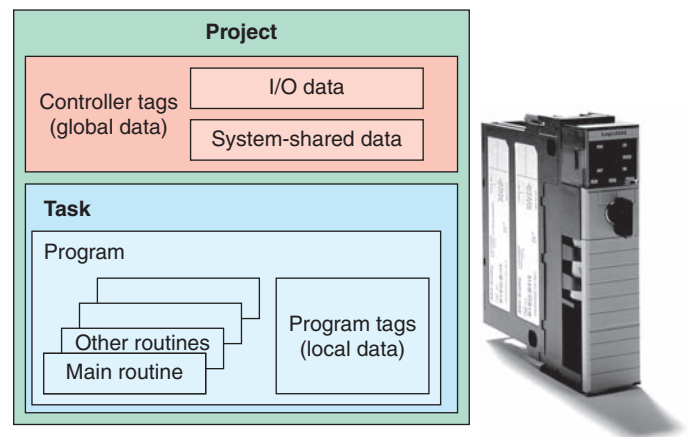


Figure 15-5 ControlLogix processor program file.
Source: Image Used with Permission of Rockwell Automation, Inc.

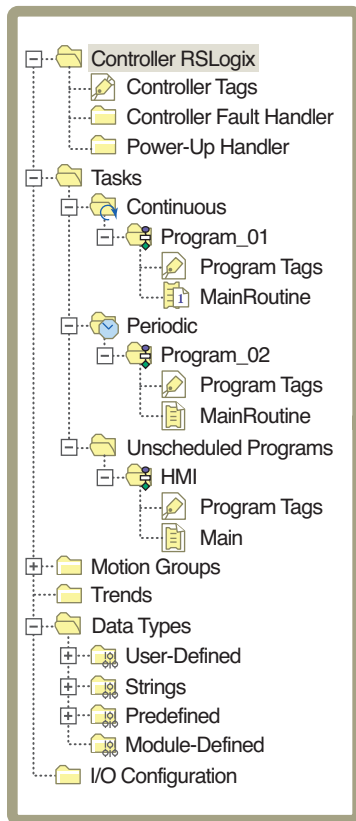


Figure 15-6 Controller organizer tree.

The RSLogix 5000 controller organizer (Figure 15-6) displays the project organization in a tree format showing tasks, programs, routines, data types, trends, I/O configuration and tags. Each folder groups common functions together. This structure simplifies the navigation and the overall view of the whole project.

In front of each folder, there is an icon containing a + sign or a – sign. The + sign indicates that the folder is closed. Click it to expand the tree display and display

the files in the folder. The – sign indicates that the folder is already open and its contents are visible. Clicking on the right mouse button brings up many different, context-sensitive popup menus. Often, you find that this is a short-cut to access the property window or menu options from the menu bar.

Tasks

Tasks are the first level of scheduling within a project. A task is a collection of scheduled programs. When a task is executed, the associated programs are executed in the order listed. This list of programs is known as the program schedule. Tasks provide scheduling based on specific conditions and do not contain any executable code. Only one task may be executing at any given time. The number of tasks a controller can support depends on the specific controller. The main types of tasks (Figure 15-7) include:

- *Continuous* tasks execute nonstop but are always interrupted by a periodic task. Continuous tasks have the lowest priority. A ControlLogix continuous task is similar to the File 2 in the SLC 500 platform. Here the continuous task is named Main Task.
- *Periodic* tasks function as timed interrupts. They interrupt the continuous task and execute for a fixed length of time at specific time intervals.
- *Event* tasks also function as interrupts. Rather than being an interrupt on a timed basis, an event task is triggered by an event that happened or failed to happen.

Programs

Programs are the second level of scheduling within a project. The function of the folders under Main Task is to determine and specify the order in which the programs

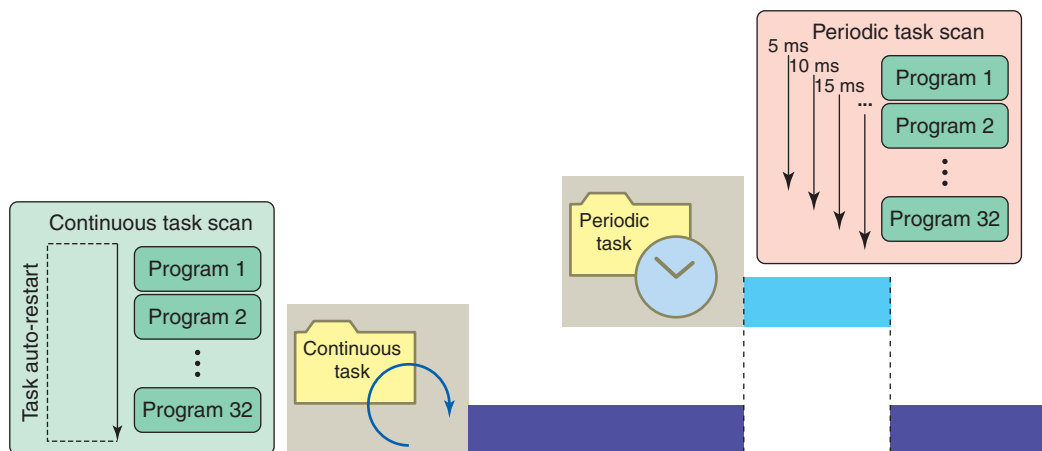


Figure 15-7 Continuous and periodic tasks.

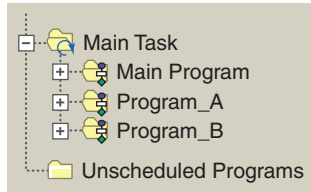


Figure 15-8 Order of execution of programs.

execute. There is no executable code within a program. Routines within programs will execute in the order listed below their associated task in the controller organizer as shown in Figure 15-8. In this example, according to the listed order, the Main Program is scheduled to execute first, Program_A second, and Program_B third. Programs that are not assigned to a task are unscheduled. Unscheduled programs are downloaded to the controller but do not execute. These programs remain unscheduled until needed. Depending on the RSLogix 5000 software version as many as 100 programs could be scheduled within each task.

Routines

Routines are the third level of scheduling within a project and provide the executable code for the project. Each routine contains a set of logic elements for a specific

programming language. When a routine is created it is specified as ladder logic, sequential function chart, function block diagram or structured text (Figure 15-9). Any one routine must be completely in the same language. The number of routines per project is limited only by the amount of controller memory. Libraries of standard routines can be created that can be reused on multiple machines or applications. A routine can be assigned as one of the following types:

- A *main routine* is one configured to execute first when the program runs. Each program will have one main routine typically followed by several or many subroutines.
- A *subroutine* is one that is called by another routine. Subroutines are used for large or complex programming tasks or tasks that require more than one programming language.
- A *fault routine* is one that executes if the controller finds a program fault. Each program can have one fault routine, if desired.

Tags

Unlike conventional controllers, ControlLogix uses a *tag-based* addressing structure. Tags are meaningful names, descriptive of your application and not merely generic

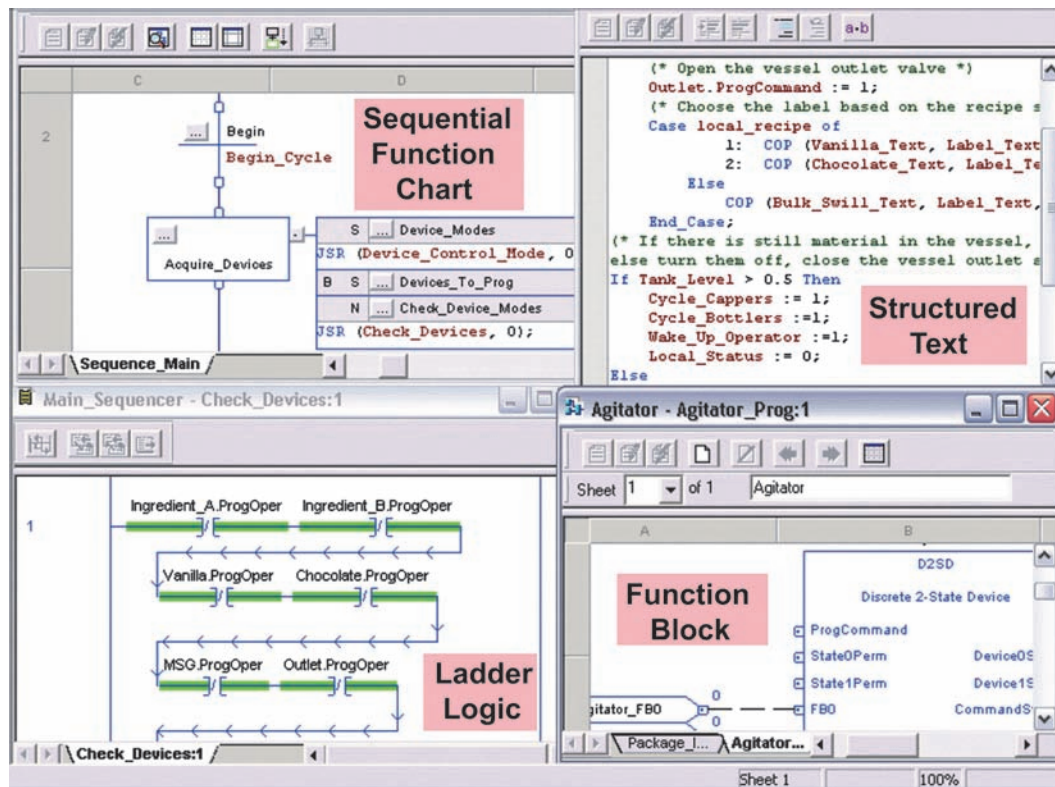


Figure 15-9 Each routine contains a set of logic elements for a specific programming language.

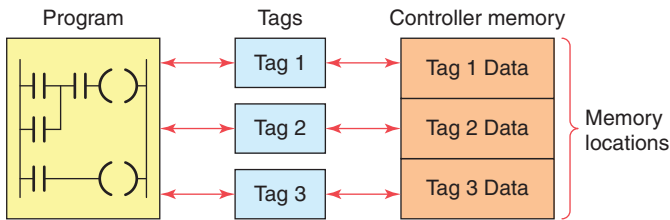


Figure 15-10 Tags used to assign memory locations.

addresses. A tag is created to represent the data and identify areas in the controller’s memory where these data are stored. In applications developed using Logix 5000 software, there are no predefined data tables such as in an SLC 500. When you want to use or monitor data in a program you use tag names to refer to the memory locations, as illustrated in Figure 15-10. This functionality allows you to name your data specifically for their functions within the control program while providing self-documented logic. Whenever you wish to group data, you create an array, which is a grouping of tags of similar types.

Scope refers to which programs have access to a tag. The scope of a tag must be specified when you create the tag. There are two scopes for tags: program scope and controller scope. A *program tag* consists of data that can be accessed only by routines within a specific program (local data). The routines in other programs cannot access program scoped tags of another program. A *controller tag* consists of data that are accessible by all routines within a controller (global data). Figure 15-11 shows two programs, A and B, within a project. Note that each program has program scope tags with identical names (Tag_1, Tag_2, and Tag_3). Because they are program scoped, there is no relationship between them, even though they have the same name. The program scope data are accessible only to the routines within a single program. The same tag name may appear in different programs as local variables because you can select the scope in which to create the tag.

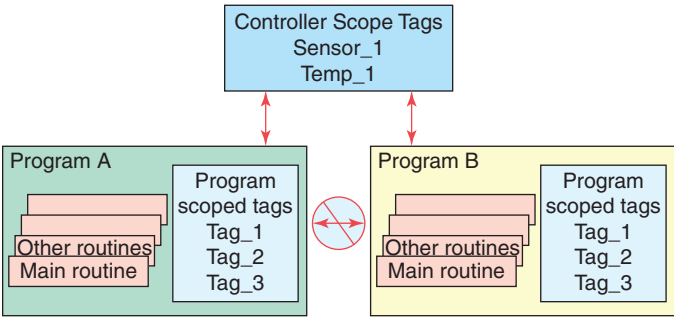


Figure 15-11 Program scoped and controller scoped tags.

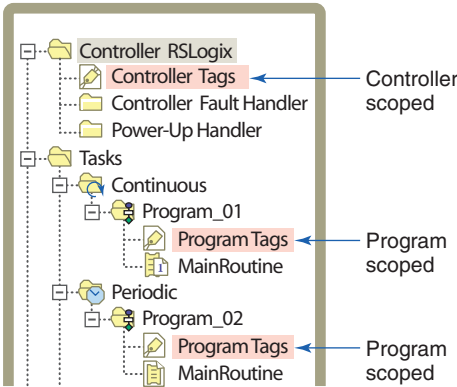


Figure 15-12 Listing of program and controller scoped tags.

The scope of a tag must be declared when you create the tag. Figure 15-12 shows program and controller scoped tags as listed in the controller organizer under the program they are assigned to. I/O tags are automatically created as controller scoped tags.

There are four different tag types: base, alias, produced, and consumed tags. The tag type defines how the tag operates within the project. A *base tag* stores various types of data for use by logic in the project. This tag defines a memory location where data are stored. Base tag memory use depends on the type of data the tag represents. An example of the base tag Local:2:O.Data.4 is shown in Figure 15-13 and is based on the following format:

Location	Network location LOCAL = same chassis as the controller
Slot	Slot number of I/O module in its chassis
Type	Type of data I = input O = output C = configuration S = status
Member	Specific data from the I/O module; depends on what type of data the module can store.
SubMember	Specific data related to a Member.
Bit	Specific point on a digital I/O module; depends on the size of the I/O module (0-31 for a 32-point module)

An *alias tag* is used to create an alternate name (alias) for a tag. The alias tag is simply another name for an already named memory location. An alias tag can refer to a base, alias, consumed, or produced tag. The alias tag is often used to create a tag name to represent a real-word input or output. Figure 15-14 shows an example of the

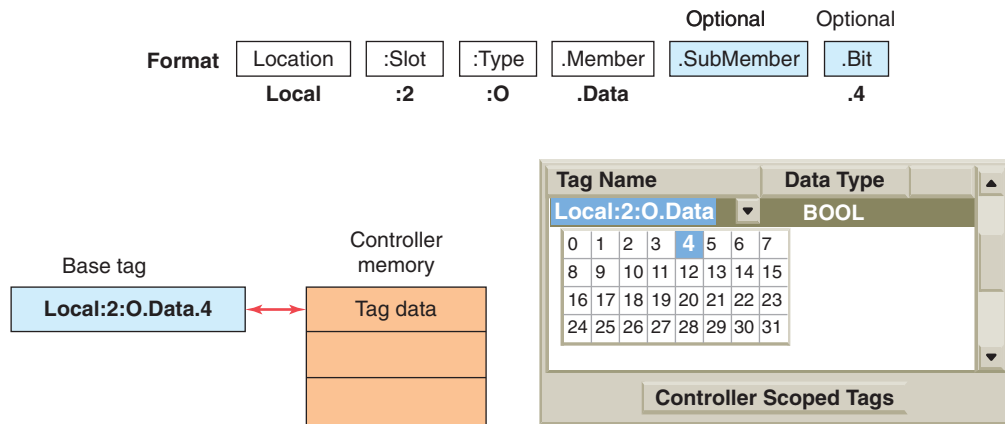


Figure 15-13 Base tag.

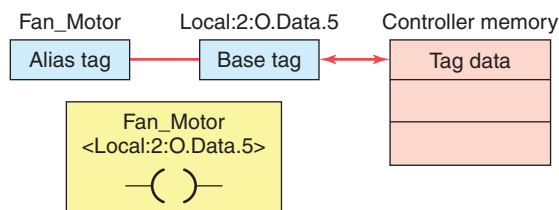


Figure 15-14 Alias tag linked to a base tag.

use of an alias tag. The alias tag (Fan_Motor) is linked to the base tag (<local:2:O.Data.5>) so that any action to the base also happens to the alias and vice versa. The alias name is easier to understand and easier to relate to the application, while the base tag contains the physical location of the output point in the ControlLogix chassis.

Produced/consumed tags are used to share tag information over a network between two or more devices. A produced tag sends data while a consumed tag receives data. Produced tags are always controller scoped. Figure 15-15 shows an example of how a controller can produce data and send them over the network to two controllers that use

or consume the data. The producing controller will have a tag that is of the produced type, whereas the consuming controllers will have a tag with the exact same name that is of the consumed type.

When you design your application, you configure it to both produce globally to other controllers in the system via the backplane and to consume tags from other controllers. This feature allows you to be selective about which data are sent and received by any controller. Likewise, multiple controllers can connect to any data being produced, thereby preventing the need to send multiple messages containing the same data.

Logix controllers are based on 32-bit operations. The types of data that can be a base tag are BOOL, SINT, INT, DINT, and REAL, as illustrated in Figure 15-16 and listed below. The controller stores all data in a minimum of 4 bytes or 32 bits of data.

- A **BOOL** or Boolean base tag is 1 bit of data stored in bit 0 of a 4 byte memory location. The other bits, 1 to 31, are unused. BOOLs have a range of 0 to 1, off or on respectively.

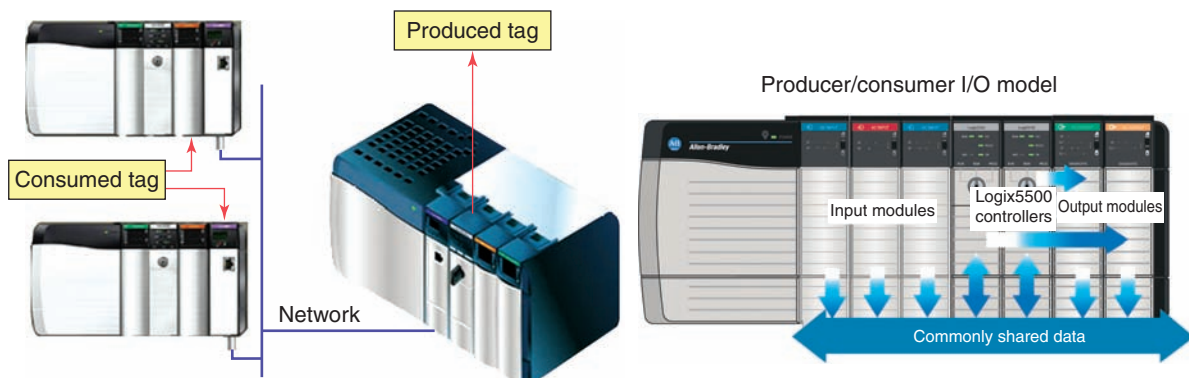


Figure 15-15 Produced/consumed tags used to share information.

Source: Image Used with Permission of Rockwell Automation, Inc.

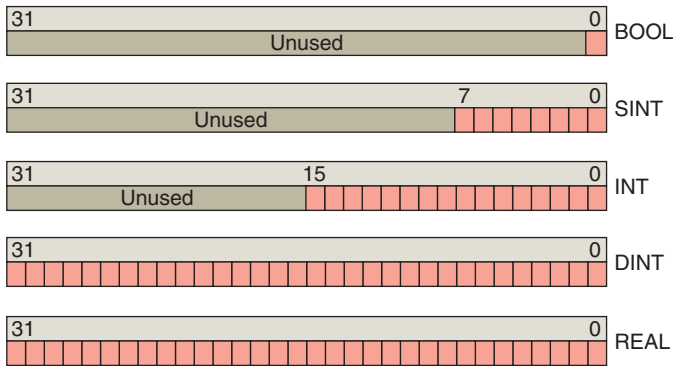


Figure 15-16 Types of base tag data.

- A **SINT** or Single Integer base tag uses 8 bits of memory and stores the data in bits 0 to 7. These bits are sometimes called the low byte. The other 3 bytes, bits 8 to 31, are unused. SINTs have a range of -128 (negative values) to 127 (positive values).
- An **INT** or Integer base tag is 16 bits, bits 0 to 15, sometimes called the lower bytes. Bits 16 to 31 are unused. INTs have a range between $-31,768$ and $32,767$.
- A **DINT** or Double Integer base tag uses 32 bits, or all 4 bytes, and has the following range: -2^{31} to $2^{31}-1$ ($-2,147,483,648$ to $2,147,483,647$).
- A **REAL** base tag also uses 32 bits of a memory location and has a range of values based on the IEEE Standard for Floating-Point Arithmetic.

Structures

There is another class of data types called structures. A structure-type tag is a grouping of different data types that function as a single unit and serve a specific purpose. An example of an RSLogix structure is shown in Figure 15-17. Each element of a structure is referred to as a member and each member of a structure can be a different data type.

	Name	Data Types	Style	Description
Members	PRE	DINT	Decimal	
	ACC	DINT	Decimal	
	EN	BOOL	Decimal	
	TT	BOOL		
	DN	BOOL		
	FS	BOOL	Decimal	
	LS	BOOL	Decimal	
	OV	BOOL	Decimal	
	ER	BOOL		

Figure 15-17 Structure-type tag.

Data type : COUNTER				
Name	Counter			
Description				
Members	Data type size : 12 byte(s)			
	Name	Data Type	Style	Description
	PRE	DINT	Decimal	
	ACC	DINT	Decimal	
	CU	BOOL	Decimal	
	CD	BOOL	Decimal	
	DN	BOOL	Decimal	
	OV	BOOL	Decimal	
	UN	BOOL	Decimal	

Figure 15-18 Predefined structure.

There are three different types of structures in a ControlLogix controller: *predefined*, *module-defined*, and *user-defined*. The controller creates *predefined* structures for you that include timers, counters, messages and PID types. An example of a predefined counter instruction structure is shown in Figure 15-18. It is made up of the preset value, the accumulated value, and the instruction's status bits.

Module-defined structures are automatically created when the I/O modules are configured for the system. When you add input or output modules a number of defined tags are automatically added to the controller tags. Figure 15-19 shows the two tags (Local:1:C and Local:1:I) created after a digital input module has been

Controller Tags - controller3(controller)				
Scope: controller3		Show...	Show All	
Name	Value	Force Mask	Style	Data Type
+ Local:1:C				AB:1756_DI_AC...
+ Local:1:I				AB:1756_DI_AC...
Monitor Tags / Edit Tags				
Name	Value	Force Mask	Style	Data Type
- Local:1:C	{...}	{...}		AB:1756_DI_AC...
- Local:1:C.Di...	0		Decimal	BOOL
+ Local:1:C.Fil...	1		Decimal	SINT
+ Local:1:C.Fil...	9		Decimal	SINT
+ Local:1:C.C...	2#0000_000...		Binary	DINT
+ Local:1:C.C...	2#0000_000...		Binary	DINT
+ Local:1:C.F...	2#0000_000...		Binary	DINT
+ Local:1:C.O...	2#0000_000...		Binary	DINT
+ Local:1:C.Fi...	2#0000_000...		Binary	DINT
- Local:1:I	{...}	{...}		AB:1756_DI_AC...
+ Local:1:I.Fault	2#0000_000...		Binary	DINT
+ Local:1:I.Data	2#0000_000...		Binary	DINT
+ Local:1:I.CS...	{...}	{...}	Decimal	DINT[2]
+ Local:1:I.Op...	2#0000_000...		Binary	DINT
+ Local:1:I.Fie...	2#0000_000...		Binary	DINT

Figure 15-19 Module-defined structure for a digital input module.

Name: Size: byte(s)

Description:

	Name	Data Type	Style	Description
	Level	INT	Decimal	Stores the Level in Inches
	Pressure	DINT	Decimal	Stores the Pressure in PSIG
	Temp	REAL	Float	The Temperature in F
	Agitator_Speed	DINT	Decimal	Speed in RPM
*				

Figure 15-20 User-defined storage tank structure.

added. Tags of these types are created to store input, output, and configuration data for the module. Input tags labeled Data contain the actual input bits from the module. Configuration tags determine the characteristics and operation of the module. The name Local indicates that these tags are in the same rack as the processor. The 1 indicates that the module occupies slot 1 in the chassis. The letters I and C indicate whether the data are input data or configuration data.

A *user-defined* structure supplements the predefined structures by providing the ability to create custom-defined structures to store and handle data as a group. Figure 15-20 illustrates a user-defined structure that contains data for a storage tank. All data relative to the tank are stored together. In the design stage the programmer creates a generic user-defined memory structure that contains all the different aspects of the storage tank. Each member has a meaningful name and is created in the appropriate data type and style like REAL (floating point) for temperature and DINT (decimal) for agitator speed

in rpm. Installation and maintenance personnel can easily locate all data associated with the operation of the tank since all the information is stored together.

Creating Tags

There is more than one way to create tags. You may create tags in the tag editor before your program is entered, enter tag names as you program, or use question marks [?] in place of tag names and assign the tags later. Figure 15-21 shows an example of a controller scope base tag created in the new tag dialog box. When defining tags, the following information has to be specified:

- A *tag name*, which must begin with an alphabetic character or an underscore (_). Names can contain only alphabetic characters, numeric characters, or underscores and may be up to 40 characters in length. They may not have consecutive or trailing underscore characters, are not case sensitive and cannot have spaces in the tag name.
- An optional *tag description*, which may be up to 120 characters in length.
- The *tag type*: base, alias, or consumed.
- The *data type*, which is obtained from the list of predefined or user-defined data types.
- The scope in which to create the tag. Your options are the controller scope or any one of the existing program scopes.
- The *display style* to be used when monitoring the tag in the programming software. The software will display the choices of available styles.
- Whether or not you want to make this tag available to other controllers and the number of other controllers that can consume the tag.

Figure 15-21 Controller scope base tag.

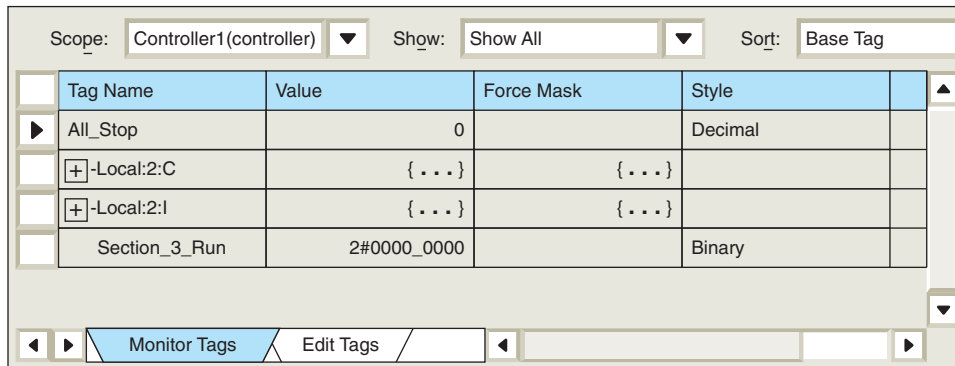


Figure 15-22 Monitor Tags window.

Monitoring and Editing Tags

After tags have been created they can be monitored using the Monitor Tags window displayed in Figure 15-22. When *Monitor Tags* is selected the actual value(s) for the tags will be shown. The Force Mask column is used to force inputs and outputs when troubleshooting. You can also create new tags or edit existing tags using the Edit Tags window displayed in Figure 15-23. When *Edit Tags*

is selected new tags may be created, and existing tag properties may be modified.

Array

Many control programs require the ability to store blocks of information in memory in the form of tables that can be accessed at runtime. An *array* is a tag type that contains a block of multiple pieces of data. Each element

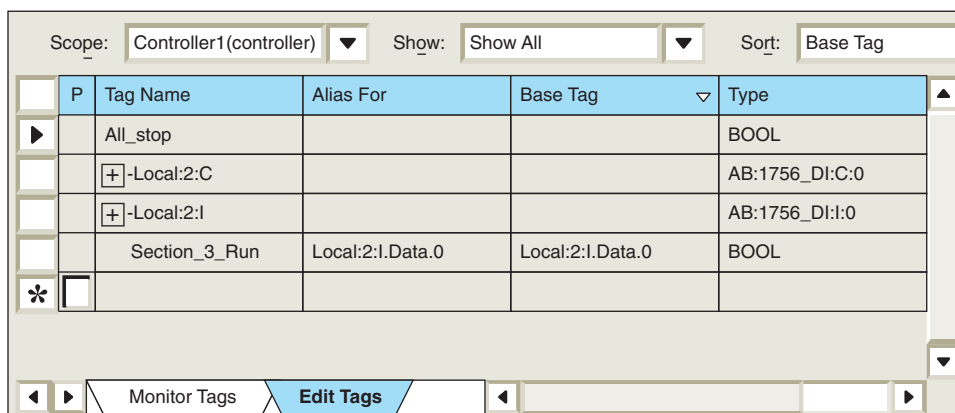


Figure 15-23 Edit Tags window.

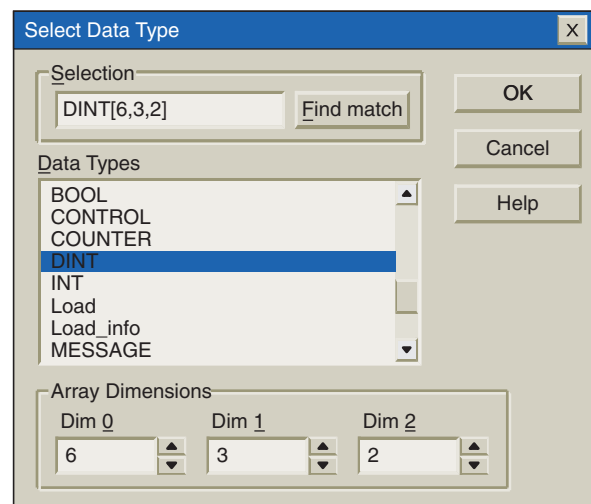
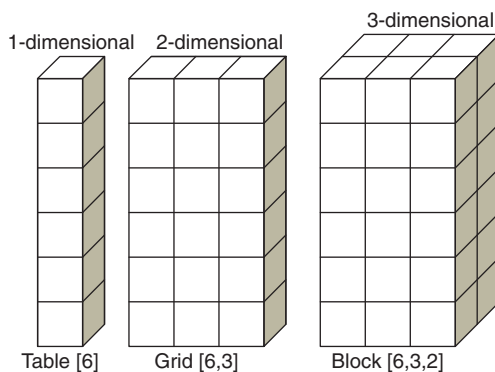


Figure 15-24 Types of arrays.

of an array must be of the *same data type* (e.g., BOOL, SINT, or INT). An array occupies a contiguous block of controller memory. Arrays are similar to tables of values. The use of arrayed data types offers the fastest data throughput (output) from a ControlLogix processor. Because arrays are numerically sequenced tags of the same data that occupy a contiguous memory location, large amounts of data can be retrieved efficiently. Arrays can be built using 1, 2 or 3 dimensions, as illustrated in Figure 15-24, to represent the data they are intended to contain.

A single tag within the array is one element. The element may be a basic data type or a structure. The elements start with 0 and extend to the number of elements minus 1. Figure 15-25 is an example of the memory

Array - Temp
Data Type - INT[5]

Temp[0]	297
Temp[1]	200
Temp[2]	180
Temp[3]	120
Temp[4]	100

Figure 15-25 Memory layout for a 1-dimensional array.

layout for a 1-dimensional (one column of values) array created to hold five temperatures. The tag name is Temp and the array consists of 5 elements numbered 0 through 4.



PART 1 REVIEW QUESTIONS

1. Compare the memory configuration of a Logix 5000 controller with that of an SLC 500 controller.
2. What does a project contain?
3. List four programming functions that can be carried out using the program organizer.
4. Explain the function of tasks within the project.
5. State the three main types of tasks.
6. What type of tasks function as timed interrupts?
7. Explain the function of programs within the project.
8. Explain the function of routines within the project.
9. Which routine is configured to execute first?
10. Name the four types of programming languages that can be used to program Logix 5000 controllers.
11. What are tags used for?
12. Compare the accessibility of program scope and controller scope tags.
13. Name the tag type used for each of the following:
 - a. Create an alternate name for a tag.
 - b. Share information over a network.
 - c. Store various types of data.
14. What is the difference between a produced tag and a consumed tag?
15. List the five types of base tag data.
16. State the data type used for each of the following:
 - a. 32-bit memory storage
 - b. On/Off toggle switch
 - c. 16-bit memory storage
 - d. 8-bit memory storage
17. Describe the make-up of a predefined structure.
18. Describe the make-up of a module-defined structure.
19. Describe the make-up of a user-defined structure.
20. Explain two ways of creating tags.
21. When defining tags what limitations are placed on the entering of a tag name?
22. What is meant by the tag display style?
23. Write an example of an array tag used to hold 4 speeds.

Part 2 Bit-Level Programming

Part Objectives

After completing this part, you will be able to:

- Know what happens during the program scan
- Demonstrate an understanding of input, output, and internal relay addressing format for a tag-based Logix controller
- Develop ladder logic programs with input instructions and output coil combinations
- Develop ladder logic programs with latched outputs

Program Scan

When a CLX controller executes a program, it must know—in real time—when external devices controlling a process are changing. During each operating cycle, the processor reads all the inputs, takes these values, and energizes or de-energizes the outputs according to the user program. This process is known as the *program scan*.

Figure 15-26 illustrates the signal flow into and out of a Logix controller during a controller's operating cycle when ladder logic is executing. During the program scan, the controller reads rungs and branches from left to right and top to bottom as follows:

- Only one rung at a time is scanned.
- As the program is scanned, the status of inputs are checked for True (1 or ON) or False (0 or OFF) conditions.

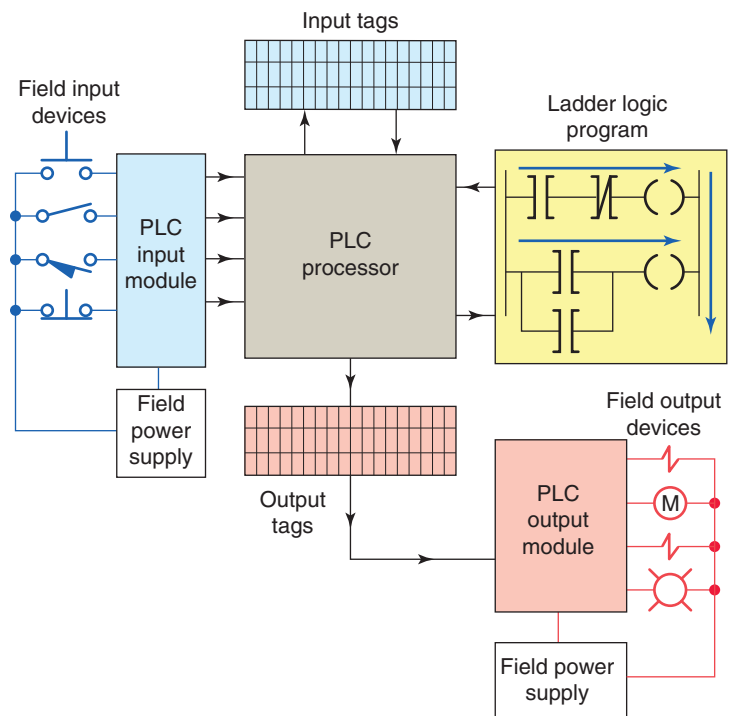


Figure 15-26 Logix controller operating cycle.

- The status signals from the inputs are sent to the input tags where they are stored.
- As the program is scanned by the processor, inputs are checked for True or False conditions and the ladder logic is evaluated based on these values.
- The resulting ON or OFF action, as a result of evaluating each rung, is then sent to the output tags for storage.
- During the output update portion of the scan, corresponding output values are sent to the process or machine by way of the output module.

- I/O updates occur asynchronously to the scan of the logic. With a ControlLogix processor two separate 32-bit unsynchronized processes gone on simultaneously—that is, asynchronously. This means that the module can update the input tag from the field and write the output tag to the field at any point (or at several points) during the processor's execution of the ladder rungs. The result is more efficiency and control over when the input field device data are updated in the input tag and when the output data resulting from the solved logic are sent to the output modules and their respective field devices.

Creating Ladder Logic

Although other programming languages are available, ladder logic is the most common programming language for PLCs. The instructions in ladder logic programming can be divided into two broad categories: input and output instructions. The most common input instruction is equivalent to a relay contact and the most common output instruction is the equivalent of a relay coil (Figure 15-27). When creating ladder I/O bit instructions, the following rules apply:

- All input instructions must be to the left of an output instruction.
- A rung cannot begin with an output instruction if it also contains an input instruction. This is because the controller tests all inputs for true or false before deciding what value the output instruction should be.
- A rung does not need to contain any input instructions, but it must contain at least one output instruction.

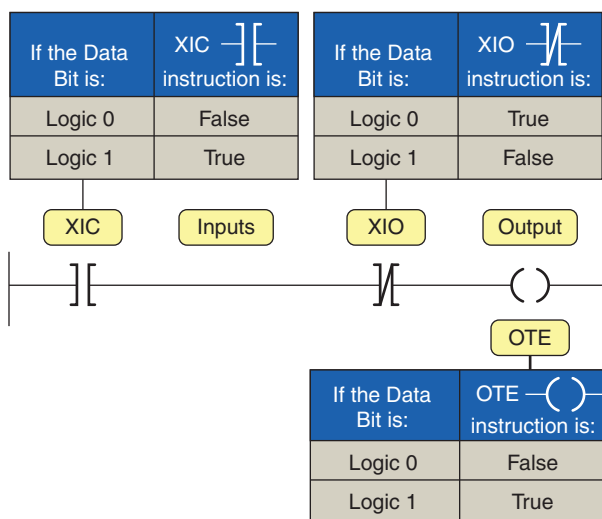


Figure 15-27 Contacts and coil instructions.

- When a rung has only one output instruction it will always be true.
- The last instruction on a rung must always be an output instruction.
- The XIC, or Examine If Closed contact instruction, checks to see if the input has a value of one. If the input is one, the XIC instruction returns a true value.
- The XIO, or Examine If Open contact instruction, checks to see if the input has a value of zero. If the input is zero, the XIO instruction returns a true value.
- The OTE or Output Energize coil instruction sets the tag associated with it to true or one when the rung has logic continuity. When true it can be used to energize an output device or simply set a value in memory to one.

ControlLogix PLCs support multiple outputs on one rung. CLX controllers allow the use of serial logic that does not conform to traditional electrical hardwired circuits or ladder logic. For example, both of the rungs shown in Figure 15-28 are valid in RSLogix 5000. However the series connection of outputs would not work if wired that way in an equivalent electrical circuit or programmed that way in RSLogix 500. In both instances in RSLogix 5000, instructions tagA and tagB must be true to energize output tag1 and tag2.

In ControlLogix output instructions can be placed between input instructions as illustrated in Figure 15-29. In this example instructions tagA and tagB must be true to energize output tag1. Instructions tagA and tagB and tagC must all be true before output tag2 is set to energize.

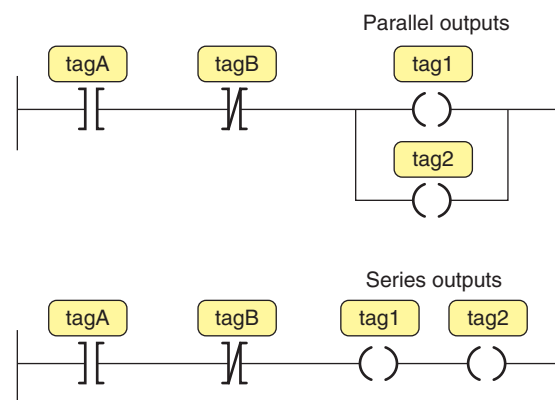


Figure 15-28 Parallel and series outputs.

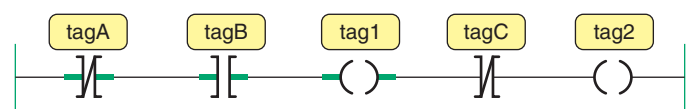


Figure 15-29 Output instruction placed between input instructions.

Tag-Based Addressing

Logix 5000 controllers use a tag-based addressing structure. A tag is a text-based name for an area of the controller where data is stored. An example of how a tag-based address is implemented using a ControlLogix controller is shown in Figure 15-30. Tag names use a meaningful description of the variable. In this application when the normally closed high limit switch is activated the program will switch the high limit output light on. The addressing format can be summarized as follows:

- The physical address for the tag Limit_switch is Local:1:I.Data.2(C). Local indicates that the module is in the same rack as the processor, 1 indicates that the module is in slot 1 in the rack, I indicates that the module is an input type, Data indicates that it

- is a digital input, 2 indicates that the limit switch is connected to terminal 2 on the module, and C indicates that it is a controller tag with global access.
- The physical address for the tag High_limit_light is Local:2:O.Data.4(C). Local indicates that the module is in the same rack as the processor, 2 indicates that the module is in slot 2 in the rack, O indicates that the module is an output type, Data indicates that it is a digital input, 4 indicates that the high limit light is connected to terminal 4 on the module, and C indicates that it is a controller tag with global access.

One advantage of the use of tag-based addressing is that the allocation of variable names for program values is not tied to specific memory locations in the memory structure, as is the case with rack/slot and rack/group type

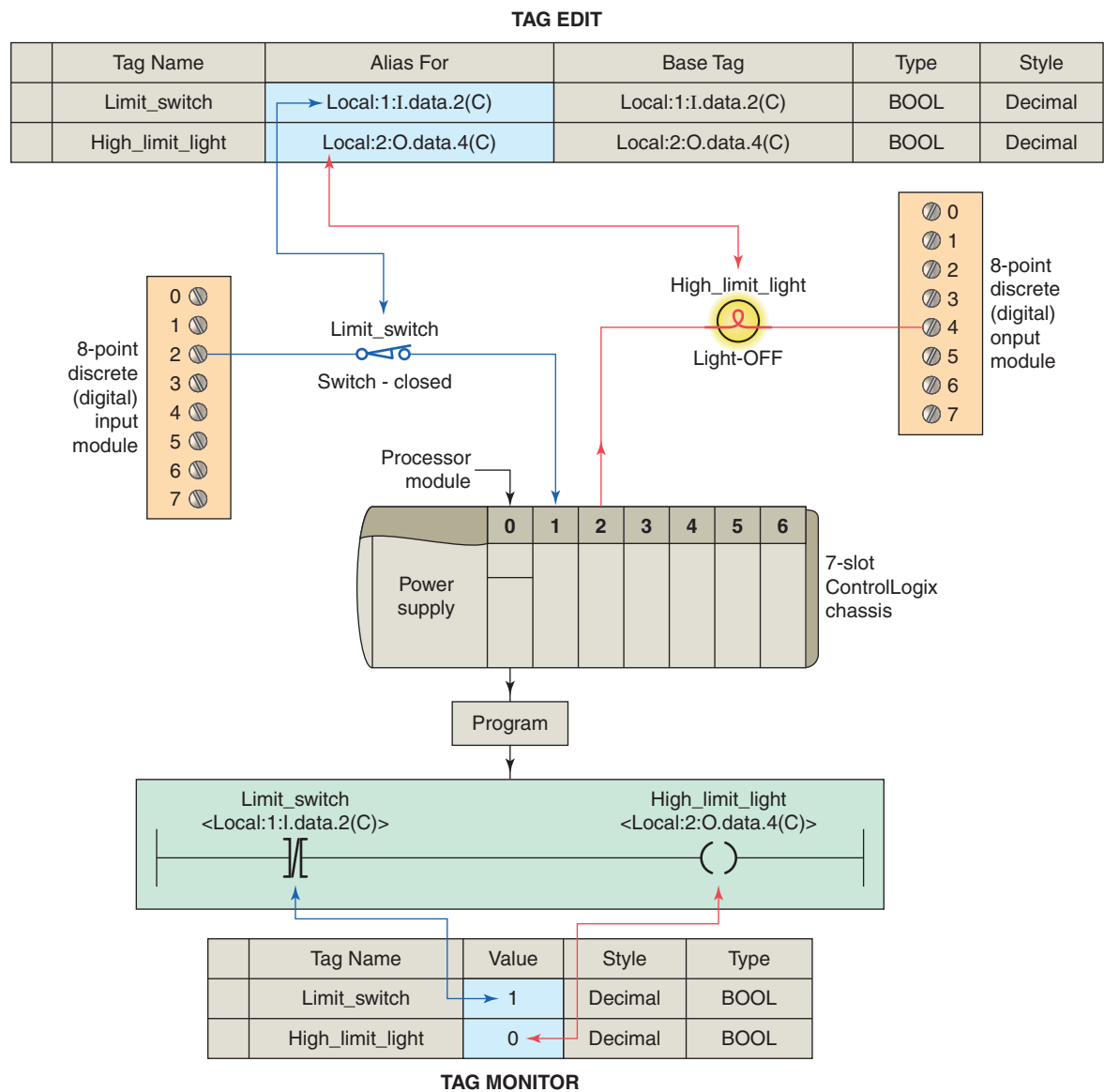


Figure 15-30 Tag-based address implementation.

systems. Initially, all program development can proceed with just the tag names and data types assigned. Using tag aliases, programmers can write code independent of electrical connection assignments. At a later date, input and output field devices are easily matched to the pin numbers on the respective module they are connected to.

Adding Ladder Logic to the Main Routine

Figure 15-31 shows the diagram for a hardwired contactor operated motor start/stop control circuit. The normally open start button is momentarily closed to energize the contactor coil and close its main contacts to start the motor. The seal-in auxiliary contact of the contactor is connected in parallel with the start button to keep the starter coil energized when the start button is released. The normally closed stop button is momentarily opened to de-energize the contactor coil and stop the motor.

Figure 15-32 shows the ladder logic program for the motor start/stop control circuit and the RSLogix 5000 toolbar used to create it. Free form editing found in RSLogix 5000 helps speed development in that you do not have to place an instruction and tie an address to the instruction

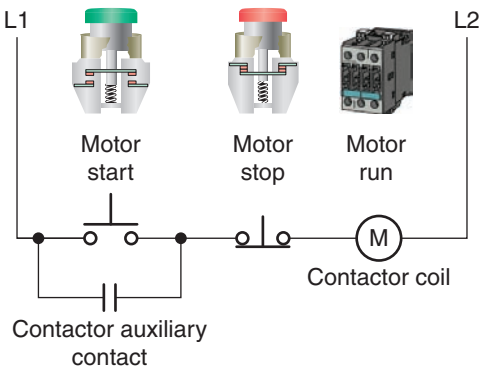


Figure 15-31 Hardwired motor start/stop control circuit.

before adding the next instruction. In this example we have chosen to use question marks [?] in place of tag names and assign the tags later. Field device wiring for the two push-button inputs and the single contactor coil output are as illustrated. The stop button is connected to terminal 3 and the start button to terminal 4 of the DC input module located in slot 1 of the rack. The contactor coil is connected to terminal 4 of the DC output module located in slot 2 of the rack. Both the start and stop buttons are examined for a closed condition (XIC) because both buttons must be closed to cause the motor starter to operate.

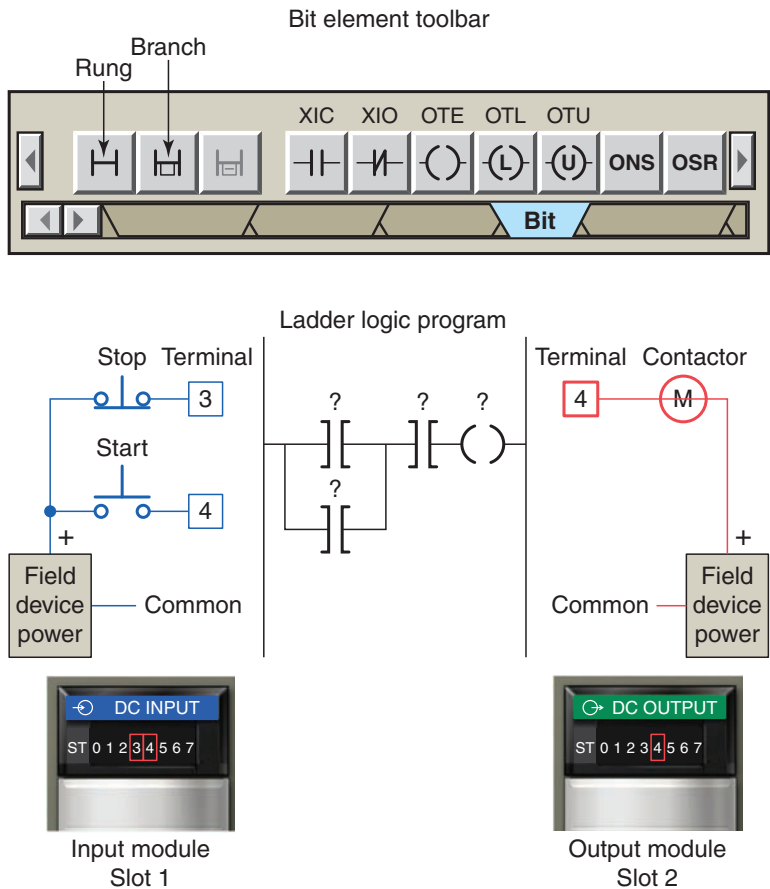


Figure 15-32 Programmed motor start/stop control circuit.

With text-based Logix systems you can use the name of the tag to document your ladder code and organize your data to mirror your application. For the programmed motor start/stop control circuit three tags Motor_Start, Motor_Stop, and Motor_Run are created. Figure 15-33 illustrates how the Motor_Start tag is created in the New Tag window. This window can be accessed by right clicking the ? mark above the XIC instruction in the ladder logic program. Since this tag represents a value from an input field device a link through the module to the field device must be created. When Local:1:I.Data is selected a dialog box for all of the terminal numbers on the input module appears. The tag name (Motor_Start) used in the program is then linked to input terminal number 3 where the field device represented by the tag name is connected.

Figure 15-34 shows what the ladder logic program would look like after all three tags have been created. Users have the ability to reference data via multiple names using Aliases. This allows the flexibility to name data differently depending on their use. The tag description provides for a more meaningful description of the tag name. Tag names are downloaded and stored in the controller

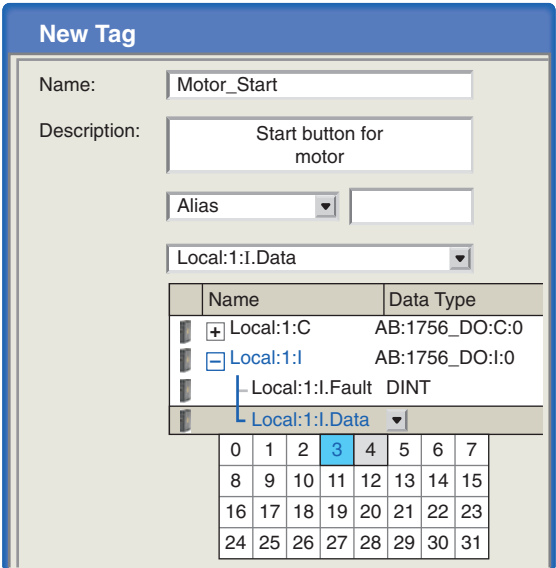


Figure 15-33 Creating the Motor_Start tag.

but the description is not as it is part of the documentation of the project.

Figure 15-35 shows the state of the tags created for the motor start/stop program as seen in the program and

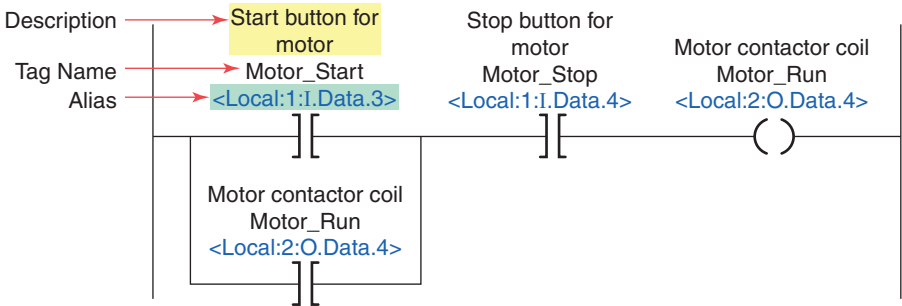


Figure 15-34 Ladder logic program after all tags have been created.

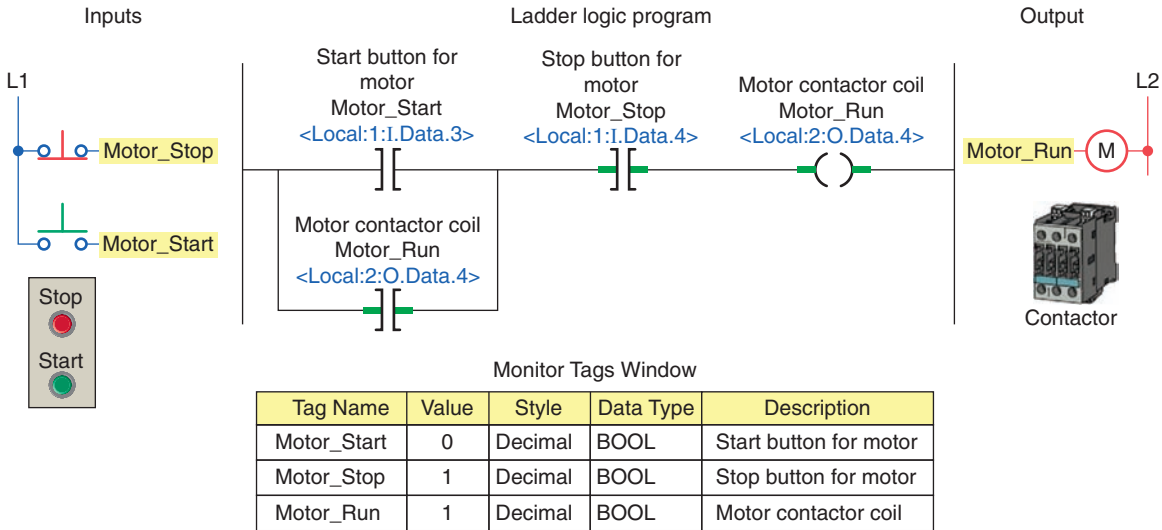


Figure 15-35 Ladder logic program and Monitor Tags window with motor operating.

Monitor Tags window, when the motor is operating. When the motor is operating:

- The XIC Motor_Start instruction is false because the NO start button is open; therefore its value is 0.
- The XIC Motor_Stop instruction is true because the NC stop button is closed; therefore its value is 1.
- The OTE Motor_Run instruction is true because the rung has logic continuity; therefore its value is 1.

Internal Relay Instructions

Internal relay instructions are used when other than real-world field devices are needed as input or output reference instructions. For example, an internal relay bit is used as an output when the logical resultant of a rung is used to control other internal logic. An internal control relay is programmed in the ControlLogix system by creating a tag (either program or controller type) and assigning a Boolean type to the tag.

Figure 15-36 shows a ControlLogix program that uses an internal relay to implement on/off control of a room light from three different entrances or positions. Three single pole switches are used for inputs in place of the two 3-way and one 4-way switches normally required for

an equivalent hardwired control circuit. The operation of the program can be summarized as follows:

- An internal relay is used to execute the logic of the circuit without having to use a real-world output.
- The status value stored in memory for all tags, when all input switches are open, is 0 and so the room light will be off.
- Closing Position_1_Switch changes the status of its XIC instruction from false to true thereby establishing logic continuity for Rung 1.
- As a result, the status of the internal relay coil and its XIC contact change from false to true.
- This establishes logic continuity for Rung 2 and switches the room light on.
- A change in the state of any of input switches will change the current state of the light.

Latch and Unlatch Instructions

The *output latch (OTL)* instruction is a retentive output instruction that is used to maintain, or latch, an output. If this output is turned on, it will stay on even if the status

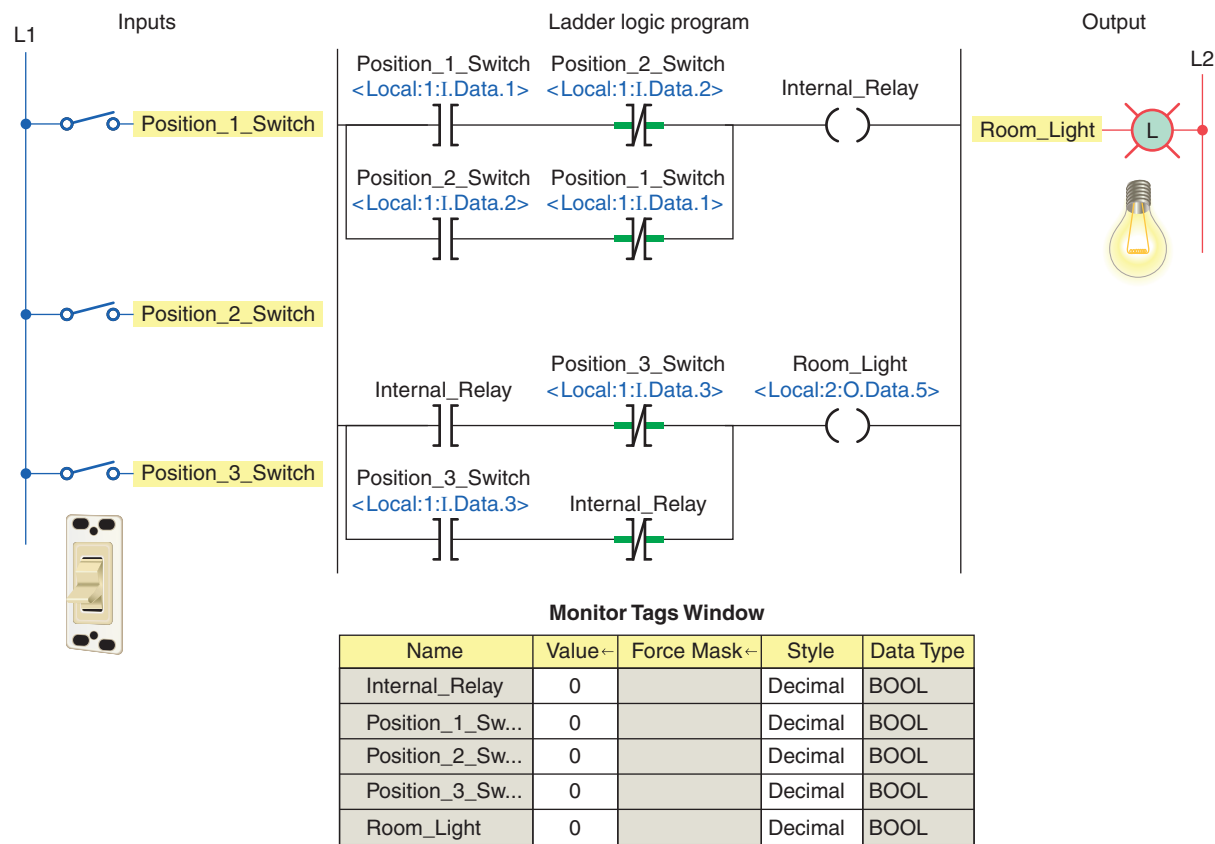


Figure 15-36 Internal relay to implement on/off control of a room light from three different entrances.

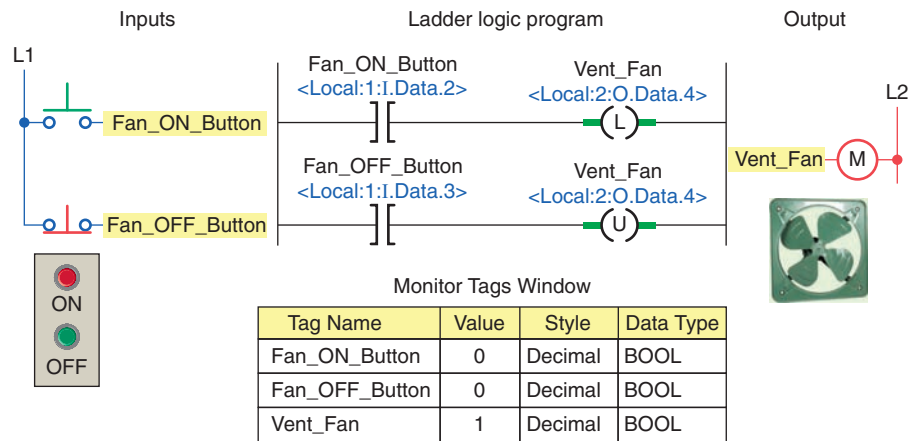


Figure 15-37 Output latch and unlatch instructions used to control a vent fan motor.

of the input logic that caused the output to energize becomes false. The OTL instruction will remain in a latched on condition until an unlatch instruction (OTU) with the same referenced tag is energized. The OTL instruction is often used in programs where the value of a variable must be maintained in instances where there is a shutdown due to a power failure or system fault. Retentive memory permits the system to be restarted with memory locations holding the values that were present when the program execution was halted.

Figure 15-37 shows a ControlLogix program that uses an output latch and unlatch instruction pair to implement the control of a vent fan motor. The operation of the program can be summarized as follows:

- The OTL instruction will write a 1 to its address when true.
- When the OTL goes false, the output address will remain a 1.
- This is true even if the processor powers down and then back up.
- The output address will remain a 1 until reset to 0 by the unlatch instruction.

- If the output address is off, both the latch and unlatch instructions are not intensified, but once the bit is turned on, you will see both the latch and unlatch intensified even though both inputs are shut off.

One-Shot Instruction

The CLX *One-Shot (ONS)* instruction is an input instruction used to turn an output on for one program scan only. The program of Figure 15-38 uses the ONS instruction with a math instruction to perform a calculation once per scan. This program is used to execute the ADD math function only once per actuation of the limit switch, no matter how long the limit switch is held closed. The operation of the program can be summarized as follows:

- On any scan for which *limit_switch_1* is cleared or *storage_1* is set, this rung has no effect.
- On any scan for which *limit_switch_1* is set and *storage_1* is cleared, the ONS instruction sets *storage_1* and the ADD instruction increments *sum* by 1.
- As long as *limit_switch_1* stays set, *sum* stays the same value. The *limit_switch_1* must go from cleared to set again for *sum* to be incremented again.

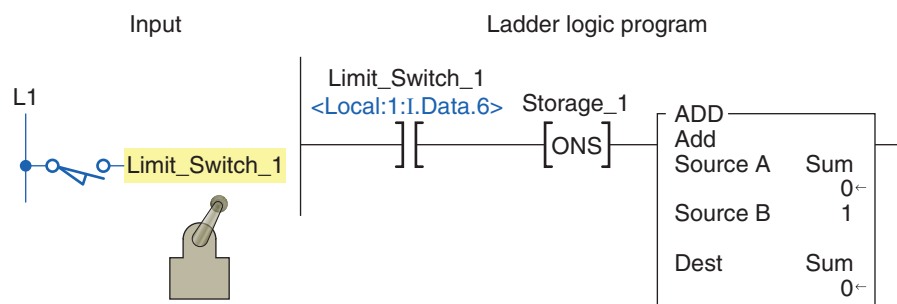


Figure 15-38 ONS instruction used to perform a calculation once per scan.



PART 2 REVIEW QUESTIONS

1. What operations are performed by the processor during the program scan?
2. With a ControlLogix processor I/O updates occur asynchronously. Explain what this means.
3. In ladder logic programming into what two broad categories can instruction types be classified?
4. A field input switch is examined using an XIC instruction.
 - a. What is the value (0 or 1) stored in its memory bit when the switch is opened and closed?
 - b. What is the state of the instruction (true or false) when the switch is opened and closed?
5. A field input switch is examined using an XIO instruction.
 - a. What is the value (0 or 1) stored in its memory bit when the switch is opened and closed?
 - b. What is the state of the instruction (true or false) when the switch is opened and closed?
6. The value of an OTE instruction as it appears in the Monitor Tags window is 1. Explain what this means as far as the status of a real-world field output and programmed XIC and XIO instructions associated with this tag are concerned.
7. Define a tag in the ControlLogix system.
8. What advantage do tag-based addressing systems have over rack/slot and rack/group types?
9. How is an internal relay programmed in the ControlLogix system?
10. The output latch instruction is a retentive output instruction. Explain what retentive means.
11. The ControlLogix ONS instruction is a one-shot instruction. Explain what this means.



PART 2 PROBLEMS

1. Modify the original ControlLogix start/stop motor control program with a second start and stop button added to the program. The additional start button is to be connected to pin 1 and the stop button to pin 2 of the digital input module.
2. Extend control of the original ControlLogix internal relay program used to control a room light from 3 entrances to 4. The additional single-pole switch is to be connected to pin 4 of the digital input module.
3. Implement the hardwired latching relay alarm circuit of Figure 15-39 in Logix format. The alarm will be latched on anytime:
 - The normally open temperature switch closes.
 - Both normally open float switches 1 and 2 close.
 - Either normally open sensor switch 1 or 2 closes while the normally closed pressure switch is closed.
4. Implement the hardwired tank filling and emptying operation shown in Figure 15-40 in Logix format.

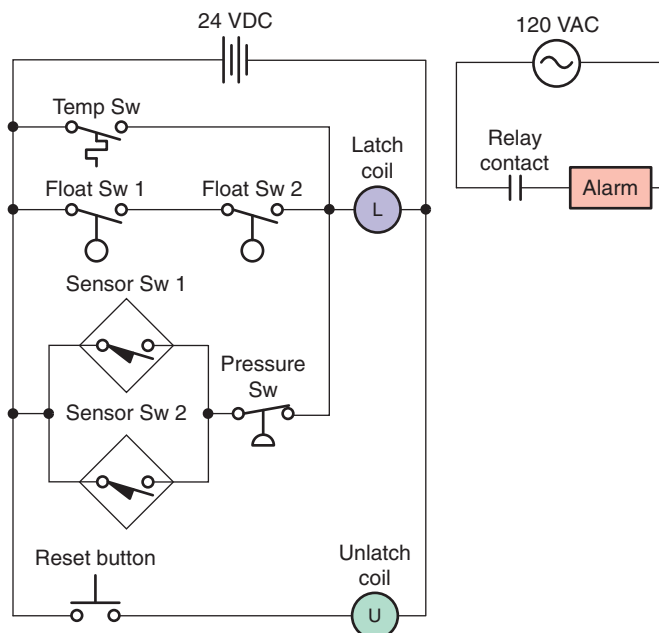


Figure 15-39 Hardwired latching relay alarm circuit for Problem 3.

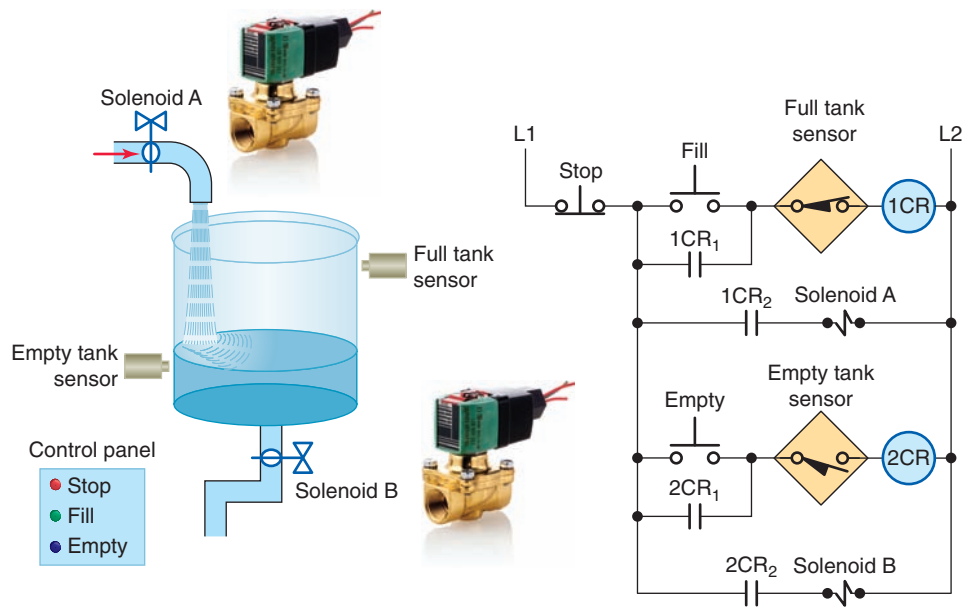


Figure 15-40 Hardwired tank filling and emptying operation for Problem 4.

Source: Photo courtesy ASCO Valve Inc., www.ascovalve.com.

The operation of the control circuit can be summarized as follows:

- Assuming the liquid level of the tank is at or below the empty level mark, momentarily pressing the FILL pushbutton will energize control relay 1CR.
- Contacts 1CR₁ and 1CR₂ will both close to seal in the 1CR coil and energize normally closed solenoid valve A to start filling the tank.
- As the tank fills, the normally open empty-level sensor switch closes.
- When the liquid reaches the full level, the normally closed full-level sensor switch opens to open the circuit to the 1CR relay coil and switch solenoid valve A to its de-energized closed state.
- Anytime the liquid level of the tank is above the empty-level mark, momentarily pressing the EMPTY pushbutton will energize control relay 2CR.
- Contacts 2CR₁ and 2CR₂ will both close to seal in the 2CR coil and energize normally closed solenoid valve B to start emptying the tank.
- When the liquid reaches the empty level, the normally open empty-level sensor switch opens to open the circuit to the 2CR relay coil and switch solenoid valve B to its de-energized closed state.
- The stop button may be pressed at any time to halt the process.

Part 3 Programming Timers

Timer Predefined Structure

Timers are used to turn outputs on and off after a time delay, turn outputs on or off for a set amount of time, and keep track of the time an output is on or off. The timer address in the SLC 500 controller is a data table address or symbol, whereas the timer address in the ControlLogix controller is a predefined structure of the TIMER data type. The TIMER structure is shown in Figure 15-41. Timer parameters and status bits include:

- **Tag Name**—User-friendly tag name for the timer (e.g., Pump_Timer). If you want to use a timer, you must create a tag of type timer.
- **Preset (PRE)**—The number of time increments that the timer must accumulate to reach the desired time delay. Specifies the value (in milliseconds) which the timer must reach before the done bit (DN) changes state. The preset value is stored as a binary

Data Type: TIMER				
Name:	Pump_Timer			
Description:				
Members:	Data Type Size: 12 byte(s)			
	Name	Data Type	Style	Description
	PRE	DINT	Decimal	
	ACC	DINT	Decimal	
	EN	BOOL	Decimal	
	TT	BOOL	Decimal	
	DN	BOOL	Decimal	

Figure 15-41 TIMER predefined structure.

Part Objectives

After completing this part, you will be able to:

- Understand ControlLogix timer tags and their members
- Utilize status bits from timers in logic
- Develop ladder logic programs using ControlLogix timers

number (DINT). The time base is always 1 msec. For example, for a 3 second timer, enter 3000 for the PRE value.

- **Accumulator (ACC)**—The accumulator value is the number of milliseconds the instruction has been enabled. The accumulator value stops changing when ACC value = PRE value.
- **Enable Bit (EN)**—The enable bit indicates the TON instruction is enabled. The EN bit is true when the rung input logic is true, and false when the rung input logic is false.
- **Timer Timing Bit (TT)**—The timing bit indicates that a timing operation is in process. The TT bit is true only when the accumulator is incrementing. TT remains true until the accumulator reaches the preset value.
- **Done Bit (DN)**—The done bit indicates that accumulated value (ACC) is equal to the preset (PRE)

value. The DN bit signals the end of the timing process by changing states from false-to-true or from true-to-false depending on the type of time contact instruction used. The DN bit is the most commonly used timer status bit.

On-Delay Timer (TON)

The *on-delay timer (TON)* is a nonretentive output instruction used when the application requires an action to occur at some time after the rung conditions for the timer become true. The ControlLogix TON on-delay instruction and timer selection toolbar are shown in Figure 15-42. When you want to use a timer, you must create a tag of type TIMER (it is a predefined data type) and enter the preset and the accumulated value. The tag must be defined before the preset and accumulated values can be entered. A value can be entered for the accumulator while programming. When the program is downloaded this value will be in the timer for the first scan. If the TON timer is not enabled the value will be set back to zero. Normally zero will be entered for the accumulator value.

The timer tag name is declared using the new tag properties dialog box shown in Figure 15-43. Tag name, description (optional), tag type, data type, and scope are selected or typed to complete the validation. A descriptive tag name, such as Solenoid_Delay, makes it easier to know what function the timer serves in the control system.

The program of Figure 15-44 is an example of a 10000 ms (10 s) TON timer. Timers generate both word level (DINT) and bit level (BOOL) data and status. The operation of the program can be summarized with reference to the Monitor Tags window.

- The status of all instruction is shown after the timer input switch has been switched from off to on (1) and accumulated 5000 ms (5 s) of time.
- At this halfway point the EN bit is 1 since the rung is true, the TT bit is 1 since the accumulated value is

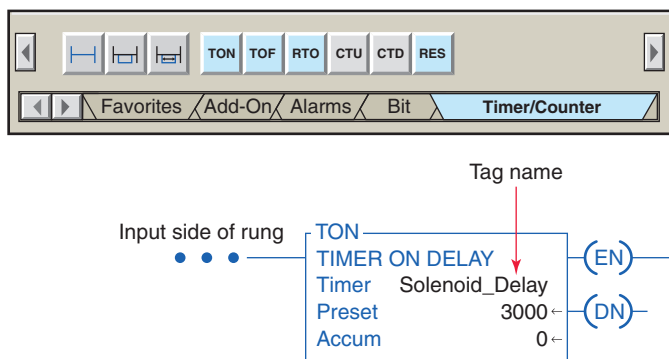


Figure 15-42 TON on-delay instruction.

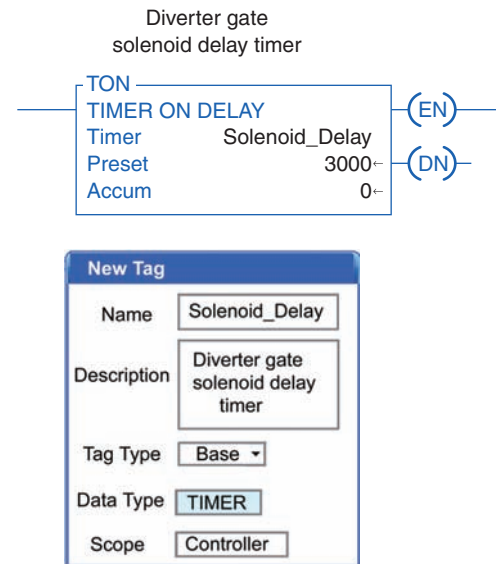


Figure 15-43 Timer tag validation.

changing, and the DN bit is 0 since the accumulated value does not yet equal the preset value.

- When the ACC equals PRE, the accumulated value stops incrementing, EN stays on for as long as the rung remains true, TT equals 0 since the accumulated value is not changing, and DN equals 1 since $ACC = PRE$.
- This will result in the DN pilot light switching on at the same time as the TT pilot light switches off.
- The EN pilot light remains on as long as the input switch is closed.
- Opening the input switch at any time causes the TON instruction to go false resetting the counter ACC value to 0 and EN, TT, and DN bits to 0. This in turn switches off all output pilot lights.
- The TON instruction is a self-resetting timer. When the rung goes false, the timer is automatically reset. A reset instruction can be used, but usually is not.

Figure 15-45 shows a TON timer used to delay the operation of a diverter gate solenoid for 3 seconds after a target has been sensed by the solenoid energize sensor. The operation of the program can be summarized as follows:

- Detection of the target causes closure of the SOL_Energize_Sensor contacts making the timer rung true and start timing.
- With passage of the target the SOL_Energize_Sensor contacts open but the rung remains true through the EN bit of the TON timer.
- After 3000 ms (3 s) delay time has elapsed, delay timer DN bit is set to 1 to energize the SOL_Gate.

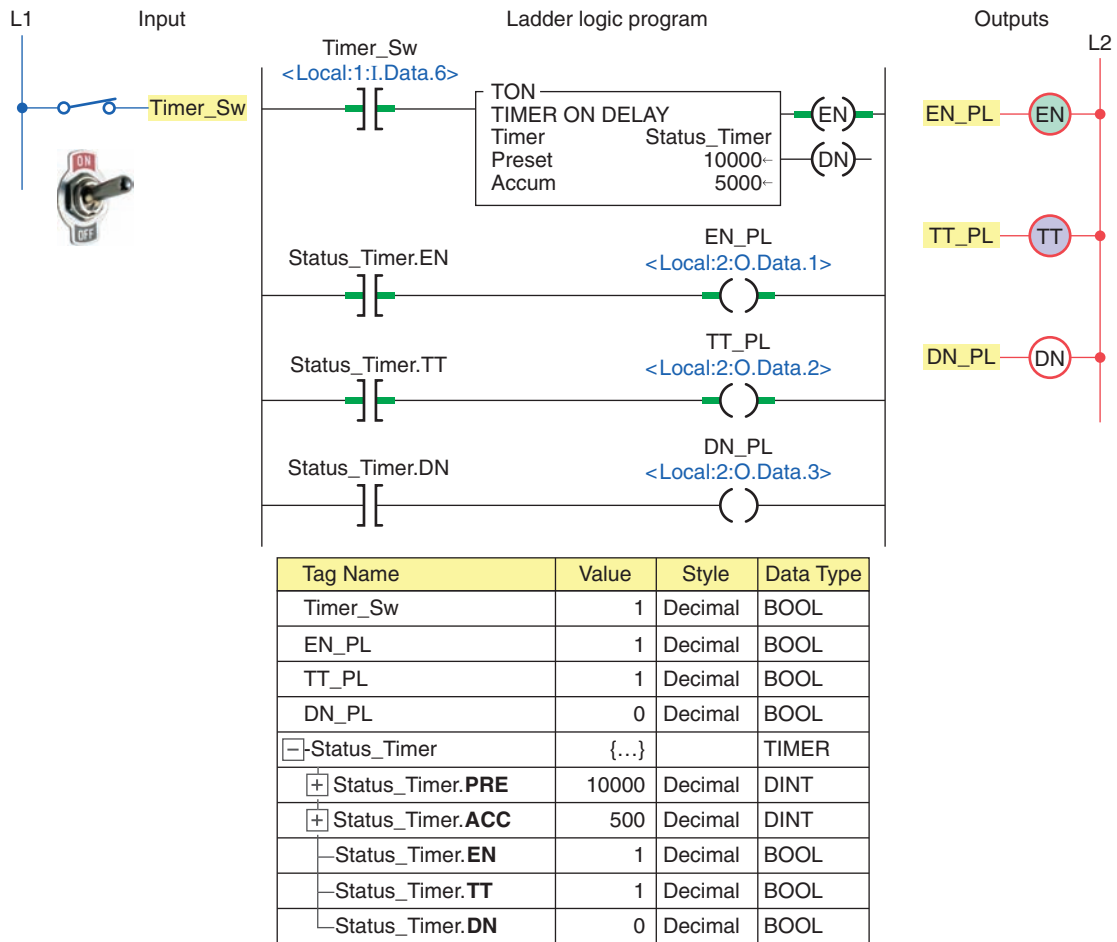


Figure 15-44 Ten-second TON timer program.

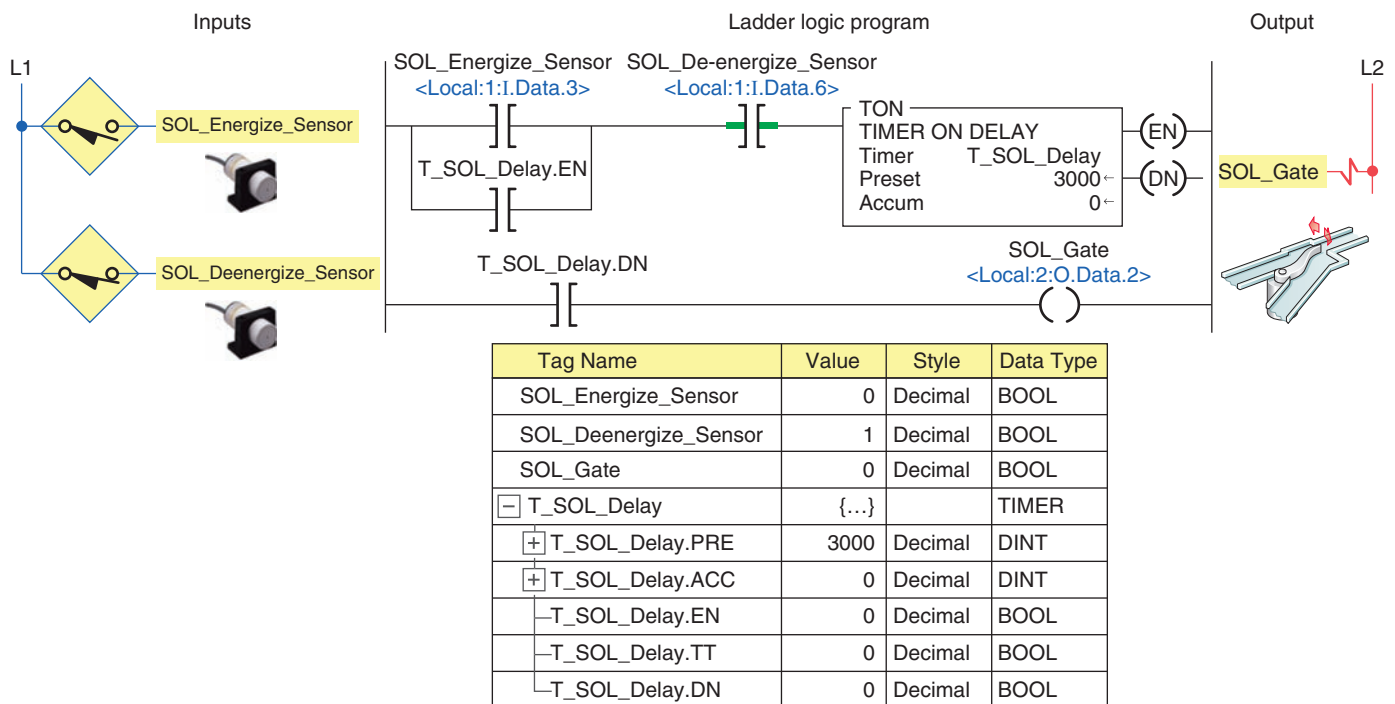


Figure 15-45 TON timer used to delay the operation of a diverter gate solenoid.

Source: Photos courtesy Omron Industrial Automation, www.ia.omron.com.

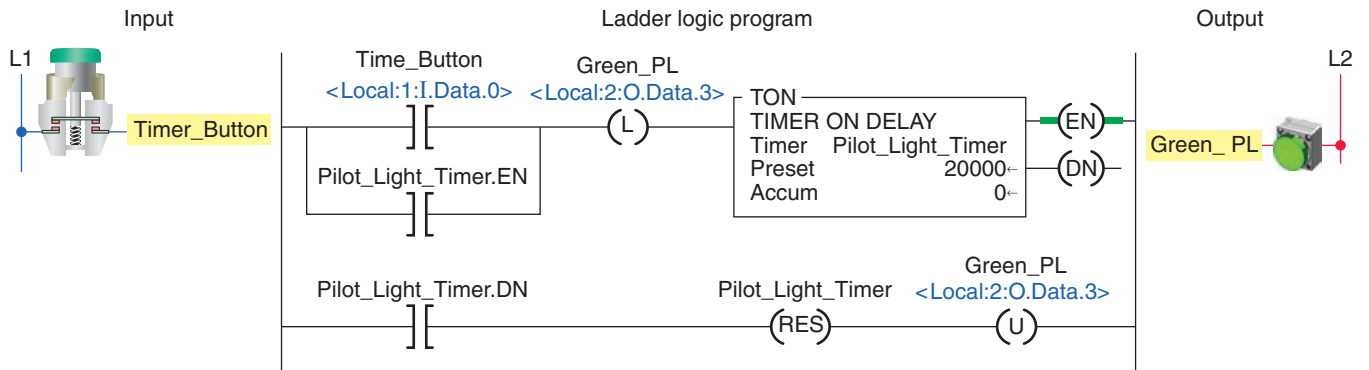


Figure 15-46 Pilot light TON timer.

- Momentary detection of the target by the SOL_ Deenergize_Sensor causes the opening of its contacts and resets the program to its original state.

Figure 15-46 shows a program that uses a TON timer to illuminate a green pilot light for 20 seconds each time a momentary button is pressed. In addition to the TON timer this program uses multiple outputs on one rung, output latch and unlatch instructions, as well as a timer reset instruction. The operation of the program can be summarized as follows:

- Initially closing the Timer_Button sets (latches) the Green_PL on and enables the Pilot_Light_Timer.

- When the button is then opened the timer rung remains true through the logic path created by the Pilot_Light_Timer.EN bit.
- After 20000 ms (20 s) have elapsed the timer DN bit is set to reset the timer to its original state and unlatch the Green_PL and switch it off.

The ControlLogix program of Figure 15-47 shows three TON timers cascaded (connected together) for traffic light control. The ladder logic used is the same as that used to program the traffic lights using the SLC 500 controller. The different tags created to fit the program are

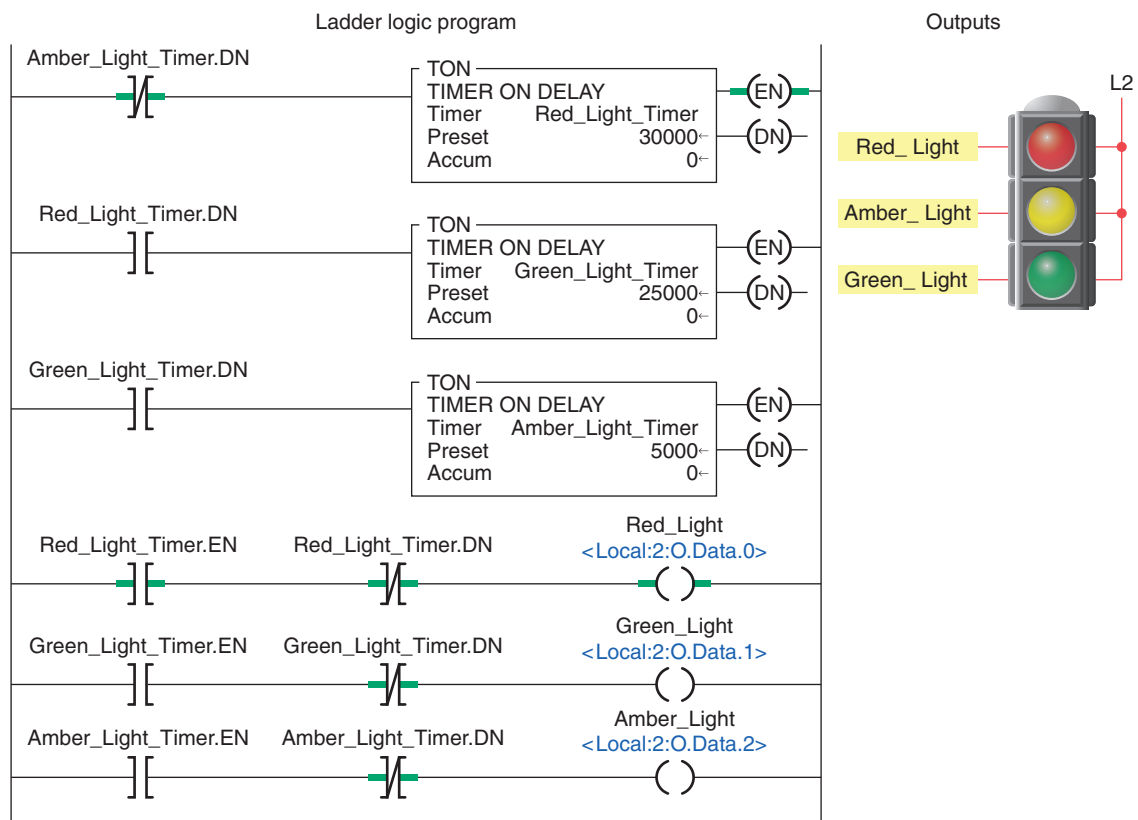


Figure 15-47 ControlLogix traffic control program.

Tag Name	Value	Style	Data Type
+Amber_Light_Timer	{...}		TIMER
+Green_Light_Timer	{...}		TIMER
-Red_Light_Timer	{...}		TIMER
+Red_Light_Timer.PRE	30000	Decimal	DINT
+Red_Light_Timer.ACC	0	Decimal	DINT
-Red_Light_Timer.EN	1	Decimal	BOOL
-Red_Light_Timer.TT	1	Decimal	BOOL
-Red_Light_Timer.DN	0	Decimal	BOOL
Red_Light	1	Decimal	BOOL
Green_Light	0	Decimal	BOOL
Amber_Light	0	Decimal	BOOL

Figure 15-48 Tags created for traffic light program.

shown in Figure 15-48. Operation of the program can be summarized as follows:

- Transition from red light to green light to amber light is accomplished by the interconnection of the EN and DN bits of the three TON timer instructions.
- The input to the Red_Light_Timer is controlled by the Amber_Light_Timer.DN bit.
- The input to the Green_Light_Timer is controlled by the Red_Light_Timer.DN bit.
- The input to the Amber_Light_Timer is controlled by the Green_Light_Timer.DN bit.
- The timed sequence of the lights is:
 - Red—30 s on
 - Green—25 s on
 - Amber—5 s on
- The sequence then repeats itself.

Off-Delay Timer (TOF)

The *off-delay timer (TOF)* operates in a fashion opposite to the TON on-delay timer. An off-delay timer will turn on immediately when the rung of ladder logic is true,

but it will delay before turning off after the rung goes false. The ControlLogix TOF off-delay timer instruction is shown in Figure 15-49. The description of the function block fields and tag references are the same as for that of a TON timer.

Figure 15-50 shows a program that uses a TOF timer to illuminate a green pilot light for 20 seconds each time a momentary button is pressed. The program code is simpler than that used to accomplish the same task using a TON timer. The operation of the program can be summarized as follows:

- When the Timer_Button is initially closed the timer rung and instruction and DN bit all become true.
- The DN bit switches on the Green_PL and the program remains in this state as long as the button is held closed.
- When the button is released the Timer_Button instruction goes false and starts the timing cycle.
- The light remains on and the timer begins accumulating time.
- When the accumulator reaches 20000 ms (20 s) the timer DN bit becomes false and the light is switched off.

The program of Figure 15-51 uses both on-delay and off-delay timers for control of a heating oven process. The different tags created to fit the program are shown

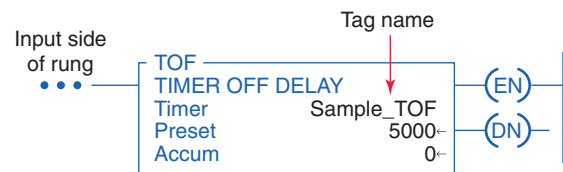


Figure 15-49 ControlLogix TOF off-delay timer instruction.

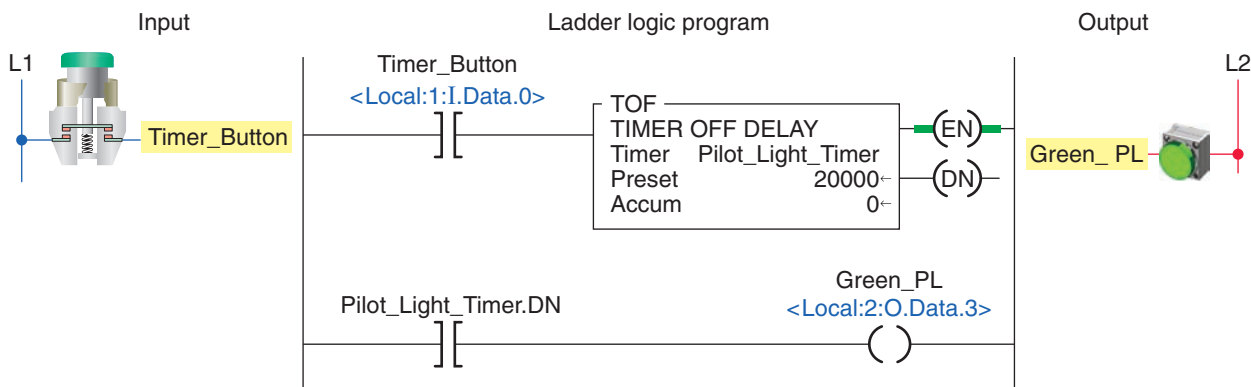


Figure 15-50 Pilot light TOF timer.

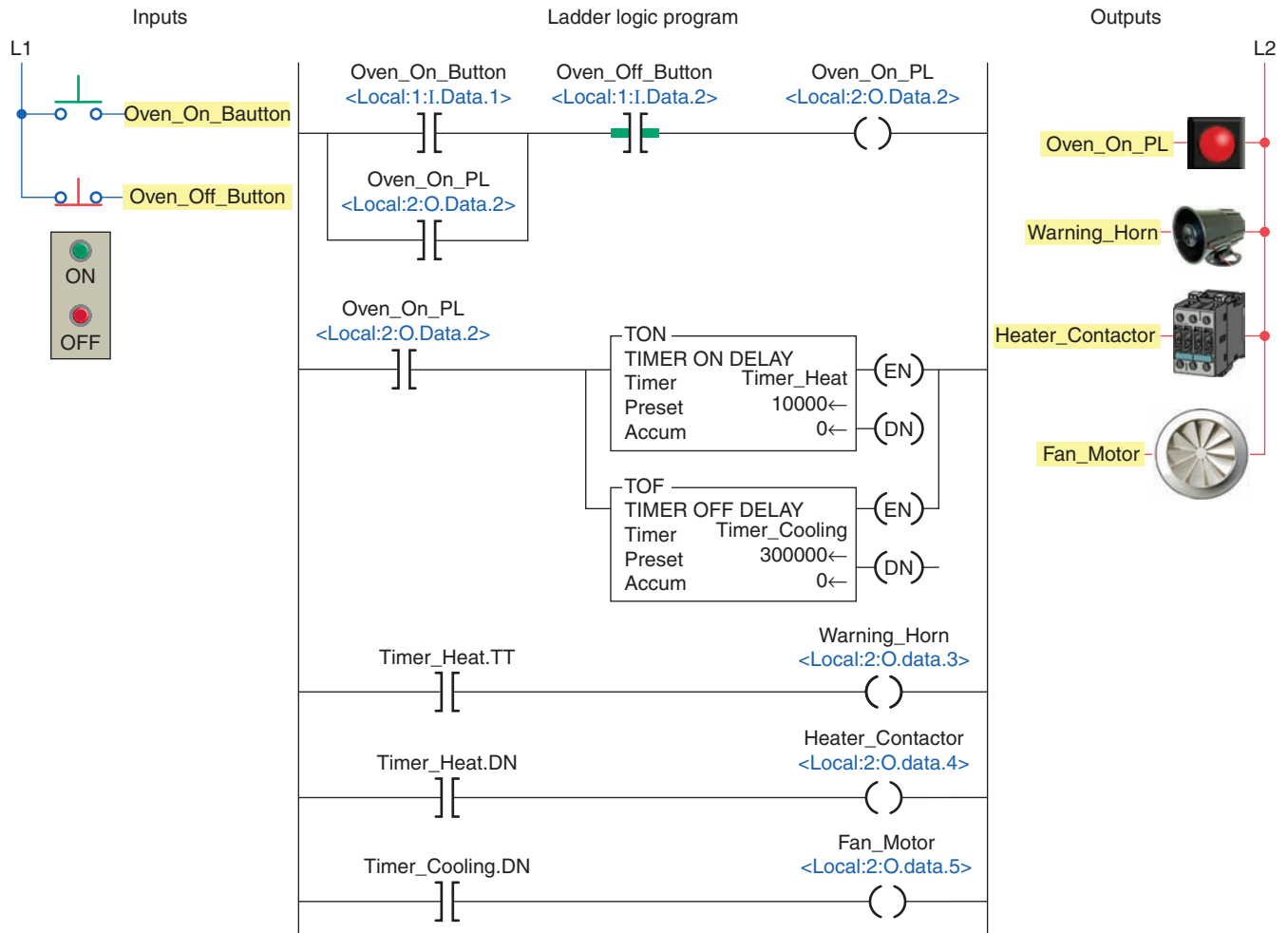


Figure 15-51 Timer control of a heating oven process.

Tag Name	Alias For	Base Tag	Data Type	Style
Warning_Horn	Local:2:O.Data.3	Local:2:O.Data.3	BOOL	Decimal
Heater_Contactor	Local:2:O.Data.4	Local:2:O.Data.4	BOOL	Decimal
Fan_Motor	Local:2:O.Data.5	Local:2:O.Data.5	BOOL	Decimal
Oven_On_PL	Local:2:O.Data.2	Local:2:O.Data.2	BOOL	Decimal
Oven_On_Button	Local:1:I.Data.1	Local:1:I.Data.1	BOOL	Decimal
Oven_Off_Button	Local:1:I.Data.2	Local:1:I.Data.2	BOOL	Decimal
+ -Timer_Heat			TIMER	
+ -Timer_Cooling			TIMER	

Figure 15-52 Tags created for heating oven process.

in Figure 15-52. Operation of the program can be summarized as follows:

- Pressing the **Oven_On_Button** energizes the **Oven_On_PL** output which seals itself in and enables the TON and TOF timer instructions.
- The **Timer_Heat.TT** bit of the TON timer becomes true which sounds the **Warning_Horn** to warn that the oven is about to come on.

- The **Timer_Cooling.DN** bit of the TOF timer becomes true which energizes the **Fan_Motor**.
- After 10 s (10000 ms) have elapsed the **Timer_Heat.TT** bit becomes false to turn off the **Warning_Horn** and the **Timer_Heat.DN** bit becomes true to energize the **Heater_Contactor** and turn on the heating coils.
- When the **Oven_Off_Button** is momentarily actuated the **Oven_On_PL** output goes false which turns the pilot light off and opens the continuity of its seal-in logic path.
- The **Timer_Heat** timer instruction and its **DN** bit instruction become false which de-energizes the **Heater_Contactor** and turns off the heating coils.
- The **Timer_Cooling** timer begins accumulating time and the fan continues to operate for the 5 minute (300000 ms) delay period after which the **Timer_Cooling.DN** bit becomes false to turn the fan off.

Retentive Timer On (RTO)

A *retentive on-delay timer (RTO)* operates the same as a TON timer, except that the retentive timer retains (remembers) its ACC value even if:

- The rung goes false.
- The processor is placed in the program mode.
- The processor faults.
- Power to the processor is temporarily interrupted and the processor battery is functioning properly.

The ControlLogix RTO retentive on-delay timer instruction is shown in Figure 15-53. The description of the function block fields and tag references are the same as for that of a TON timer; however, a RES reset instruction must be used to reset the accumulated value of a retentive timer. The RES instruction must have the same tag name as the timer you want to reset.

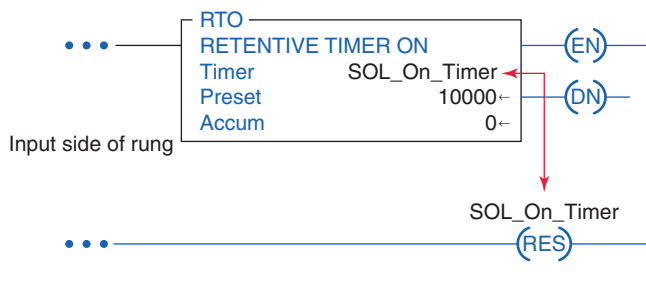


Figure 15-53 RTO retentive on-delay timer instruction.

An example application of a limit switch 2 minute (120000 ms) RTO timer program is shown in Figure 15-54. The different tags created to fit the program are shown in Figure 15-55. The operation of the program can be summarized as follows:

- The status and value of all instructions, with the timer initially reset, are as shown in the monitor tags window.
- When the Limit_Switch has been closed for 1 minute, the status and value of the instructions would be:
 - PRE – 120000
 - ACC – 60000
 - LS_Timer.EN – 1
 - LS_Timer.TT – 1
 - LS_Timer.DN – 0
 - LS_EN_PL – 1
 - LS_TT_PL – 1
 - LS_Alarm – 0
- When the Limit_Switch is opened after 1.5 minutes, the status and value of the instructions would be:
 - PRE – 120000
 - ACC – 90000
 - LS_Timer.EN – 0
 - LS_Timer.TT – 0
 - LS_Timer.DN – 0
 - LS_EN_PL – 0
 - LS_TT_PL – 0
 - LS_Alarm – 0

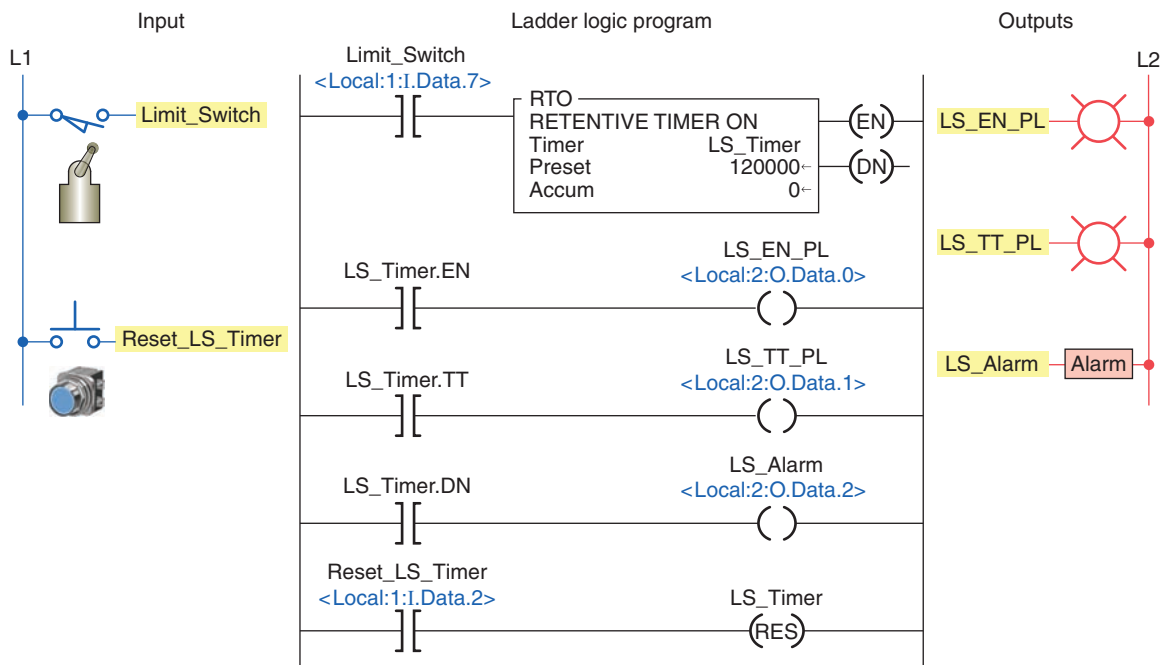


Figure 15-54 Limit switch RTO timer program.

Tag Name	Value	Style	Data Type
-LS_Timer	{...}		TIMER
-LS_Timer.PRE	120000	Decimal	DINT
-LS_Timer.ACC	0	Decimal	DINT
-LS_Timer.EN	0	Decimal	BOOL
-LS_Timer.TT	0	Decimal	BOOL
-LS_Timer.DN	0	Decimal	BOOL
Limit_Switch	0	Decimal	BOOL
LS_EN_PL	0	Decimal	BOOL
LS_TT_PL	0	Decimal	BOOL
LS_Alarm	0	Decimal	BOOL

Figure 15-55 Tags created for the RTO retentive on-delay timer program.

- When the Limit_Switch is closed and stays closed until the timer times out, the status and value of the instructions would be:
 - PRE – 120000
 - ACC –120000
- When the Limit_Switch is opened after the timer times out, the status and value of the instructions would be:
 - LS_Timer.EN – 1
 - LS_Timer.TT – 0
 - LS_Timer.DN – 1
 - LS_EN_PL – 1
 - LS_TT_PL – 0
 - LS_Alarm – 1
- When the Reset_LS_Timer is closed, the status and value of the instructions are reset to their original values.



PART 3 REVIEW QUESTIONS

1. Compare the methods used to address timers in an SLC 500 and a ControlLogix controller.
2. List the five different members of a TIMER structure.
3. What type of timing application may require you to use a TON on-delay timer?
4. What PRE value is used for a timer?
5. To what value is the accumulated value of a timer normally set?
6. What timer status bit is set to 1 when the TON timer times out?
7. The TON instruction is self-resetting. Explain what this means.
8. What number would be entered into the PRE value of a ControlLogix timer for a timing period of 4.5 minutes?
9. Compare the operation a TOF and a TON timer.
10. When does the rung of a TOF timer begin accumulating time?
11. The RTO timer is a retentive timer. Explain what this means.
12. How are the retentive timer and reset instruction related?



PART 3 PROBLEMS

1. Modify the original CLX ten-second TON timer program with an additional rung added to the program that will energize a solenoid whenever the timer is enabled and timing. The solenoid is to be connected to pin 6 of the digital output module.
2. With reference to the ladder logic of the CLX diverter gate program, assume the solenoid gate fails to energize as programmed. You suspect the problem is due to an open in the solenoid coil or wiring to it. How might observation of the solenoid output status light help confirm this?
3. You are required to extend the Green light-on time of the CLX traffic control program to 40 seconds. What changes would have to be made to the program?
4. With reference to the CLX heating oven process program, assume the oven-on pilot light burns out. In what way would the operation of the program be affected?
5. With reference to the CLX limit switch RTO program, in addition to the alarm you are required to install a warning pilot light to indicate that the timer has timed out. How would you proceed?
6. Implement the hardwired TON alarm circuit of Figure 15-56 in Logix format.

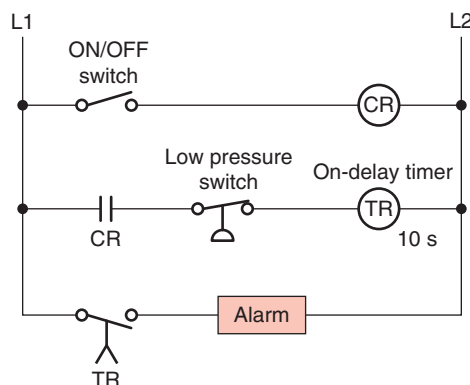


Figure 15-56 Hardwired TON alarm circuit for Problem 6.

Part 4 Programming Counters

Part Objectives

After completing this part, you will be able to:

- Understand ControlLogix counter tags and their members
- Utilize status bits from counters in logic
- Develop ladder logic programs using ControlLogix counters

Counters

Counters are similar to timers, except that a counter accumulates (counts) the changes in state of an external trigger signal whereas timers increment using an internal clock. PLC counters are generally triggered by a change in an input field device that causes a false-to-true transition of the counter ladder rung. It does not matter how long the rung stays true or false—it is only the transition that counts.

There are two basic counter types: count-up (CTU) and count-down (CTD). The ControlLogix CTU instruction and counter selection toolbar are shown in Figure 15-57. When you want to use a timer, you must create a tag of type COUNTER (it is a predefined data type) and enter the preset and the accumulated value. When entering the instruction, this tag must be defined before the preset and accumulated values can be entered. A RES reset instruction that has the same tag name as the

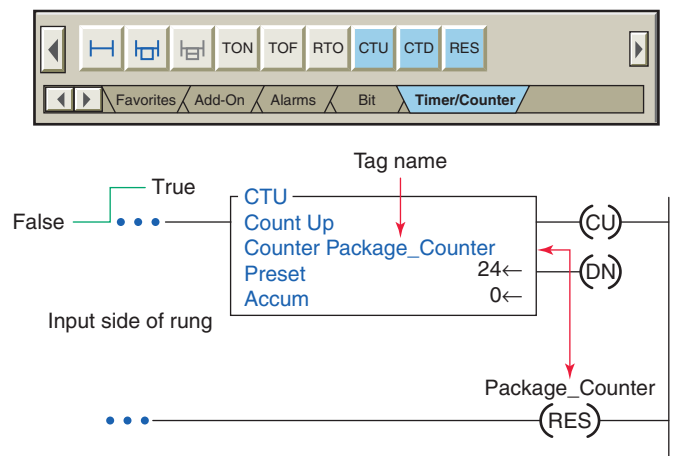


Figure 15-57 CTU count-up counter instruction.

counter must be used to reset the accumulated value of the counter to zero.

All counters are retentive in that the accumulated value of any counter is retained, even during a power failure, until reset. The on/off status of the counter done, overflow, and underflow bits are retentive as well. ControlLogix counter parameters and status bits are shown in the edit tags window of Figure 15-58 and can be summarized as follows:

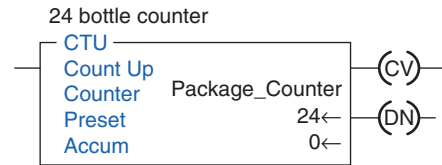
- **Preset (PRE) Value**—Specifies the value the counter must reach before the done (DN) bit turns on (1).
- **Accumulated (ACC) Value**—Is the number of false-to-true transitions of the counter run. ACC is reset to zero when a reset (RES) instruction (of the same counter address) is executed.
- **CU (Count-Up Enable Bit)**—The count-up enable bit indicates the CTU instruction is enabled.

Tag Name	Data Type	Style
[-]Part_Counter	COUNTER	Decimal
[+]Part_Counter.PRE	DINT	Decimal
[+]Part_Counter.ACC	DINT	Decimal
[-]Part_Counter.CU	BOOL	Decimal
[-]Part_Counter.CD	BOOL	Decimal
[-]Part_Counter.DN	BOOL	Decimal
[-]Part_Counter.OV	BOOL	Decimal
[-]Part_Counter.UN	BOOL	Decimal

Figure 15-58 ControlLogix counter parameters and status bits.

- **CD (Count-Down Enable Bit)**—The count-down enable bit indicates the CTD instruction is enabled.
- **DN (Count-Up Done Bit)**—Is set (1) when ACC value is equal to or greater than the PRE value. Is reset by the RES instruction.
- **OV (Overflow Bit)**—The overflow bit indicates the counter exceeded the upper limit. Is set when the ACC value is greater than +2,147,483,647 and reset when the reset instruction is executed. Note that the accumulated value keeps incrementing even after the ACC value equals the PRE value.
- **UN (Underflow Bit)**—Indicates that the counter exceeded the lower limit of -2,147,483,648.

The counter tag name is declared using the new tag properties dialog box shown in Figure 15-59. Tag name, description (optional), tag type, data type (base type is



New Tag

Name
Package_Counter

Description
24 bottle counter

Tag Type
Base

Data Type
COUNTER

Scope
Main program

Figure 15-59 Counter tag validation.

used most often), and scope are selected or typed to complete the validation.

Count-Up (CTU) Counter

Count-up (CTU) counters will cause the accumulated count to increase by 1 every time there is a false-to-true transition of the counter ladder rung. An example application of a count-up counter program used to count packets of bottles is shown in Figure 15-60. The operation of the program can be summarized as follows:

- Each open-to-close transition of the Bottle_Sensor proximity switch causes the counter to increment by 1.

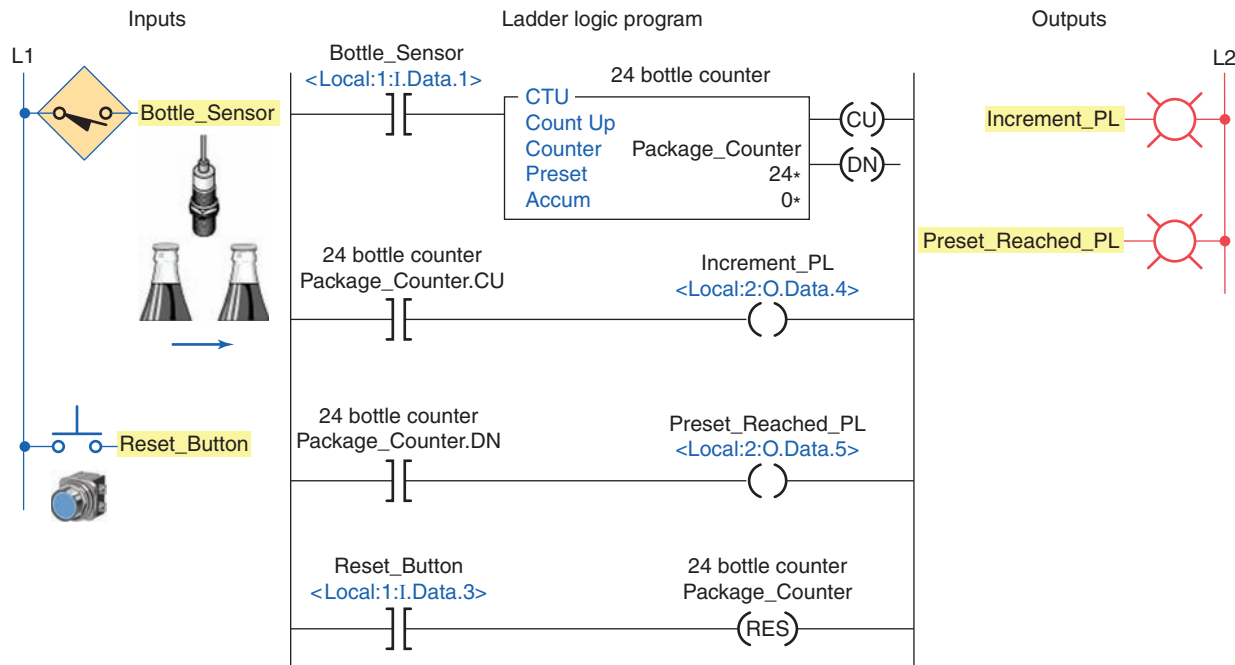


Figure 15-60 Count-up counter program used to count packets of bottles.

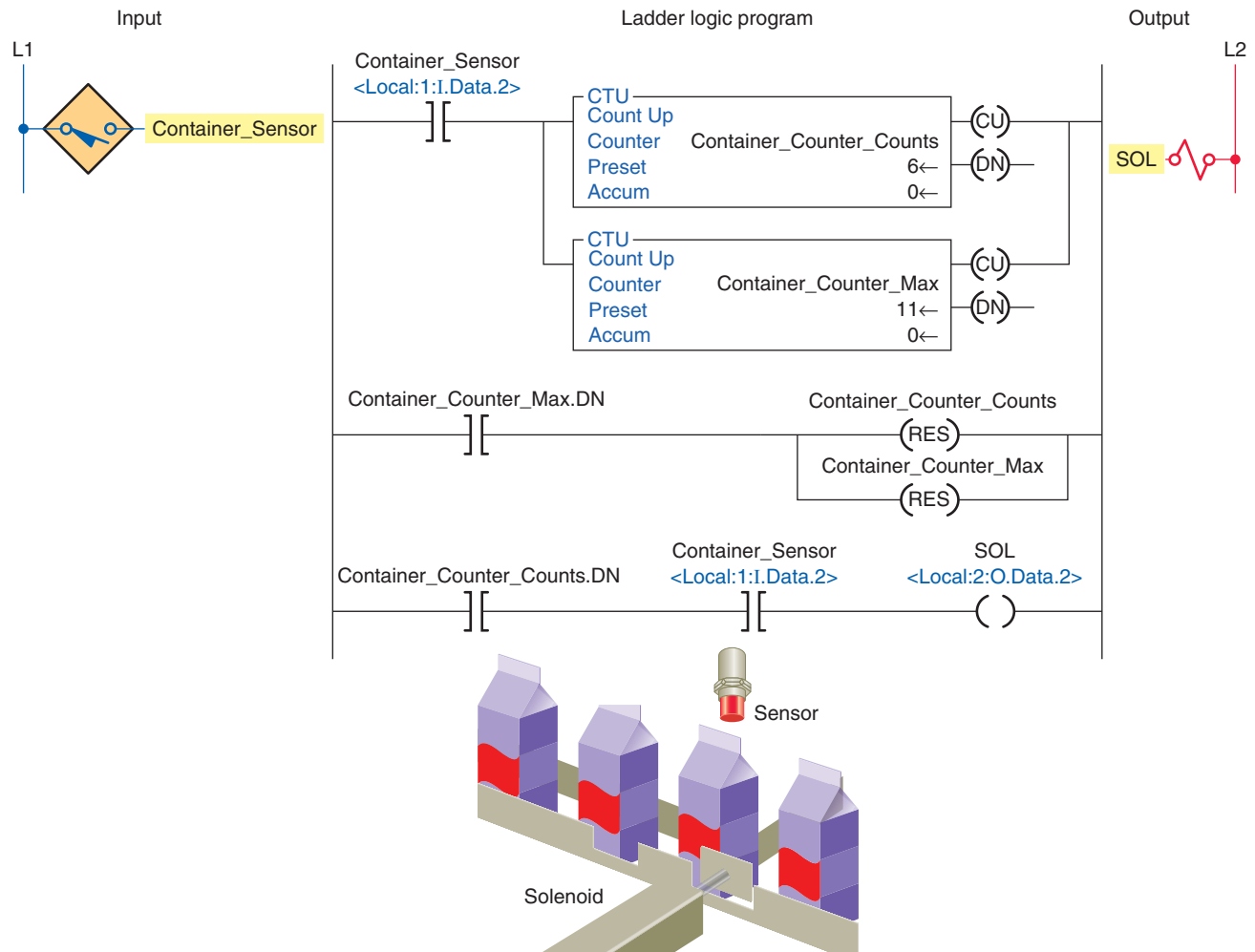


Figure 15-61 CTU program used to remove containers from a conveyor line.

- The Increment_PL controlled by the Package_Counter.CU status bit turns on and off as each bottle passes to show that the counter is incrementing.
- When the accumulated value of the counter is 24 the DN bit of the counter is set and switches on the Preset_Reached_PL.
- The counter is reset by momentarily closing the Reset_Button.

The program shown in Figure 15-61 uses two CTU instructions as part of a program to remove 5 out of every 10 containers from a conveyor line using an electric solenoid. The different tags created to fit the program are shown in Figure 15-62. The operation of the program can be summarized as follows:

- The preset for the **Container_Counter_Counts** is set for 6 and that for the **Container_Counter_Max** is set to 11.

Tag Name	Value	Style	Data Type
<input type="checkbox"/> Container_Counter_Counts	{...}		COUNTER
<input type="checkbox"/> Container_Counter_Counts .PRE	6	Decimal	DINT
<input type="checkbox"/> Container_Counter_Counts .ACC	0	Decimal	DINT
<input type="checkbox"/> Container_Counter_Counts .CU	0	Decimal	BOOL
<input type="checkbox"/> Container_Counter_Counts .CD	0	Decimal	BOOL
<input type="checkbox"/> Container_Counter_Counts .DN	0	Decimal	BOOL
<input type="checkbox"/> Container_Counter_Counts .OV	0	Decimal	BOOL
<input type="checkbox"/> Container_Counter_Counts .UN	0	Decimal	BOOL
<input type="checkbox"/> Container_Counter_Max	{...}		COUNTER
<input type="checkbox"/> Container_Counter_Max .PRE	11	Decimal	DINT
<input type="checkbox"/> Container_Counter_Max .ACC	0	Decimal	DINT
<input type="checkbox"/> Container_Counter_Max .CU	0	Decimal	BOOL
<input type="checkbox"/> Container_Counter_Max .CD	0	Decimal	BOOL
<input type="checkbox"/> Container_Counter_Max .DN	0	Decimal	BOOL
<input type="checkbox"/> Container_Counter_Max .OV	0	Decimal	BOOL
<input type="checkbox"/> Container_Counter_Max .UN	0	Decimal	BOOL
Container_Sensor	0	Decimal	BOOL
SOL	0	Decimal	BOOL

Figure 15-62 Tags created for the CTU program used to remove containers from a conveyor line.

- When the container is detected both counters will increase their accumulated values by 1.
- When the sixth part arrives the Container_Counter_Counts counter will then be done, thereby allowing the solenoid to actuate for any container after the fifth.
- The Container_Counter_Max counter will continue until the eleventh part is detected and then both of the counters will be reset.

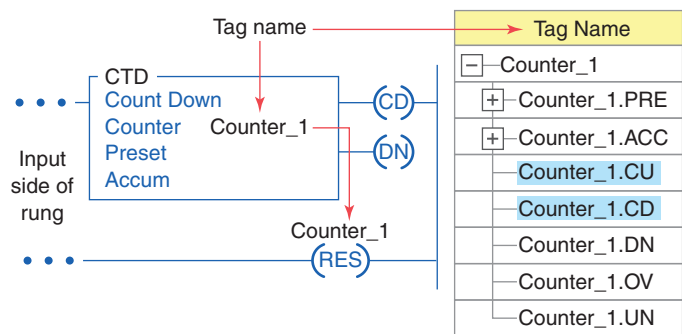


Figure 15-63 Count-down CTD counter instruction.

Count-Down (CTD) Counter

The *count-down (CTD) counter* operates in a fashion opposite to the count-up CTU counter. CTD counters will cause the accumulated count to decrease instead of increase by one every time there is a false-to-true transition of the counter ladder rung. The ControlLogix CTD down-counter instruction is shown in Figure 15-63. The descriptions of the function block fields and the tag references are the same as those associated with the CTU function block. The CTD instruction is typically used with a CTU instruction that references the same counter structure.

The application program shown in Figure 15-64 is used to limit the number of parts that can be stored in the buffer zone to a maximum of 50. A CTU counter and a CTD counter are used together *with the same*

address to form an Up/Down counter. This is the most common type of application of the CTD counter. The different tags created to fit the program are shown in Figure 15-65. The operation of the program can be summarized as follows:

- The Restart_Button is momentarily actuated at any time to reset the accumulated value of the counter to zero.
- Conveyor brings parts into a buffer zone.
- Each time a part enters the buffer zone, the Enter_Limit_Sw is actuated and Counter_1 increments by 1.

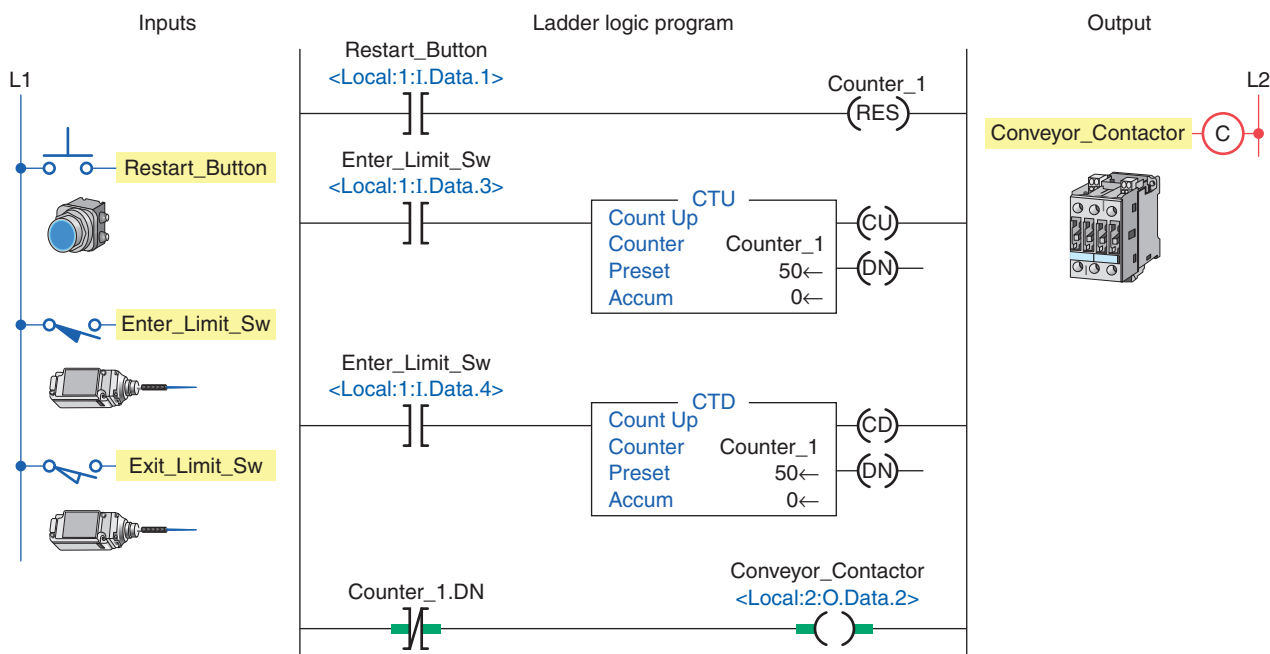


Figure 15-64 CTU counter and CTD counter used together to form an Up/Down counter.

Tag Name	Value	Style	Data Type
Counter_1	{ ... }		COUNTER
Counter_1.PRE	50	Decimal	DINT
Counter_1.ACC	0	Decimal	DINT
Counter_1.CU	0	Decimal	BOOL
Counter_1.CD	0	Decimal	BOOL
Counter_1.DN	0	Decimal	BOOL
Counter_1.OV	0	Decimal	BOOL
Counter_1.UN	0	Decimal	BOOL
Restart_Button	0	Decimal	BOOL
Enter_Limit_Sw	0	Decimal	BOOL
Exit_Limit_Sw	0	Decimal	BOOL
Conveyor_Contactor	1	Decimal	BOOL

Figure 15-65 Tags created for the Up/Down counter program.

- Each time a part leaves the buffer zone, the Exit_Limit_Sw is actuated and Counter_1 decrements by 1.
- When the number of parts in the buffer zone, at any one time, reaches 50, the Counter_1.DN bit is set.
- As a result the Conveyor_Contactor rung goes false to de-energize the conveyor contactor, automatically stopping the conveyor from bringing in any more parts until the accumulated count drops below 50.



PART 4 REVIEW QUESTIONS

1. In what way are timers and counters similar?
2. Outline the procedure followed to create a tag when you want to use a counter.
3. All counters are retentive. In what way does this affect their operation?
4. What is specified by the preset value of a counter?
5. When is each of the following counter bits set?
 - a. CU
 - b. DN
 - c. CD
6. Compare the operations of a CTU and a CTD counter.
7. What is an Up/Down counter?
8. Explain how you go about creating tags for an Up/Down counter that uses a CTU and CTD instruction.



PART 4 PROBLEMS

1. With reference to the CTU packets of bottles program, what changes to the program would be required to count 6 bottle packets?
2. With reference to the CTU program used to remove containers from a conveyor line, assume the output solenoid coil failed open. In what way would the operation of the program be affected?
3. Modify the original Up/Down counter program to include:
 - a. A red pilot light to indicate entry of a part into the buffer zone. Light to be connected to pin 4 of the digital output module.
 - b. A green pilot light to indicate exit of a part from the buffer zone. Light to be connected to pin 3 of the digital output module.
4. Write a ControlLogix program, complete with tags, for an Up/Down counter used to keep track of cars entering and exiting a parking lot. The program requirements for this application can be summarized as follows:
 - The parking lot holds 30 vehicles.
 - There is an entrance vehicle sensor and an exit vehicle sensor.
 - When the parking lot is full a Lot Full sign is illuminated.
 - Whenever a car exits the lot, a Caution Buzzer/Light is activated to warn pedestrians.

Part 5 Math, Comparison, and Move Instructions

Part Objectives

After completing this part, you will be able to:

- Utilize ControlLogix math instructions in programs
- Utilize ControlLogix comparison instructions in programs
- Utilize ControlLogix move instructions in programs
- Develop and follow the operation of programs that use math, comparison, and move instructions

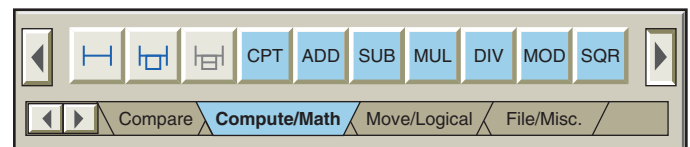


Figure 15-66 Compute/Math toolbar for the ControlLogix controller.

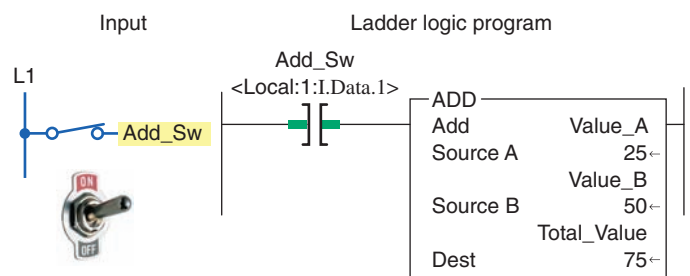
Math Instructions

ControlLogix basic math instructions include addition, subtraction, multiplication, division, square root, and clear. Figure 15-66 shows the Compute/Math toolbar for the ControlLogix controller.

The **ADD** instruction is used to add two numbers. This instruction adds these values from Source A and Source B. The source can be a constant value or a tag. The result of the ADD instruction is put in the destination (Dest) tag.

Figure 15-67 shows an example of an ADD instruction rung along with its Monitor Tags window. The operation of the rung can be summarized as follows:

- When the **ADD_Sw** is closed the rung will be true.
- The **ADD** instruction will execute to add the number from Source A (**Value_A**) and the value from Source B (**Value_B**).



Tag Name	Value	Style	Data Type
<input checked="" type="checkbox"/> Total_Value	75	Decimal	DINT
<input checked="" type="checkbox"/> Value_A	25	Decimal	DINT
<input checked="" type="checkbox"/> Value_B	50	Decimal	DINT
ADD_Sw	1	Decimal	BOOL

Figure 15-67 ADD instruction rung and its Monitor Tags window.

- The result will be stored in the Dest tag (Total_Value).
- In this example, the 25 was added to 50 and the result (75) was stored in Total_Value.

The *SUB* instruction is used to subtract two numbers. Figure 15-68 shows an example of a SUB instruction rung along with its Monitor Tags window. The operation of the rung can be summarized as follows:

- When the SUB_Sw or Calculate tag is true the SUB instruction is executed.
- Source B (Shipped_Parts) is subtracted from Source A (Parts_Stock) and the result is stored in the Dest tag named Current_Inventory.

- In this example, the 200 was subtracted from 900 and the result (700) was stored in Current_Inventory.
- Source A and Source B can be constants (numbers) or tags.

The *MUL* instruction is used to multiply two numbers. Figure 15-69 shows an example of a MUL instruction rung along with its Monitor Tags window. When multiple bottles are packed in cases, the number of bottles per case, the number of cases, and the multiply instruction will give you the total number of bottles. The operation of the rung can be summarized as follows:

- When the Sw_1 and Sw_2 are both true the MUL instruction is executed.

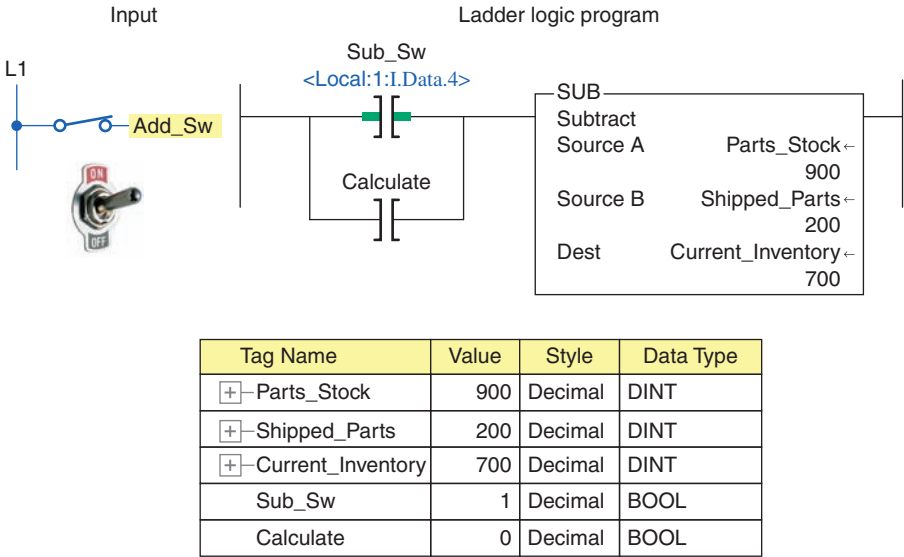


Figure 15-68 SUB instruction rung and its Monitor Tags window.

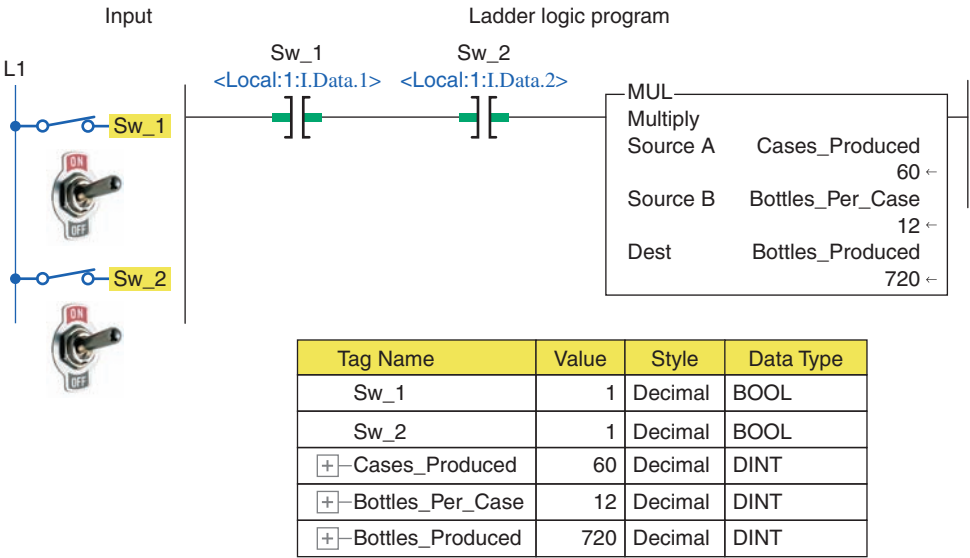
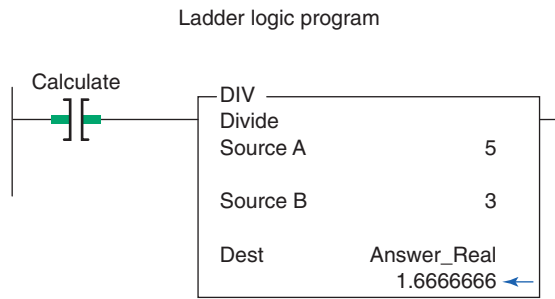


Figure 15-69 MUL instruction rung and its Monitor Tags window.



Tag Name	Value	Style	Data Type
Calculate	1	Decimal	BOOL
Answer_Real	1.6666666	Float	REAL

Figure 15-70 DIV instruction rung and its Monitor Tags window.

- Source A (the value in tag Cases_Produced) is multiplied by Source B (the value in tag Bottles_Per_Case) and the result is stored in the Dest tag Bottles_Produced.
- Source A and Source B can be constants (numbers) or tags.

The *DIV* instruction is used to divide two numbers. Figure 15-70 shows an example of a *DIV* instruction rung along with its Monitor Tags window. The operation of the rung can be summarized as follows:

- A constant (5) is used for Source A and a constant (3) for Source B. Note that tags could have been used for Source A or Source B.
- When the Calculate tag is true the *DIV* instruction is executed.
- Source A (5) is divided by Source B (3) and the result (1.6666666) is stored in the Dest tag Answer_Real. Note that in this example a Real-type tag has been used for its destination.

The program of Figure 15-71 is used as part of a parts tracking system with three conveyors. The number of parts in conveyor 1 and the number of parts in conveyor 2 are added to get the number of parts on conveyor 3. The operation of the program can be summarized as follows:

- Each time Conveyor_1_Sensor is actuated the accumulated value of Counter_1_Parts is incremented by 1.
- Each time Conveyor_2_Sensor is actuated the accumulated value of Counter_2_Parts is incremented by 1.

- The addition in the *ADD* instruction places the sum of the accumulated values of the two counters in the Conveyor_3_Parts tag.
- When the accumulated value for either counter is equal to 150 the reset (*RES*) instructions for both counters are enabled to automatically reset both counter *ACC* values to zero.
- Both counters can also be reset manually at any time by actuation of the Manual_Conveyor_Reset button.

Comparison Instructions

Compare instructions are used to compare two values. They can be used to see if two values are equal, if one value is greater or less than the other, and so on. In ControlLogix controllers compare instructions are input instructions that do comparisons by either using an expression or doing the comparison indicated by the specific instruction. Figure 15-72 shows the Compare toolbar for the ControlLogix controller.

The *equal* (*EQU*) instruction is used to test if two values are equal. Values compared can be actual values or tags that contain values. Figure 15-73 shows an example of an *EQU* instruction rung along with its Monitor Tags window. The operation of the rung can be summarized as follows:

- The value stored at Source A is compared to the value stored at Source B.
- If the values are equal, the instruction is logically true.
- If the values are unequal, the instruction is logically false.
- In this example Source A (25) is equal to Source B (25) so the instruction is true and output Equal_PL is on.
- Source A and Source B may be SINT, INT, DINT, or REAL data types.

The *not equal* (*NEQ*) instruction is used to test two values for inequality. Figure 15-74 shows an example of an *NEQ* instruction rung. When Source A is not equal to Source B, the instruction is logically true; otherwise, it is logically false. In this example the two values are not equal so the Not_Equal_PL is energized.

The *less than* (*LES*) instruction is used to check if a value from one source is less than the value from a second source. Figure 15-75 shows an example of an *LES* instruction rung. When Source A is less than Source B, the instruction is logically true; otherwise, it is logically false. In this example Value_1 (100) is less than Value_2 (300) so the Less_Than_PL is energized.

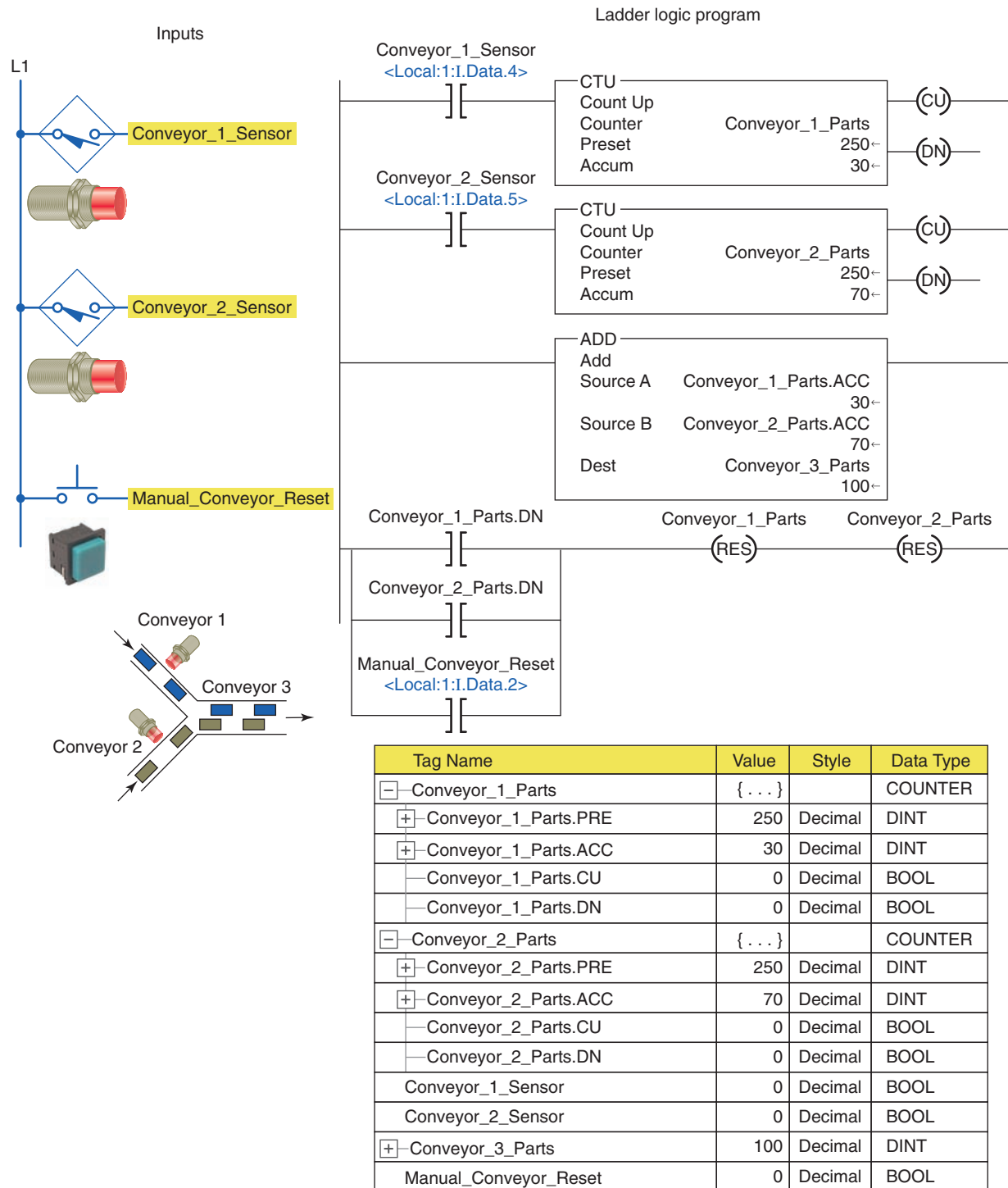


Figure 15-71 Program used as part of a parts tracking system.

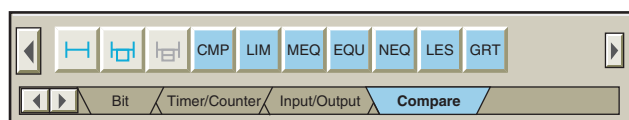


Figure 15-72 Compare toolbar for the ControlLogix controller.

The *greater than (GRT)* instruction is used to check if a value from one source is greater than the value from a second source. Figure 15-76 shows an example of a GRT instruction rung. When Source A is greater than Source B, the instruction is logically true; otherwise, it is logically false. In this example Value_1 (1420) is

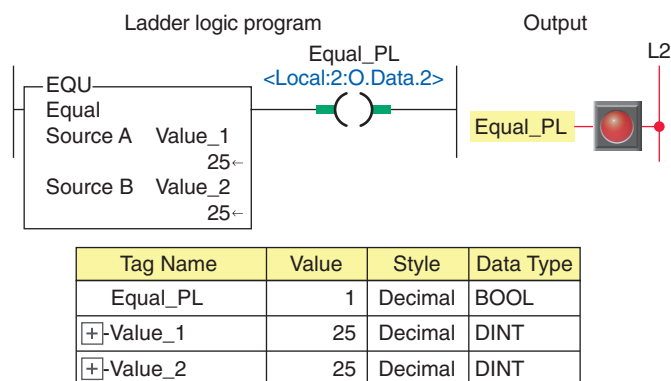


Figure 15-73 EQU instruction rung and its Monitor Tags window.

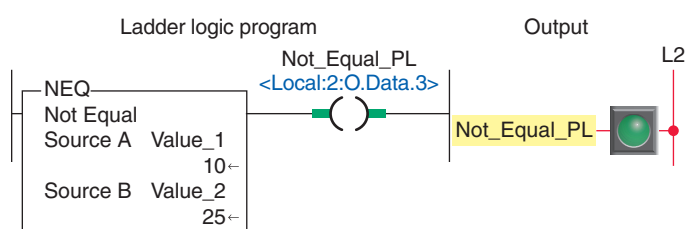


Figure 15-74 NEQ instruction rung.

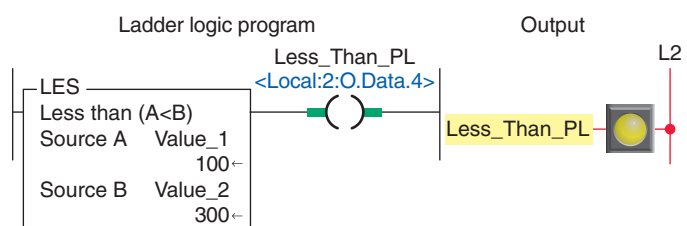


Figure 15-75 LES instruction rung.

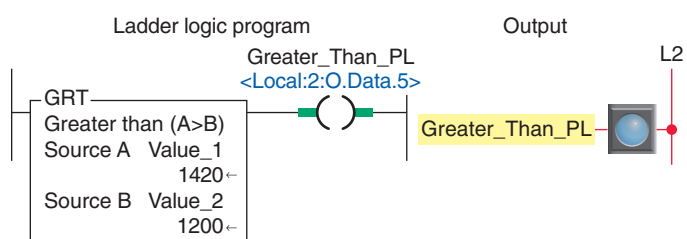


Figure 15-76 GRT instruction rung.

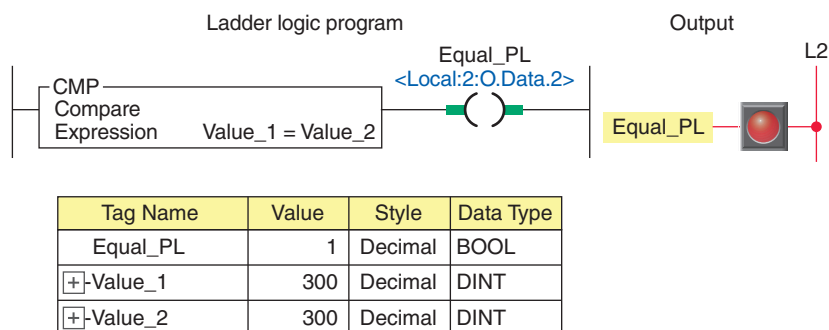


Figure 15-77 CMP instruction rung.

greater than Value_2 (1200) so the Greater_Than_PL is energized.

The *compare (CMP)* instruction performs a comparison on the arithmetic operations specified by the expression. The expression may contain arithmetic operators, comparison operators, and tags. The execution of a CMP instruction is slightly slower and uses more memory than the execution of the other comparison instructions. The advantage of the CMP instruction is that it allows you to enter complex expressions in one instruction. Figure 15-77 shows an example of a CMP instruction rung. In this example the comparison operator found in the expression is the equivalent of an EQU instruction. The comparison instruction is true because Value_1 (300) is equal to Value_2 (300).

The program of Figure 15-78 is an example of the use of comparison instructions used to test the accumulated value of a counter. The operation of the program can be summarized as follows:

- When the accumulated count is between 5 and 10 the GRT and LES instructions will both be logically true so the PL_1 pilot light will be on.
- When the accumulated count is equal to 15, the EQU instruction will be logically true so the PL_2 pilot light will be on.
- The PL_3 pilot light will be on at all times except when the accumulated count is 20 at which time the NEQ instruction is logically false.
- The counter is reset automatically when the accumulated count reaches 25 or manually anytime the Reset_PB is actuated.

Move Instructions

The *move (MOV)* instruction is an output instruction that can move a constant or the contents of one memory location to another location. Figure 15-79 shows the Move toolbar and instruction for the ControlLogix

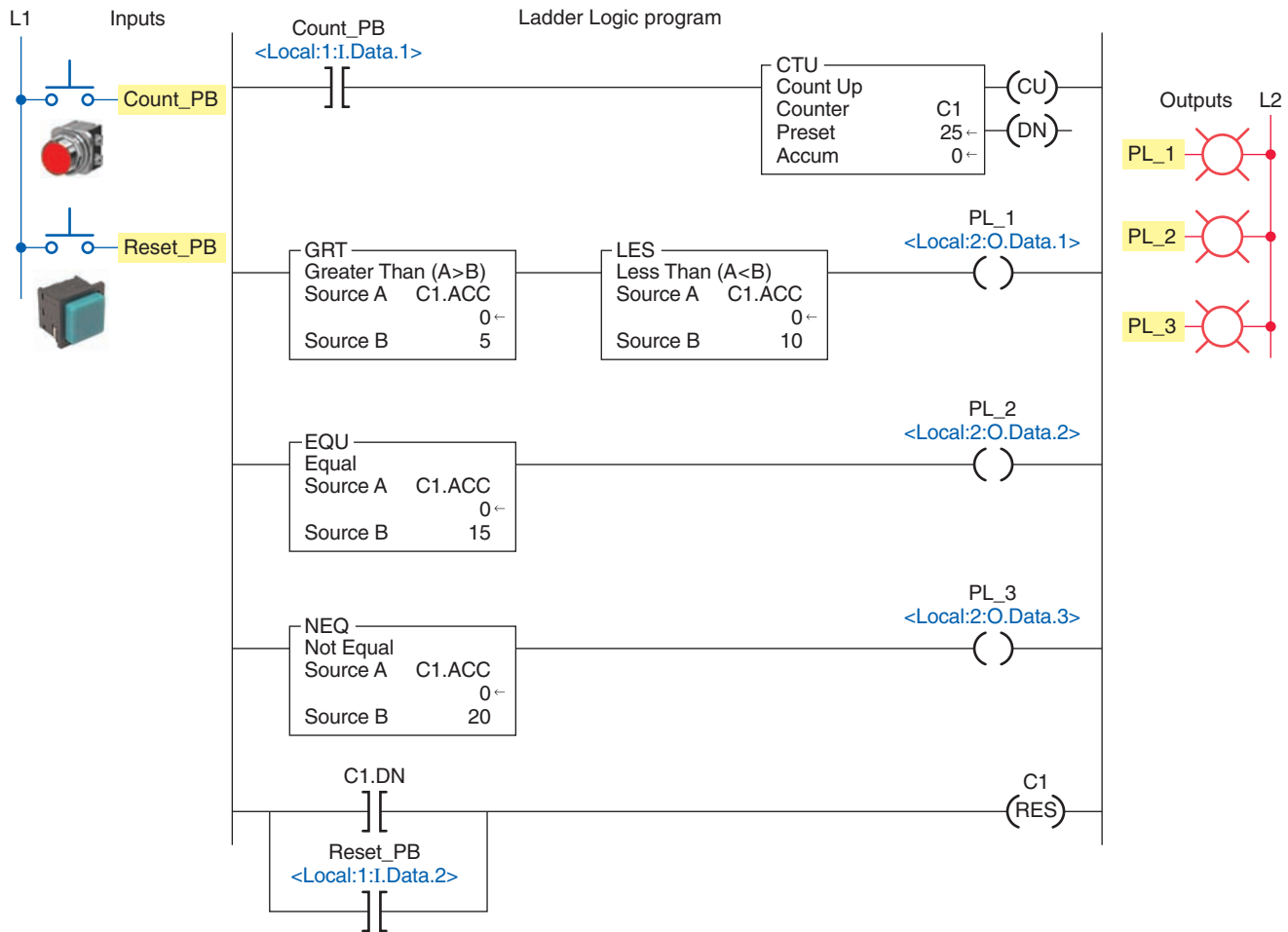


Figure 15-78 Comparison instructions used to test the accumulated value of a counter.

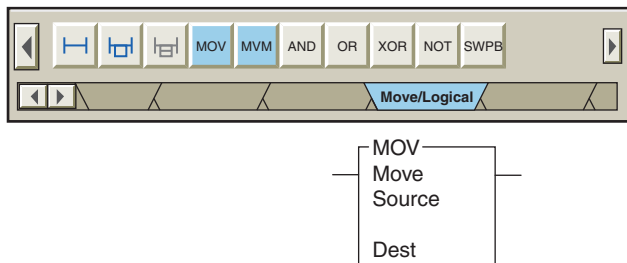


Figure 15-79 Move toolbar for the ControlLogix controller.

controller. The MOV instruction is used to copy data from a source to a destination. Both the source and the destination data type of a MOV instruction may be INT, DINT, SINT, or REAL.

The program of Figure 15-80 is an example of how the MOV instruction can be used to create a variable preset timer. The operation of the program can be summarized as follows:

- Actuating the PB_10s button executes its MOV instruction to transfer 10000 to the timer preset value setting the delay period for 10 seconds.
- Actuating the PB_15s button executes its MOV instruction to transfer 15000 to the timer preset value setting the delay period for 15 seconds.
- Closing the Timer_Start switch starts the timer timing.
- While the timer is timing, the pilot light PL_1 is on for the duration of the timer preset period.
- When the timer times out, PL_1 turns off and PL_2 turns on.

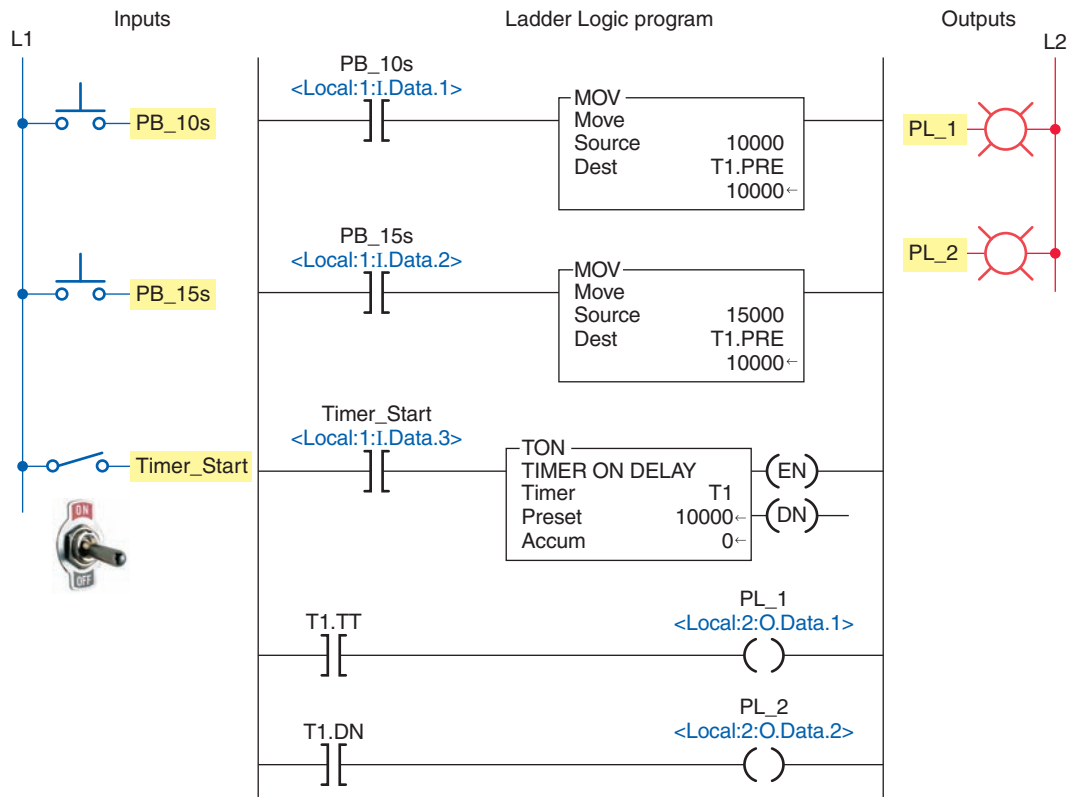


Figure 15-80 MOV instruction used to create a variable preset timer.



PART 5 REVIEW QUESTIONS

1. Construct a ControlLogix ladder rung with a math instruction that executes when a toggle switch is closed to add the tag named Pressure_A (value 680) to the constant of 50 and store the answer in the tag named Result.
2. Construct a ControlLogix ladder rung with a math instruction that executes when two normally open limit switches are closed to subtract the tag named Count_1 (value 60) from the tag named Count_2 (value 460) and store the answer in the tag named Count_Total.
3. Construct a ControlLogix ladder rung with a math instruction that executes when either one of two normally open pushbuttons is closed to multiply the tag named Cases (value 10) by the constant 24 and store the answer in the tag named Cans.
4. Construct a ControlLogix ladder rung with a compare instruction that will energize a pilot light output anytime the value stored at Data_3 is 60.
5. Construct a ControlLogix ladder rung with a compare instruction that will energize a pilot light output anytime the value stored at Data_2 is not the same as that stored at Data_6.
6. Construct a ControlLogix ladder rung with compare instructions that will energize a pilot light output anytime the pressure of a system goes above 300 psi or below 100 psi.



PART 5 PROBLEMS

1. While checking the operation of the parts tracking system with the Monitor Tags window, you note that the value of Conveyor_Sensor_1 remains at 1 with parts passing by. What can you surmise from this? Why?
2. Three conveyors are delivering the same parts in different packages. A package can hold 12, 24, or 18 parts. Proximity switches installed on each of the conveyor lines are used to advance the accumulated value of the three counters. Write a ControlLogix program that uses multiply and add instructions to calculate the sum of the parts.
3. A single pole switch is used in place of the two pushbuttons for the variable preset timer program. When this switch is closed the timer is to be set for 10 seconds and when open to 15 seconds. Make the necessary changes to the program.

Part 6 Function Block Programming

Part Objectives

After completing this part, you will be able to:

- Describe the difference between ladder logic and function block diagram programming
- Recognize the basic elements of a function block diagram
- Write and read a function block diagram

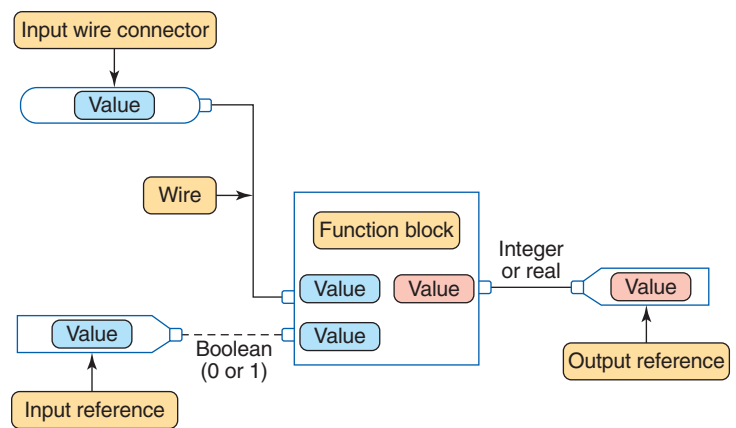


Figure 15-81 Structure of function block or routine.

Function Block Diagram (FBD)

A *function block diagram (FBD)* is a graphical depiction of process flow using simple and complex interconnecting blocks. It is similar to a ladder logic diagram, except that function blocks replace the interconnection of contacts and the coils. In addition, there are no power rails.

A function block circuit is analogous to an electrical circuit where links and wires depict signal paths between components. The workplace is known as a sheet and consists of function blocks joined together with lines called wires. The structure of a function block program, or routine, is shown in Figure 15-81. A function block diagram consists of four basic elements: function block, references, wire connectors, and wires. Data flow on a wire from wire connectors or input references, move through the function block, and then pass on to an output reference. The line type of the link between function blocks

indicates what type of data are present. A dash line indicates a Boolean signal path (e.g., 0 or 1) and a solid line indicates an integer or real value.

Function blocks are graphical representations of executable code. A function block can take one or more inputs and make decisions or calculations and then generate one or more outputs. There are many different types of function blocks included in the programming software to perform various common tasks. In addition, customized **Add-On** instructions can be created by the programmer for sets of commonly used logic. Once an Add-On instruction is defined in a project, it appears on the instruction toolbar and behaves like the standard instructions.

Figure 15-82 shows an example of a BAND (Boolean AND) function block. The information associated with a function block can be summarized as follows:

- Inputs are shown entering from the left and outputs exiting on the right.

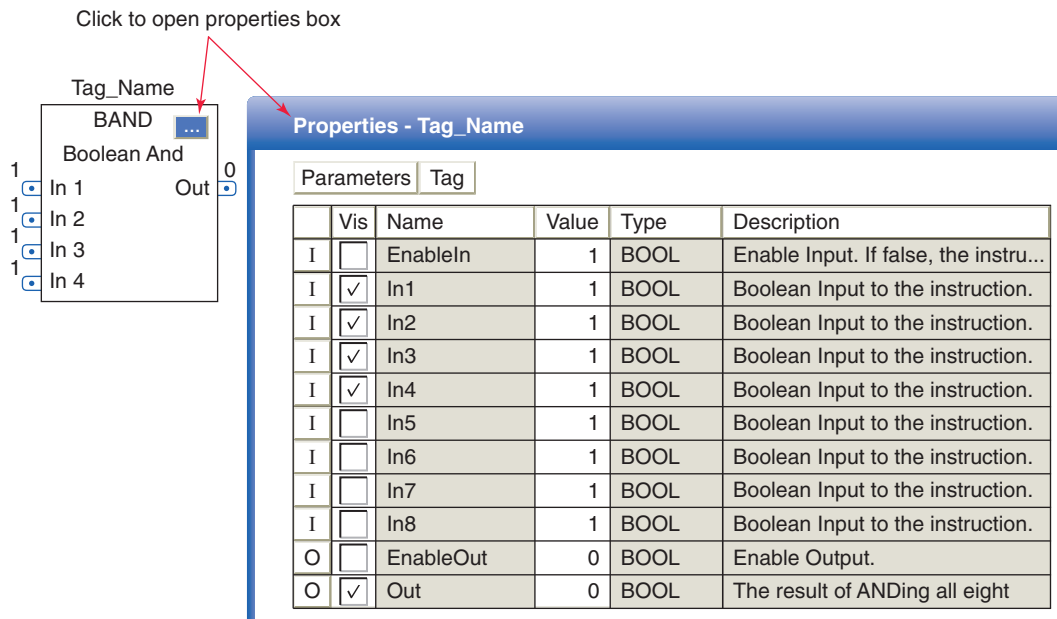


Figure 15-82 Example of a BAND (Boolean AND) function block.

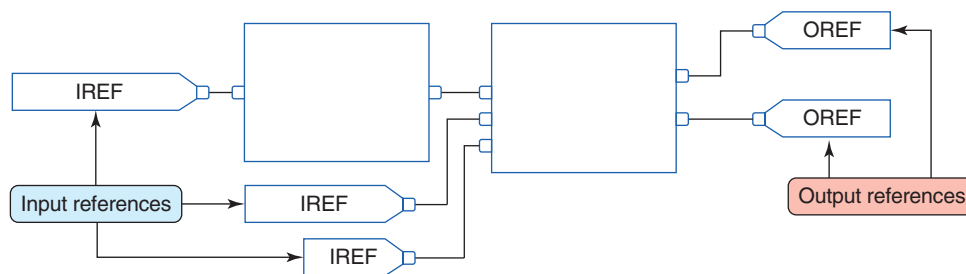


Figure 15-83 Input and output references.

- The function block type is shown within the block.
- A tag name for the block is placed above it.
- The names of the inputs and outputs are shown within the block.
- The default view of the block has some but not all of the input and output parameters visible when the box is placed into the program.
- The properties box, used to set the option of input and output parameters, is displayed by clicking the selection button located at the upper right hand corner of the block.
- The 1 and 0 next to the inputs and outputs identifies the logical state of the input and output pins for the instruction.
- The dots on the input and output pins indicate BOOL type data is required.

References represent tags that are linked to values stored in a controller's memory. The two types of references, input and output, are illustrated in Figure 15-83. An input reference, or IREF, is used to receive a value from an input device or tag. An output reference, or OREF, is used to send a value to an output device or tag. When you use an IREF or an OREF you must create a tag or assign an existing tag to the element. You may use any of the data types for an IREF or OREF.

Function blocks can be connected to other function blocks by connecting their outputs to the input of another function block using **wires and pins** (Figure 15-84). Wires map a signal's path and show the flow of controller execution. Each element in a function block diagram contains pins. Elements are connected by moving wires from input pins to output pins or vice versa. The pins on the left of a function block are input pins, and those on the

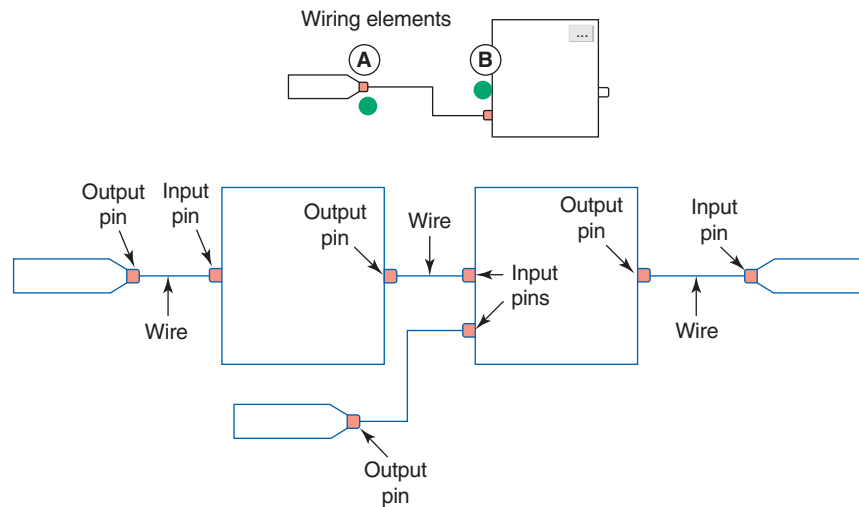


Figure 15-84 Function block diagram wire and pins.

right are output pins. To wire two elements together, click the output pin of the first element (A) and then click the input pin of the other element (B). A green dot shows a valid connection point.

Wire connectors are used to create a path without using a wire. When there are many function blocks on a sheet, or the function blocks are far apart, wire connectors used in place of wires can make the logic harder to read. Wire connectors are also used to connect function blocks that are on a different sheet of the same function block routine, as illustrated in Figure 15-85. The use of wire connectors can be summarized as follows:

- An output wire connector, or **OCON**, sends a value or signal to an input wire connector, or **ICON**.
- Each output wire connector must have at least one corresponding input wire connector.
- Each output wire connector requires a unique tag name and the corresponding input connector must have the same name.

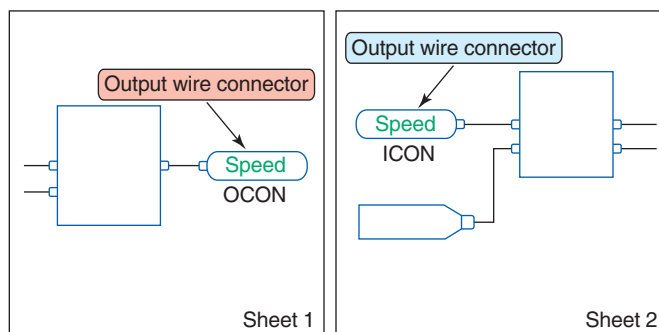


Figure 15-85 OCON and ICON wire connectors.

- Multiple input wire connectors can reference the same output wire connector. This lets you share data at several points in your function block diagram.

Figure 15-86 illustrates the signal flow and execution of an FBD program. The operation can be summarized as follows:

- Each program scan sets all the FBD blocks starting on the left side of the signal flow and continues to evaluate all blocks according to the signal flow until the final output is determined.
- The location of a block does not affect the order in which the blocks execute.
- The inputs of a block require data to be available before the controller can execute that block.
- If function blocks are not wired together, it does not matter which block executes first as there is no data flow between the blocks.
- The interconnected line between the blocks indicates what type of signal is present.

Data latching refers to how the controller verifies that the data present at the input to a function block are valid. If you use an IREF to specify input data for a function block instruction, as illustrated in Figure 15-87, the data in that IREF are latched (won't change) for the scan of the function block routine. The IREF latches data from program-scoped and controller-scoped tags. The controller updates all IREF data at the beginning of each scan. A function block routine executes in the following order:

- The controller latches all data values in IREFs.

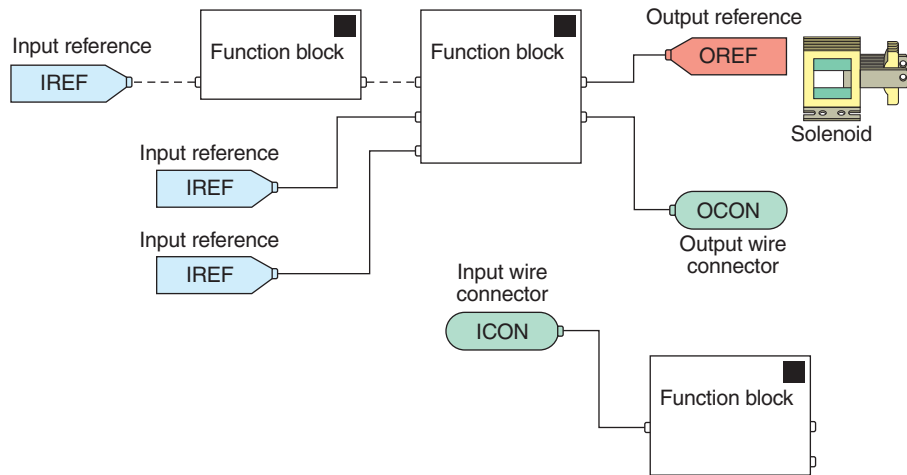


Figure 15-86 Signal flow and execution of an FBD program.

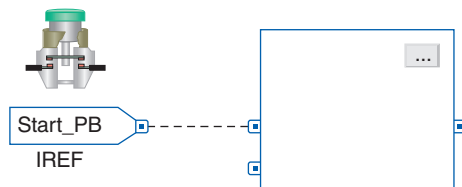


Figure 15-87 IREF is latched for the scan of the function block routine.

- The controller executes the other function blocks in order.
- The controller writes outputs in OREFs.

To create a **feedback loop** around a block, wire an output pin of the block to an input pin of the same block. The input pin will receive the value of the output that was produced on the last scan of the function block. The loop contains only a single block, so execution order does not matter. Figure 15-88 shows an example of a feedback loop used to reset an on-delay timer. When the timer finishes timing its DN bit is used to reset the timer.

When a group of function blocks are in a feedback loop, the controller cannot determine which block to execute first. This problem is resolved by placing an **Assume Data Available** indicator mark at the input pin of the function block that should be executed first. In the example shown in Figure 15-89, the input for block 1 uses the data from block 3 that were produced in the previous scan. To place the indicator click on the interconnecting wire and select the **Assume Data Available** choice.

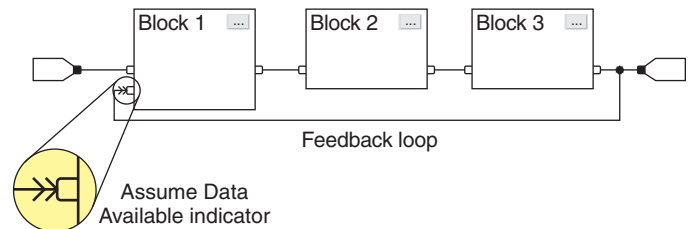


Figure 15-89 Assume Data Available indicator marker.

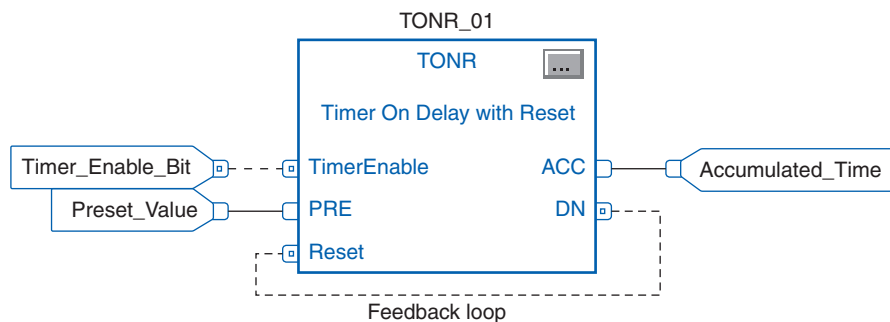


Figure 15-88 Feedback loop used to reset an on-delay timer.

FBD Programming

Figure 15-90 illustrates the setup procedure for FBD programming. The steps to be followed can be summarized as follows:

- Right click on the MainProgram file and select New Routine from the pop-up menu.
- Select the Function Block diagram entry from the Type window.
- Enter a name for the Routine (e.g., FDB_Sample).
- You will now see the new program (FDB_Sample) listed under MainProgram.
- Left clicking the FDB_Sample twice opens the graphic development window.
- FBD instructions selected from the Language Element toolbar are used in the development of the program.

- Extra sheets can be added when the current sheet is full by clicking the add sheet icon. Movement between sheets is provided by left and right arrows.

The MainRoutine is always a ladder logic program in RSLogix 5000 software, and all other routines are called from the MainRoutine. Therefore, the MainRoutine will have one unconditional rung with a jump to subroutine (JSR) calling FBD_Sample. The FBD program will execute from the JSR instruction. No subroutine or return subroutine instruction in the FBD is necessary.

Function block programs are similar to ladder logic programs, except that the process is visualized in the form of function blocks instead of ladder rungs. Figure 15-91 shows a comparison between ladder logic and the FBD equivalent for a three-input AND ladder logic rung. The operation of the FBD can be summarized as follows:

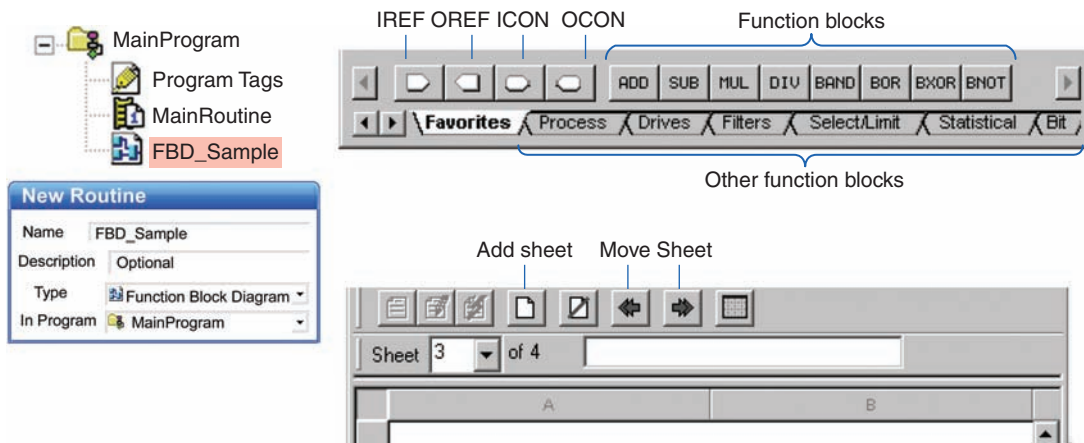


Figure 15-90 Setup procedure for FBD programming.

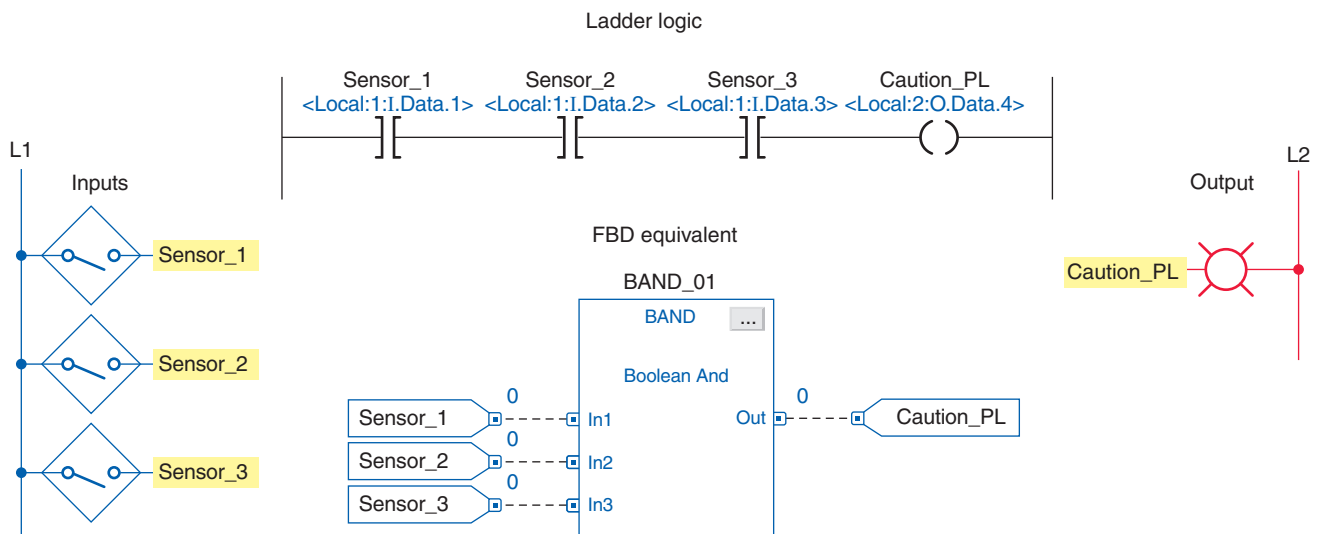


Figure 15-91 Comparison between ladder logic and the FBD equivalent for a three-input AND ladder logic rung.

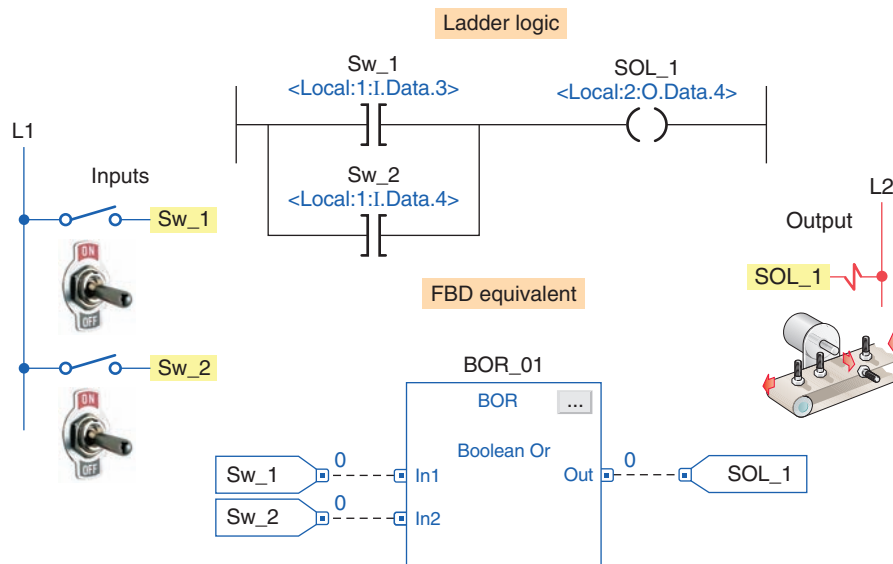


Figure 15-92 Comparison between ladder logic and the FBD equivalent for a two-input OR ladder logic rung.

- When the inputs represented by Sensor_1, Sensor_2, and Sensor_3 are true (value 1) the BAND (Boolean AND) function block will be true.
- The BAND block executes to set output Caution_PL true and switch the pilot light on.
- The 0 to the right of the input reference and out pin indicates its logic state. A 0 indicates the state of the tag is false, while a 1 signifies it is true.
- The same field input sensors and output pilot light devices and tags can be used with either program.
- The XIC and OTE contact and coil instructions have been replaced by the BAND function block.

Figure 15-92 shows a comparison between ladder logic and the FBD equivalent for a two-input OR ladder logic rung. As with ladder OR logic, if any of the two inputs is true the BOR function block will be true. In this example, with the BOR function block true, the output reference tag SOL_1 will be true energizing the solenoid.

Figure 15-93 shows a comparison between ladder logic and the FBD equivalent for a combination of multiple inputs. The operation of the FBD can be summarized as follows:

- The alarm will be energized if either input In1 or In2 to the BOR block is true.

- Input In2 of the BOR block will be true only when all three of the sensor switches are closed.
- Input In1 of the BOR block will be true only when the Temp_Sw is closed at the same time as the Press_Sw is open.
- The BNOT function block executes similarly to an XIO ladder logic contact instruction. When In is 0, Out is 1 and vice versa.

Figure 15-94 shows a comparison between ladder logic and the FBD equivalent for the motor start/stop control circuit. The logic sequence for starting and stopping the motor can be summarized as follows:

- When Motor_Start button is closed the BOR output will become true making the BAND output true.
- Motor_Run output energizes the contactor coil, the contacts of which close to start the motor operating.
- When the Motor_Start button is then opened the output of the BOR block remains true due to the 1 status of the feedback signal from the Motor_Run tag.
- When the Motor_Stop button is opened the output of the BAND block turns false to de-energize the contactor coil and stop the motor.

Figure 15-95 shows a comparison between ladder logic and the FBD equivalent for the 10 second TON (on-delay

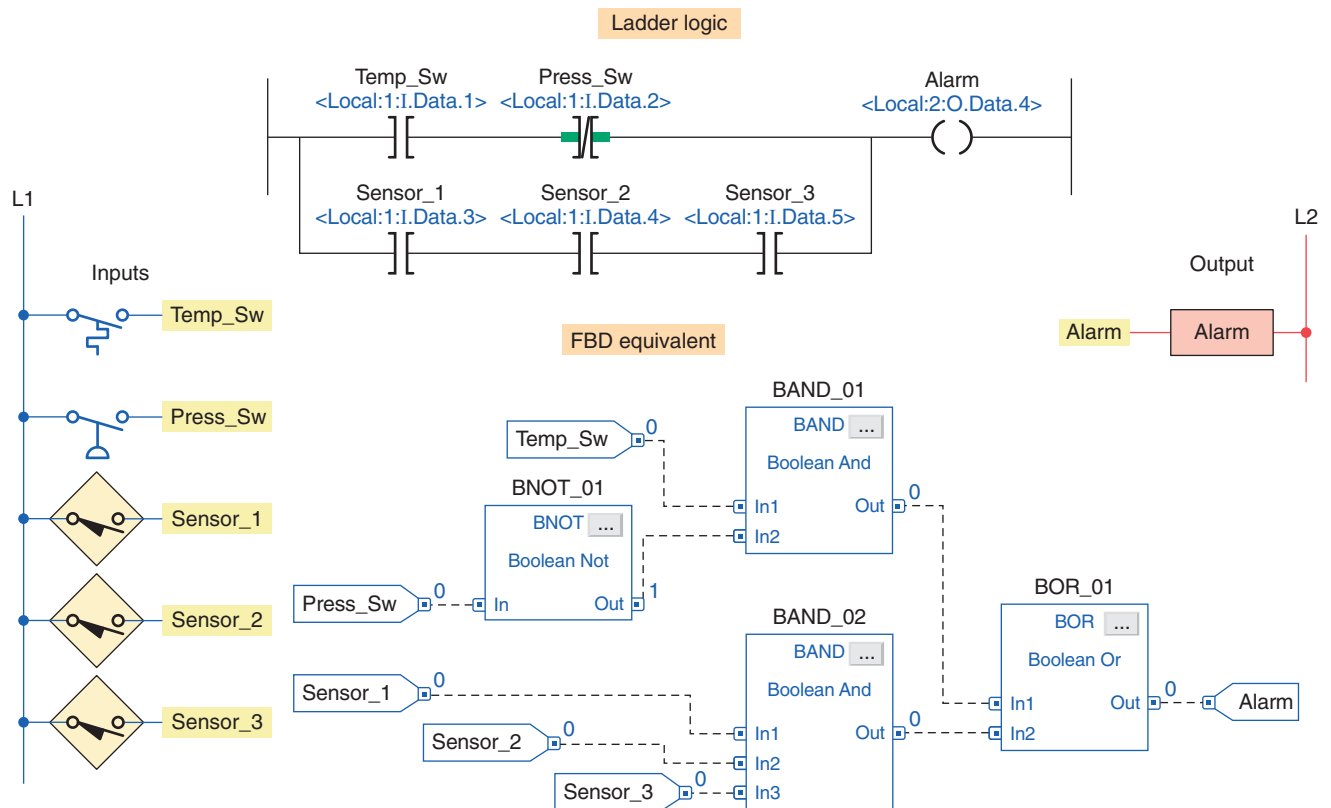


Figure 15-93 Comparison between ladder logic and the FBD equivalent for a combination of multiple inputs.

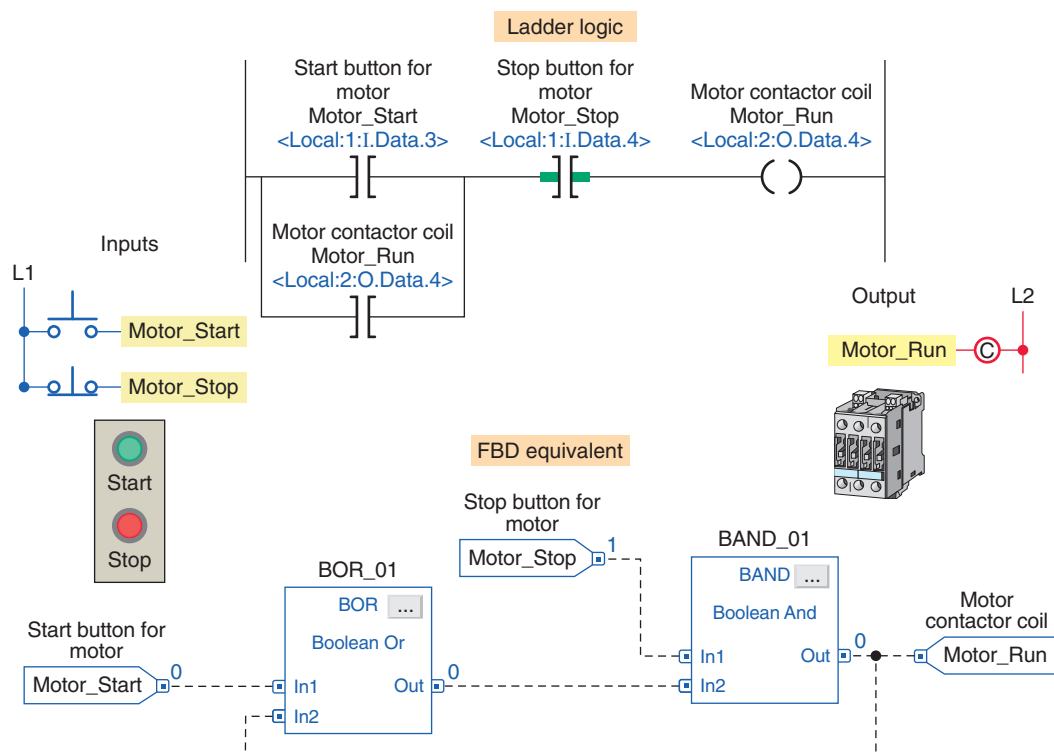


Figure 15-94 Comparison between ladder logic and the FBD equivalent for a motor start/stop control circuit.

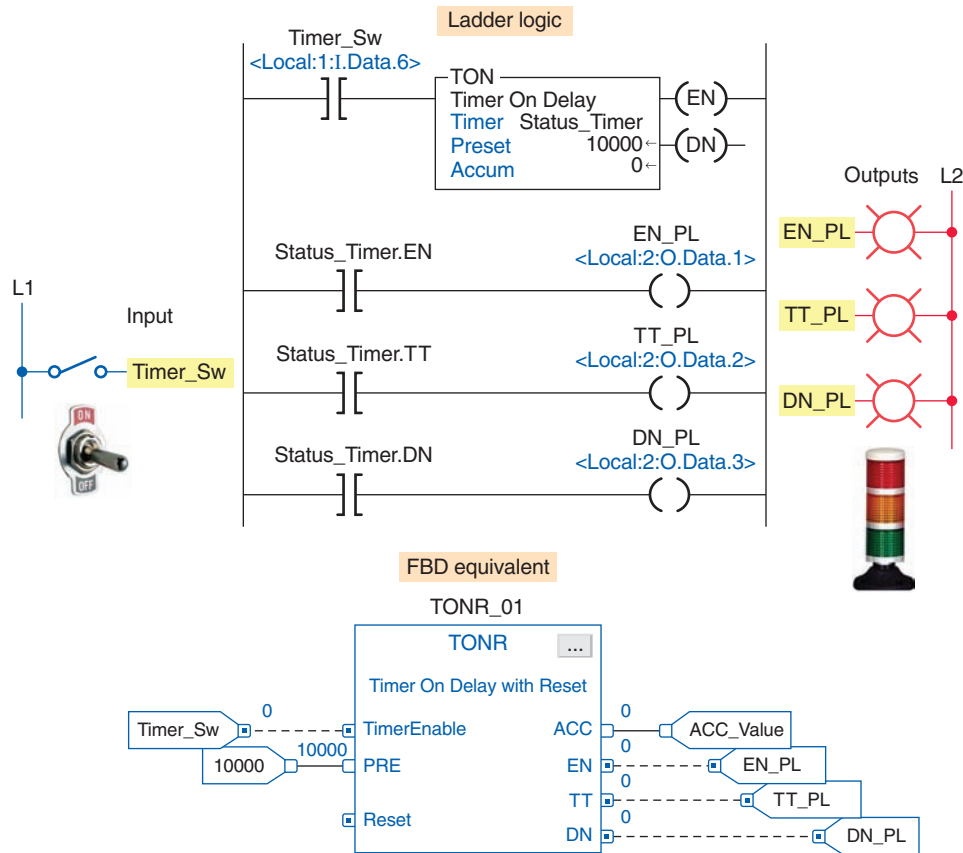


Figure 15-95 Comparison between ladder logic and the FBD equivalent for a 10 second TON and TONR timer.

timer) and TONR (on-delay with reset). The operation of the FBD can be summarized as follows:

- When the Timer_Sw is closed, the TONR function block timer turns true and starts accumulating time.
- The accumulated time is monitored by the output reference tag named ACC.
- The EN (enable bit) output changes to 1 to turn on the EN_PL.
- The TT (timer timing bit) output changes to 1 to turn on the TT_PL.
- The timer times out after 10 seconds to set the DN (done bit) to 1 and turn on the DN_PL and reset the TT bit to zero and turn off the TT_PL.
- The EN bit and EN_PL remain on as long as the Timer_Sw stays toggled closed.
- Opening the Timer_Sw resets all outputs as well as the accumulated value to zero.
- The timer can also be reset by way of the Reset input.

Figure 15-96 shows a comparison between ladder logic and the FBD equivalent for the Up/Down counter used to limit the number of parts stored in a buffer zone to 50. The operation of the FBD can be summarized as follows:

- The CTUD up/down counter function block accumulated value is initially reset by momentary actuation of the Restart_Button.
- The accumulated count is monitored by the output reference tag named ACC.
- Each time a part enters the buffer zone, the Enter_Limit_Sw is actuated and the CUEnable input turns true to increment the count by 1.
- Each time a part exits the buffer zone, the Exit_Limit_Sw is actuated and the CDEnable input turns true to decrement the count by 1.
- Whenever the number of parts in the buffer zone reaches 50 the DN bit is set to 1 and the output of the BNOT block is reset to zero. This de-energizes the Conveyor_Contactor to stop the conveyor motor from delivering more parts to the buffer zone.

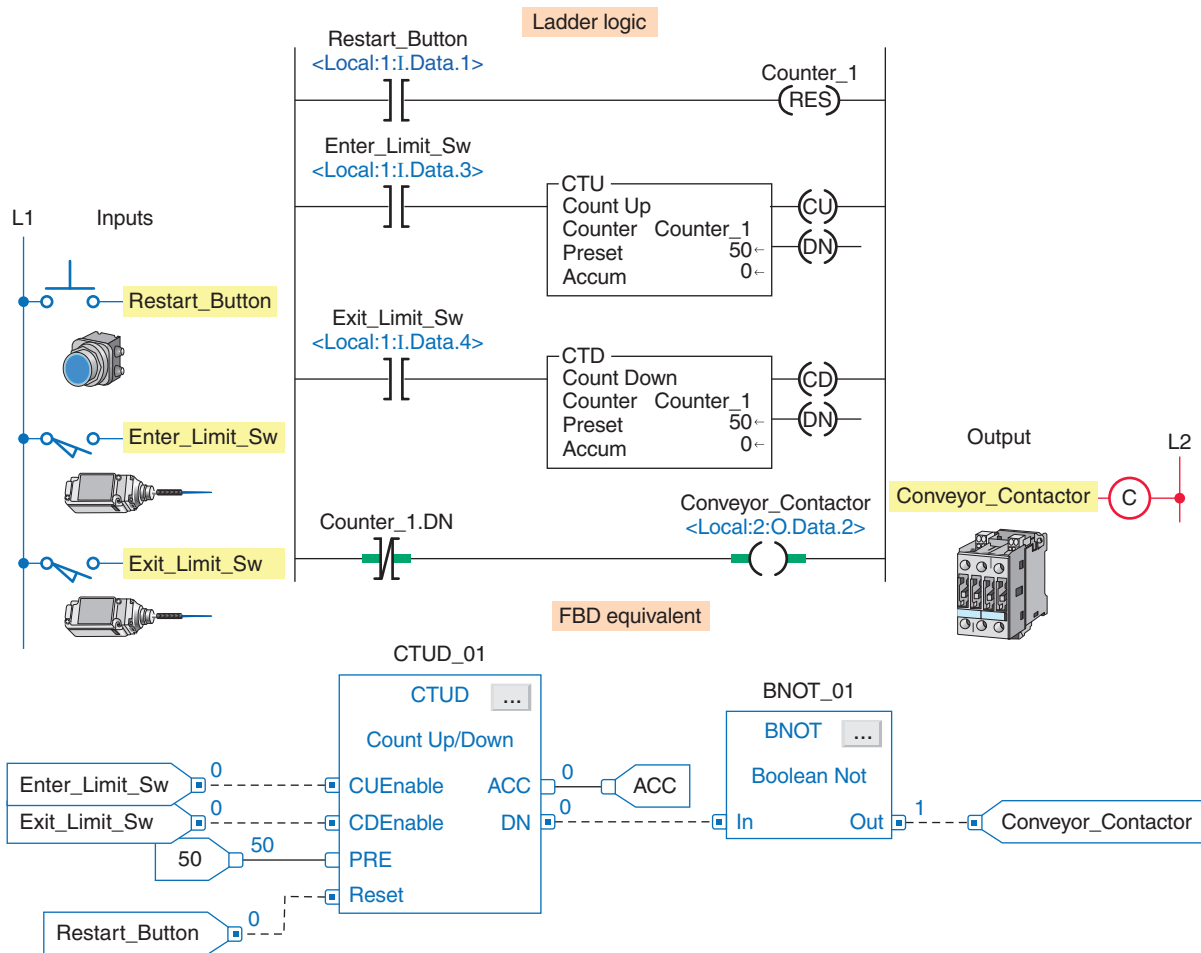


Figure 15-96 Comparison between ladder logic and the FBD equivalent for an Up/Down counter application.

Figure 15-97 shows a comparison between ladder logic and the FBD equivalent for the program used to test the accumulated value of a counter. The operation of the FBD can be summarized as follows:

- The function block routine is broken into four sheets.
- The order of the sheets does not affect the order in which the function blocks execute.
- When a function block routine executes, all sheets execute.
- Using one sheet for each device that is to be programmed helps organize your program and make it easier to understand.
- The use of the OCON and ICON named ACC enables the function blocks to be on different sheets of the same function block routine.
- The numbers and letters under the ACC output indicate the sheet number and location on the sheet where the output is used.

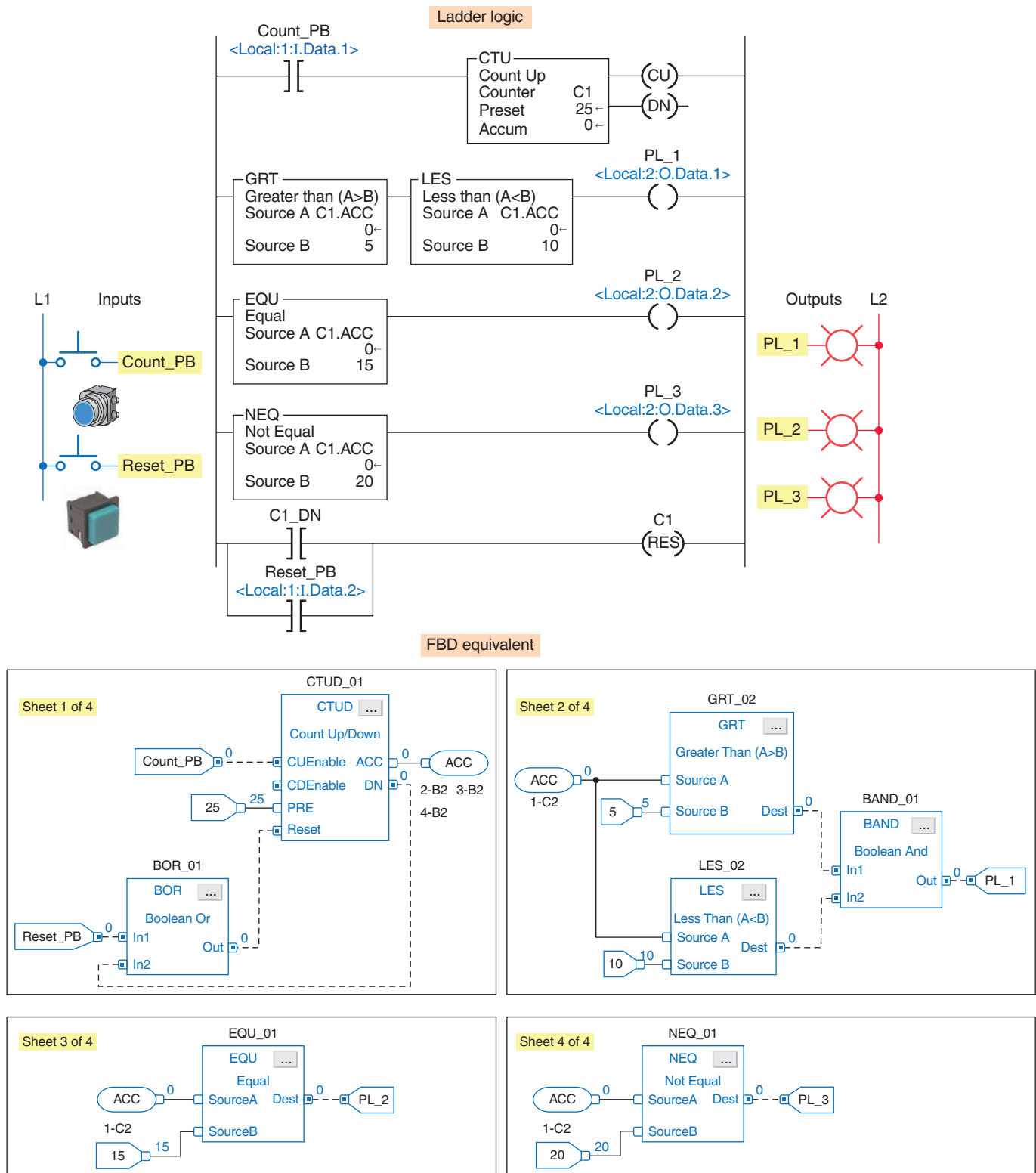


Figure 15-97 Comparison between ladder logic and the FBD equivalent for a program used to test the accumulated value of a counter.



PART 6 REVIEW QUESTIONS

1. Compare the graphical representation of a function block diagram to that of a logic ladder diagram.
2. Name the four basic elements of an FBD.
3. What do the solid and dashed interconnecting lines between FBD function blocks indicate?
4. What is an Add-On instruction?
5. How are the input and output parameter options for a function block set?
6. What does the dot on an input or output pin of a function block indicate?
7. Compare the functions of input and output reference tags.
8. Which pins of a function block are inputs and which are outputs?
9. Explain the role of input and output wire connectors.
10. How does the program scan function for an FBD program?
11. Explain data latching as it applies to function block inputs.
12. How is a function block feedback loop created?
13. What is the Assume Data Available indicator used for?
14. Outline how an FBD program is initiated.



PART 6 PROBLEMS

1. Write an FBD program that will cause the output, solenoid SOL_1, to be energized when pushbutton PB_1 is open and PB_2 is closed, and either limit switch LS_1 is open or limit switch LS_2 is closed. Assume all pushbuttons and limit switches are of the normally open type.
2. Modify the motor start/stop FBD program to include a second start/stop pushbutton station.
3. You are required to change the on-delay time of the 10 second timer program to 1 minute. What changes would have to be made to the FBD program?
4. Modify the Up/Down counter FBD program to include the following pilot lights:
 - PL_1 to come on when a part enters
 - PL_2 to come on when a part exits
 - PL_3 to come on when the buffer zone is full
5. Modify the test accumulated value of a counter FBD program as follows:
 - PL_1 to be on for an accumulated count between 0 and 5
 - PL_2 to be on for an accumulated count of 12
 - PL_3 to be on at all times except for when the accumulated count is 15