Allen-Bradley

SFC and ST Programming Languages

Excerpt from the *Logix5000 Controllers Common Procedures*, publication 1756-PM001

Programming Manual

Rockwell Automation

Important User Information

Solid state equipment has operational characteristics differing from those of electromechanical equipment. *Safety Guidelines for the Application, Installation and Maintenance of Solid State Controls* (Publication SGI-1.1 available from your local Rockwell Automation sales office or online at http://www.ab.com/manuals/gi) describes some important differences between solid state equipment and hard-wired electromechanical devices. Because of this difference, and also because of the wide variety of uses for solid state equipment, all persons responsible for applying this equipment must satisfy themselves that each intended application of this equipment is acceptable.

In no event will Rockwell Automation, Inc. be responsible or liable for indirect or consequential damages resulting from the use or application of this equipment.

The examples and diagrams in this manual are included solely for illustrative purposes. Because of the many variables and requirements associated with any particular installation, Rockwell Automation, Inc. cannot assume responsibility or liability for actual use based on the examples and diagrams.

No patent liability is assumed by Rockwell Automation, Inc. with respect to use of information, circuits, equipment, or software described in this manual.

Reproduction of the contents of this manual, in whole or in part, without written permission of Rockwell Automation, Inc. is prohibited.

Throughout this manual we use notes to make you aware of safety considerations.

WARNING

Identifies information about practices or circumstances that can cause an explosion in a hazardous environment, which may lead to personal injury or death, property damage, or economic loss.

IMPORTANT

Identifies information that is critical for successful application and understanding of the product.

ATTENTION

Identifies information about practices or circumstances that can lead to personal injury or death, property damage, or economic loss. Attentions help you:

- identify a hazard
- avoid a hazard
- recognize the consequence

SHOCK HAZARD

Labels may be located on or inside the drive to alert people that dangerous voltage may be present.

BURN HAZARD

Labels may be located on or inside the drive to alert people that surfaces may be dangerous temperatures.

Purpose of this Manual

This manual is an excerpt of *Logix5000 Controllers Common Procedures*, publication 1756-PM001. It provides step-by-step procedures on how to perform the following tasks, which are common to all Logix5000™ controllers:

- Design, program, and force a sequential function chart
- Program a routine using the structured text programming language

The term *Logix5000 controller* refers to any controller that is based on the Logix5000 operating system, such as:

- CompactLogix™ controllers
- ControlLogix® controllers
- FlexLogix™ controllers
- PowerFlex® 700S with DriveLogix controllers
- SoftLogix5800™ controllers

Who Should Use this Manual

This manual is intended for those individuals who program applications that use Logix5000 controllers, such as:

- software engineers
- control engineers
- application engineers
- instrumentation technicians

When to Use this Manual

Use this manual when you perform these actions:

- develop the basic code for your application
- modify an existing application
- perform isolated tests of your application

As you integrate your application with the I/0 devices, controllers, and networks in your system:

- Refer to the user manual for your specific type of controller.
- Use this manual as a reference, when needed.

How to Use this Manual

This manual is divided into the basic tasks that you perform while programming a Logix5000 controller.

- Each chapter covers a task.
- The tasks are organized in the sequence that you will typically perform them.

As you use this manual, you will see some terms that are formatted differently from the rest of the text:

Text that is:	Identifies:	For example:	Means:
Italic	the actual name of an item that you see on your screen or in an example	Right-click *User-Defined* …	Right-click on the item that is named User-Defined.
bold	an entry in the "Glossary"	Type a **name** …	If you want additional information, refer to **name** in the "Glossary." If you are viewing the PDF file of the manual, click **name** to jump to the glossary entry.
`courier`	information that you must supply based on your application (a variable)	Right-click `name_of_program` …	You must identify the specific program in your application. Typically, it is a name or variable that you have defined.
enclosed in brackets	a keyboard key	Press [Enter].	Press the Enter key.

Design a Sequential Function Chart

Chapter 5

Chapter 6

Program a Sequential Function Chart

Chapter 7

Program Structured Text

Chapter 14

Force Logic Elements

Design a Sequential Function Chart

When to Use This Procedure

Use this procedure to design a **sequential function chart (SFC)** for your process or system. An SFC is similar to a flowchart of your process. It defines the steps or states through which your system progresses. Use the SFC to:

- organize the functional specification for your system
- program and control your system as a series of steps and transitions

By using an SFC to specify your process, you gain these advantages:

- Since an SFC is a graphical representation of your process, it is easier to organize and read than a textual version. In addition, RSLogix 5000 software lets you:
  - add notes that clarify steps or capture important information for use later on
  - print the SFC to share the information with other individuals
- Since Logix5000 controllers support SFCs, you do not have to enter the specification a second time. You are programming your system as you specify it.

By using an SFC to program your process, you gain these advantages:

- graphical division of processes into its major logic pieces (steps)
- faster repeated execution of individual pieces of your logic
- simpler screen display
- reduced time to design and debug your program
- faster and easier troubleshooting
- direct access to the point in the logic where a machine faulted
- easy updates and enhancements

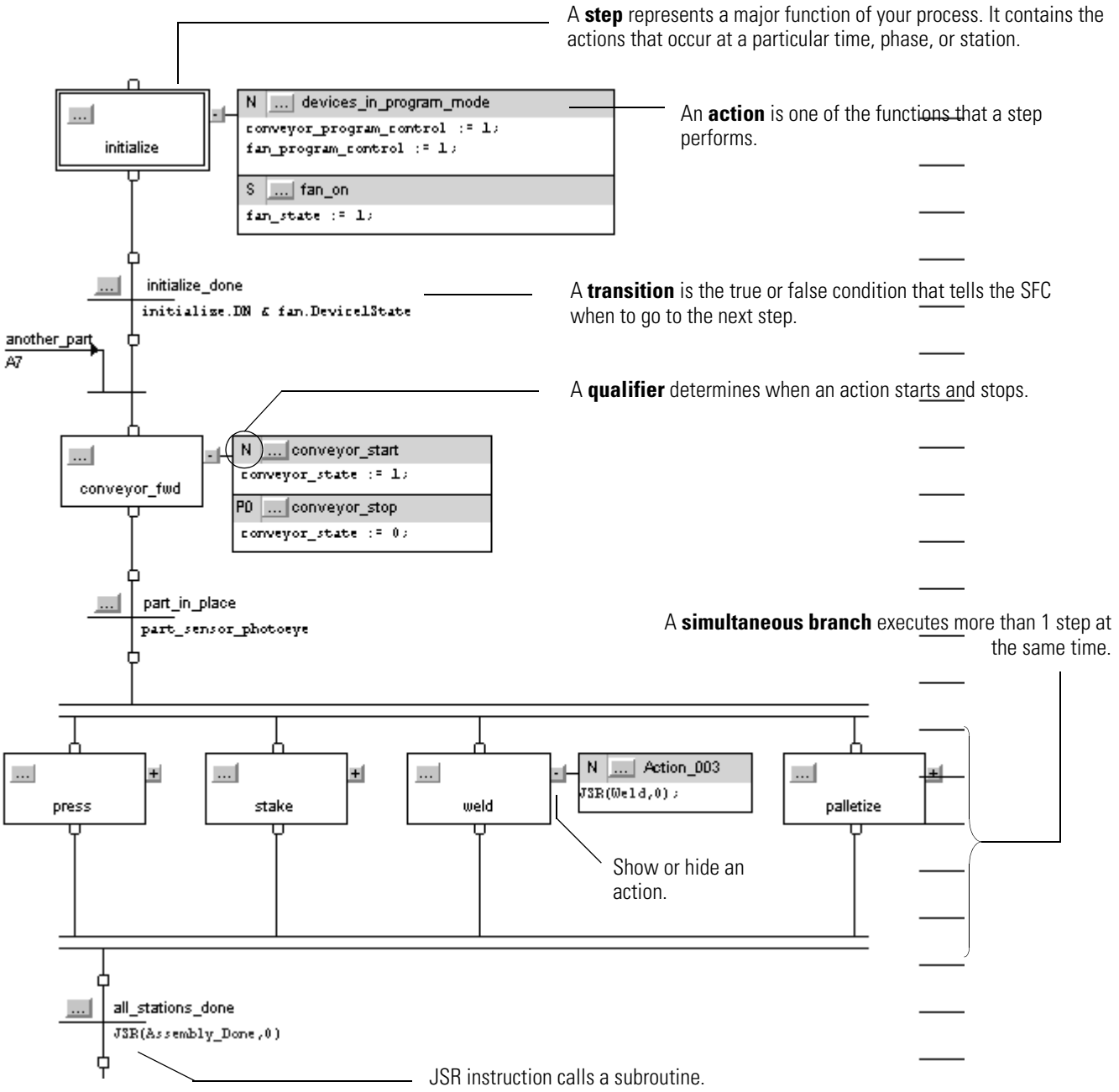
How to Use This Procedure

Typically, the development of an SFC is an iterative process. If you prefer, you can use RSLogix 5000 software to draft and refine your SFC. For specific procedures on how to enter an SFC, see “Program a Sequential Function Chart” on page 6-1.

What is a Sequential Function Chart?

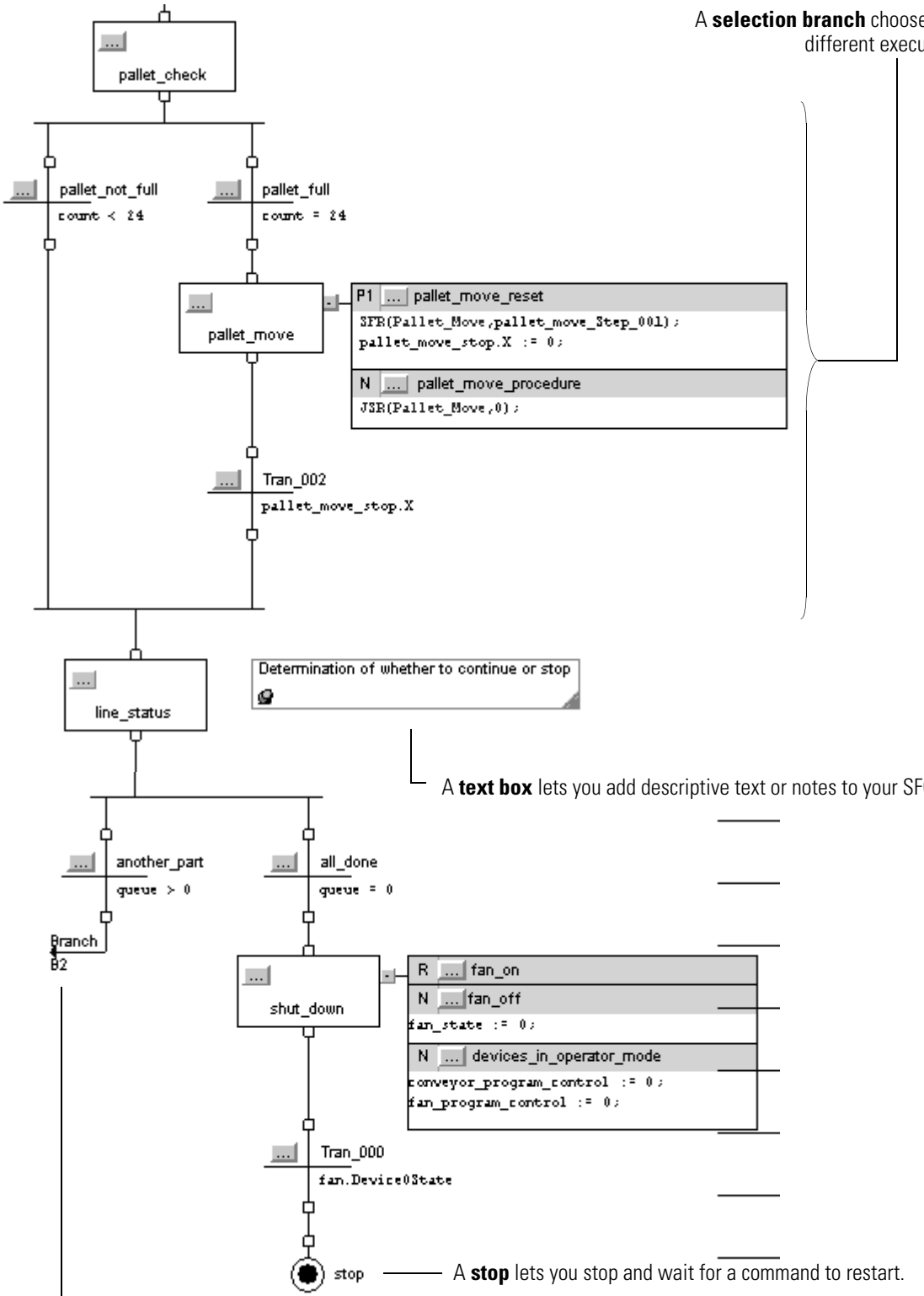
A **sequential function chart** (SFC) is similar to a flowchart. It uses steps and transitions to perform specific operations or actions. Figure 5.1 and Figure 5.2 is an example that shows the elements of an SFC:

Figure 5.1 SFC Example



(continued on next page)

Figure 5.2 SFC Example (continued from previous page)



How to Design an SFC: Overview

To design an SFC, you perform these tasks:

- ☐ Define the Tasks
- ☐ Choose How to Execute the SFC
- ☐ Define the Steps of the Process
- ☐ Organize the Steps
- ☐ Add Actions for Each Step
- ☐ Describe Each Action in Pseudocode
- ☐ Choose a Qualifier for an Action
- ☐ Define the Transition Conditions
- ☐ Transition After a Specified Time
- ☐ Turn Off a Device at the End of a Step
- ☐ Keep Something On From Step-to-Step
- ☐ End the SFC
- ☐ Nest an SFC
- ☐ Configure When to Return to the OS/JSR
- ☐ Pause or Reset an SFC

The remaining sections of this chapter describe in detail how to perform each task.

Define the Tasks

The first step in the development of an SFC is to separate the configuration and regulation of devices from the commands to those devices. Logix5000 controllers let you divide your project into one **continuous task** and multiple **periodic tasks**.

1. Organize your project as follows:

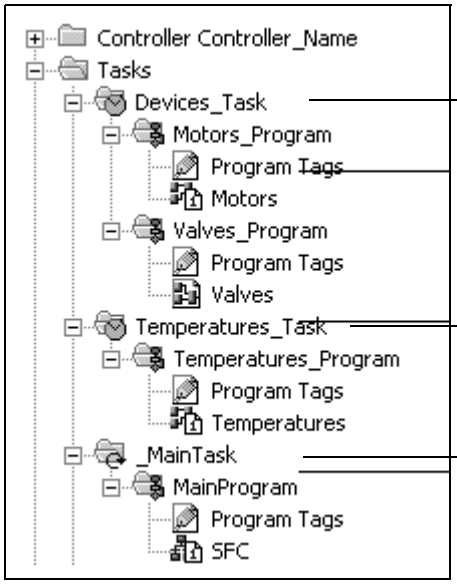
These functions:	Go here:
configure and regulate devices	periodic task
command a device to a specific state	SFC in the continuous task
sequence the execution of your process	

2. For those functions that go in a periodic task, group the functions according to similar update rates. Create a periodic task for each update rate.

   For example, your 2-state devices may require faster updates than your PID loops. Use separate periodic tasks for each.

The following example shows a project that uses two periodic tasks to regulate motors, valves, and temperature loops. The project uses an SFC to control the process.

EXAMPLE Define the Tasks



This task (periodic) uses function block diagrams to turn on or off motors and open or close valves. The SFC in *MainTask* commands the state for each device. The function block diagrams set and maintain that state.

This task (periodic) uses function block diagrams to configure and regulate temperature loops. The SFC in *MainTask* commands the temperatures. The function block diagrams set and maintain those temperatures.

This task (continuous) executes the sequential function chart (SFC). The SFC commands the specific state or temperature for each device or temperature loop.

Choose How to Execute the SFC

To execute an SFC, either configure it as the main routine for a program or call it as a subroutine.

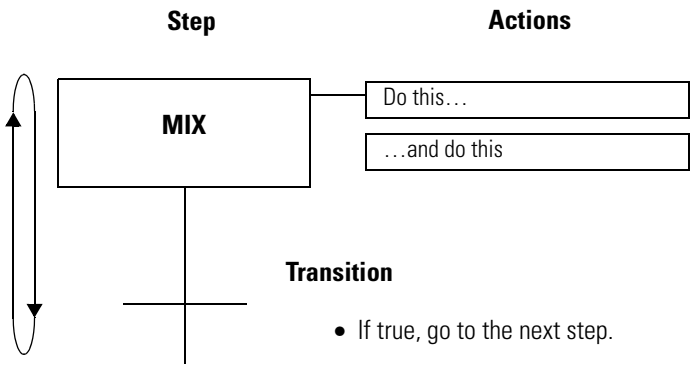
If:	Then:
The SFC is the only routine in the program.	Configure the SFC as the main routine for the program.
The SFC calls *all* the other routines of the program.	
The program requires other routines to execute independent of the SFC.	1. Configure another routine as the main routine for the program. 2. Use the main routine to call the SFC as a subroutine.
The SFC uses boolean actions.	

If the SFC uses boolean actions, then other logic must run independent of the SFC and monitor status bits of the SFC.

Define the Steps of the Process

A **step** represents a major function of your process. It contains the actions that occur at a particular time, phase, or station.

A step executes continuously until a transition tells the SFC to go to the next step.

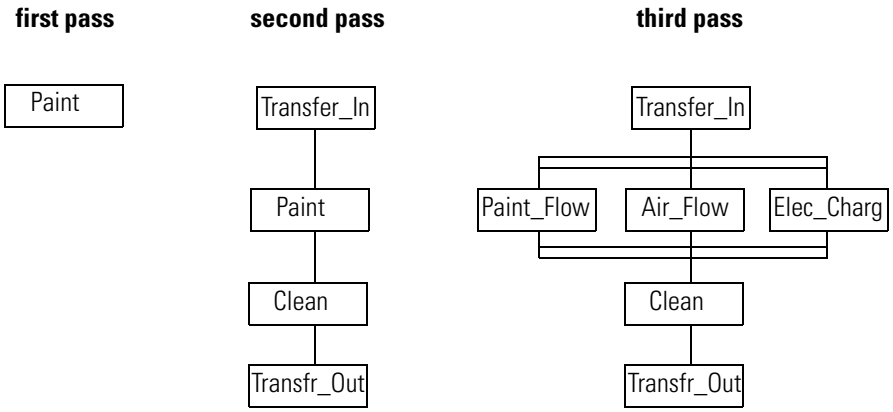


A **transition** ends a step. The transition defines the physical conditions that must occur or change in order to go to the next step.

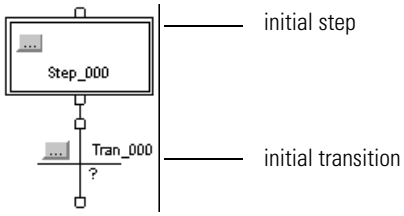
Follow These Guidelines

As you define the steps of your process, follow these guidelines:

- Start with large steps and refine the steps in several passes.



- When you first open an SFC routine, it contains an initial step and transition. Use this step to initialize your process.



- To identify a step, look for a physical change in your system, such as new part that is in position, a temperature that is reached, a preset time that is reached, or a recipe selection that occurs. The step is the actions that take place before that change.

- Stop when your steps are in meaningful increments. For example:

This organization of steps:	Is:
produce_solution	probably too large
set_mode, close_outlet, set_temperature, open_inlet_a, close_inlet_a, set_timer, reset_temperature, open_outlet, reset_mode	probably too small
preset_tank, add_ingredient_a, cook, drain	probably about right

SFC_STEP Structure

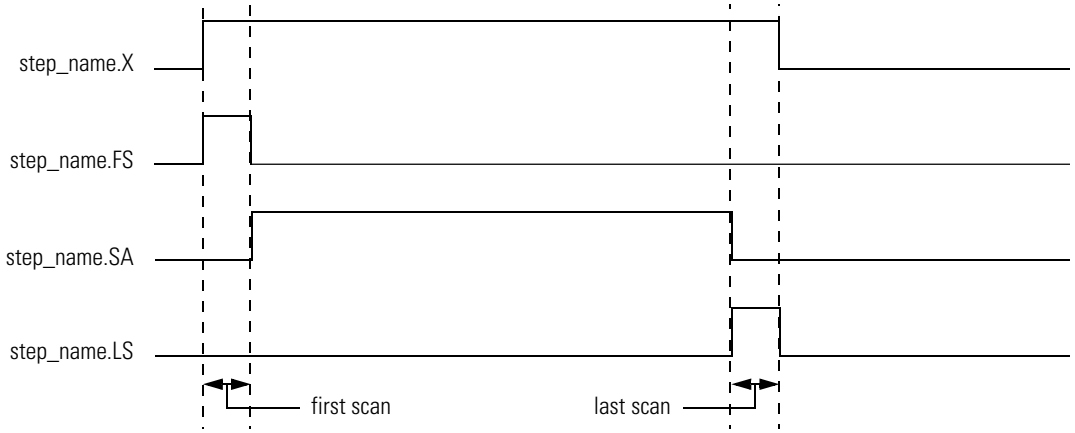
Each step uses a tag to provide information about the step. Access this information via either the *Step Properties* dialog box or the *Monitor Tags* tab of the *Tags* window:

If you want to:	Then check or set this member:	Data type:	Details:
determine how long a step has been active (milliseconds)	T	DINT	When a step becomes active, the Timer (T) value resets and then starts to count up in milliseconds. The timer continues to count up until the step goes inactive, regardless of the Preset (PRE) value.
flag when the step has been active for a specific length of time (milliseconds)	PRE	DINT	Enter the time in the Preset (PRE) member. When the Timer (T) reaches the Preset value, the Done (DN) bit turns on and stays on until the step becomes active again. As an option, enter a numeric expression that calculates the time at runtime.
	DN	BOOL	When the Timer (T) reaches the Preset (PRE) value, the Done (DN) bit turns on and stays on until the step becomes active again.
flag if a step did not execute long enough	LimitLow	DINT	Enter the time in the LimitLow member (milliseconds). • If the step goes inactive before the Timer (T) reaches the LimitLow value, the AlarmLow bit turns on. • The AlarmLow bit stays on until you reset it. • To use this alarm function, turn on (check) the AlarmEnable (AlarmEn) bit. As an option, enter a numeric expression that calculates the time at runtime.
	AlarmEn	BOOL	To use the alarm bits, turn on (check) the AlarmEnable (AlarmEn) bit.
	AlarmLow	BOOL	If the step goes inactive before the Timer (T) reaches the LimitLow value, the AlarmLow bit turns on. • The bit stays on until you reset it. • To use this alarm function, turn on (check) the AlarmEnable (AlarmEn) bit.

If you want to:	Then check or set this member:	Data type:	Details:
flag if a step is executing too long	LimitHigh	DINT	Enter the time in the LimitHigh member (milliseconds). • If the Timer (T) reaches the LimitHigh value, the AlarmHigh bit turns on. • The AlarmHigh bit stays on until you reset it. • To use this alarm function, turn on (check) the AlarmEnable (AlarmEn) bit. As an option, enter a numeric expression that calculates the time at runtime.
	AlarmEn	BOOL	To use the alarm bits, turn on (check) the AlarmEnable (AlarmEn) bit.
	AlarmHigh	BOOL	If the Timer (T) reaches the LimitHigh value, the AlarmHigh bit turns on. • The bit stays on until you reset it. • To use this alarm function, turn on (check) the AlarmEnable (AlarmEn) bit.
do something while the step is active (including first and last scan)	X	BOOL	The X bit is on the entire time the step is active (executing). Typically, we recommend that you use an action with a *N Non-Stored* qualifier to accomplish this.
do something one time when the step becomes active	FS	BOOL	The FS bit is on during the first scan of the step. Typically, we recommend that you use an action with a *P1 Pulse (Rising Edge)* qualifier to accomplish this.
do something while the step is active, *except* on the first and last scan	SA	BOOL	The SA bit is on when the step is active except during the first and last scan of the step.
do something one time on the last scan of the step	LS	BOOL	The LS bit is on during the last scan of the step. Use this bit only if you do the following: On the *Controller Properties* dialog box, *SFC Execution* tab, set the *Last Scan of Active Step* to *Don't Scan* or *Programmatic reset*. Typically, we recommend that you use an action with a *P0 Pulse (Falling Edge)* qualifier to accomplish this.

If you want to:	Then check or set this member:	Data type:	Details:
determine the target of an SFC Reset (SFR) instruction	Reset	BOOL	An SFC Reset (SFR) instruction resets the SFC to a step or stop that the instruction specifies. • The Reset bit indicates to which step or stop the SFC will go to begin executing again. • Once the SFC executes, the Reset bit clears.
determine the maximum time that a step has been active during any of its executions	TMax	DINT	Use this for diagnostic purposes. The controller clears this value only when you choose the *Restart Position* of *Restart at initial step* and the controller changes modes or experiences a power cycle.
determine if the Timer (T) value rolls over to a negative value	OV	BOOL	Use this for diagnostic purposes.
determine how many times a step has become active	Count	DINT	This is *not* a count of scans of the step. • The count increments each time the step becomes active. • It increments again only after the step goes inactive and then active again. • The count resets only if you configure the SFC to restart at the initial step. With that configuration, it resets when the controller changes from program mode to run mode.
use one tag for the various status bits of this step	Status	DINT	**For this member:** / **Use this bit:** Reset / 22 AlarmHigh / 23 AlarmLow / 24 AlarmEn / 25 OV / 26 DN / 27 LS / 28 SA / 29 FS / 30 X / 31

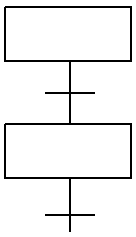
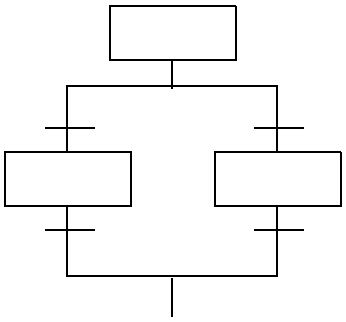
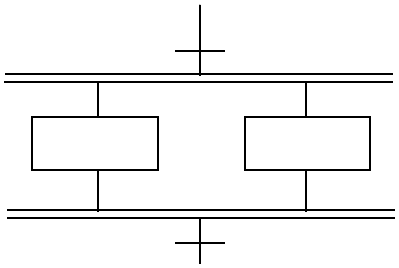
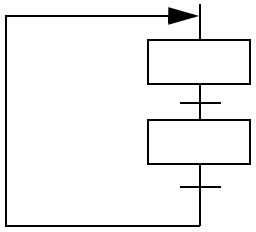
The following diagram shows the relationship of the X, FS, SA, and LS bits.


step_name.X
step_name.FS
step_name.SA
step_name.LS
first scan
last scan

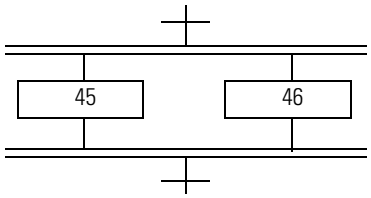
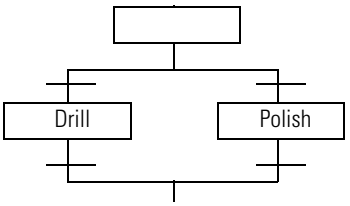
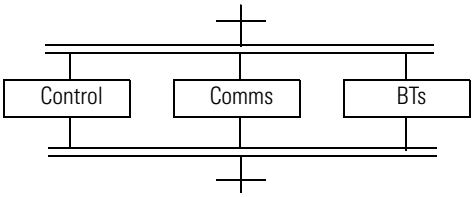
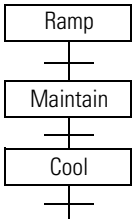
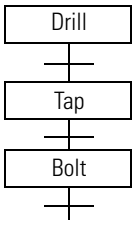
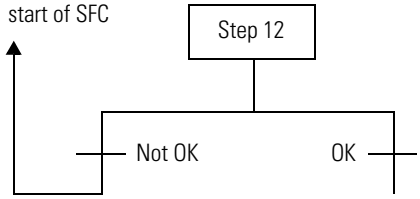
Organize the Steps

Once you define the steps of your process, organize them into sequences, simultaneous branches, selection branches, or loops.

Overview

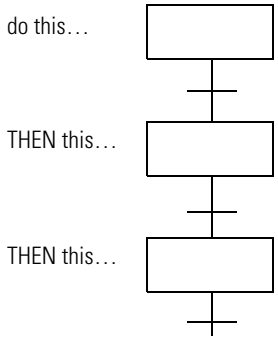
To:	Use this structure:	With these considerations:
Execute 1 or more steps in sequence: • One executes repeatedly. • Then the next executes repeatedly.	**Sequence** 	The SFC checks the transition at the end of the step: • If true, the SFC goes to the next step. • If false, the SFC repeats the step.
• Choose between alternative steps or groups of steps depending on logic conditions • Execute a step or steps or skip the step or steps depending on logic conditions	**Selection Branch** 	• It is OK for a path to have no steps and only a transition. This lets the SFC skip the selection branch. • By default, the SFC checks from left to right the transitions that start each path. It takes the first true path. • If no transitions are true, the SFC repeats the previous step. • RSLogix 5000 software lets you change the order in which the SFC checks the transitions.
Execute 2 or more steps at the same time. All paths must finish before continuing the SFC	**Simultaneous Branch** 	• A single transition ends the branch. • The SFC checks the ending transition after the last step in each path has executed at least once. If the transition is false, the SFC repeats the previous step.
Loop back to a previous step	**Wire to a Previous Step** 	• Connect the wire to the step or simultaneous branch to which you want to go. • *Do not* wire into, out of, or between a simultaneous branch.

Here are some examples of SFC structures for different situations:

Example situation:	Example solution:
Station 45 and 46 of an assembly line work on parts simultaneously. When both stations are done, the parts move down 1 station.	**Simultaneous Branch** 
Depending on the build code, a station either drills or polishes.	**Selection Branch** 
To simplify my programming, I want to separate communications and block transfers from other control logic. All occur at the same time.	**Simultaneous Branch** 
In a heat treating area, the temperature ramps up at a specific rate, maintains that temperature for a specific duration, and then cools at a specific rate.	**Sequence** 
At station 12, the machine drills, taps, and bolts a part. The steps occur one after the other.	**Sequence** 
Step 12 inspects a process for the correct mix of chemicals. • If OK, then continue with the remaining steps. • If not OK, go to the top of the SFC and purge the system.	**Wire** 

Sequence

A sequence is a group of steps that execute one after the other.



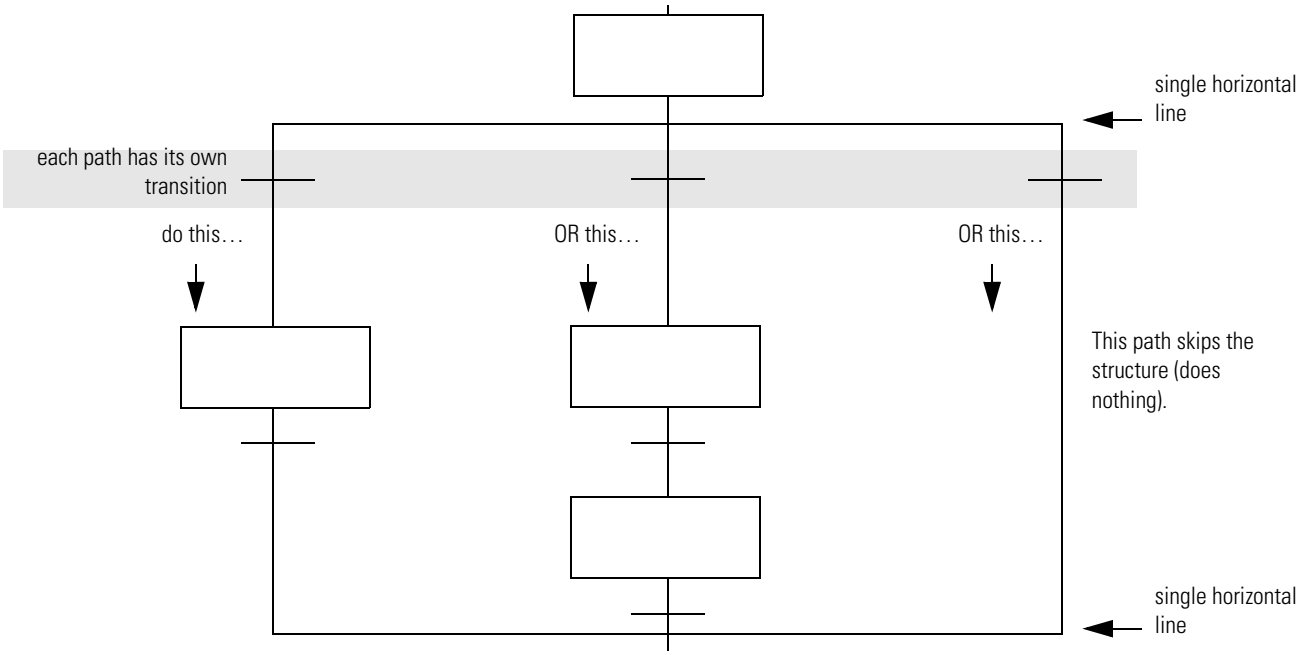
For a detailed diagram of the execution of a sequence of steps, see Figure 5.5 on page 5-52.

To override the state of a transition, see "Force Logic Elements" on page 14-1.

Selection Branch

A selection branch represents a choice between one path (step or group of steps) or another path (i.e., an OR structure).

- Only one path executes.
- By default the SFC checks the transitions from left to right.
  - The SFC takes the first true path.
  - RSLogix 5000 software lets you change the order in which the SFC checks the transitions. See "Program a Sequential Function Chart" on page 6-1.



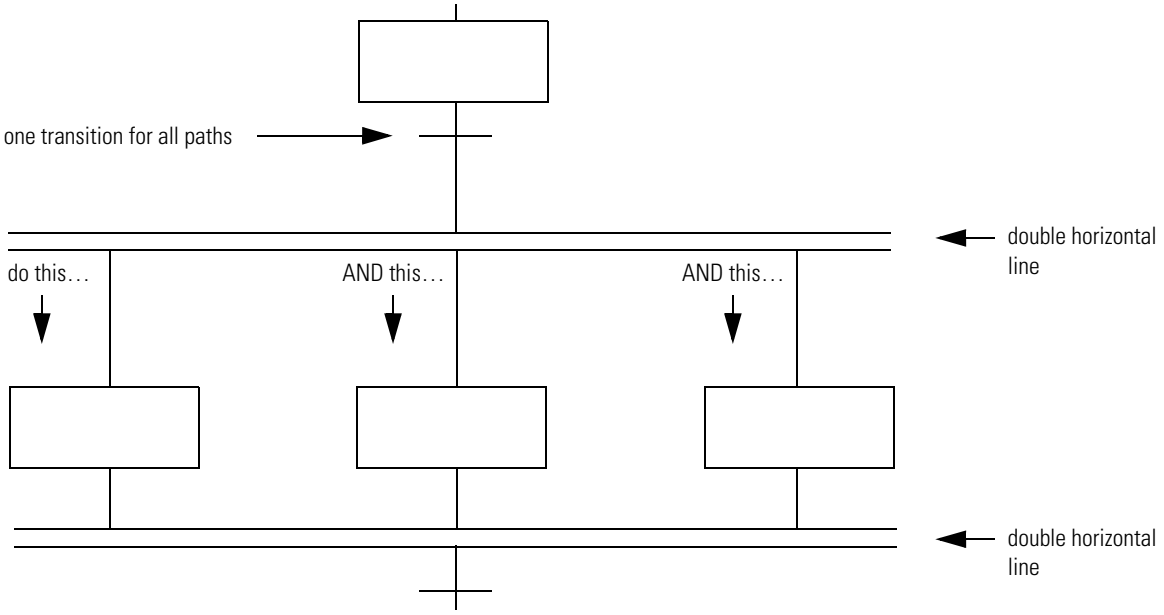
For a detailed diagram of the execution of a selection branch, see Figure 5.7 on page 5-54.

To override the state of a transition, see "Force Logic Elements" on page 14-1.

Simultaneous Branch

A simultaneous branch represents paths (steps or group of steps) that occur at the same time (i.e., an AND structure).

- All paths execute.
- All paths must finish before continuing with the SFC.
- The SFC checks the transition after the last step of each path has executed at least once.



For a detailed diagram of the execution of a simultaneous branch, see Figure 5.6 on page 5-53.

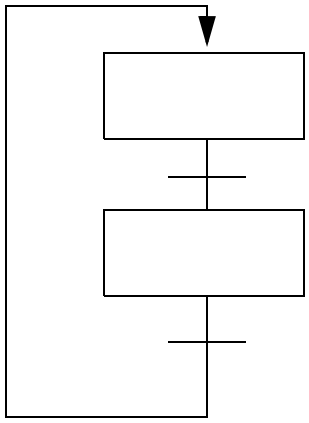
To override the branch and prevent a path from executing, see "Force Logic Elements" on page 14-1.

Wire to a Previous Step

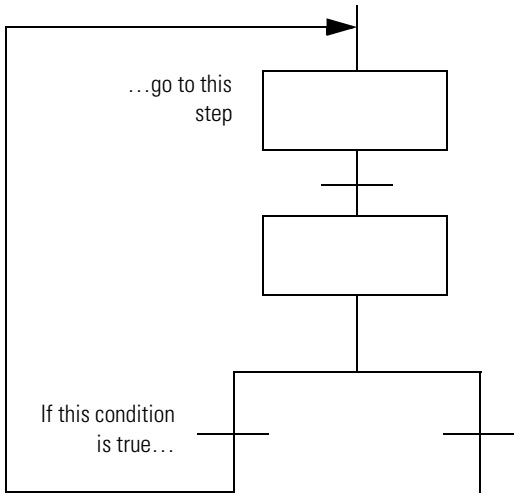
In addition to connecting steps in sequences, simultaneous branches, and selection branches, you can connect a step to a previous point in your SFC. This lets you:

- loop back and repeat steps
- return to the beginning of the SFC and start over

For example:



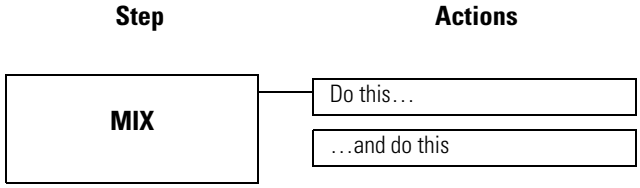
simple loop that repeats the entire SFC



path of a selection branch that returns to a previous step

Add Actions for Each Step

Use **actions** to divide a step into the different functions that the step performs, such as commanding a motor, setting the state of a valve, or placing a group of devices in a specific mode.



How Do You Want to Use the Action?

There are two types of actions:

If you want to:	Then:
execute structured text directly in the SFC	Use a Non-Boolean Action
call a subroutine	
use the automatic reset option to reset data upon leaving a step	
only set a bit and program other logic to monitor the bit to determine when to execute.	Use a Boolean Action

Use a Non-Boolean Action

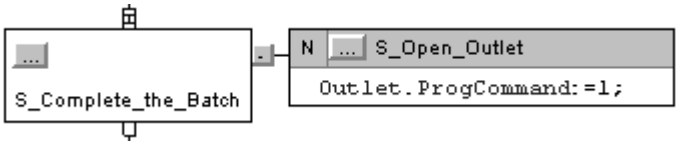
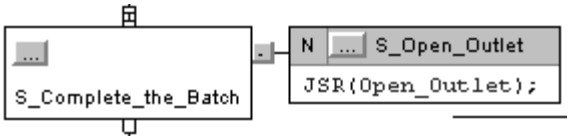
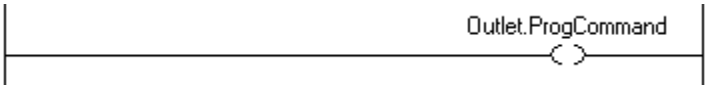
A non-boolean action contains the logic for the action. It uses structured text to execute assignments and instructions or call a subroutine.

With non-boolean actions, you also have the option to **postscan** (automatically reset) the assignments and instructions before leaving a step:

- During postscan the controller executes the assignments and instructions as if all conditions are false.
- The controller postscans both embedded structured text and any subroutine that the action calls.

To automatically reset assignments and instructions, see “Turn Off a Device at the End of a Step” on page 5-32.

To program a non-boolean action, you have the following options:

If you want to:	Then:
• execute your logic without additional routines • use structured text assignments, constructs, and instructions	Embed structured text. For example:  When the *S_Complete_the_Batch* step is active, the *S_Open_Outlet* action executes. The action sets the *Outlet.ProgCommand* tag equal to 1, which opens the outlet valve.
• re-use logic in multiple steps • use another language to program the action, such as ladder logic • nest an SFC	Call a subroutine. For example:  When the *S_Complete_the_Batch* step is active, the *S_Open_Outlet* action executes. The action calls the *Open_Outlet* routine. **Open_Outlet Routine**  When the *Open_Outlet* routine executes, the OTE instruction sets the *Outlet.ProgCommand* tag equal to 1, which opens the outlet valve.

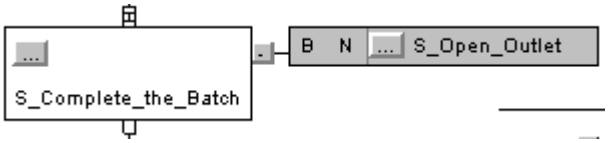
You *cannot* reuse a non-boolean action within the same SFC except to reset a stored action. Only one instance of a specific non-boolean action is permitted per SFC.

Use a Boolean Action

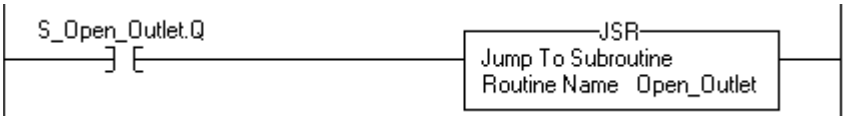
A boolean action contains no logic for the action. It simply sets a bit in its tag (SFC_ACTION structure). To do the action, other logic must monitor the bit and execute when the bit is on.

With boolean actions, you have to manually reset the assignments and instructions that are associated with the action. Since there is no link between the action and the logic that performs the action, the automatic reset option does not effect boolean actions.

Here is an example:



When the *S_Complete_the_Batch* step is active, the *S_Open_Outlet* action executes. When the action is active, its Q bit turns on.



A ladder logic routine monitors the Q bit (*S_Open_Outlet.Q*). When the Q bit is on, the JSR instruction executes and opens the outlet valve.

You can reuse a boolean action multiple times within the same SFC.

SFC_ACTION Structure

Each action (non-boolean and boolean) uses a tag to provide information about the action. Access this information via either the

Action Properties dialog box or the *Monitor Tags* tab of the *Tags* window:

If you want to:	Then check or set this member:	Data type:	Details:
determine when the action is active	Q	BOOL	The status of the Q bit depends on whether the action is a boolean action or non-boolean action: **If the action is:** / **Then the Q bit is:** boolean: on (1) the entire time the action is active, including the last scan of the action non-boolean: on (1) while the action is active but off (0) at the last scan of the action To use a bit to determine when an action is active, use the Q bit.
	A	BOOL	The A bit is on the entire time the action is active.
determine how long an action has been active (milliseconds)	T	DINT	When an action becomes active, the Timer (T) value resets and then starts to count up in milliseconds. The timer continues to count up until the action goes inactive, regardless of the Preset (PRE) value.
use one of these time-based qualifiers: L, SL, D, DS, SD	PRE	DINT	Enter the time limit or delay in the Preset (PRE) member. The action starts or stops when the Timer (T) reaches the Preset value. As an option, enter a numeric expression that calculates the time at runtime.
determine how many times an action has become active	Count	DINT	This is *not* a count of scans of the action. • The count increments each time the action becomes active. • It increments again only after the action goes inactive and then active again. • The count resets only if you configure the SFC to restart at the initial step. With that configuration, it resets when the controller changes from program mode to run mode.
use one tag for the various status bits of this action	Status	DINT	**For this member:** / **Use this bit:** Q: 30 A: 31

Describe Each Action in Pseudocode

To organize the logic for an action, first describe the action in pseudocode. If you are unfamiliar with pseudocode, follow these guidelines:

- Use a series of short statements that describe exactly what should happen.

- Use terms or symbols such as: if, then, otherwise, until, and, or, =, >, <.
- Sequence the statements in the order that they should execute.
- If necessary, name the conditions to check first (when 1st) and then the action to take second (what 2nd).

Enter the pseudocode into the body of the action. After you enter the pseudocode, you can:

- Refine the pseudocode so it executes as structured text.
- Use the pseudocode to design your logic and leave the pseudocode as comments. Since all structured text comments download to the controller, your pseudocode is always available as documentation for the action.

To convert the pseudocode to structured text comments, add the following comment symbols:

For a comment:	Use one of these formats:
on a single line	`//comment`

that spans more than one line	`(*start of comment . . . end of comment*)`
	
	`/*start of comment . . . end of comment*/`

Choose a Qualifier for an Action

Each action (non-boolean and boolean) uses a **qualifier** to determine when it starts and stops.

The default qualifier is *Non-Stored*. The action starts when the step is activated and stops when the step is deactivated.

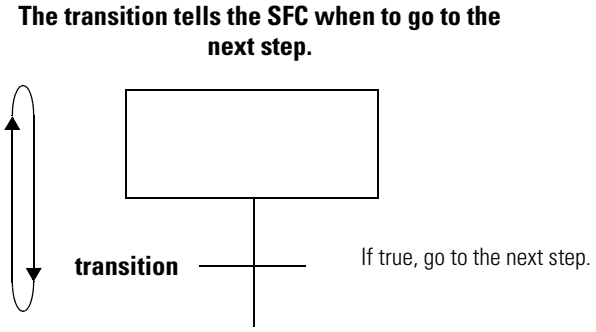
To change when an action starts or stops, assign a different qualifier:

Table 5.1 Choose a Qualifier for an Action

If you want the action to:	And:	Then assign this qualifier:	Which means:
start when the step is activated	stop when the step is deactivated	N	Non-Stored
	execute only once	P1	Pulse (Rising Edge)
	stop before the step is deactivated or when the step is deactivated	L	Time Limited
	stay active until a *Reset* action turns off this action	S	Stored
	stay active until a *Reset* action turns off this action or a specific time expires, even if the step is deactivated	SL	Stored and Time Limited
start a specific time after the step is activated *and* the step is still active	stop when the step is deactivated	D	Time Delayed
	stay active until a *Reset* action turns off this action	DS	Delayed and Stored
start a specific time after the step is activated, even if the step is deactivated before this time	stay active until a *Reset* action turns off this action	SD	Stored and Time Delayed
execute once when the step is activated	execute once when the step is deactivated	P	Pulse
start when the step is deactivated	execute only once	P0	Pulse (Falling Edge)
turn off (reset) a stored action: • S Stored • SL Stored and Time Limited • DS Delayed and Stored • SD Stored and Time Delayed	——————————►	R	Reset

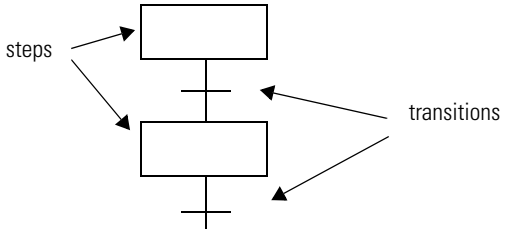
Define the Transition Conditions

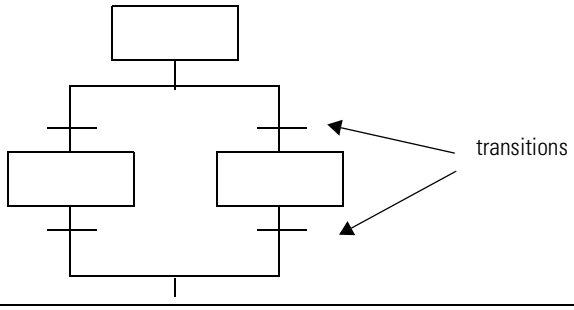
The transition is the physical conditions that must occur or change in order to go to the next step.

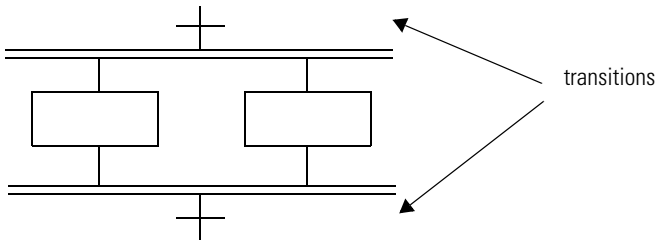


Transitions occur in the following places:

For this structure:	Make sure that:
sequence	A transition is between each step.
selection branch	Transitions are inside the horizontal lines.
simultaneous branch	Transitions are outside the horizontal lines.







Here are two examples of transitions:

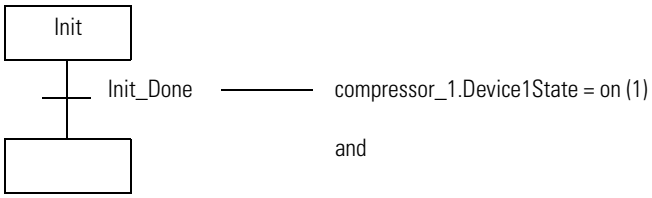
EXAMPLE

You want to:

a. Turn on 2 compressors. When a compressor is on, the Device1State bit is on.

b. When both compressors are on, go to the next step.

Solution:



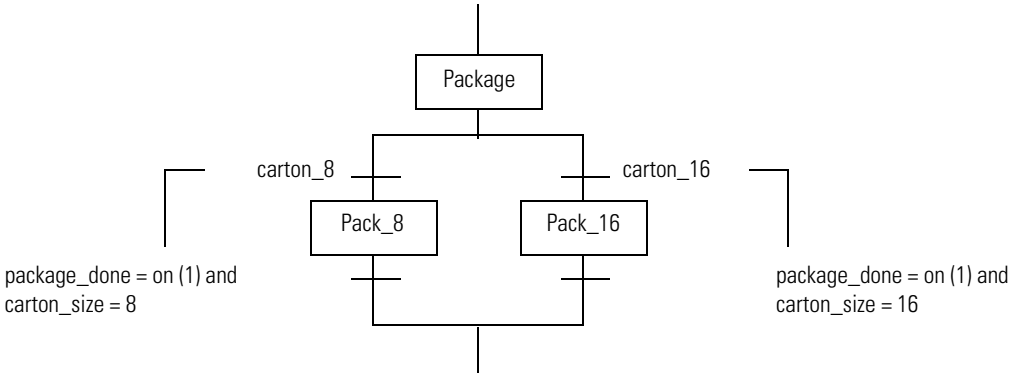
EXAMPLE

You want to:

a. Package the product. When the product is in the package, the *package_done* bit turns on.

b. Pack the product either 8 per carton or 16 per carton.

Solution:



To override the state of a transition, see "Force Logic Elements" on page 14-1.

Transition Tag

Each transition uses a BOOL tag to represent the true or false state of the transition.

If the transition is:	The value is:	And:
true	1	The SFC goes to the next step.
false	0	The SFC continues to execute the current step.

How Do You Want to Program the Transition?

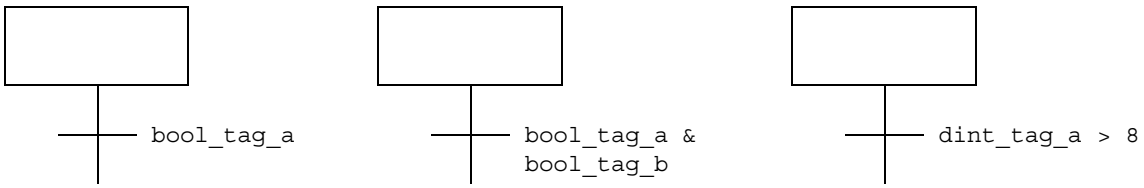
To program the transition, you have these options:

If you want to:	Then:
enter the conditions as an expression in structured text	Use a BOOL Expression
enter the conditions as instructions in another routine	Call a Subroutine
use the same logic for multiple transitions	

Use a BOOL Expression

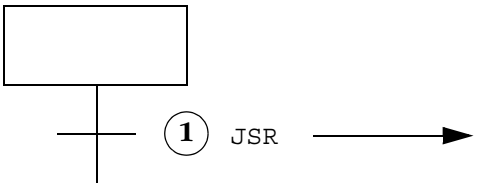
The simplest way to program the transition is to enter the conditions as a **BOOL expression** in structured text. A BOOL expression uses bool tags, relational operators, and logical operators to compare values or check if conditions are true or false. For example, `tag1>65`.

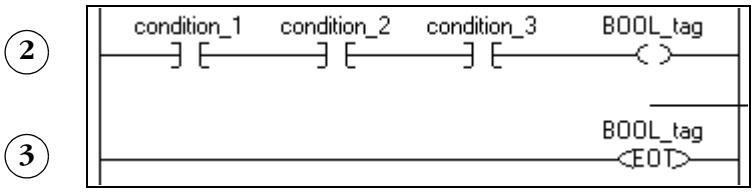
Here are some examples of BOOL expressions.



Call a Subroutine

To use a subroutine to control a transition, include an End Of Transition (EOT) instruction in the subroutine. The EOT instruction returns the state of the conditions to the transition, as shown below.





- or -

2

```
If condition_1 & condition_2 &
condition_3 then
     BOOL_tag := 1;
Else
     BOOL_tag := 0;
End_if;
```

3

```
EOT(BOOL_tag);
```

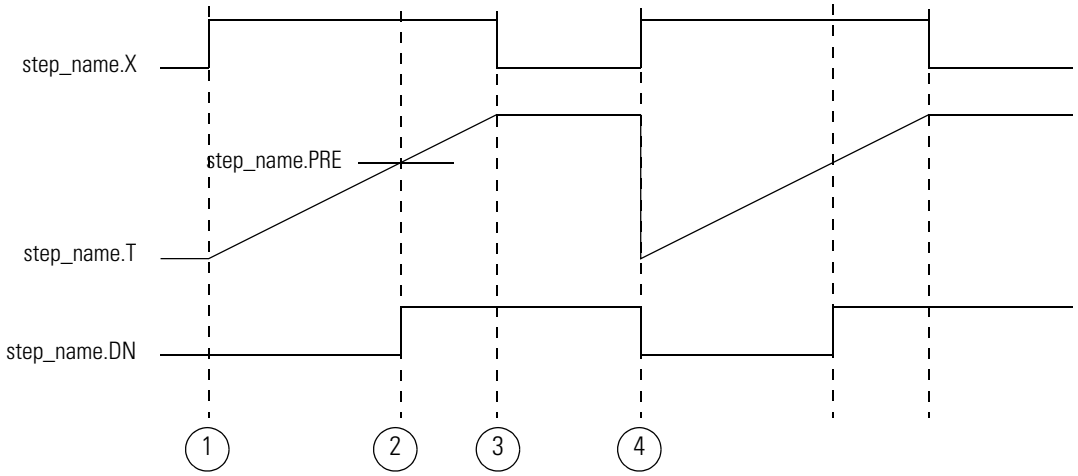
1. Call a subroutine.
2. Check for the required conditions. When those conditions are true, turn on a BOOL tag.
3. Use an EOT instruction to set the state of the transition equal to the value of the BOOL tag. When the BOOL tag is on (true), the transition is true.

Transition After a Specified Time

Each step of the SFC includes a millisecond timer that runs whenever the step is active. Use the timer to:

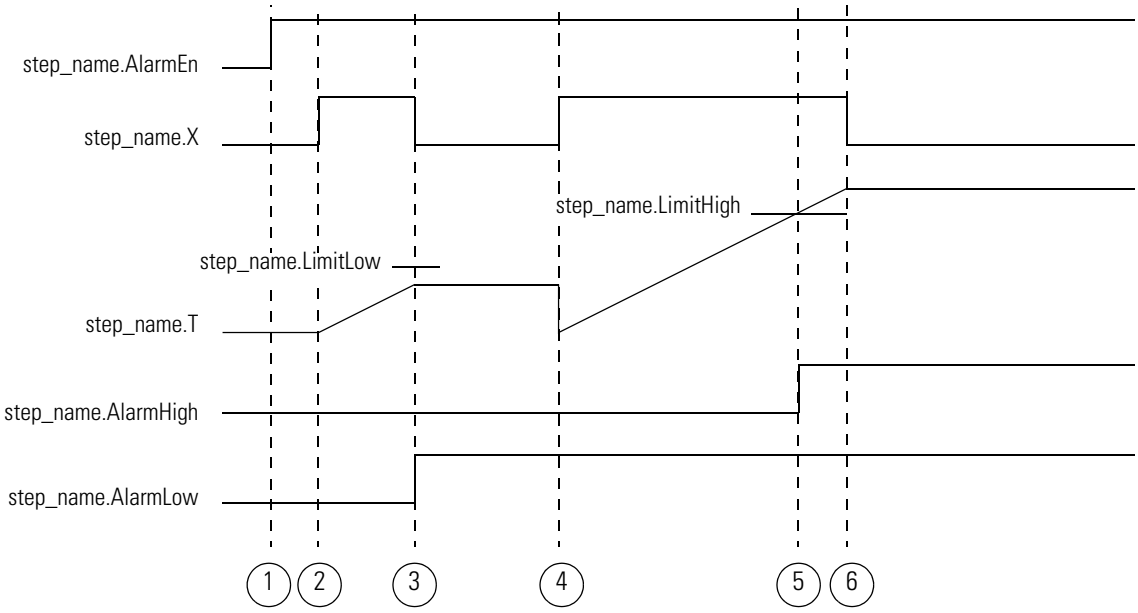
- signal when the step has run for the required time and the SFC should go to the next step
- signal when the step has run too long and the SFC should go to an error step

Figure 5.3 The following diagram shows the action of the timer and associated bits of a step:



Description:
1. Step becomes active. X bit turns on. Timer (T) begins to increment.
2. Timer reaches the Preset (PRE) value of the step. DN bit turns on. Timer continues to increment.
3. Step becomes inactive. X bit turns off. Timer retains its value. DN remains on.
4. Step becomes active. X bit turns on. Timer clears and then begins to increment. DN bit turns off.

Figure 5.4 The following diagram shows the action of the low and high alarms for a step:



Description:
1. AlarmEn is on. To use the low and high alarms turn this bit on. Turn the bit on via the properties dialog box or the tag for the step.
2. Step becomes active. X bit turns on. Timer (T) begins to increment.
3. Step becomes inactive. X bit turns off. Timer retains its value. Since Timer is less than LimitLow, AlarmLow bit turns on.

Description:
4. Step becomes active. X bit turns on. Timer clears and then begins to increment. AlarmLow stays on. (You have to manually turn it off.)
5. Timer reaches the LimitHigh value of the step. AlarmHigh bit turns on. Timer continues to increment.
6. Step becomes inactive. X bit turns off. Timer retains its value. AlarmHigh stays on. (You have to manually turn it off.)

Here is an example of the use of the Preset time of a step.

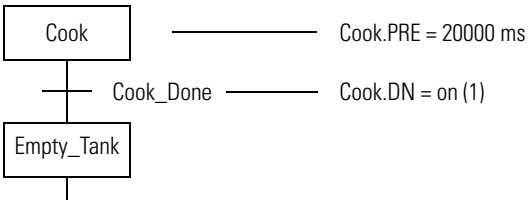
EXAMPLE

Functional specification says:

a. Cook the ingredients in the tank for 20 seconds.

b. Empty the tank.

Solution:



Here is an example of the use of the high alarm of a step.

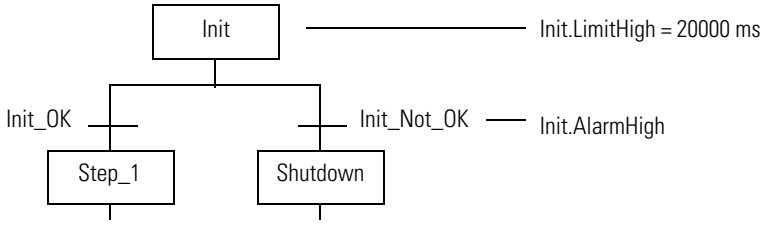
EXAMPLE

Functional specification says:

a. Home 8 devices.

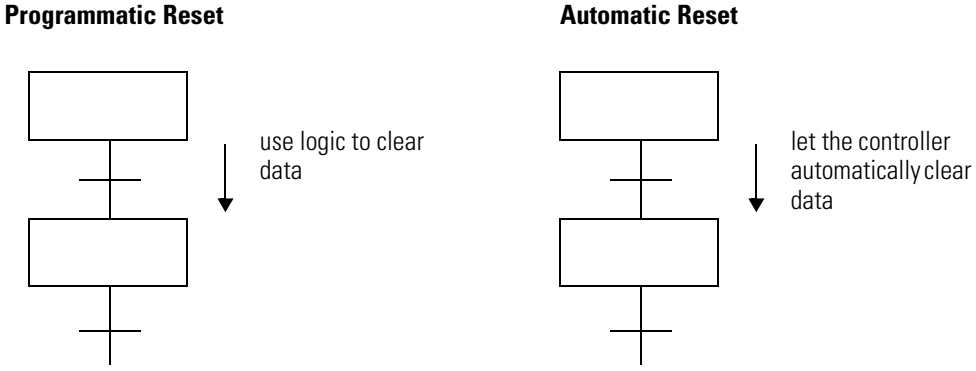
b. If all 8 devices are not home within 20 seconds, then shutdown the system.

Solution:



Turn Off a Device at the End of a Step

When the SFC leaves a step, you have several options on how to turn off devices that the step turned on.



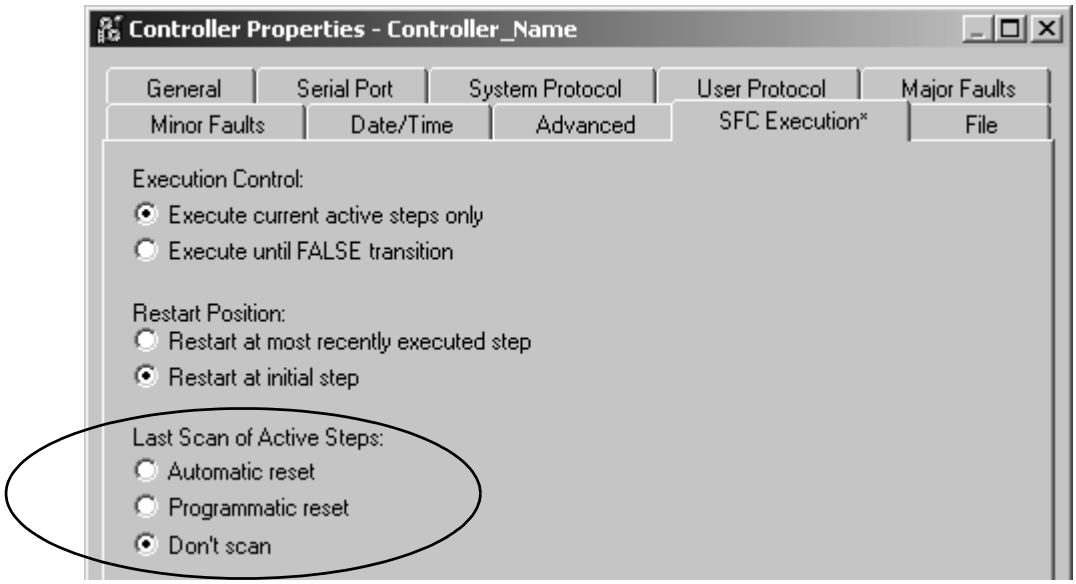
Each option requires you to make the following choices:

1. Choose a last scan option.
2. Based on the last scan option, develop your logic so that the last scan returns data to the desired values.

Choose a Last Scan Option

On the last scan of each step, you have the following options. The option that you choose applies to all steps in all SFCs of this controller.

If you want to:	And on the last scan of a step:	Then:	See:
control which data to clear	Execute *only* P and P0 actions and use them to clear the required data.	Use the Don't Scan Option	page 5-34
	Execute *all* actions and use either of these options to clear the required data: • status bits of the step or action to condition logic • P and P0 actions	Use the Programmatic Reset Option	page 5-35
let the controller clear data	→	Use the Automatic Reset Option	page 5-38


Controller Properties - Controller_Name
General
Serial Port
System Protocol
User Protocol
Major Faults
Minor Faults
Date/Time
Advanced
SFC Execution*
File
Execution Control:
Execute current active steps only
Execute until FALSE transition
Restart Position:
Restart at most recently executed step
Restart at initial step
Last Scan of Active Steps:
Automatic reset
Programmatic reset
Don't scan

The following table compares the different options for handling the last scan of a step:

Characteristic:	During the last scan of a step, this option does the following:		
	Don't scan	**Programmatic reset**	**Automatic reset**
execution actions	Only P and P0 actions execute. They execute according to their logic.	All actions execute according to their logic.	• P and P0 actions execute according to their logic. • All other actions execute in postscan mode. • On the next scan of the routine, the P and P0 actions execute in postscan mode.
retention of data values	All data keeps its current values.	All data keeps its current values.	• Data reverts to its values for postscan. • Tags to the left of [:=] assignments clear to zero.
method for clearing data	Use P and P0 actions.	Use either: • status bits of the step or action to condition logic • P and P0 actions	Use either: • [:=] assignment (non-retentive assignment) • instructions that clear their data during postscan
reset of a nested SFC	A nested SFCs remains at its current step.	A nested SFCs remains at its current step.	For the *Restart Position* property, if you choose the *Restart at initial step* option, then: • A nested SFC resets to its initial step. • The X bit of a stop element in a nested SFC clears to zero.

Use the Don't Scan Option

The default option for handling the last scan of a step is *Don't scan*. With this option, all data keeps its current values when the SFC leaves a step. This requires you to use additional assignments or instructions to clear any data that you want to turn off at the end of a step.

To turn off a device at the end of a step:

1. Make sure that the *Last Scan of Active Steps* property is set to the *Don't scan* option (default).

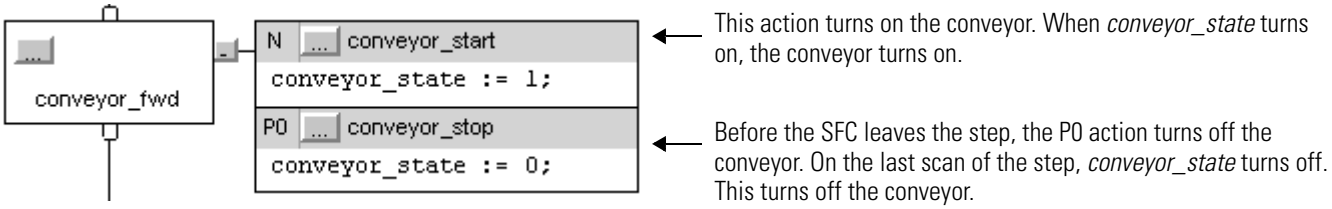
2. Use a *P0 Pulse (Falling Edge)* action to clear the required data. Make sure that the P0 action or actions are last in the order of actions for the step.

During the last scan of the step, the *Don't scan* option executes only P and P0 actions. The assignments and instructions of the actions execute according to their logic conditions.

- The controller *does not* execute a **postscan** of assignments or instructions.
- When the SFC leaves the step, all data keeps its current values.

The following example uses an action to turn on a conveyor at the start of a step. A different action turns off the conveyor at the end of the step.

EXAMPLE Use the Don't Scan Option



Use the Programmatic Reset Option

An optional method to programmatically turn off (clear) devices at the end of a step is to execute all actions on the last scan of the step. This lets you execute your normal logic as well as turn off (clear) devices at the end of a step.

1. In the *Last Scan of Active Steps* property, choose the *Programmatic reset* option:

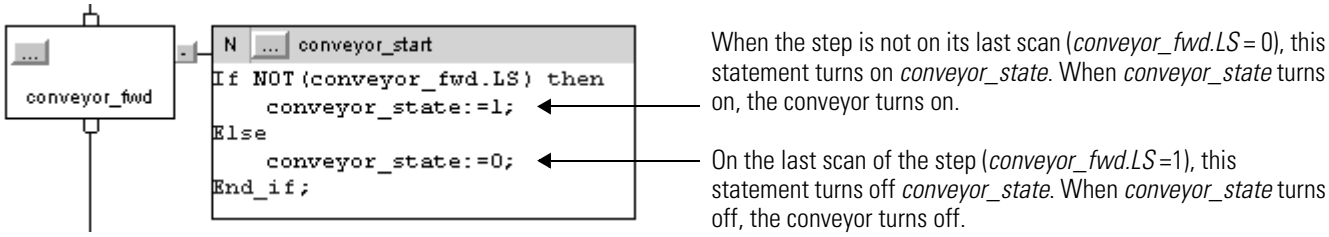
2. Clear the required data using any of the following methods:
   - To your normal logic, add logic that clears the required data. Use the LS bit of the step or the Q bit of the action to condition the execution of the logic.
   - Use a *P0 Pulse (Falling Edge)* action to clear the required data. Make sure that the P0 action or actions are last in the order of actions for the step.

During the last scan of the step, the *Programmatic reset* option executes all assignments and instructions according to logic conditions.

- The controller *does not* **postscan** the assignments or instructions.
- When the SFC leaves the step, all data keeps its current value.

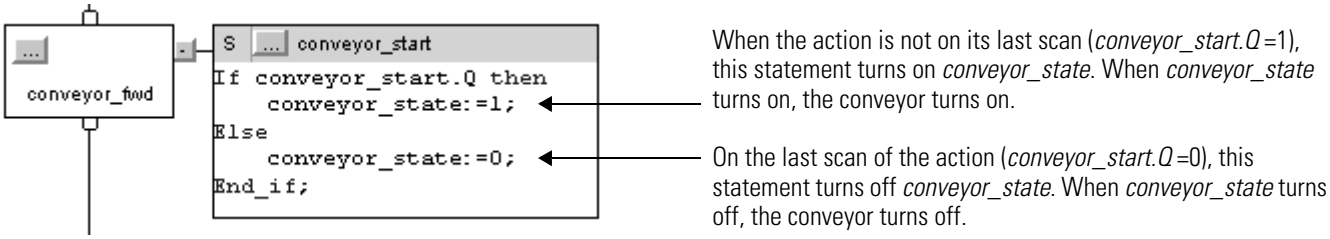
The following example uses a single action to turn on and off a conveyor. The LS bit of the step conditions the execution of the logic. See "SFC_STEP Structure" on page 5-8.

EXAMPLE Use the Programmatic Reset Option and the LS Bit



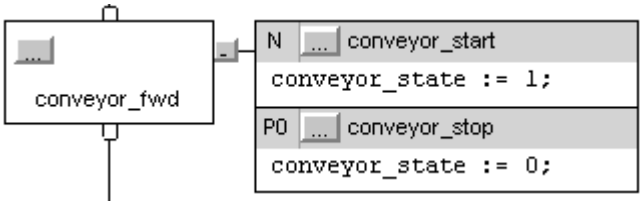
For an action that uses one of the stored qualifiers, use the Q bit of the action to condition your logic. See "SFC_ACTION Structure" on page 5-20.

EXAMPLE Use the Programmatic Reset Option and the Q Bit



You can also use a *P0 Pulse (Falling Edge)* action to clear data. The following example uses an action to turn on a conveyor at the start of a step. A different action turns off the conveyor at the end of the step.

EXAMPLE Use the Programmatic Reset Option and a P0 Action



This action turns on the conveyor. When *conveyor_state* turns on, the conveyor turns on.

Before the SFC leaves the step, the P0 action turns off the conveyor. On the last scan of the step, *conveyor_state* turns off. This turns off the conveyor.

Use the Automatic Reset Option

To automatically turn off (clear) devices at the end of a step:

1. In the *Last Scan of Active Steps* property, choose the *Automatic reset* option:

2. To turn off a device at the end of the step, control the state of the device with an assignment or instruction such as:
   - [:=] assignment (non-retentive assignment)
   - Output Energize (OTE) instruction in a subroutine

During the last scan of each step, the *Automatic reset* option does the following:

- execute P and P0 actions according to their logic conditions
- clear tags to the left of [:=] assignments
- execute a **postscan** of embedded structured text
- execute a postscan of any subroutine that an action calls via a Jump to Subroutine (JSR) instruction
- reset any nested SFC (SFC that an action calls as a subroutine)

IMPORTANT The postscan of an action actually occurs when the action goes from active to inactive. Depending on the qualifier of the action, the postscan could occur before or after the last scan of the step.

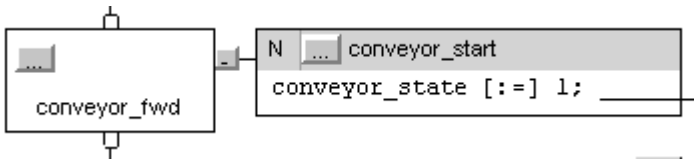
As a general rule, the postscan executes instructions as if all conditions are false. For example, the Output Energize (OTE) instruction clears its data during postscan.

Some instructions *do not* follow the general rule during postscan. For a description of how a specific instruction executes during postscan, see the following manuals:

- *Logix5000 Controllers General Instructions Reference Manual*, publication 1756-RM003
- *Logix5000 Controllers Process and Drives Instructions Reference Manual*, publication 1756-RM006
- *Logix5000 Controllers Motion Instruction Set Reference Manual*, publication 1756-RM007

Here is an example that uses a non-retentive assignment to control a conveyor. It turns on a conveyor at the start of a step and automatically turns off the conveyor when the step is done.

EXAMPLE Automatically Clear Data

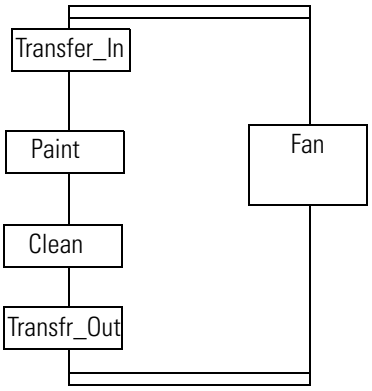
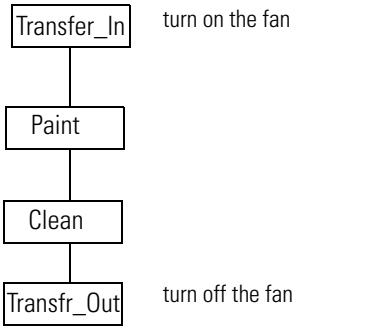
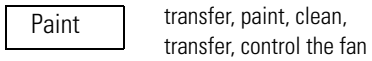


This action turns on the conveyor. When *conveyor_state* turns on, the conveyor turns on.

Keep Something On From Step-to-Step

How Do You Want to Control the Device?

To provide bumpless control of a device during more than one time or phase (step), do one of the following:

Option:	Example:
Use a Simultaneous Branch Make a separate step that controls the device.	
Store and Reset an Action Note the step that turns on the device and the step that turns off the device. Later, define a Stored and Reset Action pair to control the device.	
Use One Large Step Use one large step that contains all the actions that occur while the device is on.	

Use a Simultaneous Branch

A simple way to control a device or devices during one or more steps is to create a separate step for the devices. Then use a simultaneous branch to execute the step during the rest of the process.

Here is an example:

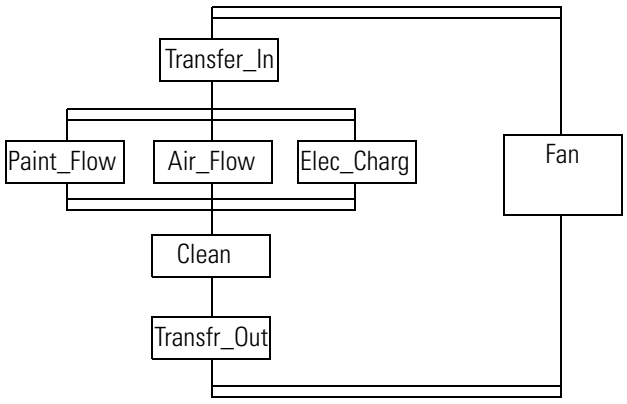
EXAMPLE

A paint operation does the following:

1. Transfer the product into the paint shop.
2. Paint the product using 3 separate paint guns.
3. Clean the guns.
4. Transfer the product to the paint ovens.

During the entire process, the system must control the shop fans.

Solution:



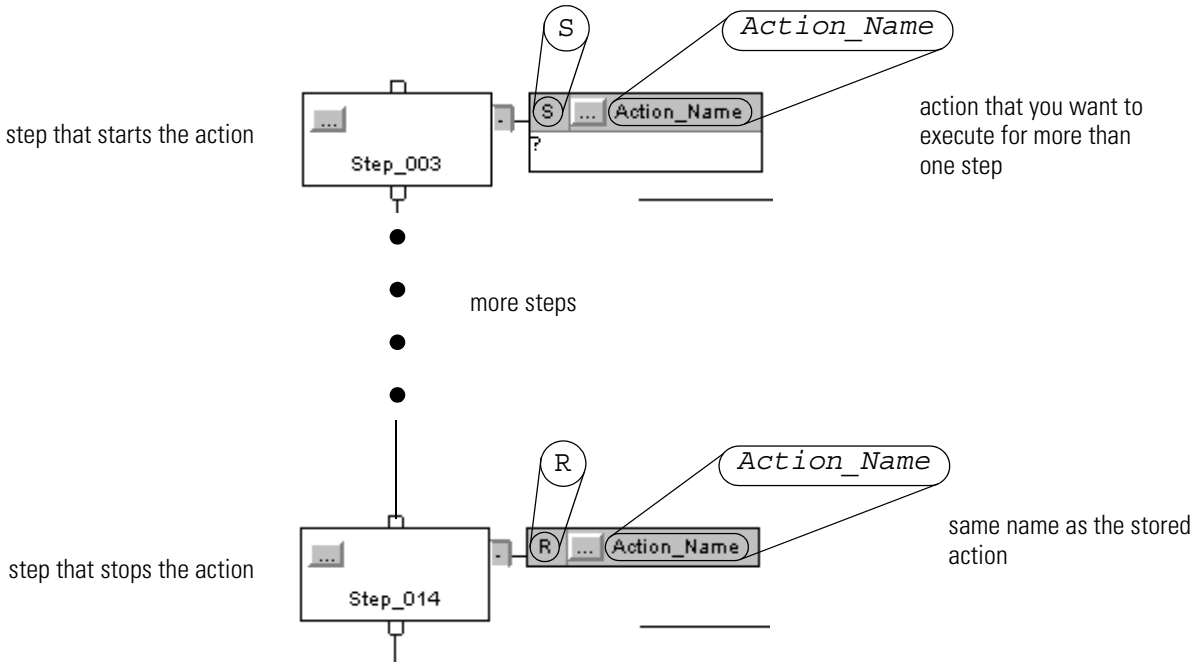
Store and Reset an Action

Typically, an action turns off (stops executing) when the SFC goes to the next step. To keep a device on from step to step without a bump, store the action that controls the device:

1. In the step that turns on the device, assign a stored qualifier to the action that controls the device. For a list of stored qualifiers, see Table 5.1 on page 5-23.

2. In the step that turns off the device, use a *Reset* action.

The following figure shows the use of a stored action.

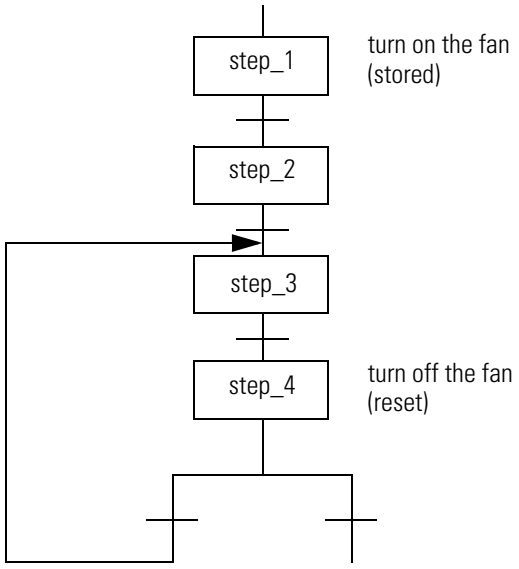


When the SFC leaves the step that stores the action, RSLogix 5000 software continues to show the stored action as active. (By default, a green border displays around the action.) This lets you know that the SFC is executing the logic of that action.

To use a stored action, follow these guidelines:

- The *Reset* action only turns off the stored action. It *does not* automatically turn off the devices of the action. To turn off the device, follow the *Reset* action with another action that turns off the device. Or use the *Automatic reset* option described on page 5-38.
- Before the SFC reaches a stop element, reset any stored actions that you *do not* want to execute at the stop. An active stored action remains active even if the SFC reaches a stop.
- Use caution when you jump in between a step that stores an action and a step that resets the action. Once you reset an action, it only starts when you execute the step that stores the action.

  In the following example, steps 1 - 4 require a fan to be on. At the end of *step_4*, the fan is reset (turned off). When the SFC jumps back to *step_3*, the fan remains off.



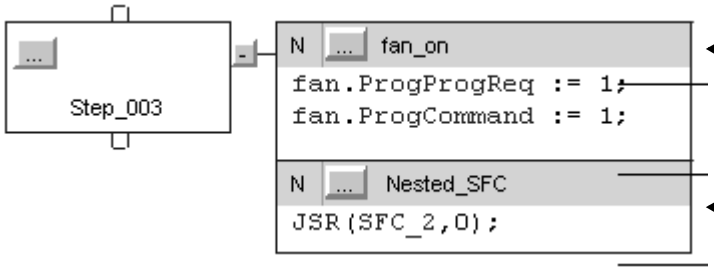
To turn the fan back on, the SFC has to jump back to *step_1*.

Use One Large Step

If you use one large step for multiple functions, then use additional logic to sequence the functions. One option is to nest an SFC within the large step.

In the following example, a step turns on a fan and then calls another SFC. The nested SFC sequences the remaining functions of the step. The fan stays on throughout the steps of the nested SFC.

EXAMPLE Use a Large Step



This action turns on a fan:

- *fan.ProgProgReq* lets the SFC command the state of the fan.
- *fan.ProgCommand* turns on the fan.

This action calls another SFC. The SFC sequences the remaining functions of the step.

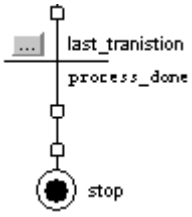
For additional information on how to nest an SFC, see "Nest an SFC" on page 5-49.

End the SFC

Once an SFC completes its last step, it *does not* automatically restart at the first step. You must tell the SFC what to do when it finishes the last step.

At the End of the SFC, What Do You Want to Do?

To:	Do this:
automatically loop back to an earlier step	Wire the last transition to the top of the step to which you want to go. See "Wire to a Previous Step"on page 5-17.
stop and wait for a command to restart	Use a Stop Element. See "Use a Stop Element" on page 5-45.



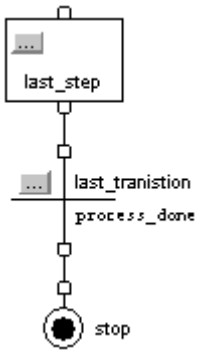
Use a Stop Element

The stop element lets you stop the execution of an entire SFC or a path of a simultaneous branch and wait to restart. When an SFC reaches a stop element, the following occurs:

- The X bit of the stop element turns on. This signals that the SFC is at the stop element.
- Stored actions remain active.
- Execution stops for part or all of the SFC:

If the stop element is at the end of a:	Then:
sequence	entire SFC stops
selection branch	
path within a simultaneous branch	only that path stops while the rest of the SFC continues to execute.

EXAMPLE Use a Stop Element



When the SFC reaches *last_step* and *process_done* is true, the execution of the SFC stops.

Restart (Reset) the SFC

Once at the stop element, you have several options to restart the SFC:

If the SFC is:	And the *Last Scan of Active Steps* option is:	Then:
nested (i.e., another SFC calls this SFC as a subroutine)	Automatic reset	At the end of the step that calls the nested SFC, the nested SFC automatically resets: • The nested SFC resets to the initial step. • The X bit of the stop element in the nested SFC clears to zero.
	Programmatic reset	1. Use an SFC Reset (SFR) instruction to restart the SFC at the required step. 2. Use logic to clear the X bit of the stop element.
	Don't scan	
NOT nested (i.e., *no* SFC calls this SFC as a subroutine)	→	1. Use an SFC Reset (SFR) instruction to restart the SFC at the required step. 2. Use logic to clear the X bit of the stop element.

The following example shows the use of the SFC Reset (SFR) instruction to restart the SFC and clear the X bit of the stop element.

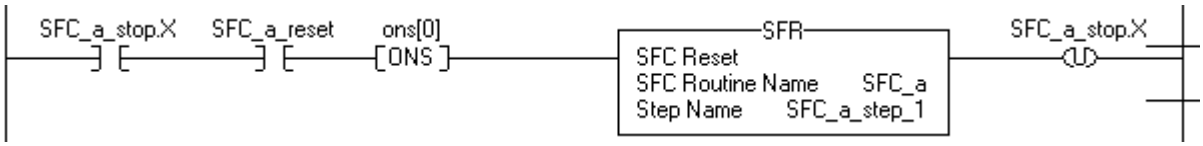
EXAMPLE

Restart (Reset) the SFC

If SFC_a_stop.X = on (SFC_a is at the stop) and SFC_a_reset = on (time to reset the SFC) then for one scan (ons[0] = on):

Reset SFC_a to SFC_a_Step_1

SFC_a_stop.X = 0



SFC_STOP Structure

Each stop uses a tag to provide the following information about the stop element:

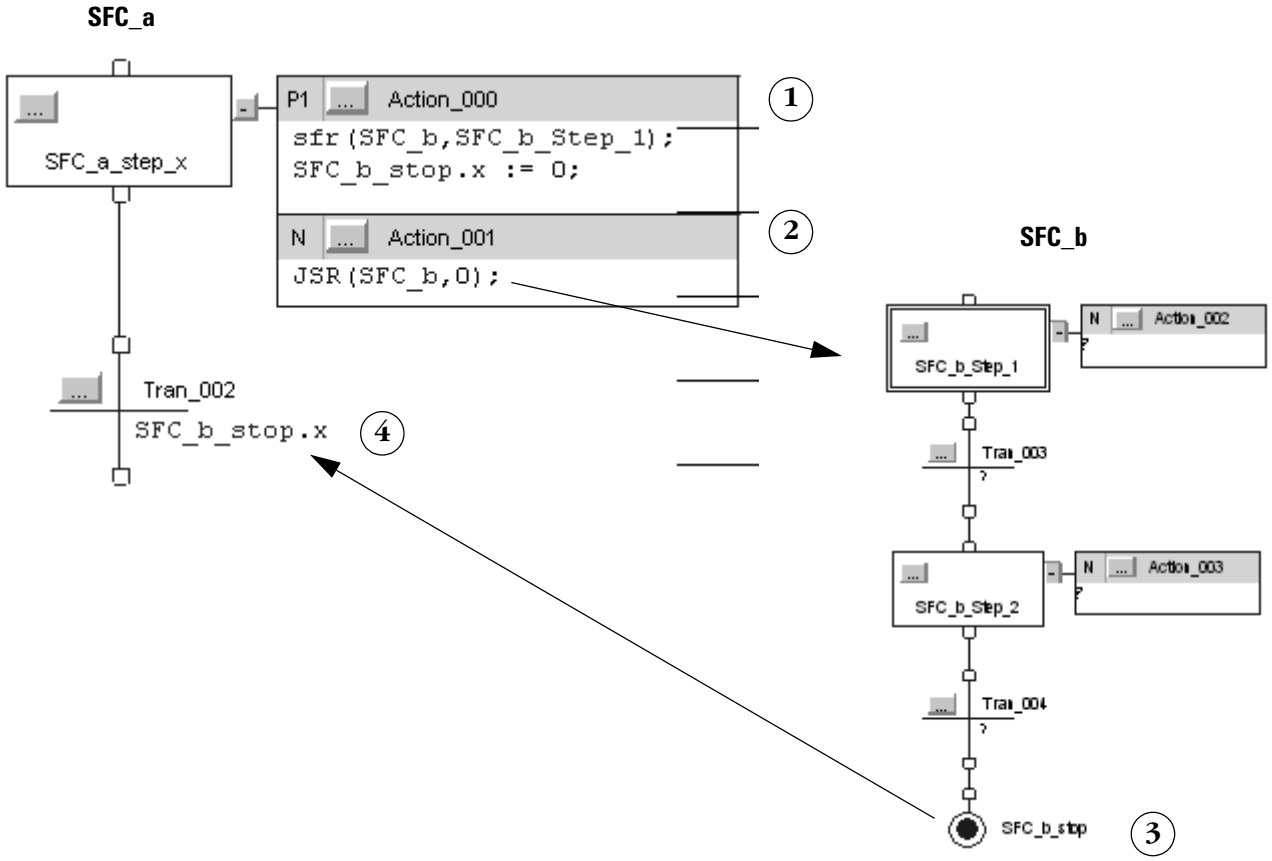
If you want to:	Then check or set this member:	Data type:	Details:
determine when the SFC is at the stop	X	BOOL	• When the SFC reaches the stop, the X bit turns on. • The X bit clears if you configure the SFCs to restart at the initial step and the controller changes from program to run mode. • In a nested SFC, the X bit also clears if you configure the SFCs for automatic reset and the SFC leaves the step that calls the nested SFC.
determine the target of an SFC Reset (SFR) instruction	Reset	BOOL	An SFC Reset (SFR) instruction resets the SFC to a step or stop that the instruction specifies. • The Reset bit indicates to which step or stop the SFC will go to begin executing again. • Once the SFC executes, the Reset bit clears.
determine how many times a stop has become active	Count	DINT	This is *not* a count of scans of the stop. • The count increments each time the stop becomes active. • It increments again only after the stop goes inactive and then active again. • The count resets only if you configure the SFC to restart at the initial step. With that configuration, it resets when the controller changes from program mode to run mode.

If you want to:	Then check or set this member:	Data type:	Details:	
use one tag for the various status bits of this stop	Status	DINT	**For this member:**	**Use this bit:**
			Reset	22
			X	31

Nest an SFC

One method for organizing your project is to create one SFC that provides a high-level view of your process. Each step of that SFC calls another SFC that performs the detailed procedures of the step (nested SFC).

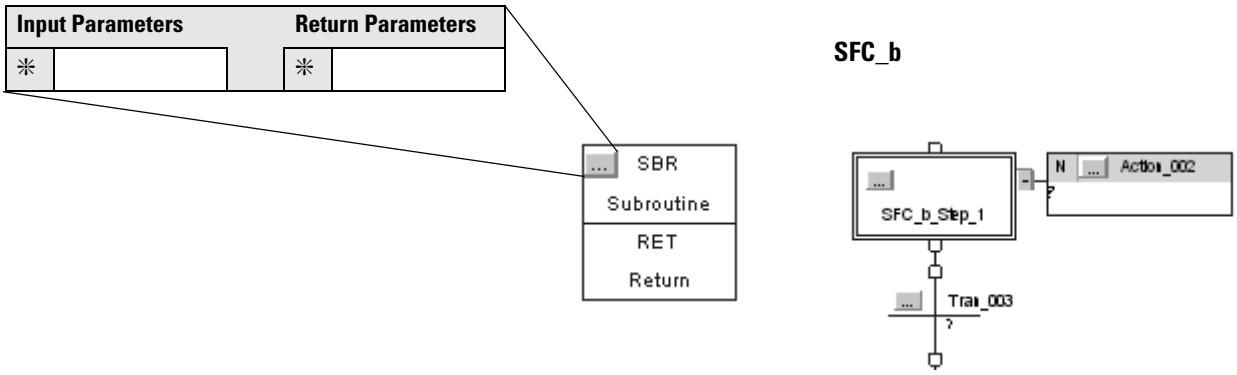
The following figure shows one way to nest an SFC. In this method, the last scan option of the SFC is configured for either *Programmatic reset* or *Don't scan*. If you configure the SFC for *Automatic reset*, then step 1 is unnecessary.



1. Reset the nested SFC:
   - The SFR instruction restarts the *SFC_b* at *SFC_b_Step_1*. Each time the *SFC_a* leaves this step and then returns, you have to reset the *SFC_b*.
   - The action also clears the X bit of the stop element.
2. Call the *SFC_b*.
3. Stop the *SFC_b*. This sets the X bit of the stop element.
4. Use the X bit of the stop element to signal that the *SFC_b* is done and it is time to go to the next step.

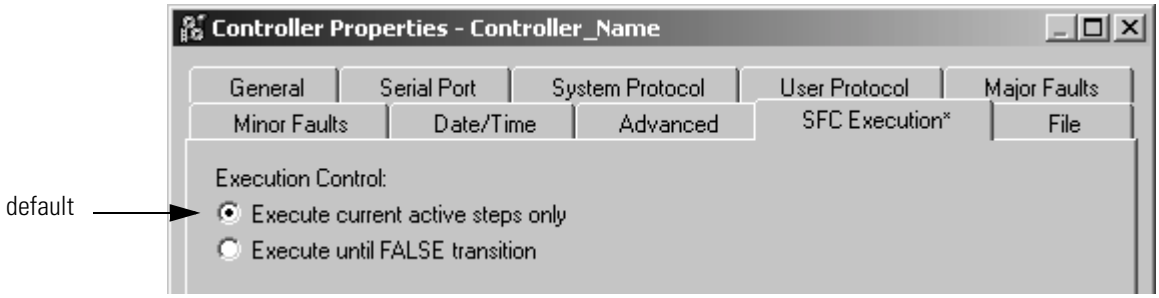
Pass Parameters

To pass parameters to or from an SFC, place a Subroutine/Return element in the SFC.



Configure When to Return to the OS/JSR

By default, an SFC executes a step or group of simultaneous steps and then returns to the operating system (OS) or the calling routine (JSR).



You have the option of letting the SFC execute until it reaches a false transition. If several transitions are true at the same time, this option reduces the time to get to the desired step.

Use the *Execute until FALSE transition* option only when:

1. You don't have to update JSR parameters before each step. Parameters update only when the SFC returns to the JSR. See "Pass Parameters" on page 5-50.

2. A false transition occurs within the watchdog timer for the task. If the time that it takes to return to a JSR and complete the rest of the task is greater than the watchdog timer, a major fault occurs.

For a detailed diagram of the execution of each option, see Figure 5.9 on page 5-55.

Pause or Reset an SFC

Two optional instructions are available that give you further control over the execution of your SFC:

If you want to:	Then use this instruction:
pause an SFC	Pause SFC (SFP)
reset an SFC to a specific step or stop	Reset SFC (SFR)

Both instructions are available in the ladder logic and structured text programming languages.

For more information, use either of the following resources:

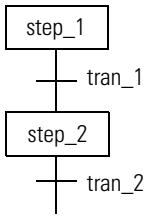
- In RSLogix 5000 software, from the *Help* menu, choose *Instruction Help*. Look in the *Program Control Instructions* category.
- See *Logix5000 Controllers General Instructions Reference Manual*, publication 1756-RM003.

Execution Diagrams

The following diagrams show the execution of an SFC with different organizations of steps or different selections of execution options. Use the diagrams if you require a more detailed understanding of how your SFC executes.

Figure 5.5 Execution of a Sequence

This…



…executes like this

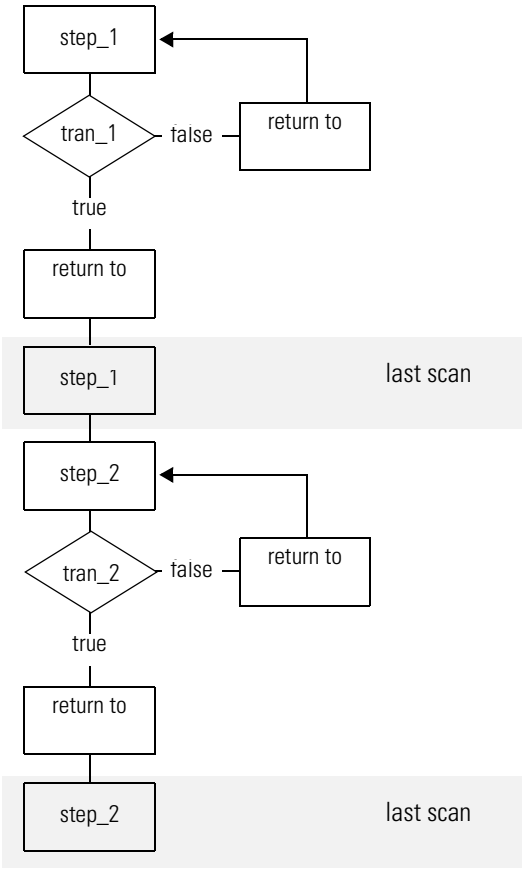
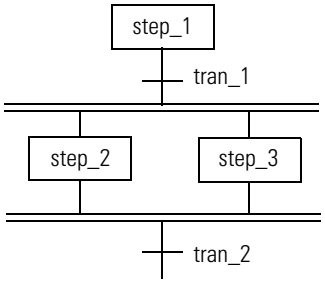


Figure 5.6 Execution of a Simultaneous Branch

This…



…executes like this

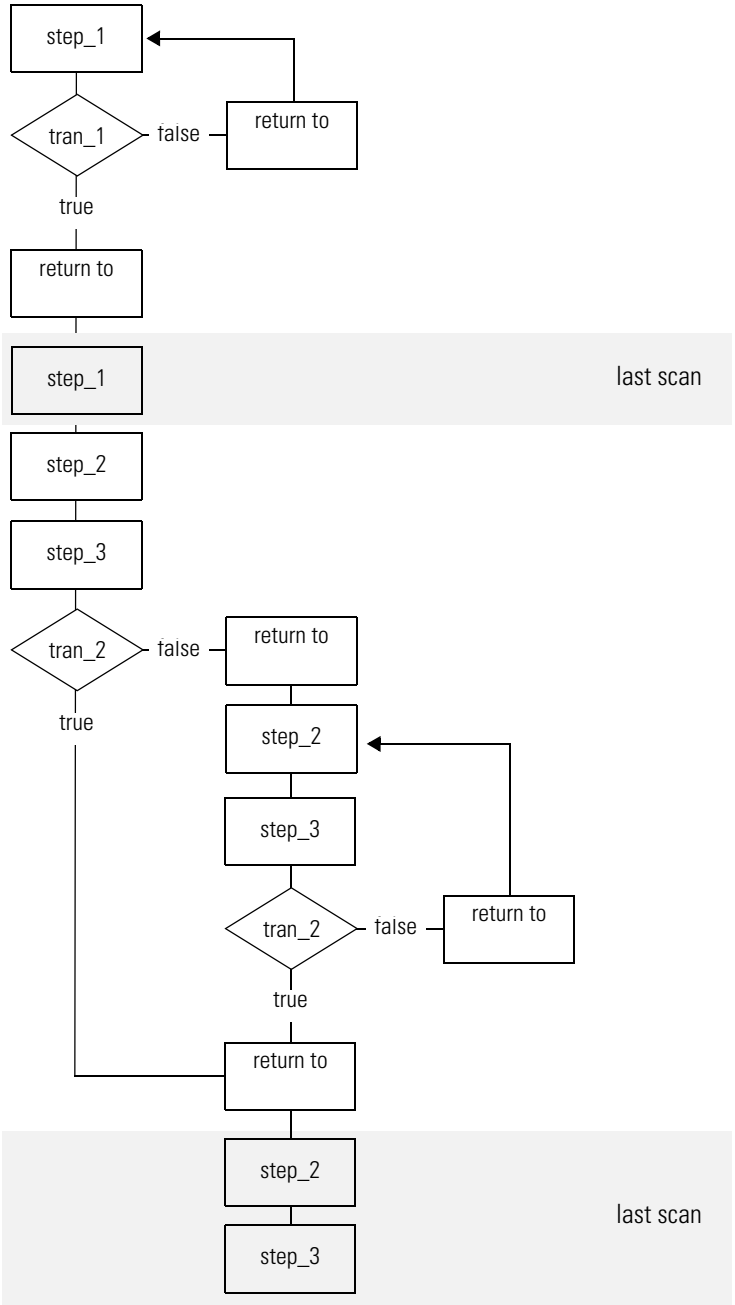
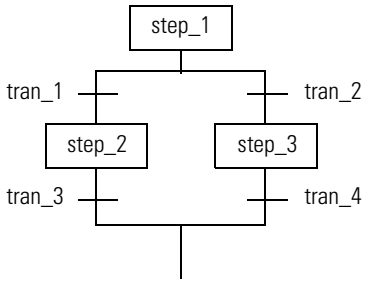


Figure 5.7 Execution of a Selection Branch

This…



…executes like this

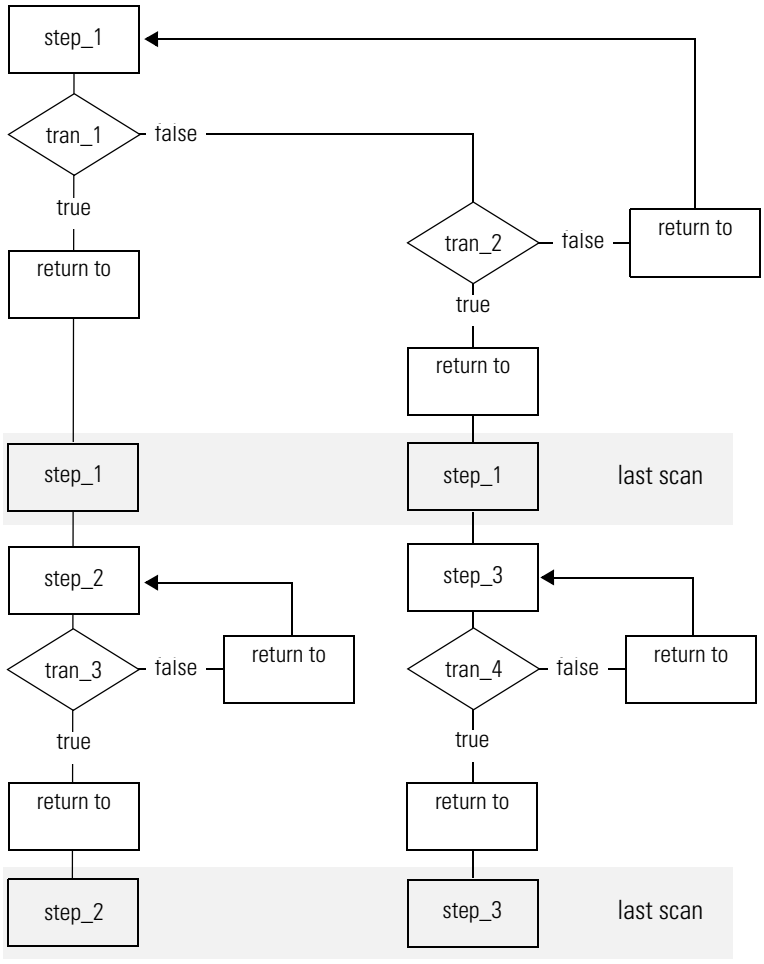


Figure 5.8 When parameters enter and exit an SFC

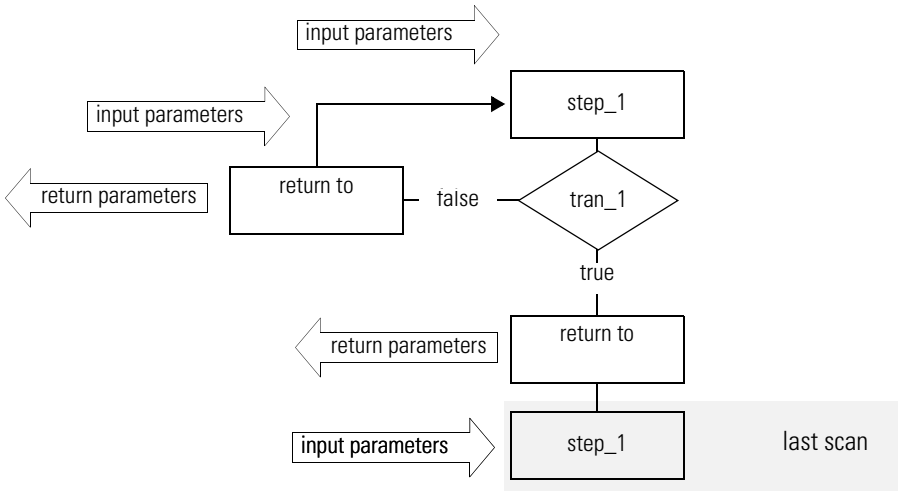
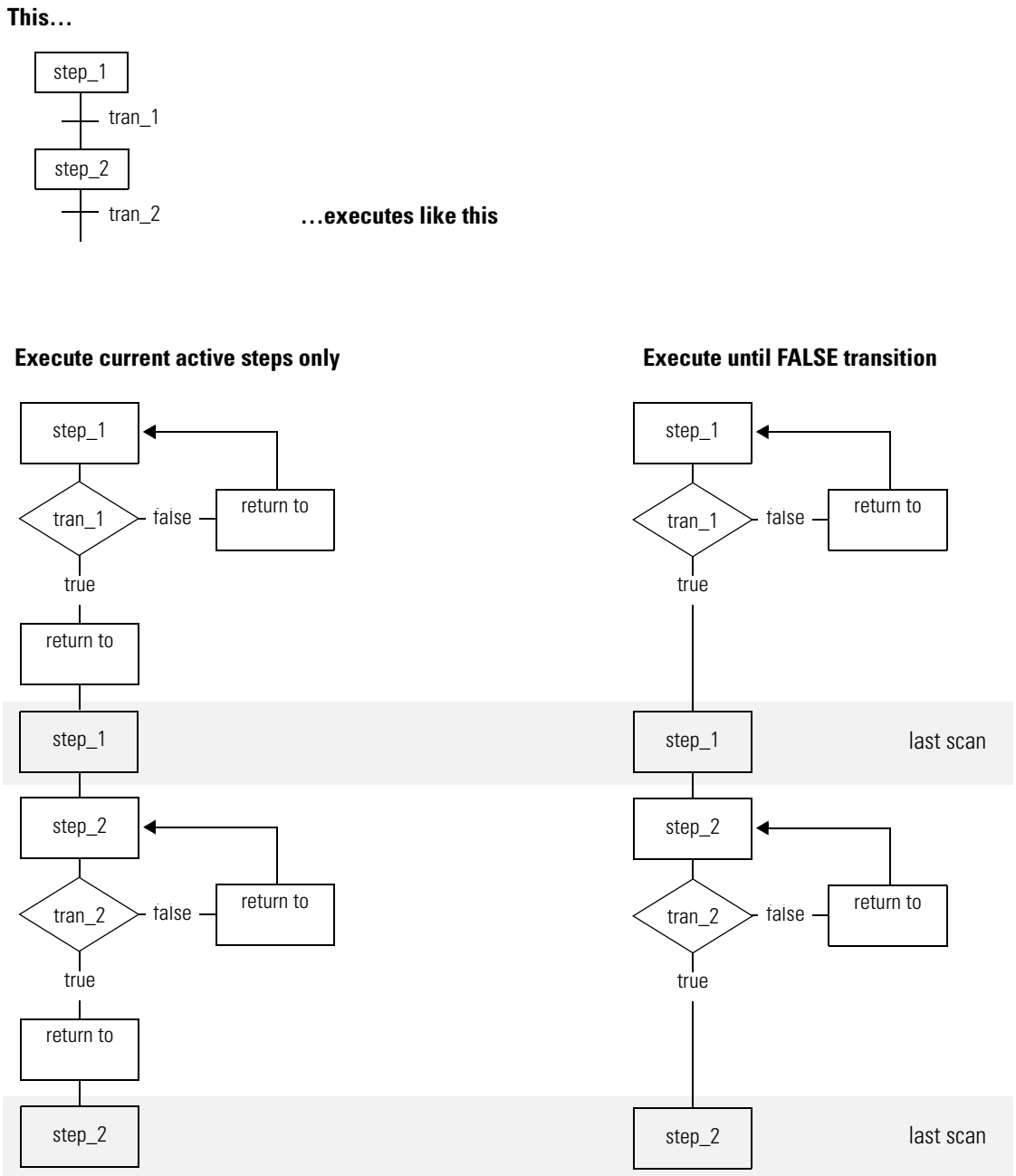


Figure 5.9 Options for Execution Control



Notes:

Program a Sequential Function Chart

When to Use This Procedure

Use this procedure to enter a **sequential function chart** (SFC) into RSLogix 5000 software. Enter the SFC as you design it. Or first design the SFC and then enter it. To design the SFC, see "Design a Sequential Function Chart" on page 5-1.

Before You Use This Procedure

Before you use this procedure, make sure you are able to perform the following tasks:

- ☑ Navigate the Controller Organizer
- ☑ Identify the Programming Languages That Are Installed

For more information on any of those tasks, see "Getting Started" on page 1-1.

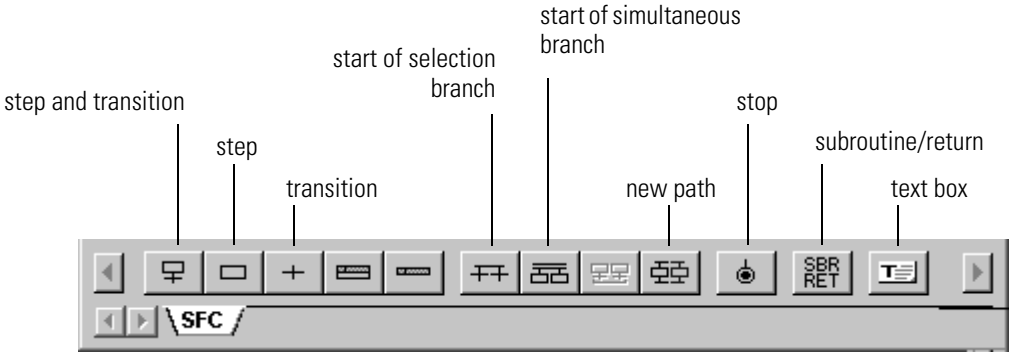
How to Use This Procedure

To program an SFC:

- ☐ Add an SFC Element
- ☐ Create a Simultaneous Branch
- ☐ Create a Selection Branch
- ☐ Set the Priorities of a Selection Branch
- ☐ Return to a Previous Step
- ☐ Rename a Step
- ☐ Configure a Step
- ☐ Rename a Transition
- ☐ Program a Transition
- ☐ Add an Action
- ☐ Rename an Action
- ☐ Configure an Action
- ☐ Program an Action
- ☐ Assign the Execution Order of Actions
- ☐ Document the SFC
- ☐ Show or Hide Text Boxes or Tag Descriptions
- ☐ Configure the Execution of the SFC
- ☐ Verify the Routine

Add an SFC Element

To add SFC elements, use the SFC toolbar.



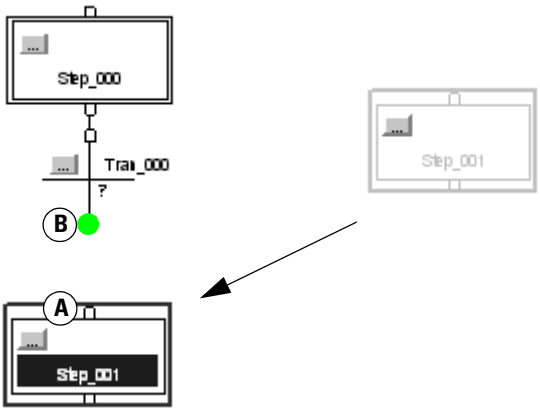
To add an element to your SFC, you have these options:

- ☐ Add and Manually Connect Elements
- ☐ Add and Automatically Connect Elements
- ☐ Drag and Drop Elements

Add and Manually Connect Elements

1. On the SFC toolbar, click the button for the item that you want to add.
2. Drag the element to the required location on the SFC.

For example:



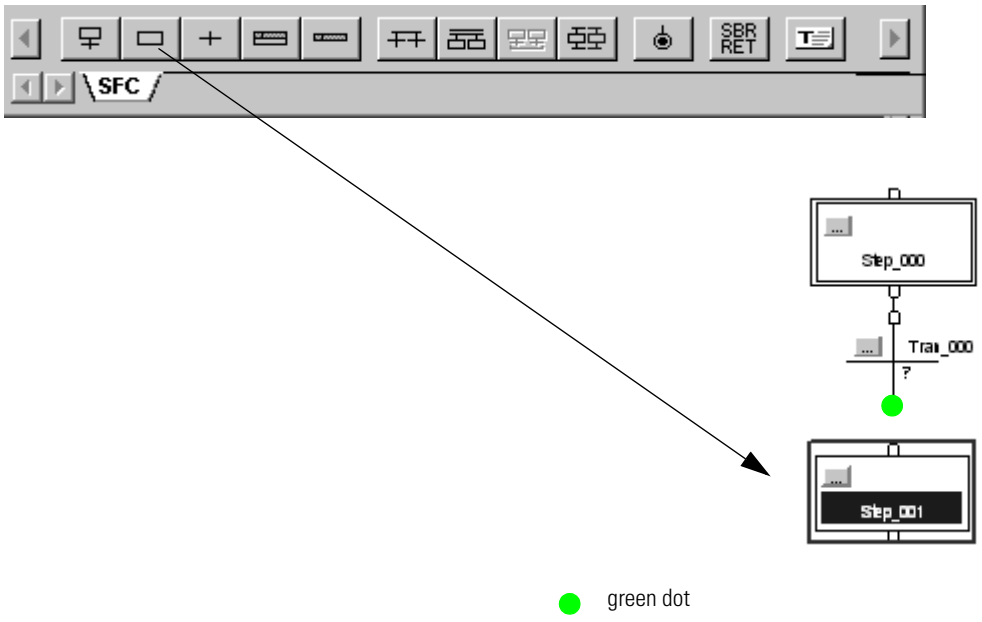
3. To wire (connect) two elements together, click a pin on one of the elements (A) and then click the pin on the other element (B). A green dot shows a valid connection point.

Add and Automatically Connect Elements

1. Select (click) the element to which you want to connect a new element.
2. With the element still selected, click the toolbar button for the next element.

Drag and Drop Elements

From the SFC toolbar, drag the button for the required element to the desired connection point on the SFC. A green dot shows a valid connection point.

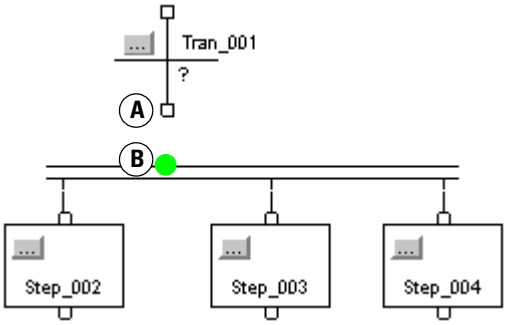


Create a Simultaneous Branch

Start a Simultaneous Branch

1. On the SFC toolbar, click the  button. Then drag the new branch to the desired location.

2. To add a path to the branch, select (click) the first step of the path that is to the left of where you want to add the new path. Then click the  button.



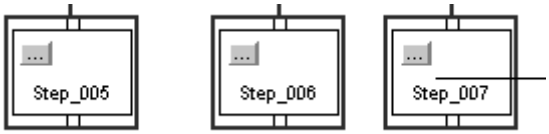
green dot

3. To wire the simultaneous branch to the preceding transition, click the bottom pin of the transition (A) and then click the horizontal line of the branch (B). A green dot shows a valid connection point.

End a Simultaneous Branch

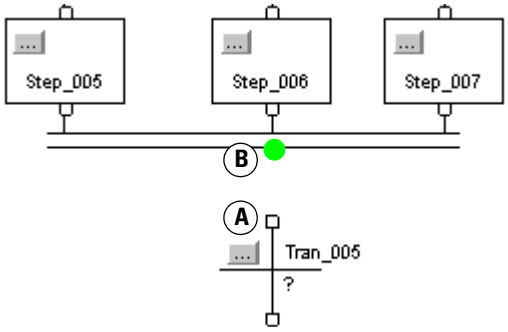
1. Select the last step of each path in the branch. To select the steps, you can either:

   - Click and drag the pointer around the steps that you want to select.
   - Click the first step. Then press and hold [Shift] and click the rest of the steps that you want to select.



2. On the SFC toolbar, click the  button.

3. Add the transition that follows the simultaneous branch.



green dot

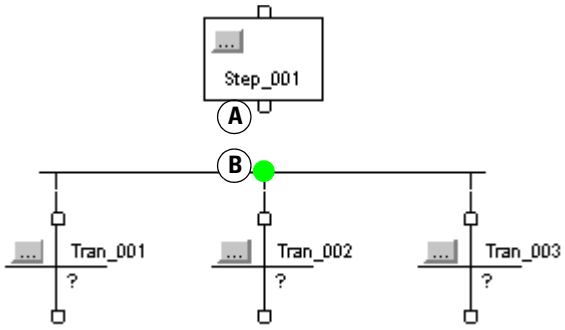
4. To wire the simultaneous branch to the transition, click the top pin of the transition (A) and then click the horizontal line of the branch (B). A green dot shows a valid connection point.

Create a Selection Branch

Start a Selection Branch

1. On the SFC toolbar, click the button. Then drag the new branch to the desired location.

2. To add a path to the branch, select (click) the first transition of the path that is to the left of where you want to add the new path. Then click the button.



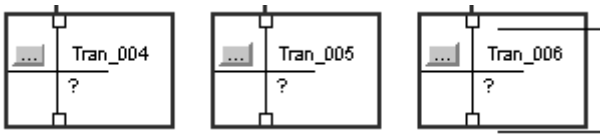
green dot

3. To wire the selection branch to the preceding step, click the bottom pin of the step (A) and then click the horizontal line of the branch (B). A green dot shows a valid connection point.

End a Selection Branch

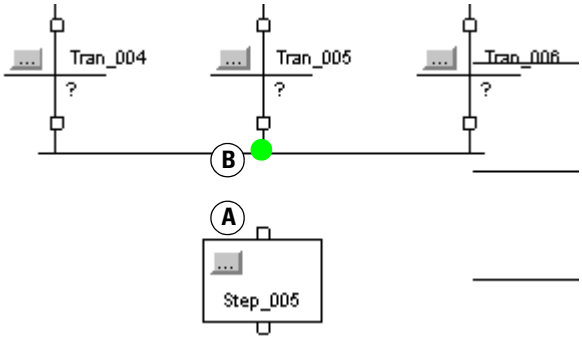
1. Select the last transition of each path in the branch. To select the transitions, you can either:

   - Click and drag the pointer around the transitions that you want to select.
   - Click the first transition. Then press and hold [Shift] and click the rest of the transitions that you want to select.



2. On the SFC toolbar, click the  button.

3. Add the step that follows the selection branch.



green dot

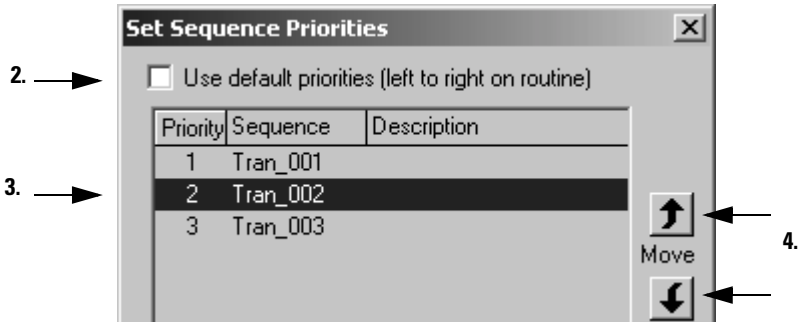
4. To wire the selection branch to the step, click the top pin of the step (A) and then click the horizontal line of the branch (B). A green dot shows a valid connection point.

Set the Priorities of a Selection Branch

By default, the SFC checks the transitions that start a selection branch from left to right. If you want to check a different transition first, assign a priority to each path of the selection branch. For example, it is a good practice to check for error conditions first. Then check for normal conditions.

To assign priorities to a selection branch:

1. Right click the horizontal line that starts the branch and choose *Set Sequence Priorities*.



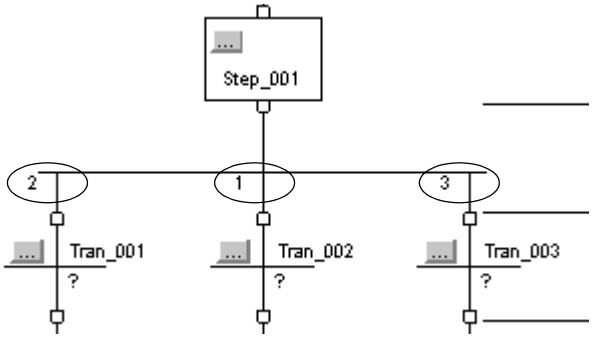
2. Clear (uncheck) the *Use default priorities* check box.

3. Select a transition.

4. Use the *Move* buttons to raise or lower the priority of the transition.

5. When all the transitions have the desired priority, choose OK

When you clear (uncheck) the *Use default priorities* check box, numbers show the priority of each transition.



Return to a Previous Step

To jump to a different step in your SFC:

- Connect a Wire to the Step
- Hide a Wire
- Show a Hidden Wire

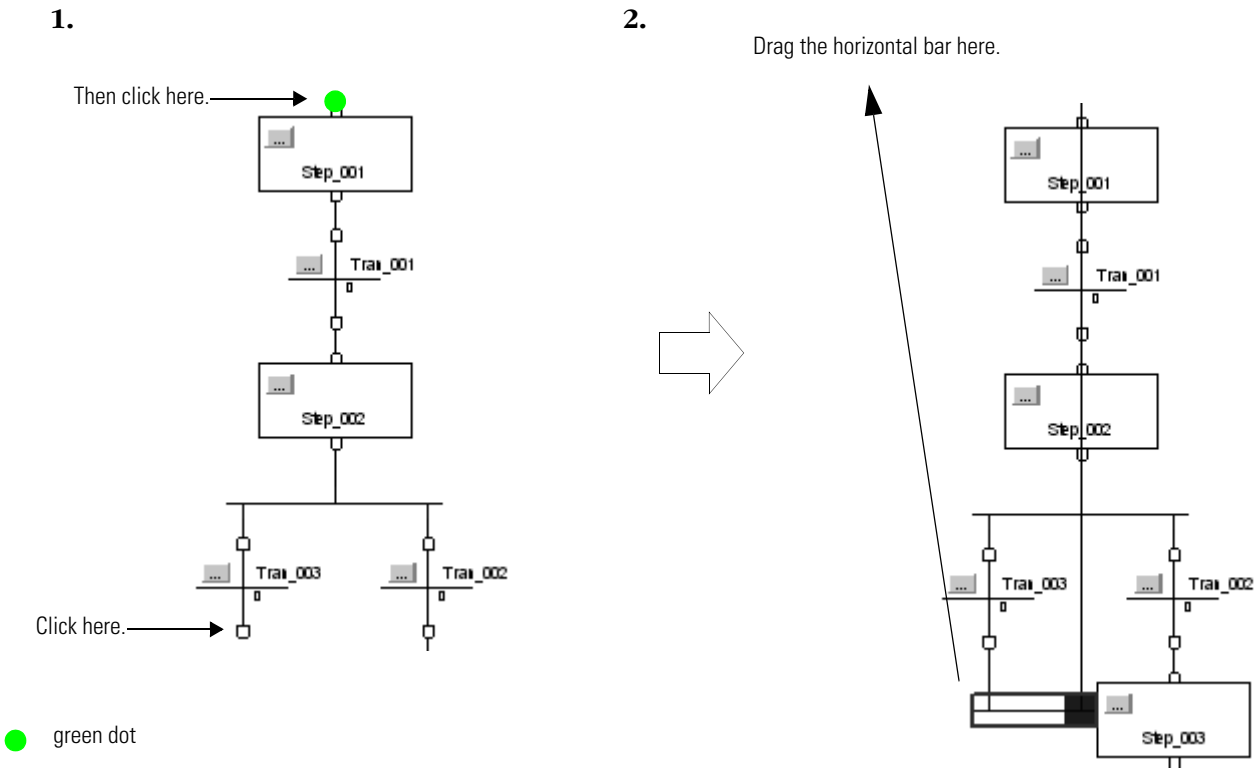
Connect a Wire to the Step

1. Click the lower pin of the transition that signals the jump. Then click the top pin of the step to which you want to go. A green dot shows a valid connection point.

   Typically, the resulting connection orients itself along the center of the flowchart and is hard to see.

2. To make the jump easier to read, drag its horizontal bar above the step to which the jump goes. You may also have to reposition some of the SFC elements.

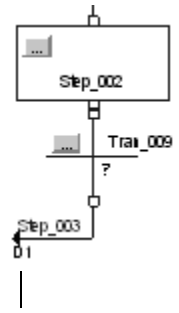
For example, to go to *Step_001* from *Tran_003*:



Hide a Wire

If a wire gets in the way of other parts of your SFC, hide the wire to make the SFC easier to read.

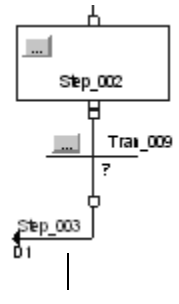
To hide a wire, right-click the wire and choose *Hide Wire*.



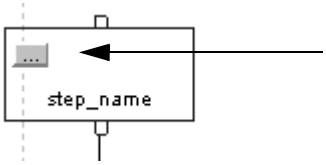
To see the SFC element to which the wire goes, click the grid location on the wire.

Show a Hidden Wire

To show a wire that is hidden, right-click a visible part of the wire and choose *Show Wire*.

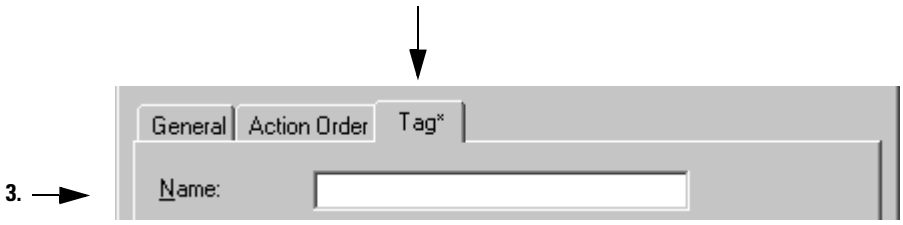


Rename a Step



Each step uses a tag to store configuration and status information about the step. To rename the tag of the step:

1. Click the [...] button of the step.
2. Click the *Tag* tab.



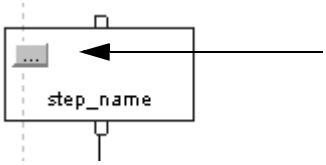
3. Type the new name for the step (tag).
4. Choose OK

Configure a Step

To configure a step, you have these options:

- Assign the Preset Time for a Step
- Configure Alarms for a Step
- Use an Expression to Calculate a Time

Assign the Preset Time for a Step



1. Click the [...] button of the step.

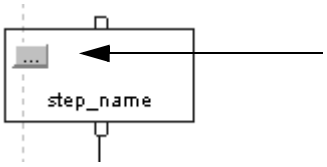
2. Type the time for the step, in milliseconds.
3. Choose OK

When the step is active for the preset time (Timer = Preset), the DN bit of the step turns on.

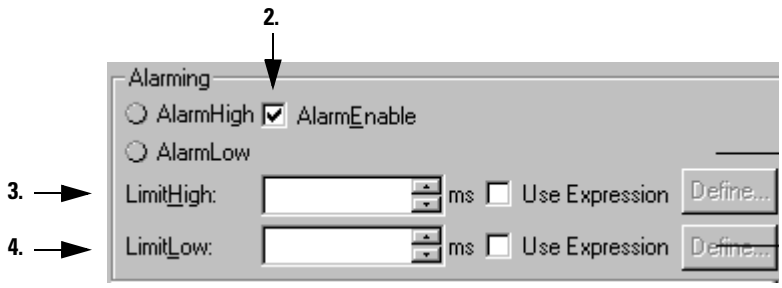
To calculate the preset time for a step at runtime, see "Use an Expression to Calculate a Time" on page 6-12.

Configure Alarms for a Step

To turn on an alarm if a step executes too long or not long enough:



1. Click the ... button of the step.
2. Check the *AlarmEnable* check box.



3. Type the time for the high alarm, in milliseconds.
4. Type the time for the low alarm, in milliseconds.
5. Choose OK

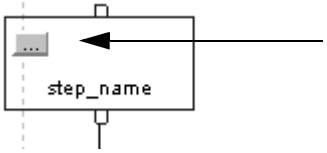
To calculate the time for an alarm at runtime, see "Use an Expression to Calculate a Time" on page 6-12.

Use an Expression to Calculate a Time

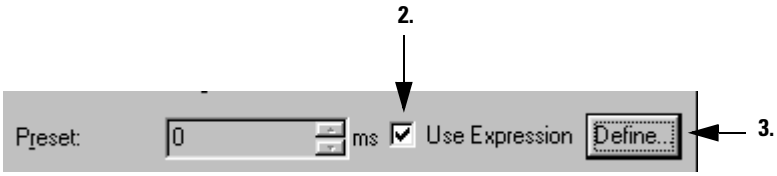
To calculate a time based on tags in your project, enter the time as a **numeric expression**. You can use an expression to calculate the following times:

- Preset
- LimitHigh
- LimitLow

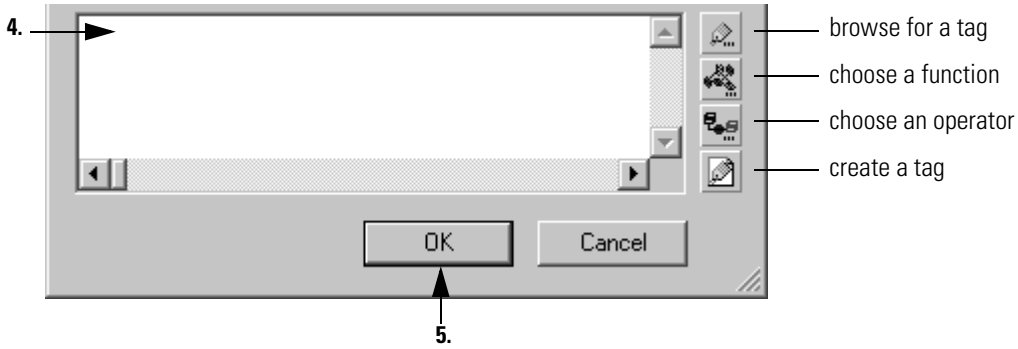
To enter a time as an expression:



1. Click the ... button of the step.
2. Select (check) the *Use Expression* check box.



3. Click the *Define* button.



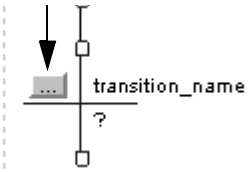
4. Type a **numeric expression** that defines the time.

- Use the buttons alongside the dialog box to help you complete the expression.
- For information on numeric expressions, see "Expressions" on page 7-4.

5. Choose OK

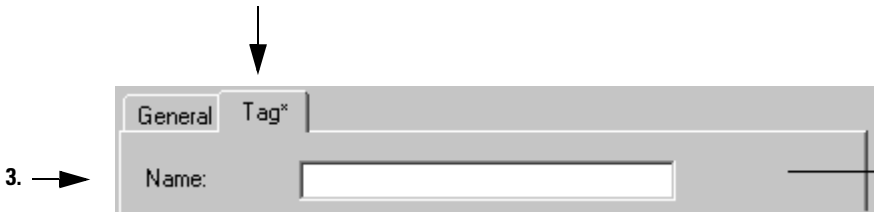
6. To close the *Step Properties* dialog box, choose OK

Rename a Transition



Each transition uses a tag to store the status of the transition. To rename the tag of the transition:

1. Click the ... button of the transition.
2. Click the *Tag* tab.



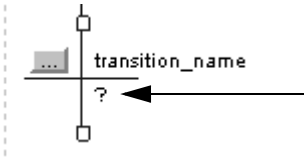
3. Type the new name for the transition (tag).
4. Choose OK

Program a Transition

To program a transition, you have these options:

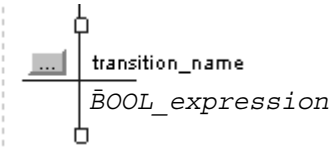
- Enter a BOOL Expression
- Call a Subroutine

Enter a BOOL Expression



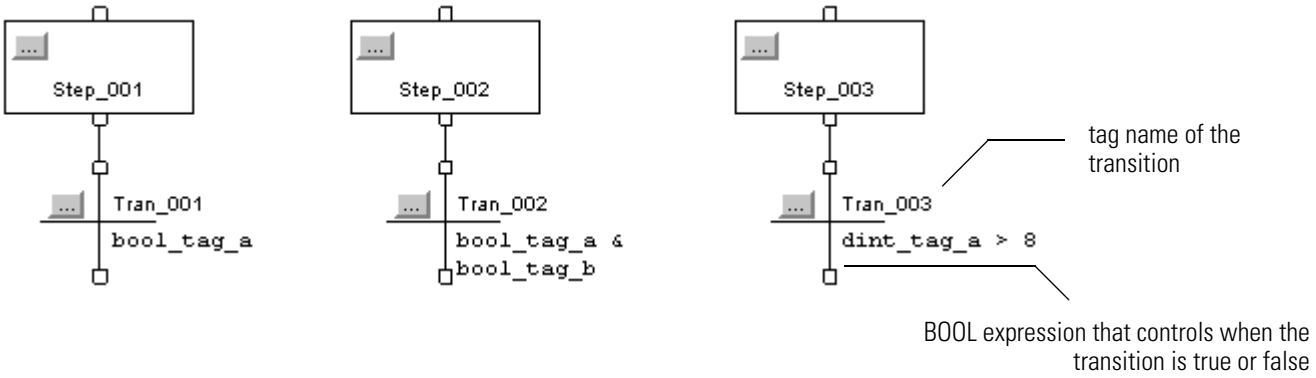
The simplest way to program the transition is to enter the conditions as a **BOOL expression** in structured text. For information on BOOL expressions, see "Expressions" on page 7-4.

1. Double-click the text area of the transition.
2. Type the BOOL expression that determines when the transition is true or false.
3. To close the text entry window, press [Ctrl] + [Enter].

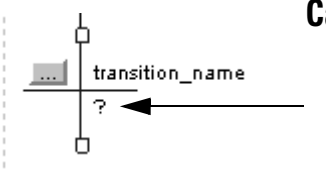


The following example shows three transitions that use a BOOL expression.

EXAMPLE Enter a BOOL Expression



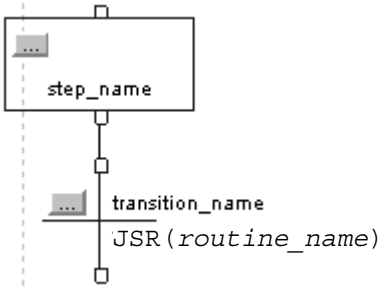
Call a Subroutine



1. Right-click the transition and choose *Set JSR*.

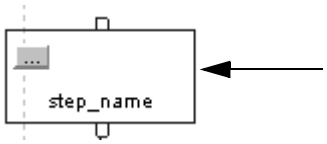
2. Choose the routine that contains the logic for the transition.

3. Choose OK

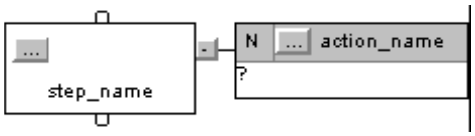


Add an Action

To add an action to a step:

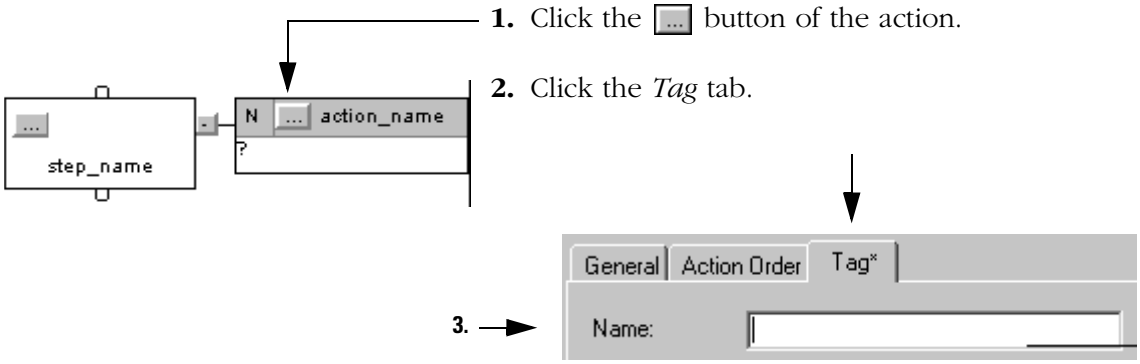


Right-click the step in which the action executes and choose *Add Action*.



Rename an Action

To change the name of an action to something that is specific to your application:



1. Click the [...] button of the action.
2. Click the *Tag* tab.
3. Type the new name for the action (tag).
4. Choose [OK]

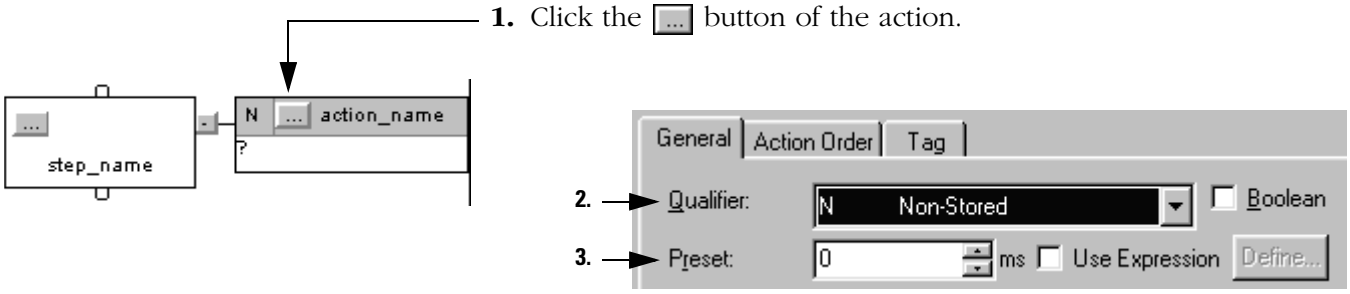
Configure an Action

To configure an action, you have these options:

- Change the Qualifier of an Action
- Calculate a Preset Time at Runtime
- Mark an Action as a Boolean Action

Change the Qualifier of an Action

A qualifier determines when an action starts and stops. The default qualifier is *N Non-Stored*. The action starts when the step is activated and stops when the step is deactivated. For more information, see "Choose a Qualifier for an Action" on page 5-23.



1. Click the ... button of the action.

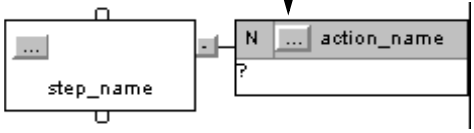
2. Assign the qualifier for the action.

3. If you chose a timed qualifier, type the time limit or delay for the action, in milliseconds. Timed qualifiers include:
   - L Time Limited
   - SL Stored and Time Limited
   - D Time Delayed
   - DS Delayed and Stored
   - SD Stored and Time Delayed

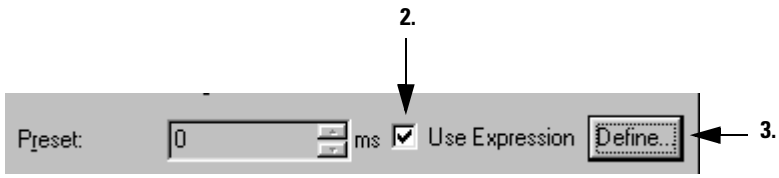
4. Choose OK

Calculate a Preset Time at Runtime

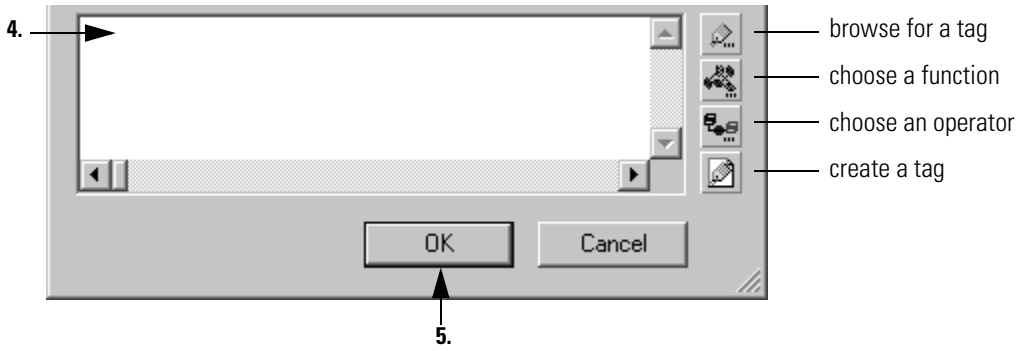
To calculate a preset value based on tags in your project, enter the value as a **numeric expression**.



1. Click the [...] button of the action.
2. Select (check) the *Use Expression* check box.



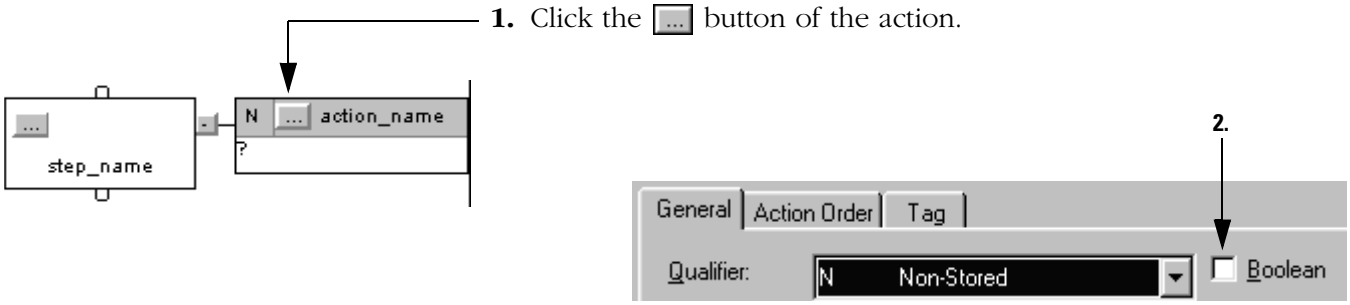
3. Click the *Define* button.



4. Type a **numeric expression** that defines the preset time.
   - Use the buttons alongside the dialog box to help you complete the expression.
   - For information on numeric expressions, see "Expressions" on page 7-4.
5. Choose [OK]
6. To close the *Action Properties* dialog box, choose [OK]

Mark an Action as a Boolean Action

Use a boolean action to only set a bit when the action executes. For more information, see "Use a Boolean Action" on page 5-20.



1. Click the [...] button of the action.
2. Click the *Boolean* check box.
3. Choose [OK]

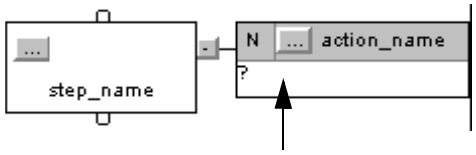
Program an Action

To program an action, you have these options:

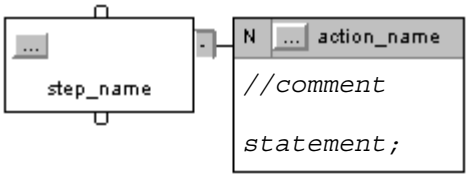
- Enter Structured Text
- Call a Subroutine

Enter Structured Text

The easiest way to program an action is to write the logic as structured text within the body of the action. When the action turns on, the controller executes the structured text.



1. Double-click the text area of the action.
2. Type the required structured text.
3. To close the text entry window, press [Ctrl] + [Enter].

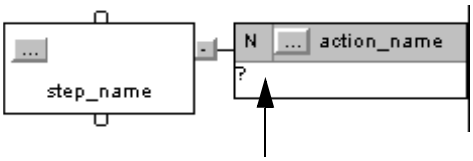


For information on structured text:

For this structured text information:	See:
general information about assignments, operators, functions, instructions, or comments	"Program Structured Text" on page 7-1
information about a specific instruction	• *Logix5000 Controllers General Instructions Reference Manual*, publication 1756-RM003 • *Logix5000 Controllers Process and Drives Instructions Reference Manual*, publication 1756-RM006 • *Logix5000 Controllers Motion Instruction Set Reference Manual*, publication 1756-RM007

Call a Subroutine

Use a Jump to Subroutine (JSR) instruction to execute a subroutine when the action is active.



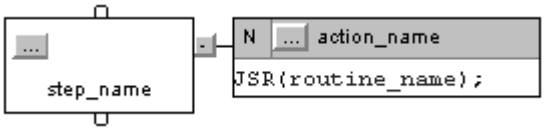
1. In the SFC, right-click the text entry area of the action and choose *Set JSR*.

2. Choose the routine that you want to call.

3. To pass a parameter to the routine, click the empty *Input Parameters* text box. Then use the down arrow to choose the tag that contains the parameter.

4. To receive a parameter from the routine, click the empty *Return Parameters* text box. Then use the down arrow to choose the tag in which to store the parameter from the routine.

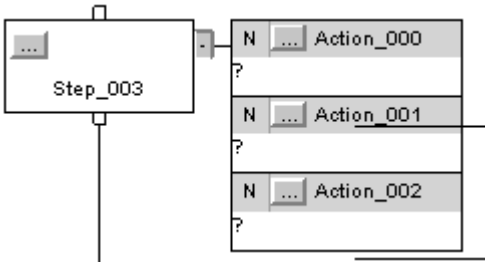
5. Choose OK



Assign the Execution Order of Actions

Actions execute in the order in which they appear.

For example:

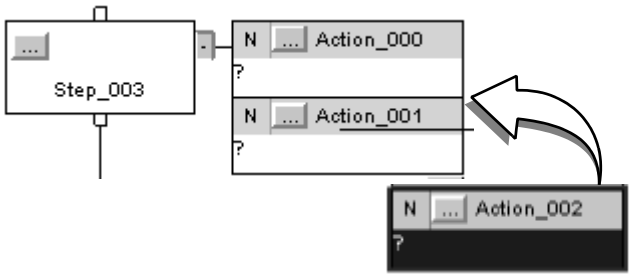


When *Step_003* is active, its actions execute in this order:

1. *Action_000*
2. *Action_001*
3. *Action_002*

To change the order in which an action executes, drag the action to the desired location in the sequence. A green bar shows a valid placement location.

For example:



Document the SFC

To document an SFC, you have the following options:

To document this:	And you want to:	Do this:
general information about the SFC	→	Add a Text Box
step	→	Add a Text Box -or- Add a Tag Description
transition	download the documentation to the controller	Add Structured Text Comments
	have the option of showing or hiding the documentation	Add a Text Box -or- Add a Tag Description
	position the documentation anywhere in the SFC	
action	download the documentation to the controller	Add Structured Text Comments
stop	→	Add a Text Box -or- Add a Tag Description
other element (e.g., selection branch)	→	

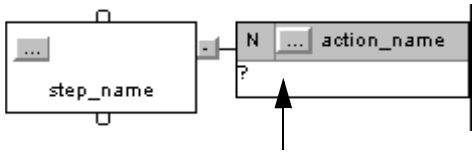
Add Structured Text Comments

Use the following table to format your comments:

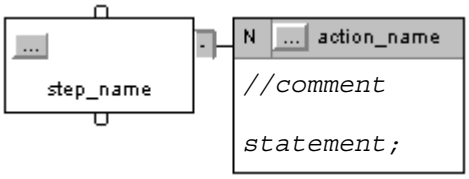
To add a comment:	Use one of these formats:
on a single line	`//comment`
at the end of a line of structured text	`(*comment*)` `/*comment*/`
within a line of structured text	`(*comment*)` `/*comment*/`
that spans more than one line	`(*start of comment . . . end of comment*)` `/*start of comment . . .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

For more information, see “Comments” on page 7-28.

To enter the comments:

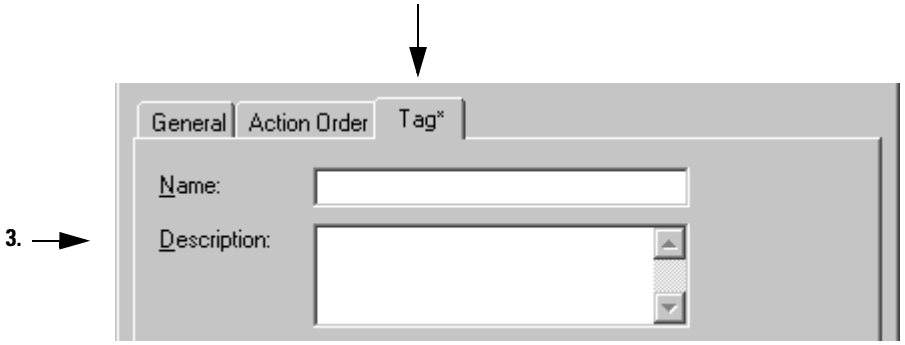


1. Double-click the text area of the action.
2. Type the comments.
3. To close the text entry window, press [Ctrl] + [Enter].



Add a Tag Description

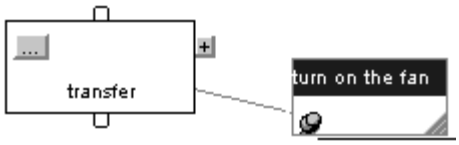
1. Click the [...] button of the element.
2. Click the *Tag* tab.

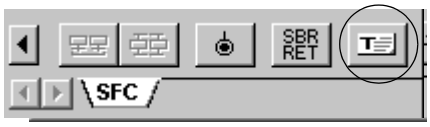


3. Type the description for the element (tag).
4. Choose [OK]
5. Drag the description box to the desired location on the SFC.

Add a Text Box

A text box lets you add notes that clarify the function of an SFC element (step, transition, stop, etc.). Or use a text box to capture information that you will use later on. For example:





1. Click 

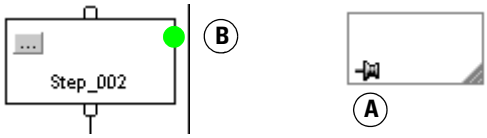
   A text box appears. 

2. Drag the text box to a location near the element to which it applies.

3. Double-click the text box and type the note. Then press [Ctrl] + [Enter].

4. As you move the element on the SFC, what do you want the text box to do?

If you the text box to:	Then:
stay in the same spot	Stop. You are done.
move with the element to which it applies	Go to step 5.



green dot

5. Click the pin symbol in the text box and then click the SFC element to which you want to attach the text box. A green dot shows a valid connection point.

Show or Hide Text Boxes or Tag Descriptions

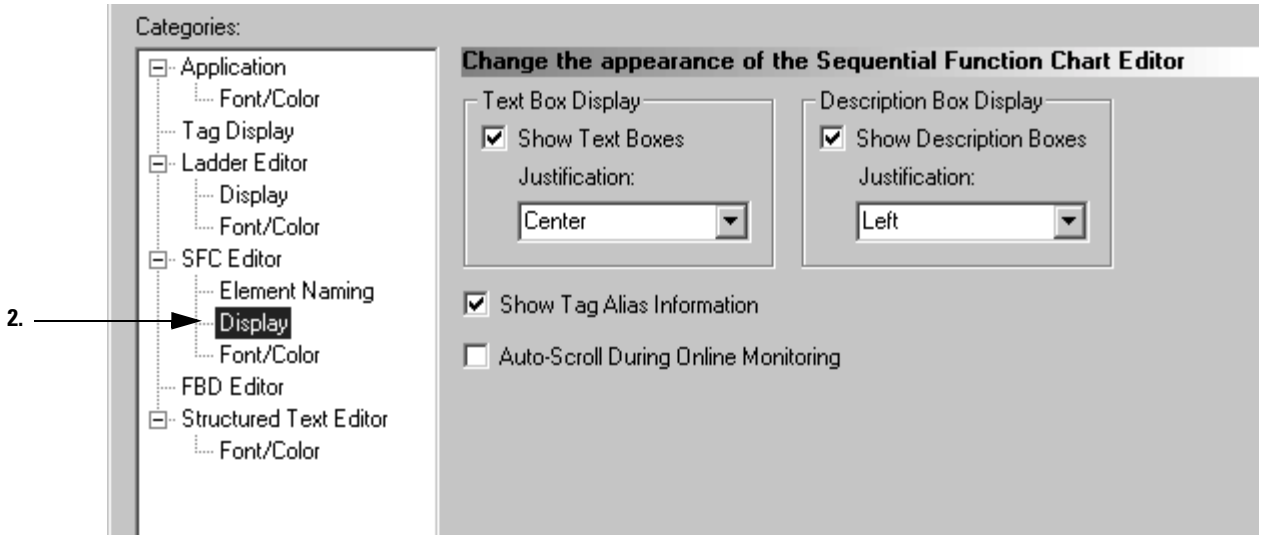
You have the option of showing or hiding both text boxes and tag descriptions. If you choose to show descriptions, the SFC window only shows the descriptions for steps, transitions, and stops (not actions).

To show or hide text boxes or descriptions, you have these options:

- Show or Hide Text Boxes or Descriptions
- Hide an Individual Tag Description

Show or Hide Text Boxes or Descriptions

1. From the *Tools* menu, choose *Options*.



2. Under *SFC Editor*, choose the *Display* category.

3. Choose the desired option.

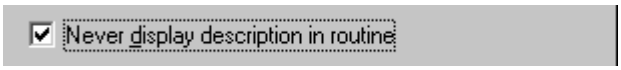
If you want to:	Then:
show text boxes or descriptions	check the corresponding check box
hide text boxes or descriptions	clear (uncheck) the corresponding check box

4. Choose OK

Hide an Individual Tag Description

To hide the description of a specific element while showing other descriptions:

1. Click the [...] button of the element whose description you want to hide.
2. Check the *Never display description in routine* check box.



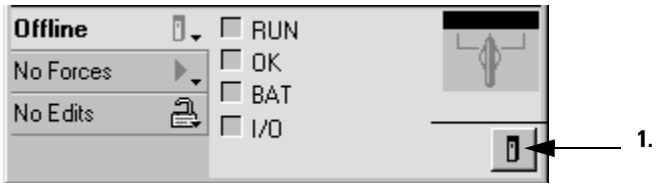
3. Choose [OK]

To show other descriptions, see "Show or Hide Text Boxes or Descriptions" on page 6-26.

Configure the Execution of the SFC

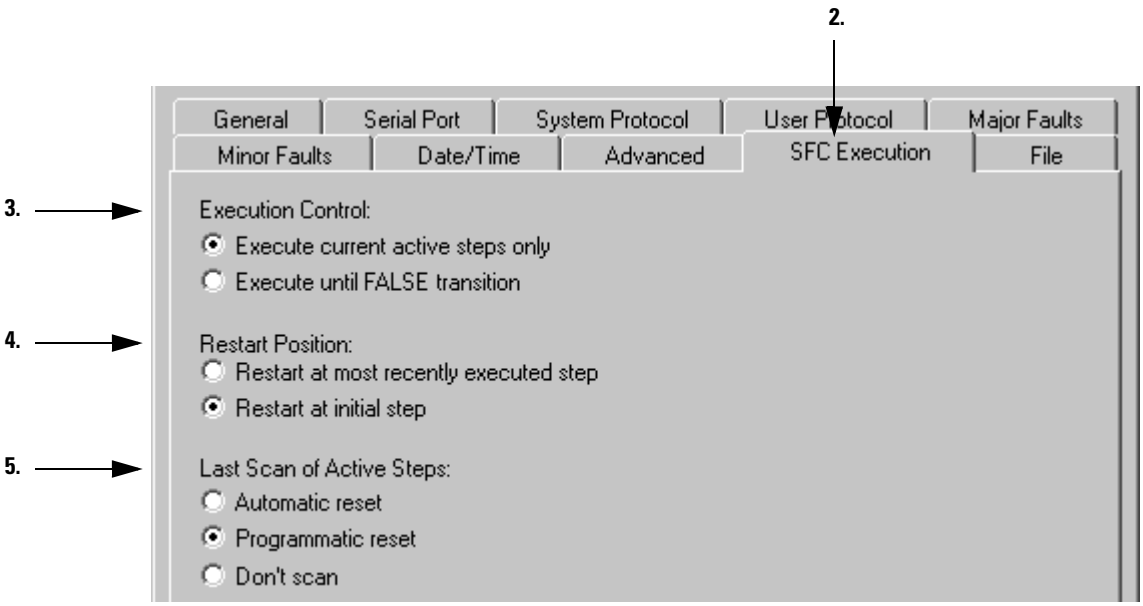
The SFC Execution tab of the controller properties lets you configure the following:

- what to do when a transition is true
- where to start after a transition to the run mode or recovery from a power loss
- what to do on the last scan of a step



1. On the Online toolbar, click controller properties button.

2. Click the *SFC Execution* tab.



3. Choose whether or not to return to the OS/JSR if a transition is true.

4. Choose where to restart the SFC after a transition to run mode or recovery from a power loss.

5. Choose what to do on the last scan of a step.

6. Choose OK

Verify the Routine

As you program your routine, periodically verify your work:

1. In the top-most toolbar of the RSLogix 5000 window, click 
2. If any errors are listed at the bottom of the window:
   a. To go to the first error or warning, press [F4].
   b. Correct the error according to the description in the Results window.
   c. Go to step 1.
3. To close the Results window, press [Alt] + [1].

Notes:

Program Structured Text

When to Use This Chapter

Use this chapter to write and enter structured text for a:

- structured text routine
- action of a sequential function chart (SFC)
- transition of sequential function chart (SFC)

Structured Text Syntax

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

- Structured text is not case sensitive.
- Use tabs and carriage returns (separate lines) to make your structured text easier to read. They have no effect on the execution of the structured text.

Structured text can contain these components:

Term:	Definition:		Examples:
assignment (see page 7-2)	Use an assignment statement to assign values to tags. The := operator is the assignment operator. Terminate the assignment with a semi colon ";".		`tag := expression;`
expression (see page 7-4)	An expression is part of a complete assignment or construct statement. An expression evaluates to a number (numerical expression) or to a true or false state (BOOL expression). An expression contains:		
	tags	A named area of the memory where data is stored (BOOL, SINT,INT,DINT, REAL, string).	`value1`
	immediates	A constant value.	`4`
	operators	A symbol or mnemonic that specifies an operation within an expression.	`tag1 + tag2` `tag1 >= value1`
	functions	When executed, a function yields one value. Use parentheses to contain the operand of a function. Even though their syntax is similar, functions differ from instructions in that functions can only be used in expressions. Instructions cannot be used in expressions.	`function(tag1)`

Term:	Definition:	Examples:
instruction (see page 7-11)	An instruction is a standalone statement. An instruction uses parenthesis to contain its operands. Depending on the instruction, there can be zero, one, or multiple operands. When executed, an instruction yields one or more values that are part of a data structure. Terminate the instruction with a semi colon ";". Even though their syntax is similar, instructions differ from functions in that instructions cannot be used in expressions. Functions can only be used in expressions.	*instruction();* *instruction(operand);* *instruction(operand1, operand2,operand3);*
construct (see page 7-12)	A conditional statement used to trigger structured text code (i.e, other statements). Terminate the construct with a semi colon ";".	IF...THEN CASE FOR...DO WHILE...DO REPEAT...UNTIL EXIT
comment (see page 7-28)	Text that explains or clarifies what a section of structured text does. • Use comments to make it easier to interpret the structured text. • Comments do not affect the execution of the structured text. • Comments can appear anywhere in structured text.	*//comment* (**start of comment . . . end of comment**) */*start of comment . . . end of comment*/*

Assignments

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

tag := *expression* ;

where:

Component:	Description:	
tag	represents the tag that is getting the new value the tag must be a BOOL, SINT, INT, DINT, or REAL	
:=	is the assignment symbol	
expression	represents the new value to assign to the tag	
	If *tag* is this data type:	**Use this type of expression:**
	BOOL	BOOL expression
	SINT INT DINT REAL	numeric expression
;	ends the assignment	

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

The expression can be simple, such as an immediate value or another tag name, or the expression can be complex and include several operators and/or functions. See the next section “Expressions“on page 7-4 for details.

Specify a non-retentive assignment

The non-retentive assignment is different from the regular assignment described above in that the tag in a non-retentive assignment is reset to zero each time the controller:

- enters the RUN mode
- leaves the step of an SFC if you configure the SFC for *Automatic reset* (This applies only if you embed the assignment in the action of the step or use the action to call a structured text routine via a JSR instruction.)

A non-retentive assignment has this syntax:

tag [:=] *expression* ;

where:

Component:	Description:	
tag	represents the tag that is getting the new value the tag must be a BOOL, SINT, INT, DINT, or REAL	
[:=]	is the non-retentive assignment symbol	
expression	represents the new value to assign to the tag	
	If *tag* is this data type:	**Use this type of expression:**
	BOOL	BOOL expression
	SINT INT DINT REAL	numeric expression
;	ends the assignment	

Assign an ASCII character to a string

Use the assignment operator to assign an ASCII character to an element of the DATA member of a string tag. To assign a character, specify the value of the character or specify the tag name, DATA member, and element of the character. For example:

This is OK:	This is *not* OK.
`string1.DATA[0]:= 65;`	`string1.DATA[0] := A;`
`string1.DATA[0]:= string2.DATA[0];`	`string1 := string2;`

To add or insert a string of characters to a string tag, use either of these ASCII string instructions:

To:	Use this instruction:
add characters to the end of a string	CONCAT
insert characters into a string	INSERT

Expressions

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

- tag name that stores the value (variable)
- number that you enter directly into the expression (immediate value)
- functions, such as: ABS, TRUNC
- operators, such as: +, -, <, >, And, Or

As you write expressions, follow these general rules:

- Use any combination of upper-case and lower-case letter. For example, these three variations of "AND" are acceptable: AND, And, and.
- For more complex requirements, use parentheses to group expressions within expressions. This makes the whole expression easier to read and ensures that the expression executes in the desired sequence. See “Determine the order of execution“on page 7-10.

In structured text, you use two types of expressions:

BOOL expression: An expression that produces either the BOOL value of 1 (true) or 0 (false).

- A bool expression uses bool tags, relational operators, and logical operators to compare values or check if conditions are true or false. For example, `tag1>65`.
- A simple bool expression can be a single BOOL tag.
- Typically, you use bool expressions to condition the execution of other logic.

Numeric expression: An expression that calculates an integer or floating-point value.

- A numeric expression uses arithmetic operators, arithmetic functions, and bitwise operators. For example, `tag1+5`.
- Often, you nest a numeric expression within a bool expression. For example, `(tag1+5)>65`.

Use the following table to choose operators for your expressions:

If you want to:	Then:
Calculate an arithmetic value	"Use arithmetic operators and functions"on page 7-6.
Compare two values or strings	"Use relational operators"on page 7-7.
Check if conditions are true or false	"Use logical operators"on page 7-9.
Compare the bits within values	"Use bitwise operators"on page 7-10.

Use arithmetic operators and functions

You can combine multiple operators and functions in arithmetic expressions.

Arithmetic operators calculate new values.

To:	Use this operator:	Optimal data type:
add	+	DINT, REAL
subtract/negate	-	DINT, REAL
multiply	*	DINT, REAL
exponent (x to the power of y)	**	DINT, REAL
divide	/	DINT, REAL
modulo-divide	MOD	DINT, REAL

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

For:	Use this function:	Optimal data type:
absolute value	ABS (*numeric_expression*)	DINT, REAL
arc cosine	ACOS (*numeric_expression*)	REAL
arc sine	ASIN (*numeric_expression*)	REAL
arc tangent	ATAN (*numeric_expression*)	REAL
cosine	COS (*numeric_expression*)	REAL
radians to degrees	DEG (*numeric_expression)*	DINT, REAL
natural log	LN (*numeric_expression*)	REAL
log base 10	LOG (*numeric_expression*)	REAL
degrees to radians	RAD (*numeric_expression*)	DINT, REAL
sine	SIN (*numeric_expression*)	REAL
square root	SQRT (*numeric_expression*)	DINT, REAL
tangent	TAN (*numeric_expression*)	REAL
truncate	TRUNC (*numeric_expression*)	DINT, REAL

For example:

Use this format:	**Example:**	
	For this situation:	**You'd write:**
value1 operator value2	If *gain_4* and *gain_4_adj* are DINT tags and your specification says: "Add 15 to *gain_4* and store the result in *gain_4_adj*."	`gain_4_adj := gain_4+15;`
operator value1	If alarm and *high_alarm* are DINT tags and your specification says: "Negate *high_alarm* and store the result in *alarm*."	`alarm:= -high_alarm;`
function(numeric_expression)	If *overtravel* and *overtravel_POS* are DINT tags and your specification says: "Calculate the absolute value of *overtravel* and store the result in *overtravel_POS*."	`overtravel_POS := ABS(overtravel);`
value1 operator (function((value2+value3)/2)	If *adjustment* and *position* are DINT tags and *sensor1* and *sensor2* are REAL tags and your specification says: "Find the absolute value of the average of *sensor1* and *sensor2*, add the *adjustment*, and store the result in *position*."	`position := adjustment + ABS((sensor1 + sensor2)/2);`

Use relational operators

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

If the comparison is:	**The result is:**
true	1
false	0

Use the following relational operators:

For this comparison:	**Use this operator:**	**Optimal Data Type:**
equal	=	DINT, REAL, string
less than	<	DINT, REAL, string
less than or equal	<=	DINT, REAL, string
greater than	>	DINT, REAL, string
greater than or equal	>=	DINT, REAL, string
not equal	<>	DINT, REAL, string

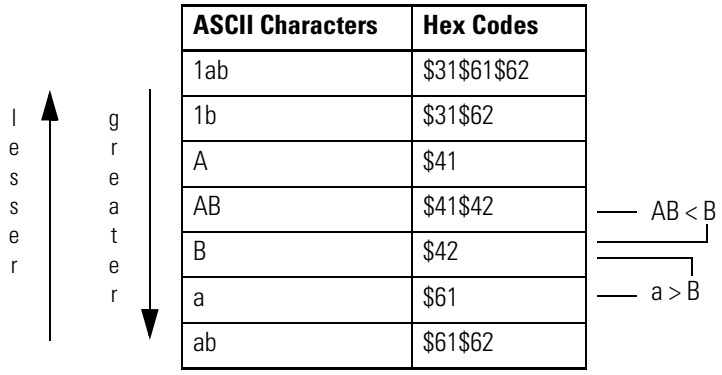
For example:

Use this format:	Example:	
	For this situation:	**You'd write:**
value1 operator value2	If *temp* is a DINT tag and your specification says: "If *temp* is less than 100° then…"	`IF temp<100 THEN...`
stringtag1 operator stringtag2	If *bar_code* and *dest* are string tags and your specification says: "If *bar_code* equals *dest* then…"	`IF bar_code=dest THEN...`
char1 operator char2 To enter an ASCII character directly into the expression, enter the decimal value of the character.	If *bar_code* is a string tag and your specification says: "If *bar_code.DATA[0]* equals 'A' then…"	`IF bar_code.DATA[0]=65 THEN...`
bool_tag := bool_expressions	If *count* and *length* are DINT tags, *done* is a BOOL tag, and your specification says "If *count* is greater than or equal to *length*, you are done counting."	`done := (count >= length);`

How Strings Are Evaluated

The hexadecimal values of the ASCII characters determine if one string is less than or greater than another string.

- When the two strings are sorted as in a telephone directory, the order of the strings determines which one is greater.



ASCII Characters	Hex Codes
1ab	$31$61$62
1b	$31$62
A	$41
AB	$41$42
B	$42
a	$61
ab	$61$62

- Strings are equal if their characters match.
- Characters are case sensitive. Upper case "A" ($41) is *not* equal to lower case "a" ($61).

For the decimal value and hex code of a character, see the back cover of this manual.

Use logical operators

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

If the comparison is:	The result is:
true	1
false	0

Use the following logical operators:

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

For example:

Use this format:	Example:	
	For this situation:	**You'd write:**
BOOLtag	If *photoeye* is a BOOL tag and your specification says: "If *photoeye_1* is on then…"	`IF photoeye THEN...`
NOT *BOOLtag*	If *photoeye* is a BOOL tag and your specification says: "If *photoeye* is off then…"	`IF NOT photoeye THEN...`
expression1 & *expression2*	If *photoeye* is a BOOL tag, *temp* is a DINT tag, and your specification says: "If *photoeye* is on and *temp* is less than 100° then…".	`IF photoeye & (temp<100) THEN...`
expression1 OR *expression2*	If *photoeye* is a BOOL tag, *temp* is a DINT tag, and your specification says: "If *photoeye* is on or *temp* is less than 100° then…".	`IF photoeye OR (temp<100) THEN...`
expression1 XOR *expression2*	If *photoeye1* and *photoeye2* are BOOL tags and your specification says: "If: • *photoeye1* is on while *photoeye2* is off or • *photoeye1* is off while *photoeye2* is on then…"	`IF photoeye1 XOR photoeye2 THEN...`
BOOLtag := *expression1* & *expression2*	If *photoeye1* and *photoeye2* are BOOL tags, *open* is a BOOL tag, and your specification says: "If *photoeye*1 and *photoeye2* are both on, set *open* to true".	`open := photoeye1 & photoeye2;`

Use bitwise operators

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

For:	Use this operator:	Optimal Data Type:
bitwise AND	&, AND	DINT
bitwise OR	OR	DINT
bitwise exclusive OR	XOR	DINT
bitwise complement	NOT	DINT

For example:

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

Determine the order of execution

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

- Operations of equal order are performed from left to right.
- If an expression contains multiple operators or functions, group the conditions in parenthesis "( )" . This ensures the correct order of execution and makes it easier to read the expression.

Order:	Operation:
1.	( )
2.	function (…)
3.	**
4.	– (negate)
5.	NOT
6.	*, /, MOD
7.	+, - (subtract)
8.	<, <=, >, >=
9.	=, <>
10.	&, AND
11.	XOR
12.	OR

Instructions

Structured text statements can also be instructions. See the Locator Table at the beginning of this manual for a list of the instructions available in structured text. A structured text instruction executes each time it is scanned. A structured text instruction within a construct executes every time the conditions of the construct are true. If the conditions of the construct are false, the statements within the construct are not scanned. There is no rung-condition or state transition that triggers execution.

This differs from function block instructions that use EnableIn to trigger execution. Structured text instructions execute as if EnableIn is always set.

This also differs from relay ladder instructions that use rung-condition-in to trigger execution. Some relay ladder instructions only execute when rung-condition-in toggles from false to true. These are transitional relay ladder instructions. In structured text, instructions will execute each time they are scanned unless you pre-condition the execution of the structured text instruction.

For example, the ABL instruction is a transitional instruction in relay ladder. In this example, the ABL instruction only executes on a scan when *tag_xic* transitions from cleared to set. The ABL instruction does not execute when *tag_xic* stays set or when *tag_xic* is cleared.

In structured text, if you write this example as:

```
IF tag_xic THEN ABL(0,serial_control);

END_IF;
```

the ABL instruction will execute every scan that *tag_xic* is set, not just when *tag_xic* transitions from cleared to set.

If you want the ABL instruction to execute only when *tag_xic* transitions from cleared to set, you have to condition the structured text instruction. Use a one shot to trigger execution.

```
osri_1.InputBit := tag_xic;
OSRI(osri_1);

IF (osri_1.OutputBit) THEN
     ABL(0,serial_control);
END_IF;
```

Constructs

Constructs can be programmed singly or nested within other constructs.

If you want to:	Use this construct:	Available in these languages:	See page:
do something if or when specific conditions occur	IF...THEN	structured text	7-13
select what to do based on a numerical value	CASE...OF	structured text	7-16
do something a specific number of times before doing anything else	FOR...DO	structured text	7-19
keep doing something as long as certain conditions are true	WHILE...DO	structured text	7-22
keep doing something until a condition is true	REPEAT...UNTIL	structured text	7-25

IF...THEN

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

Operands:

Structured Text

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

Operand:	Type:	Format:	Enter:
bool_ expression	BOOL	tag expression	BOOL tag or expression that evaluates to a BOOL value (BOOL expression)

Description:

The syntax is:

```
IF bool_expression1 THEN
      <statement >;        <-- statements to execute when bool_expression1 is true
                .
                .
                .
ELSIF bool_expression2 THEN   (optional)
      <statement>;         <-- statements to execute when bool_expression2 is true
                .
                .
                .
ELSE                          (optional)
      <statement>;         <-- statements to execute when both expressions are false
                .
                .
                .
END_IF;
```

To use ELSIF or ELSE, follow these guidelines:

1. To select from several possible groups of statements, add one or more ELSIF statements.
   - Each ELSIF represents an alternative path.
   - Specify as many ELSIF paths as you need.
   - The controller executes the first true IF or ELSIF and skips the rest of the ELSIFs and the ELSE.
2. To do something when all of the IF or ELSIF conditions are false, add an ELSE statement.

The following table summarizes different combinations of IF, THEN, ELSIF, and ELSE.

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

Arithmetic Status Flags: not affected

Fault Conditions: none

Example 1: IF…THEN

If you want this:	Enter this structured text:
IF rejects > 3 then	`IF rejects > 3 THEN`
conveyor = off (0)	`conveyor := 0;`
alarm = on (1)	`alarm := 1;`
	`END_IF;`

Example 2: IF…THEN…ELSE

If you want this:	Enter this structured text:
If conveyor direction contact = forward (1) then	`IF conveyor_direction THEN`
light = off	`light := 0;`
Otherwise light = on	`ELSE`
	`light [:=] 1;`
	`END_IF;`

The [:=] tells the controller to clear *light* whenever the controller:

- enters the RUN mode
- leaves the step of an SFC if you configure the SFC for *Automatic reset* (This applies only if you embed the assignment in the action of the step or use the action to call a structured text routine via a JSR instruction.)

Example 3: IF…THEN…ELSIF

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

The [:=] tells the controller to clear *Sugar.Inlet* whenever the controller:

- enters the RUN mode
- leaves the step of an SFC if you configure the SFC for *Automatic reset* (This applies only if you embed the assignment in the action of the step or use the action to call a structured text routine via a JSR instruction.)

Example 4: IF…THEN…ELSIF…ELSE

If you want this:	Enter this structured text:
If tank temperature > 100	`IF tank.temp > 200 THEN`
then pump = slow	`pump.fast :=1; pump.slow :=0; pump.off :=0;`
If tank temperature > 200	`ELSIF tank.temp > 100 THEN`
then pump = fast	`pump.fast :=0; pump.slow :=1; pump.off :=0;`
otherwise pump = off	`ELSE`
	`pump.fast :=0; pump.slow :=0; pump.off :=1;`
	`END_IF;`

CASE...OF

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

Operands:

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

Structured Text

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

IMPORTANT If you use REAL values, use a range of values for a selector because a REAL value is more likely to be within a range of values than an exact match of one, specific value.

Description: The syntax is:

See the table on the next page for valid selector values.

The syntax for entering the selector values is:

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

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

Arithmetic Status Flags: not affected

Fault Conditions: none

Example

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

The [:=] tells the controller to also clear the outlet tags whenever the controller:

- enters the RUN mode
- leaves the step of an SFC if you configure the SFC for *Automatic reset* (This applies only if you embed the assignment in the action of the step or use the action to call a structured text routine via a JSR instruction.)

FOR…DO

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

Operands:

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

Structured Text

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

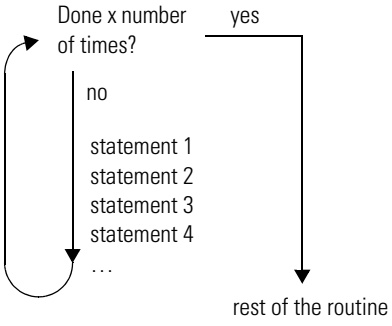
IMPORTANT Make sure that you *do not* iterate within the loop too many times in a single scan.

- The controller *does not* execute any other statements in the routine until it completes the loop.
- If the time that it takes to complete the loop is greater than the watchdog timer for the task, a major fault occurs.
- Consider using a different construct, such as IF...THEN.

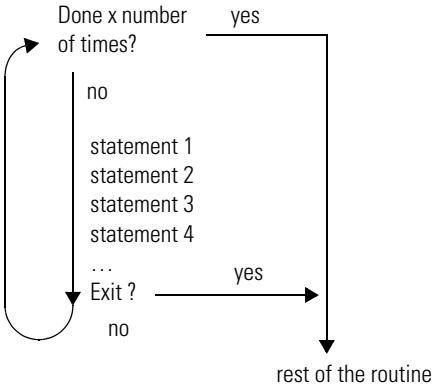
Description: The syntax is:

```
FOR count := initial_value
   TO final_value
optional { BY increment                    If you don't specify an increment, the loop
                                           increments by 1.
   DO
   <statement>;
   IF bool_expression THEN
optional     EXIT;          <-------       If there are conditions when you want to
   END_IF;                                 exit the loop early, use other statements,
                                           such as an IF...THEN construct, to
                                           condition an EXIT statement.
END_FOR;
```

The following diagrams show how a FOR...DO loop executes and how an EXIT statement leaves the loop early.



The FOR…DO loop executes a specific number of times.



To stop the loop before the count reaches the last value, use an EXIT statement.

Arithmetic Status Flags: not affected

Fault Conditions:

A major fault will occur if:	Fault type:	Fault code:
the construct loops too long	6	1

Example 1:

If you want this:	Enter this structured text:

Clear bits 0 - 31 in an array of BOOLs:
1. Initialize the *subscript* tag to 0.
2. Clear *array[subscript]* . For example, when *subscript* = 5, clear *array[5]*.
3. Add 1 to *subscript*.
4. If *subscript* is ≤ to 31, repeat 2 and 3. Otherwise, stop.
	`For subscript:=0 to 31 by 1 do`
	`  array[subscript] := 0;`
	`End_for;`

Example 2:

If you want this:	Enter this structured text:

A user-defined data type (structure) stores the following information about an item in your inventory:
• Barcode ID of the item (string data type)
• Quantity in stock of the item (DINT data type)
An array of the above structure contains an element for each different item in your inventory. You want to search the array for a specific product (use its bar code) and determine the quantity that is in stock.
1. Get the size (number of items) of the *Inventory* array and store the result in *Inventory_Items* (DINT tag).
2. Initialize the *position* tag to 0.
3. If *Barcode* matches the ID of an item in the array, then:
a. Set the *Quantity* tag = *Inventory[position].Qty*. This produces the quantity in stock of the item.
b. Stop.
Barcode is a string tag that stores the bar code of the item for which you are searching. For example, when *position* = 5, compare *Barcode* to *Inventory[5].ID*.
4. Add 1 to *position*.
5. If *position* is ≤ to (*Inventory_Items -1)*, repeat 3 and 4. Since element numbers start at 0, the last element is 1 less than the number of elements in the array. Otherwise, stop.
	`SIZE(Inventory,0,Inventory_Items);`
	`For position:=0 to Inventory_Items - 1 do`
	`  If Barcode = Inventory[position].ID then`
	`  Quantity := Inventory[position].Qty;`
	`  Exit;`
	`  End_if;`
	`End_for;`

WHILE...DO

Use the WHILE…DO loop to keep doing something as long as certain conditions are true.

Operands:

Structured Text

```
WHILE bool_expression DO
    <statement>;
END_WHILE;
```

Operand:	Type:	Format:	Enter:
bool_ expression	BOOL	tag expression	BOOL tag or expression that evaluates to a BOOL value

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

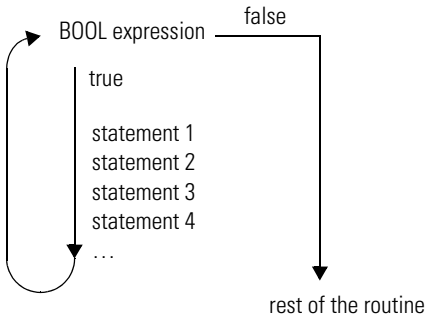
- The controller *does not* execute any other statements in the routine until it completes the loop.
- If the time that it takes to complete the loop is greater than the watchdog timer for the task, a major fault occurs.
- Consider using a different construct, such as IF...THEN.

Description: The syntax is:

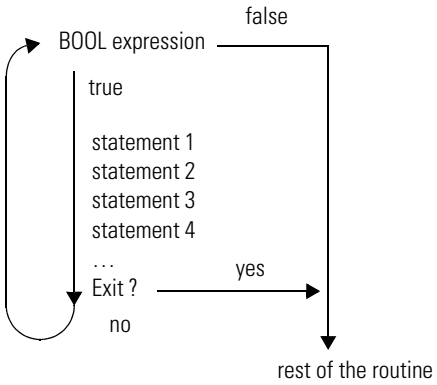
```
WHILE bool_expression1 DO
   <statement>;          <-- statements to execute while bool_expression1 is true
   IF bool_expression2 THEN
     EXIT;               <-- If there are conditions when you want to exit the loop early, use other statements, such as an IF...THEN construct, to condition an EXIT statement.
   END_IF;
END_WHILE;
```

optional

The following diagrams show how a WHILE...DO loop executes and how an EXIT statement leaves the loop early.



While the *bool_expression* is true, the controller executes only the statements within the WHILE…DO loop.



To stop the loop before the conditions are true, use an EXIT statement.

Arithmetic Status Flags: not affected

Fault Conditions:

A major fault will occur if:	Fault type:	Fault code:
the construct loops too long	6	1

Example 1:

If you want this:	Enter this structured text:

The WHILE...DO loop evaluates its conditions first. If the conditions are true, the controller then executes the statements within the loop.
This differs from the REPEAT...UNTIL loop because the REPEAT...UNTIL loop executes the statements in the construct and then determines if the conditions are true before executing the statements again. The statements in a REPEAT...UNTIL loop are always executed at least once. The statements in a WHILE...DO loop might never be executed.
	`pos := 0;`
	`While ((pos <= 100) & structarray[pos].value <> targetvalue)) do`
	`pos := pos + 2;`
	`String_tag.DATA[pos] := SINT_array[pos];`
	`end_while;`

Example 2:

If you want this:	Enter this structured text:

Move ASCII characters from a SINT array into a string tag. (In a SINT array, each element holds one character.) Stop when you reach the carriage return.
1. Initialize *Element_number* to 0.
2. Count the number of elements in *SINT_array* (array that contains the ASCII characters) and store the result in *SINT_array_size* (DINT tag).
3. If the character at *SINT_array[element_number]* = 13 (decimal value of the carriage return), then stop.
4. Set *String_tag[element_number]* = the character at *SINT_array[element_number]*.
5. Add 1 to *element_number*. This lets the controller check the next character in *SINT_array*.
6. Set the Length member of *String_tag* = *element_number*. (This records the number of characters in *String_tag* so far.)
7. If *element_number* = *SINT_array_size,* then stop. (You are at the end of the array and it does not contain a carriage return.)
8. Go to 3.
	`element_number := 0;`
	`SIZE(SINT_array, 0, SINT_array_size);`
	`While SINT_array[element_number] <> 13 do`
	`  String_tag.DATA[element_number] := SINT_array[element_number];`
	`  element_number := element_number + 1;`
	`  String_tag.LEN := element_number;`
	`  If element_number = SINT_array_size then`
	`  exit;`
	`  end_if;`
	`end_while;`

REPEAT…UNTIL

Use the REPEAT…UNTIL loop to keep doing something until conditions are true.

Operands:

Structured Text

```
REPEAT
    <statement>;
UNTIL bool_expression
END_REPEAT;
```

Operand:	Type:	Format:	Enter:
bool_ expression	BOOL	tag expression	BOOL tag or expression that evaluates to a BOOL value (BOOL expression)

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

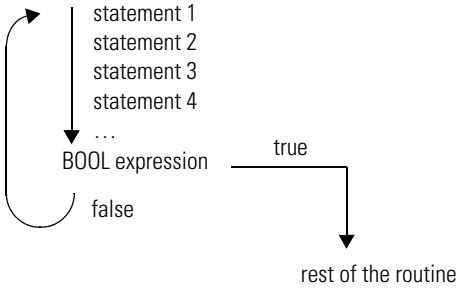
- The controller *does not* execute any other statements in the routine until it completes the loop.
- If the time that it takes to complete the loop is greater than the watchdog timer for the task, a major fault occurs.
- Consider using a different construct, such as IF...THEN.

Description: The syntax is:

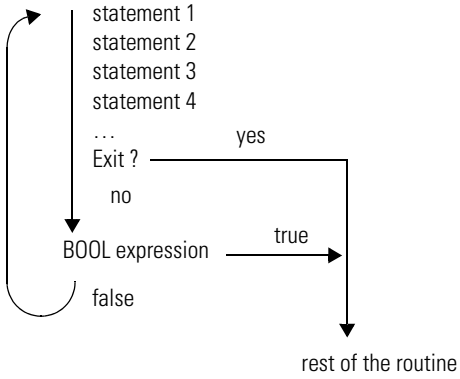
```
REPEAT
   <statement>;          <-- statements to execute while bool_expression1 is false
   IF bool_expression2 THEN
     EXIT;               <-- If there are conditions when you want to exit the loop early, use other statements, such as an IF...THEN construct, to condition an EXIT statement.
   END_IF;
UNTIL bool_expression1
END_REPEAT;
```

(IF...END_IF is optional)

The following diagrams show how a REPEAT...UNTIL loop executes and how an EXIT statement leaves the loop early.



While the *bool_expression* is false, the controller executes only the statements within the REPEAT…UNTIL loop.



To stop the loop before the conditions are false, use an EXIT statement.

Arithmetic Status Flags: not affected

Fault Conditions:

A major fault will occur if:	Fault type:	Fault code:
the construct loops too long	6	1

Example 1:

If you want this:	Enter this structured text:

The REPEAT...UNTIL loop executes the statements in the construct and then determines if the conditions are true before executing the statements again.
This differs from the WHILE...DO loop because the WHILE...DO The WHILE...DO loop evaluates its conditions first. If the conditions are true, the controller then executes the statements within the loop. The statements in a REPEAT...UNTIL loop are always executed at least once. The statements in a WHILE...DO loop might never be executed.
	`pos := -1;`
	`REPEAT`
	`  pos := pos + 2;`
	`UNTIL ((pos = 101) OR (structarray[pos].value = targetvalue))`
	`end_repeat;`

Example 2:

If you want this:	Enter this structured text:

Move ASCII characters from a SINT array into a string tag. (In a SINT array, each element holds one character.) Stop when you reach the carriage return.
1. Initialize *Element_number* to 0.
2. Count the number of elements in *SINT_array* (array that contains the ASCII characters) and store the result in *SINT_array_size* (DINT tag).
3. Set *String_tag[element_number]* = the character at *SINT_array[element_number]*.
4. Add 1 to *element_number*. This lets the controller check the next character in *SINT_array*.
5. Set the Length member of *String_tag* = *element_number*. (This records the number of characters in *String_tag* so far.)
6. If *element_number* = *SINT_array_size,* then stop. (You are at the end of the array and it does not contain a carriage return.)
7. If the character at *SINT_array[element_number]* = 13 (decimal value of the carriage return), then stop. Otherwise, go to 3.
	`element_number := 0;`
	`SIZE(SINT_array, 0, SINT_array_size);`
	`Repeat`
	`  String_tag.DATA[element_number] := SINT_array[element_number];`
	`  element_number := element_number + 1;`
	`  String_tag.LEN := element_number;`
	`  If element_number = SINT_array_size then`
	`  exit;`
	`  end_if;`
	`Until SINT_array[element_number] = 13`
	`end_repeat;`

Comments

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

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

To add comments to your structured text:

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

For example:

Format:	Example:

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

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

*/*comment*/*	`Sugar.Inlet:=0;/*close the inlet*/`
	
	`IF bar_code=65 /*A*/ THEN...`
	
	`/*Gets the number of elements in the Inventory array and stores the value in the Inventory_Items tag*/`
	`SIZE(Inventory,0,Inventory_Items);`

Force Logic Elements

When to Use This Procedure

Use a force to override data that your logic either uses or produces. For example, use forces in the following situations:

- test and debug your logic
- check wiring to an output device
- temporarily keep your process functioning when an input device has failed

Use forces only as a temporary measure. They are *not* intended to be a permanent part of your application.

How to Use This Procedure

If you want to:	See:
review the precautions that you should take whenever you add, change, remove, or disable forces	"Precautions" on page 14-2
determine current state of forces in your project	"Check Force Status" on page 14-4
determine which type of element to force in your project	"What to Force" on page 14-6
review general information about I/O forces, including which elements you are permitted to force and how an I/O force effects your project	"When to Use an I/O Force" on page 14-6
force an I/O value	"Add an I/O Force" on page 14-8
review general information about stepping through a transition or a simultaneous path	"When to Use Step Through" on page 14-9
step through an active transition	"Step Through a Transition or a Force of a Path" on page 14-9
step through a simultaneous path that is forced false	
review general information about SFC forces, including which elements you are permitted to force and how the forces effect the execution of your SFC	"When to Use an SFC Force" on page 14-9
force a transition or simultaneous path within an SFC	"Add an SFC Force" on page 14-12
stop the effects of a force	"Remove or Disable Forces" on page 14-13

Precautions

When you use forces, take the following precautions:

ATTENTION

Forcing can cause unexpected machine motion that could injure personnel. Before you use a force, determine how the force will effect your machine or process and keep personnel away from the machine area.

- Enabling I/O forces causes input, output, produced, or consumed values to change.
- Enabling SFC forces causes your machine or process to go to a different state or phase.
- Removing forces may still leave forces in the enabled state.
- If forces are enabled and you install a force, the new force immediately takes effect.

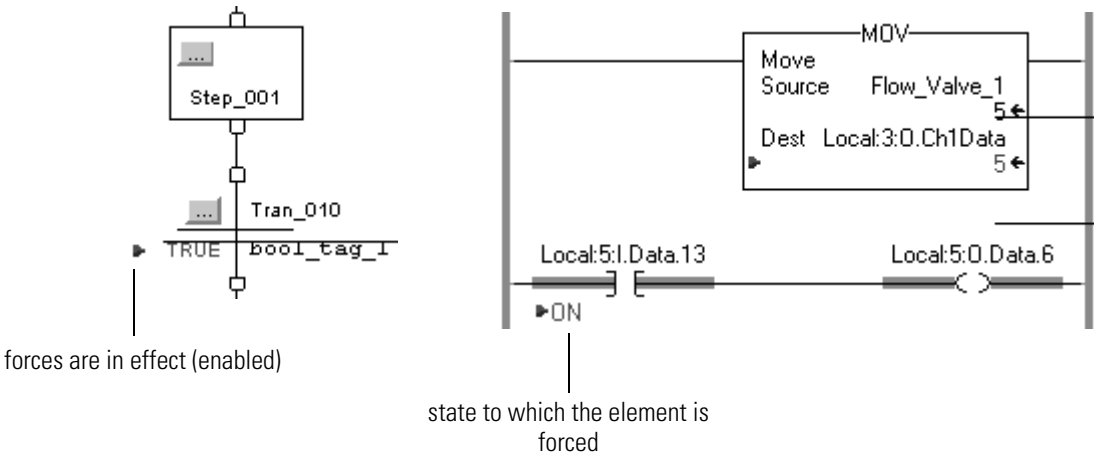
Enable Forces

For a force to take effect, you enable forces. You can only enable and disable forces at the controller level.

- You can enable I/O forces and SFC forces separately or at the same time.
- You cannot enable or disable forces for a specific module, tag collection, or tag element.

IMPORTANT If you download a project that has forces enabled, the programming software prompts you to enable or disable forces after the download completes.

When forces are in effect (enabled), a ▶ appears next to the forced element.



Disable or Remove a Force

To stop the effect of a force and let your project execute as programmed, disable or remove the force.

- You can disable or remove I/O and SFC forces at the same time or separately.
- Removing a force on an alias tag also removes the force on the base tag.

ATTENTION

Changes to forces can cause unexpected machine motion that could injure personnel. Before you disable or remove forces, determine how the change will effect your machine or process and keep personnel away from the machine area.

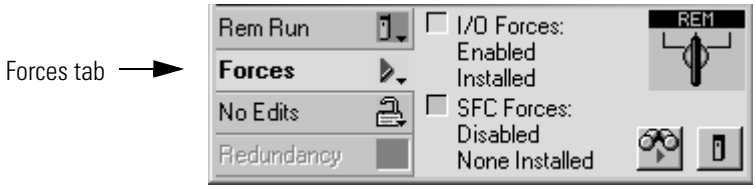
Check Force Status

Before you use a force, determine the status of forces for the controller. You can check force status in the following ways:

To determine the status of:	Use any of the following:
I/O forces	• Online Toolbar • FORCE LED • GSV Instruction
SFC forces	Online Toolbar

Online Toolbar

The Online toolbar shows the status of forces. It shows the status of I/O forces and SFC forces separately.



This:	Means:
Enabled	• If the project contains any forces of this type, they *are* overriding your logic. • If you add a force of this type, the new force immediately takes effect
Disabled	Forces of this type are inactive. If the project contains any forces of this type, they *are not* overriding your logic.
Installed	At least one force of this type exists in the project.
None Installed	No forces of this type exist in the project.

FORCE LED

If your controller has a FORCE LED, use the LED to determine the status of any I/O forces.

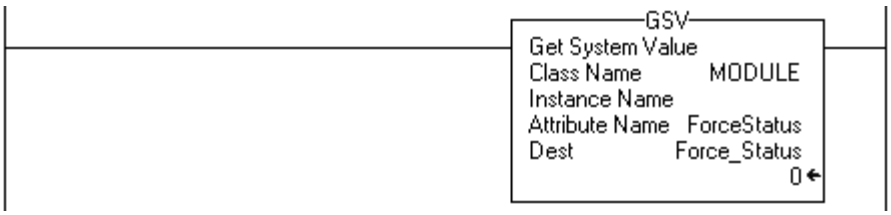
IMPORTANT The FORCE LED shows only the status of I/O forces. It *does not* show that status of SFC forces.

If the FORCE LED is:	Then:
off	• No tags contain force values. • I/O forces are inactive (disabled).
flashing	• At least one tag contains a force value. • I/O forces are inactive (disabled).
solid	• I/O forces are active (enabled). • Force values may or may not exist.

GSV Instruction

IMPORTANT The ForceStatus attribute shows only the status of I/O forces. It *does not* show the status of SFC forces.

The following example shows how to use a GSV instruction to get the status of forces.



where:

Force_Status is a DINT tag.

To determine if:	Examine this bit:	For this value:
forces are installed	0	1
no forces are installed	0	0
forces are enabled	1	1
forces are disabled	1	0

What to Force

You can force the following elements of your project:

If you want to:	Then:
override an input value, output value, produced tag, or consumed tag	Add an I/O Force
override the conditions of a transition one time to go from an active step to the next step	Step Through a Transition or a Force of a Path
override one time the force of a simultaneous path and execute the steps of the path	
override the conditions of a transition in a sequential function chart	Add an SFC Force
execute some but not all the paths of a simultaneous branch of a sequential function chart	

When to Use an I/O Force

Use an I/O force to accomplish the following:

- override an input value from another controller (i.e., a consumed tag)
- override an input value from an input device
- override your logic and specify an output value for another controller (i.e., a produced tag)
- override your logic and specify the state of an output device

IMPORTANT Forcing increases logic execution time. The more values you force, the longer it takes to execute the logic.

IMPORTANT I/O forces are held by the controller and not by the programming workstation. Forces remain even if the programming workstation is disconnected.

When you force an I/O value:

- You can force all I/O data, except for configuration data.
- If the tag is an array or structure, such as an I/O tag, force a BOOL, SINT, INT, DINT, or REAL element or member.
- If the data value is a SINT, INT, or DINT, you can force the entire value or you can force individual bits within the value. Individual bits can have a force status of:
  - no force
  - force on
  - force off

- You can also force an alias to an I/O structure member, produced tag, or consumed tag.
  - An alias tag shares the same data value as its base tag, so forcing an alias tag also forces the associated base tag.
  - Removing a force from an alias tag removes the force from the associated base tag.

Force an Input Value

Forcing an input or consumed tag:

- overrides the value regardless of the value of the physical device or produced tag
- does not affect the value received by other controllers monitoring that input or produced tag

Force an Output Value

Forcing an output or produced tag overrides the logic for the physical device or other controller (s). Other controllers monitoring that output module in a listen-only capacity will also see the forced value.

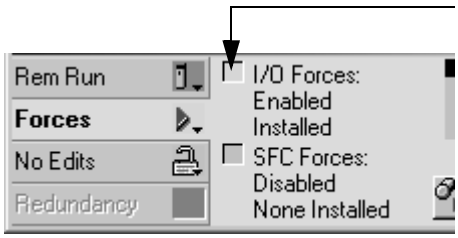
Add an I/O Force

To override an input value, output value, produced tag, or consumed tag, use an I/O force:

ATTENTION

Forcing can cause unexpected machine motion that could injure personnel. Before you use a force, determine how the force will effect your machine or process and keep personnel away from the machine area.

- Enabling I/O forces causes input, output, produced, or consumed values to change.
- If forces are enabled and you install a force, the new force immediately takes effect.



1. What is the state of the I/O Forces indicator?

If:	Then note the following:
off	No I/O forces currently exist.
flashing	No I/O forces are active. But at least one force already exists in your project. When you enable I/O forces, *all* existing I/O forces will also take effect.
solid	I/O forces are enabled (active). When you install (add) a force, it immediately takes effect.

2. Open the routine that contains the tag that you want to force.

3. Right-click the tag and choose *Monitor...* If necessary, expand the tag to show the value that you want to force (e.g., BOOL value of a DINT tag).

4. Install the force value:

To force a:	Do this:
BOOL value	Right-click the tag and choose *Force ON* or *Force OFF*.
non-BOOL value	In the *Force Mask* column for the tag, type the value to which you want to force the tag. Then press the *Enter* key.

5. Are I/O forces enabled? (See step 1.)

If:	Then:
no	From the *Logic* menu, choose *I/O Forcing* ⇒ *Enable All I/O Forces*. Then choose *Yes* to confirm.
yes	Stop.

When to Use Step Through

To override a false transition one time and go from an active step to the next step, use the *Step Through* option. With the *Step Through* option:

- You *do not* have to add, enable, disable, or remove forces.
- The next time the SFC reaches the transition, it executes according to the conditions of the transition.

This option also lets you override one time the false force of a simultaneous path. When you step through the force, the SFC executes the steps of the path.

Step Through a Transition or a Force of a Path

To step through the transition of an active step or a force of a simultaneous path:

1. Open the SFC routine.
2. Right-click the transition or the path that is forced and choose *Step Through*.

When to Use an SFC Force

To override the logic of an SFC, you have the following options:

If you want to:	Then:
override the conditions of a transition each time the SFC reaches the transition	Force a Transition
prevent the execution of one or more paths of a simultaneous branch	Force a Simultaneous Path

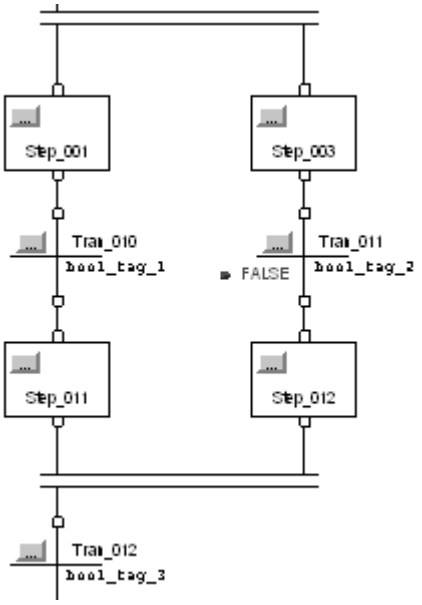
Force a Transition

To override the conditions of a transition through repeated executions of an SFC, force the transition. The force remains until you remove it or disable forces

If you want to:	Then:
prevent the SFC from going to the next step	force the transition false
cause the SFC go to the next step regardless of transition conditions	force the transition true

If you force a transition within a simultaneous branch to be false, the SFC stays in the simultaneous branch as long as the force is active (installed and enabled).

- To leave a simultaneous branch, the last step of each path must execute at least one time and the transition below the branch must be true.
- Forcing a transition false prevents the SFC from reaching the last step of a path.
- When you remove or disable the force, the SFC can execute the rest of the steps in the path.



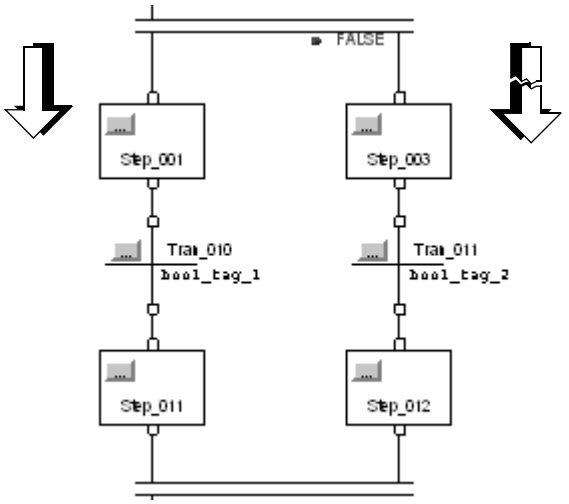
For example, to exit this branch, the SFC must be able to:

- execute *Step_011* at least once
- get past *Tran_011* and execute *Step_012* at least once
- determine that *Tran_012* is

Force a Simultaneous Path

To prevent the execution of a path of a simultaneous branch, force the path false. When the SFC reaches the branch, it executes only the un-forced paths.

This path executes.



This path *does not* execute.

If you force a path of a simultaneous branch to be false, the SFC stays in the simultaneous branch as long as the force is active (installed and enabled).

- To leave a simultaneous branch, the last step of each path must execute at least one time and the transition below the branch must be true.
- Forcing a path false prevents the SFC from entering a path and executing its steps.
- When you remove or disable the force, the SFC can execute the steps in the path.

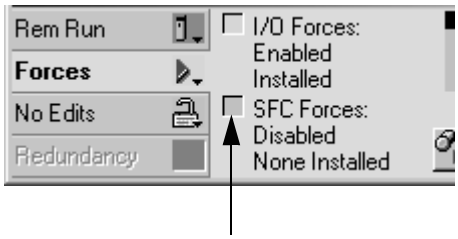
Add an SFC Force

To override the logic of an SFC, use an SFC force:

ATTENTION

Forcing can cause unexpected machine motion that could injure personnel. Before you use a force, determine how the force will effect your machine or process and keep personnel away from the machine area.

- Enabling SFC forces causes your machine or process to go to a different state or phase.
- If forces are enabled and you install a force, the new force immediately takes effect.



1. What is the state of the SFC Forces indicator?

If:	Then note the following:
off	No SFC forces currently exist.
flashing	No SFC forces are active. But at least one force already exists in your project. When you enable SFC forces, *all* existing SFC forces will also take effect.
solid	SFC forces are enabled (active). When you install (add) a force, it immediately takes effect.

2. Open the SFC routine.

3. Right-click the transition or start of a simultaneous path that you want to force, and choose either *Force TRUE* (only for a transition) or *Force FALSE*.

4. Are SFC forces enabled? (See step 1.)

If:	Then:
no	From the *Logic* menu, choose *SFC Forcing* ⇒ *Enable All SFC Forces*. Then choose *Yes* to confirm.
yes	Stop.

Remove or Disable Forces

ATTENTION

Changes to forces can cause unexpected machine motion that could injure personnel. Before you disable or remove forces, determine how the change will effect your machine or process and keep personnel away from the machine area.

If you want to:	And:	Then:
stop an individual force	leave other forces enabled and in effect	Remove an Individual Force
stop all I/O forces but leave all SFC forces active	leave the I/O forces in the project	Disable All I/O Forces
	remove the I/O forces from the project	Remove All I/O Forces
stop all SFC forces but leave all I/O forces active	leave the SFC forces in the project	Disable All SFC Forces
	remove the SFC forces from the project	Remove All SFC Forces

Remove an Individual Force

ATTENTION

If you remove an individual force, forces remain in the enabled state and any new force immediately takes effect.

Before you remove a force, determine how the change will effect your machine or process and keep personnel away from the machine area.

1. Open the routine that contains the force that you want to remove.
2. What is the language of the routine?

If:	Then:
SFC	Go to step 4.
ladder logic	Go to step 4.
function block	Go to step 3.
structured text	Go to step 3.

3. Right-click the tag that has the force and choose *Monitor…* If necessary, expand the tag to show the value that is forced (e.g., BOOL value of a DINT tag).
4. Right-click the tag or element that has the force and choose *Remove Force*.

Disable All I/O Forces

From the *Logic* menu, choose *I/O Forcing* ⇒ *Disable All I/O Forces*. Then choose *Yes* to confirm.

Remove All I/O Forces

From the *Logic* menu, choose *I/O Forcing* ⇒ *Remove All I/O Forces*. Then choose *Yes* to confirm.

Disable All SFC Forces

From the *Logic* menu, choose *SFC Forcing* ⇒ *Disable All SFC Forces*. Then choose *Yes* to confirm.

Remove All SFC Forces

From the *Logic* menu, choose *SFC Forcing* ⇒ *Remove All SFC Forces*. Then choose *Yes* to confirm.

S

T

V

W

How Are We Doing?

Your comments on our technical publications will help us serve you better in the future. Thank you for taking the time to provide us feedback.

You can complete this form and mail it back to us, visit us online at www.ab.com/manuals, or email us at RADocumentComments@ra.rockwell.com

vr

Pub. Title/Type SFC and ST Programming Languages

Cat. No.	Excerpt from the *Logix5000 Controllers Common Procedures*, publication 1756-PM001	Pub. No.	1756-PM003G-EN-E (excerpt of 1756-PM001G)	Pub. Date	March 2004	Part No.	957867-43

Please complete the sections below. Where applicable, rank the feature (1=needs improvement, 2=satisfactory, and 3=outstanding).

Overall Usefulness 1 2 3	How can we make this publication more useful for you?	
Completeness (all necessary information is provided) 1 2 3	Can we add more information to help you?	
	procedure/step	illustration feature
	example	guideline other
	explanation	definition
Technical Accuracy (all provided information is correct) 1 2 3	Can we be more accurate?	
	text	illustration
Clarity (all provided information is easy to understand) 1 2 3	How can we make things clearer?	
Other Comments	You can add additional comments on the back of this form.	

Your Name _____ Location/Phone _____

Your Title/Function _____

Would you like us to contact you regarding your comments?

___No, there is no need to contact me

___Yes, please call me

___Yes, please email me at _____

___Yes, please contact me via _____

Return this form to: Allen-Bradley Marketing Communications, 1 Allen-Bradley Dr., Mayfield Hts., OH 44124-9705

Phone: 440-646-3176 Fax: 440-646-3525 Email: RADocumentComments@ra.rockwell.com

PLEASE FASTEN HERE (DO NOT STAPLE)

Other Comments

PLEASE FOLD HERE

PLEASE REMOVE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 18235 CLEVELAND OH

POSTAGE WILL BE PAID BY THE ADDRESSEE

Allen-Bradley
RELIANCE ELECTRIC
DODGE
ROCKWELL SOFTWARE
Rockwell Automation

1 ALLEN-BRADLEY DR
MAYFIELD HEIGHTS OH 44124-9705

ASCII Character Codes

Character	Dec	Hex	Character	Dec	Hex	Character	Dec	Hex	Character	Dec	Hex
[ctrl-@] NUL	0	$00	SPACE	32	$20	@	64	$40	‘	96	$60
[ctrl-A] SOH	1	$01	!	33	$21	A	65	$41	a	97	$61
[ctrl-B] STX	2	$02	“	34	$22	B	66	$42	b	98	$62
[ctrl-C] ETX	3	$03	#	35	$23	C	67	$43	c	99	$63
[ctrl-D] EOT	4	$04	$	36	$24	D	68	$44	d	100	$64
[ctrl-E] ENQ	5	$05	%	37	$25	E	69	$45	e	101	$65
[ctrl-F] ACK	6	$06	&	38	$26	F	70	$46	f	102	$66
[ctrl-G] BEL	7	$07	‘	39	$27	G	71	$47	g	103	$67
[ctrl-H] BS	8	$08	(	40	$28	H	72	$48	h	104	$68
[ctrl-I] HT	9	$09	)	41	$29	I	73	$49	i	105	$69
[ctrl-J] LF	10	$I ($0A)	*	42	$2A	J	74	$4A	j	106	$6A
[ctrl-K] VT	11	$0B	+	43	$2B	K	75	$4B	k	107	$6B
[ctrl-L] FF	12	$0C	,	44	$2C	L	76	$4C	l	108	$6C
[ctrl-M] CR	13	$r ($0D)	-	45	$2D	M	77	$4D	m	109	$6D
[ctrl-N] SO	14	$0E	.	46	$2E	N	78	$4E	n	110	$6E
[ctrl-O] SI	15	$0F	/	47	$2F	O	79	$4F	o	111	$6F
[ctrl-P] DLE	16	$10	0	48	$30	P	80	$50	p	112	$70
[ctrl-Q] DC1	17	$11	1	49	$31	Q	81	$51	q	113	$71
[ctrl-R] DC2	18	$12	2	50	$32	R	82	$52	r	114	$72
[ctrl-S] DC3	19	$13	3	51	$33	S	83	$53	s	115	$73
[ctrl-T] DC4	20	$14	4	52	$34	T	84	$54	t	116	$74
[ctrl-U] NAK	21	$15	5	53	$35	U	85	$55	u	117	$75
[ctrl-V] SYN	22	$16	6	54	$36	V	86	$56	v	118	$76
[ctrl-W] ETB	23	$17	7	55	$37	W	87	$57	w	119	$77
[ctrl-X] CAN	24	$18	8	56	$38	X	88	$58	x	120	$78
[ctrl-Y] EM	25	$19	9	57	$39	Y	89	$59	y	121	$79
[ctrl-Z] SUB	26	$1A	:	58	$3A	Z	90	$5A	z	122	$7A
ctrl-[ESC	27	$1B	;	59	$3B	[	91	$5B	{	123	$7B
[ctrl-\] FS	28	$1C	<	60	$3C	\	92	$5C	|	124	$7C
ctrl-] GS	29	$1D	=	61	$3D	]	93	$5D	}	125	$7D
[ctrl-^] RS	30	$1E	>	62	$3E	^	94	$5E	~	126	$7E
[ctrl-_] US	31	$1F	?	63	$3F	_	95	$5F	DEL	127	$7F

Rockwell Automation Support

Rockwell Automation provides technical information on the web to assist you in using our products. At http://support.rockwellautomation.com, you can find technical manuals, a knowledge base of FAQs, technical and application notes, sample code and links to software service packs, and a MySupport feature that you can customize to make the best use of these tools.

For an additional level of technical phone support for installation, configuration and troubleshooting, we offer TechConnect Support programs. For more information, contact your local distributor or Rockwell Automation representative, or visit http://support.rockwellautomation.com.

Installation Assistance

If you experience a problem with a hardware module within the first 24 hours of installation, please review the information that's contained in this manual. You can also contact a special Customer Support number for initial help in getting your module up and running:

United States	1.440.646.3223 Monday – Friday, 8am – 5pm EST
Outside United States	Please contact your local Rockwell Automation representative for any technical support issues.

New Product Satisfaction Return

Rockwell tests all of our products to ensure that they are fully operational when shipped from the manufacturing facility. However, if your product is not functioning and needs to be returned:

United States	Contact your distributor. You must provide a Customer Support case number (see phone number above to obtain one) to your distributor in order to complete the return process.
Outside United States	Please contact your local Rockwell Automation representative for return procedure.

www.rockwellautomation.com

Corporate Headquarters
Rockwell Automation, 777 East Wisconsin Avenue, Suite 1400, Milwaukee, WI, 53202-5302 USA, Tel: (1) 414.212.5200, Fax: (1) 414.212.5201

Headquarters for Allen-Bradley Products, Rockwell Software Products and Global Manufacturing Solutions
Americas: Rockwell Automation, 1201 South Second Street, Milwaukee, WI 53204-2496 USA, Tel: (1) 414.382.2000, Fax: (1) 414.382.4444
Europe: Rockwell Automation SA/NV, Vorstlaan/Boulevard du Souverain 36-BP 3A/B, 1170 Brussels, Belgium, Tel: (32) 2 663 0600, Fax: (32) 2 663 0640
Asia Pacific: Rockwell Automation, 27/F Citicorp Centre, 18 Whitfield Road, Causeway Bay, Hong Kong, Tel: (852) 2887 4788, Fax: (852) 2508 1846

Headquarters for Dodge and Reliance Electric Products
Americas: Rockwell Automation, 6040 Ponders Court, Greenville, SC 29615-4617 USA, Tel: (1) 864.297.4800, Fax: (1) 864.281.2433
Europe: Rockwell Automation, Brühlstraße 22, D-74834 Elztal-Dallau, Germany, Tel: (49) 6261 9410, Fax: (49) 6261 17741
Asia Pacific: Rockwell Automation, 55 Newton Road, #11-01/02 Revenue House, Singapore 307987, Tel: (65) 351 6723, Fax: (65) 355 1733

Publication 1756-PM003G-EN-E (excerpt of 1756-PM001G) - March 2004 PN 957867-43
Supersedes Publication 1756-PM003F-EN-E - June 2003