Logix5000 Controllers Add-On Instructions

Programming Manual
Catalog Numbers 1756 ControlLogix, 1768 CompactLogix, 1769 CompactLogix, 1789 SoftLogix, 1794 FlexLogix, PowerFlex 700S with DriveLogix
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://literature.rockwellautomation.com) 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, when necessary, we use notes to make you aware of safety considerations.

<table>
<thead>
<tr>
<th>WARNING</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPORTANT</td>
<td>Identifies information that is critical for successful application and understanding of the product.</td>
</tr>
<tr>
<td>ATTENTION</td>
<td>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, and recognize the consequence</td>
</tr>
<tr>
<td>SHOCK HAZARD</td>
<td>Labels may be on or inside the equipment, for example, a drive or motor, to alert people that dangerous voltage may be present.</td>
</tr>
<tr>
<td>BURN HAZARD</td>
<td>Labels may be on or inside the equipment, for example, a drive or motor, to alert people that surfaces may reach dangerous temperatures.</td>
</tr>
</tbody>
</table>

Allen-Bradley, Rockwell Automation, and TechConnect are trademarks of Rockwell Automation, Inc.

Trademarks not belonging to Rockwell Automation are property of their respective companies.
# Table of Contents

## Preface
- Purpose of this Manual ..................................................... 5
- How to Use this Manual ..................................................... 5

## Designing Add-On Instructions
- Introduction ................................................................. 7
- What are the Parts of an Add-On Instruction Definition? ........ 8
  - General Information .................................................... 8
  - Parameters ............................................................... 9
  - Local Tags ............................................................... 9
  - Data Type ............................................................... 9
  - Logic Routine ........................................................... 10
  - Optional Scan Mode Routines ....................................... 11
  - Change History ......................................................... 11
  - Help ............................................................................. 11
- Deciding to Develop an Add-On Instruction. ......................... 12
  - Instruction Functionality .......................................... 12
  - Encapsulation ............................................................ 12
  - Available Languages ................................................... 13
  - Instruction Size ........................................................ 13
  - Runtime Editing of Functionality ................................... 13
  - Nesting Add-On Instructions ....................................... 13
  - Using a Routine Versus an Add-On Instruction ............... 14
  - Programmatic Access to Data ....................................... 15
  - Unavailable Instructions within Add-On Instructions ....... 16
- Planning the Design of an Add-On Instruction ....................... 18
  - Developing an Add-On Instruction ................................ 18
- Creating an Add-On Instruction ......................................... 20
  - Considerations when Selecting a Language for an Add-On Instruction ......................................................... 21
  - Transitional Instructions ............................................. 21
  - Create Parameters ...................................................... 22
    - Motor Starter Parameters Tab ................................ 24
    - Deadtime and Lag Simulation Parameters Tab .......... 25
  - Considerations When Passing Arguments to Parameters by Reference or by Value ........................................... 25
  - Considerations for Selecting a Data Type for a Parameter ... 26
  - Consideration when Using Single Dimension Array as InOut Parameter ......................................................... 26
- Determining Which Parameters to Make Visible or Required .... 26
  - Simulation Instruction in Function Block ....................... 27
  - Simulation Instruction in Ladder ................................... 27
Table of Contents

Create Local Tags ......................................................... 28
Motor Starter Local Tags ............................................. 30
Deadtime and Lag Simulation Local Tags Tab ................... 30
Enter the Logic of the Add-On Instruction .................... 31
Add-On Instructions Execution Considerations .............. 31
Optimizing Performance Considerations ..................... 31
Defining the Operation of an Instruction in Different Scan Modes .......................... 32
Enabling Scan Modes .................................................... 33
Prescan Routine .......................................................... 33
Postscan Routine .......................................................... 35
EnableInFalse Routine .................................................... 37
Using the EnableIn and EnableOut Parameters ............. 38
EnableIn Parameter and Ladder Diagram ...................... 39
EnableIn Parameter and Function Blocks ..................... 39
EnableIn Parameter and Structured Text ..................... 39
Testing the Add-On Instruction ....................................... 40
Before You Test .......................................................... 40
Test the Flow ............................................................ 40
Monitor Logic with Data Context Views ...................... 40
Tips for Validating the Add-On Instruction in Different Scan Modes .......................... 42
Storing Your Instructions ............................................... 42
Do You Want to Store Many Add-On Instructions That Share a Set of Common Add-On Instructions or User-Defined Types in a Common Location? .................. 43
Do You Want to Distribute an Add-On Instruction Definition as One File or to Manage Each Add-On Instruction as a Stand-alone Definition? .......................... 44
Updating an Add-On Instruction to a Newer Revision ........ 45
Defining Source Protection for an Add-On Instruction ........ 48
Applying Source Protection ............................................ 48
Enable the Source Protection Feature ......................... 49
Apply Source Protection .............................................. 49
Observe Source Protection .......................................... 52
### Table of Contents

- Creating Instruction Help .................................................. 53
- Define the Documentation for Add-On Instructions .......... 53
- Write Clear Descriptions .................................................. 53
- Document an Add-On Instruction ...................................... 55
- Language Switching .......................................................... 57
- Motor Starter Instruction Example ................................. 58
  - Motor Starter Ladder Diagram ................................. 60
  - Motor Starter Function Block Diagram ............. 60
  - Motor Starter Structured Text ............................. 60
- Simulation Instruction Example ...................................... 61
  - Ladder Diagram Configuration .................................. 62
  - Function Block Diagram Configuration .............. 63
  - Structured Text Configuration .............................. 63

### Chapter 2

#### Using an Add-On Instruction

- Introduction ................................................................. 65
- Programming Tips when Using an Add-On Instruction .......... 69
- Programmatically Accessing a Parameter ......................... 70
  - Using the Jog Command in Ladder Diagram .......... 71
  - Using the Jog Command In Function Block Diagram .... 71
  - Using the Jog Command in Structured Text .......... 73
- Monitoring the Value of a Parameter .......................... 73
- View Logic and Monitor with Data Context ................. 74
  - Is the Add-On Instruction Source Protected? .............. 76
- Copying an Add-On Instruction Definition .................... 77
- Importing an Add-On Instruction Definition .................. 78
  - Exporting an Add-On Instruction Definition .......... 78
Preface

Purpose of this Manual

This manual shows how to design, configure, and program Add-On Instructions. This manual is one of a set of related manuals that show common procedures for programming and operating Logix5000 controllers. For a complete list of common procedures manuals, see the Logix 5000 Controllers Common Procedures Programming Manual, publication 1756-PM001.

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

- CompactLogix controllers
- ControlLogix controllers
- DriveLogix controllers
- FlexLogix controllers
- SoftLogix5800 controllers

How to Use this Manual

Some text is formatted differently from the rest of the text.

<table>
<thead>
<tr>
<th>Text that is</th>
<th>Identifies</th>
<th>For example</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Italic</em></td>
<td>the actual name of an item that you see on your screen or in an example</td>
<td>Right-click <em>User-Defined</em> …</td>
<td>Right-click the item that is named User-Defined.</td>
</tr>
<tr>
<td><em>courier</em></td>
<td>information that you must supply based on your application (a variable)</td>
<td>Right-click <em>name_of_program</em> …</td>
<td>You must identify the specific program in your application. Typically, it is a name or variable that you have defined.</td>
</tr>
<tr>
<td>enclosed in brackets</td>
<td>a keyboard key</td>
<td>Press [Enter].</td>
<td>Press the Enter key.</td>
</tr>
</tbody>
</table>
Designing Add-On Instructions

Introduction

Add-On Instructions are available beginning with RSLogix 5000 version 16 software. Add-On Instructions are custom instructions that you design and create. With Add-On Instructions, you can create new instructions for sets of commonly-used logic, provide a common interface to this logic, and provide documentation for the instruction.

Add-On Instructions are intended to be used to encapsulate commonly used functions or device control. They are not intended to be a high-level hierarchical design tool. Programs with routines are better suited to contain code for the area or unit levels of your application.

These are some benefits to using an Add-On Instruction.

- Reuse code
  - You can use Add-On Instructions to promote consistency between projects by reusing commonly-used control algorithms.
  - If you have an algorithm that will be used multiple times in the same project or across multiple projects, it may make sense to incorporate that code inside an Add-On Instruction to make it modular and easier to reuse.

- Provide an easier to understand interface
  - You can place complicated algorithms inside of an Add-On Instruction, and then provide an easier to understand interface by making only essential Parameters visible or required.
  - You can reduce documentation development time through automatically generating instruction help.

- Protection of intellectual property
  - You can place your proprietary code inside of an Add-On Instruction, then use Source Protection to prevent others from viewing or changing your code.

- Simplify maintenance
  - You can simplify code maintenance since Add-On Instruction logic animates for a single instance.

An Add-On-Instruction can be used across multiple projects. You can define the instructions, or the instructions can be provided to you by someone else, or they can be copied from another project.
Once defined in a project, they behave similarly to the built-in instructions already available in the RSLogix 5000 software. They appear on the instruction toolbar and in the instruction browser for easy access, just like built-in RSLogix 5000 software instructions.

Add-On Instructions are made up of the following parts.

**General Information**

The General tab contains the information from when you first created the instruction. You can use this tab to update the information. The description, revision, revision note, and vendor information is copied into the custom help for the instruction. The revision is not automatically managed by the software. You are responsible in defining how it is used and when it is updated.
Parameters

The Parameters define the instruction interface; how the instruction appears when used. The Parameter order defines the order that the Parameters appear on the instruction call.

Local Tags

Local Tags are hidden members and are not visible outside the instruction and cannot be referenced by other programs or routines. They are private to the instruction.

Data Type

Parameters and Local Tags are used to define the data type that is used when executing the instruction. The software builds the associated data type. The software orders the members of the data type that correspond to the Parameters in the order that the Parameters are defined. Local Tags are added as hidden members.
Logic Routine

The Logic routine of the Add-On Instruction defines the primary functionality of the instruction. It is the code that executes whenever the instruction is called. Shown below is the interface of an Add-On Instruction and its primary Logic routine that defines what the instruction does.
Optional Scan Mode Routines

You can define additional routines for Scan mode behavior.

Change History

The Change History tab displays the creation and latest edit information that is tracked by the software. The By fields show who made the change based on the Windows user name at the time of the change.

Help

The Name, Revision, Description, and Parameter definitions are used to automatically build the Instruction help. Use the Extended Description Text to provide additional Help documentation for the Add-On Instruction. The Instruction Help Preview shows how your instruction will appear in the various languages, based on Parameters defined as Required or Visible.
Deciding to Develop an Add-On Instruction

When deciding whether to develop an Add-On Instruction, consider the following aspects.

Instruction Functionality

Complex instructions tend to be highly application specific and not reusable, or require extensive configuration support code. As with the built-in instructions, Add-On Instructions need to do one thing well and support modular coding. Consider how the instruction will be used and manage interface complexity for the end user or application.

Add-On Instructions are best at providing a specific type of functionality or device control.

Encapsulation

Add-On Instructions are designed to fully encapsulate the code and data associated with the instruction. The logic inside an Add-On Instruction only uses the Parameters and Local Tags defined by the instruction definition. There is no direct programmatic access to controller or program scope tags. This lets the Add-On Instruction be a standalone component that can execute in any application that calls it by using the Parameters interface. It can be validated once and then locked to prevent edits.
Available Languages

Add-On Instruction routines have a choice of three RSLogix 5000 software programming languages:

- Ladder Diagram
- Function Block Diagram
- Structured Text

Once created, the instruction can be called from any of the RSLogix 5000 languages. These languages are available in RSLogix 5000 software, depending on what package edition you purchase.

An Add-On Instruction written in one language can be used as an instruction through a call in another language.

Instruction Size

Add-On Instructions have one primary Logic routine that defines the behavior of the instruction when executed. This Logic routine is like any other routine in the project and has no additional restrictions in length. The total number of Input Parameters plus Output Parameters plus Local Tags can be up to 512. There are no limits on the number of InOut Parameters. The maximum data instance supported (which includes Inputs, Outputs, and Local Tags) is two megabytes. The data type size displays on the bottom of the Parameter and Local Tabs in the Add-On Instruction Definition.

Runtime Editing of Functionality

Add-On Instructions definitions can only be edited offline. If the intended functionality may need to be changed in a running controller, consider carefully if an Add-On Instruction is suitable.

Nesting Add-On Instructions

Add-On Instructions can call other Add-On Instructions in their routines. This provides the ability to design more modular code by creating simpler instructions that can be used to build more complex functionality by nesting instructions. The instructions can be nested to seven levels deep.

Add-On Instructions cannot call other routines via a JSR. You must use a nested instruction if you need complex functionality consisting of multiple routines.
Using a Routine Versus an Add-On Instruction

You can write your code in three basic ways: to run in-line as a main routine, using subroutine calls, or as Add-On Instructions. The following table summarizes how each of these ways has its advantages and disadvantages.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Main Routine</th>
<th>Subroutine</th>
<th>Add-On Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessibility</td>
<td>N/A</td>
<td>Within program (multiple copies, one for each program)</td>
<td>Anywhere in controller (single copy for the entire project)</td>
</tr>
<tr>
<td>Parameters</td>
<td>N/A</td>
<td>Pass by value</td>
<td>Pass by value via Input and Output Parameter or by reference via InOut Parameter</td>
</tr>
<tr>
<td>Numeric Parameters</td>
<td>N/A</td>
<td>No conversion, user must manage</td>
<td>Automatic data type conversion for Input and Output Parameters</td>
</tr>
<tr>
<td>Parameters data types</td>
<td>N/A</td>
<td>Atomic, arrays, structures</td>
<td>Atomic for any Parameter Arrays and structures must be an InOut Parameter</td>
</tr>
<tr>
<td>Parameter checking</td>
<td>N/A</td>
<td>None, user must manage</td>
<td>Verification checks that correct type of argument has been provided for a Parameter</td>
</tr>
<tr>
<td>Data encapsulation</td>
<td>N/A</td>
<td>All data at program or controller scope (accessible to anything)</td>
<td>Local data is isolated (only accessible within instruction)</td>
</tr>
<tr>
<td>Monitor/debug</td>
<td>In-line code with its data</td>
<td>Mixed data from multiple calls, which complicates debugging</td>
<td>Single calling instance data, which simplifies debugging</td>
</tr>
<tr>
<td>Supported programming languages</td>
<td>FBD, LD, SFC, ST</td>
<td>FBD, LD, SFC, ST</td>
<td>FBD, LD, ST</td>
</tr>
<tr>
<td>Callable from</td>
<td>N/A</td>
<td>FBD, LD, SFC, ST</td>
<td>FBD, LD, SFC via ST, ST</td>
</tr>
<tr>
<td>Protection</td>
<td>Locked and View Only</td>
<td>Locked and View Only</td>
<td>Locked and View Only</td>
</tr>
<tr>
<td>Documentation</td>
<td>Routine, rung, textbox, line</td>
<td>Routine, rung, textbox, line</td>
<td>Instruction, revision info, vendor, rung, textbox, line, extended help</td>
</tr>
<tr>
<td>Execution performance</td>
<td>Fastest</td>
<td>JSR/SBR/RTN instructions add overhead All data is copied Indexed reference impact</td>
<td>Call is more efficient InOut Parameters are passed by reference, which is faster than copying data for many types Parameter references are automatically offset from passed in instruction tag location</td>
</tr>
<tr>
<td>Memory use</td>
<td>Most used</td>
<td>Very compact</td>
<td>Compact call requires more memory than a subroutine call All references need an additional word</td>
</tr>
<tr>
<td>Edit</td>
<td>Online/offline</td>
<td>Online/offline</td>
<td>Offline only</td>
</tr>
<tr>
<td>Import/export</td>
<td>Only LD rungs with referenced tags / UDTs</td>
<td>Only LD rungs with referenced tags / UDTs</td>
<td>Full instruction definition including routines and tags to L5X</td>
</tr>
</tbody>
</table>
Programmatic Access to Data

Input and Output Parameters and Local Tags are used to define an instruction-defined data type. Each Parameter or local tag has a member in the data type, although local tag members are hidden from external use. Each call to an Add-On Instruction utilizes a tag of this data type to provide the data instance for the instruction's execution.

The Parameters of an Add-On Instruction are directly accessible in the controller's programming via this instruction-defined tag within the normal tag scoping rules.

The Local Tags are not accessible programmatically through this tag. This has impact on the usage of the Add-On Instruction. If a structured (including UDTs), array, or nested Add-On Instruction type is used as a Local Tag (not InOut Parameters), then they are not programmatically available outside the Add-On Instruction definition.
Unavailable Instructions within Add-On Instructions

Most built-in instructions can be used within Add-On Instructions. The following instructions cannot be used.

<table>
<thead>
<tr>
<th>Unavailable Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRK</td>
<td>Break</td>
</tr>
<tr>
<td>EOT</td>
<td>End of Transition</td>
</tr>
<tr>
<td>EVENT</td>
<td>Event Task Trigger</td>
</tr>
<tr>
<td>FOR</td>
<td>For (For/Next Loop)</td>
</tr>
<tr>
<td>IOT</td>
<td>Immediate Output</td>
</tr>
<tr>
<td>JSR</td>
<td>Jump to Subroutine</td>
</tr>
<tr>
<td>JXR</td>
<td>Jump to External Routine</td>
</tr>
<tr>
<td>MAOC</td>
<td>Motion Arm Output Cam</td>
</tr>
<tr>
<td>PATT</td>
<td>Attach to Equipment Phase</td>
</tr>
<tr>
<td>PCLF</td>
<td>Equipment Phase Clear Failure</td>
</tr>
<tr>
<td>PCMD</td>
<td>Equipment Phase Command</td>
</tr>
<tr>
<td>PDET</td>
<td>Detach from Equipment Phase</td>
</tr>
<tr>
<td>POVR</td>
<td>Equipment Phase Override Command</td>
</tr>
<tr>
<td>RET</td>
<td>Return</td>
</tr>
<tr>
<td>Safety Instructions</td>
<td>Any of the Safety Instructions</td>
</tr>
<tr>
<td>SBR</td>
<td>Subroutine</td>
</tr>
<tr>
<td>SFP</td>
<td>SFC Pause</td>
</tr>
<tr>
<td>SFR</td>
<td>SFC Reset</td>
</tr>
</tbody>
</table>

In addition, the following instructions may be used in an Add-On Instruction, but the data instances must be passed as an InOut Parameter.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALMA</td>
<td>Analog alarm</td>
</tr>
<tr>
<td>ALMD</td>
<td>Digital alarm</td>
</tr>
<tr>
<td>Motion Instructions</td>
<td>All Motion Instructions</td>
</tr>
<tr>
<td>MSG</td>
<td>Message</td>
</tr>
</tbody>
</table>
GSV/SSV Instruction Class Limitations

When using GSV and SSV instructions inside an Add-On Instruction, the supported classes are limited to the following.

- Controller
- Controller Device
- CST
- DFI
- Fault Log
- Program(1)
- Routine(1)
- Redundancy
- Safety
- Serial Port
- Task(1)
- Wall Clock Time

(1) The classes that represent programming components - Task, Program, Routine - only support THIS for the instance.
Planning the Design of an Add-On Instruction

Take time to plan your instruction design. Advance planning can identify problems that you can avoid later. When defining the requirements of an instruction, you are also determining the interface. Keep the following aspects in mind when defining your instruction requirements and design.

Developing an Add-On Instruction

Now that you have decided to develop an Add-On Instruction, next you need to plan the design and then create your new instruction.

Intended Behavior

· What is the purpose for creating the Add-On Instruction?
· What problem is it expected to solve?
· How is it intended to function?

Parameters

· What data needs to be passed to the instruction? What needs to be accessible outside of the instruction? How will the Parameters display?
· Determine which Parameters to make required or just visible. The order of the Parameters defines the appearance of instruction.

Naming Conventions

· What naming conventions are you going to use?
· The instruction name will be used as the mnemonic for your instruction. Although the name can be up to 40 characters long, you will typically want to use shorter, more manageable names.

Source Protection

· What type of source protection needs to be defined, if any?
· Source protection can be used to provide read only access of the Add-On Instruction definition or to completely lock the definition's routines and Local Tags to hide the implementation.

Nesting - Reuse Instructions

· Are there other Add-On Instructions that you can reuse?
· Do you need to design your instructions to share common code?
Local Tags

- What data is needed for your logic to execute but is not public?
- Identify Local Tags you might use in your instruction. Local Tags are useful for items such as intermediate calculation values that you do not want to expose to users of your instruction.

Languages

- What language do you want to use to program your instruction?
- Determine which RSLogix 5000 software programming language to use based on the use and type of application. The primary logic of your instruction will consist of a single routine of code.
- If execution time and memory usage are critical factors, refer to the Logix5000 Controllers Execution Time and Memory Use Reference Manual, publication 1756-RM087.

Scan Mode Routines

- Do you need to provide Scan mode routines?
- You can optionally define the scan behavior of the instruction in different Scan modes. This lets you define unique initialization behaviors on controller startup (Program -> Run), SFC step postscan, or EnableIn False condition.
- Decide on what language any Scan mode routines will be written.

Test

- How will you test the operation of your Add-On Instruction before commissioning it?
- What possible unexpected inputs could the instruction receive, and how will the instruction handle these cases?

Help Documentation

- What information needs to be in the instruction help?
- When you are creating an instruction you have the opportunity to enter information into various description fields. You will also need to develop information on how to use the instruction and how it operates.
Creating an Add-On Instruction

Follow these steps to create a new Add-On Instruction.

1. Open an existing or new project.

2. Right-click the Add-On Instructions folder in the Controller Organizer and select New Add-On Instruction.

The New Add-On Instruction dialog displays.

3. Enter a name for the new instruction.

   **TIP**

   The name can be up to 40 characters long. It must start with an alpha or underscore, and must contain only alphas, numerics, or underscores. The name must also not match the name of a built-in instruction or an existing Add-On Instruction in your project.

4. Enter a description for the new instruction, maximum 120 characters.
5. Choose a programming language for the Add-On Instruction Logic.

   **TIP** Once you pick a language for an Add-On Instruction, it can’t be changed. You must delete an instruction to change the language of the Add-On Instruction Logic routine.

6. Assign a Revision level for the instruction.

7. Write a Revision Note, if necessary.

8. Write a note about the Vendor, if necessary.

9. Click OK to create the instruction.

---

**Considerations when Selecting a Language for an Add-On Instruction**

Before you create an instruction you need to decide on what programming language to use. You need to select the language type based on the type of application you are developing. Ladder Diagram, Function Block Diagram, and Structured Text can be used for Add-On Instruction logic.

Each of the programming languages supported in RSLogix 5000 software is targeted for different types of applications and programming styles. In general, Ladder Diagram will execute simple boolean logic, timers, and counters the fastest. Function Block Diagrams and Structured Text may be more efficient if you take advantage of the more advanced process and drives instructions available in those languages.

You cannot compare execution times for the same Add-On Instruction written in different programming languages. There are fundamental differences on how the different languages execute and are compiled.

---

**Transitional Instructions**

Some instructions only execute (or retrigger) when rung-condition-in toggles from false to true. These are transitional-relay Ladder instructions. When used in an Add-On Instruction, these instructions will not detect the rung-in transition to the false state. When the EnableIn bit is false, the Add-On Instruction Logic routine no longer executes and thus the transitional instruction does not detect the transition to the false state. Extra conditional logic will be required to handle triggering of transitional instructions contained in an Add-On Instruction.
Some examples of transitional instructions include: ONS, MSG, PXRQ, SRT, some of the ASCII instructions, and some of the Motion instructions.

**TIP**

The EnableInFalse routine can be used to provide the conditioning required to retriggle transitional instructions contained in an Add-On Instruction. However, this method will not work for calls to this Add-On Instruction contained in a Structured Text routine, since EnableIn is always true for calls in Structured Text.

Create Parameters

Use the Add-On Instruction Definition Editor to create the Parameters for your instructions. Follow these steps to define the Parameters for your instruction.

1. In the Controller Organizer, right-click an Add-On Instruction and select Open Definition.

2. Click the Parameter tab and enter a name for a Parameter.

3. Define the Usage, based on the type of Parameters: Input, Output, InOut.
4. Select a data type.

Add-On Instruction Parameter usage and data type options are as follows:

- Input Parameter - passed by value into the Add-On Instruction. Must be a SINT, INT, DINT, REAL, or BOOL data type.
- Output Parameter - passed by value out of the Add-On Instruction. Must be a SINT, INT, DINT, REAL, or BOOL data type.
- InOut Parameter - passed by reference into the Add-On Instruction. It can be any data type including structures and arrays.

**TIP**

An instruction with only input Parameters, except EnableOut, is treated as an input instruction in a Ladder diagram. The EnableOut parameter is used for the rung condition.

5. Set the default values.

Default values are loaded from the Add-On Instruction definition into the tag of the Add-On Instruction data type when it is created, and anytime a new input or output parameter is added from the Instruction Definition editor to the Add-On Instruction definition.

**TIP**

One way to be sure that a value is applied to all instances of your instruction is to assign a literal value to the parameter in your code for your Add-On Instruction.

6. Set the display style.

7. Write a description, maximum 120 characters.

This description appears in the instruction’s help.

8. Determine whether the Parameter should be set to Required or Visible.

Refer to Determining Which Parameters to Make Visible or Required on page 26.

If you decide to make the Parameter required, it will also be visible.

9. Repeat for additional Parameters.

**TIP**

You can also create Parameters using the Tag Editor, New Parameter or Local Tag dialog, or by right-clicking on tag name in the logic of your routine.
The following are examples of the Parameters tab of the Add-On Instruction Definition Editor for two different instructions.

**Motor Starter Parameters Tab**

The order that you create the Parameters is how they will appear in the data type and on the instruction face. Select the Parameter row and use Move Up and Down to rearrange the order.
Considerations When Passing Arguments to Parameters by Reference or by Value

The following information will help you understand the differences between passing argument tags to Parameters by reference or by value.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>By Value (Input or Output)</th>
<th>By Reference (InOut)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Synchronous - the Parameter’s value does not change during Add-On Instruction execution.</td>
<td>Asynchronous- the Parameter’s value may change during Add-On Instruction execution. Any access by the instruction’s logic directly reads or writes the passed tag’s value.</td>
</tr>
<tr>
<td>Performance</td>
<td>Argument values are copied in and out of the Parameters of the Add-On Instruction. This takes more time to execute a call to the instruction.</td>
<td>Parameters access argument tags directly by reference, which leads to faster execution of instruction calls.</td>
</tr>
<tr>
<td>Memory usage</td>
<td>Most amount.</td>
<td>Least amount.</td>
</tr>
<tr>
<td>Parameter data types supported</td>
<td>Atomic.</td>
<td>Atomic, Arrays, and structures.</td>
</tr>
</tbody>
</table>
Considerations for Selecting a Data Type for a Parameter

The Logix5000 controllers perform DINT (32 bit) and REAL (32 bit) math operations, which causes DINT data types to execute faster than other integer data types. Data conversion rules of SINT to INT to DINT are applied automatically, and can add overhead. Whenever possible, use DINT data types for the Add-On Instruction input and output Parameters.

Consideration when Using Single Dimension Array as InOut Parameter

The InOut Parameter can be defined to be a single dimension array. When specifying the size of this array consider the following:

- The user of your instruction can pass an array tag that is the same size as your definition or,
- They can pass an array tag that is larger than your definition.

To accommodate this flexibility in your logic, you can determine the actual size of the referenced array by using the SIZE instruction.

TIP

When you monitor an array InOut Parameter inside of the Logic routine, the parameter definition is used to determine the size of the array. For example, if you define an InOut parameter to be a 10-element array of DINTs and the end user passes in an array of 100 DINTs. Then if you open the Add-On Instruction logic, select the appropriate context for that call, and monitor the array parameter, only 10 elements will be displayed.

Determining Which Parameters to Make Visible or Required

To help be sure that certain data is passed into the Add-On Instruction, you can use required Parameters. A required Parameter must be passed as an argument in order for a call to the instruction to verify. In Ladder Diagram and Structured Text, this is done by specifying an argument tag for these Parameters. In a Function Block Diagram, required input and output Parameters must be wired, and InOut Parameters must have an argument tag. If a required Parameter does not have an argument associated, as described above, then the routine containing the call to the Add-On Instruction will not verify.

For output Parameters, making a Parameter visible is useful if you don’t usually need to pass the Parameter value out to an argument, but you do want to display its value prominently for troubleshooting.

Required Parameters are always visible, and InOut Parameters are always required and visible. All Input and Output Parameters, regardless of being marked as Required or Visible, can be programmatically accessed as a member of the instruction’s tag.
Simulation Instruction in Function Block

If you want a pin that is displayed in Function Block, but wiring to it is optional, set it as Visible (non-Required) Input Parameter.

Simulation Instruction in Ladder

- If you want the Parameter's value displayed on the instruction face in Ladder, set the Parameter as Visible.
- An Output Parameter of the BOOL tag type that is not required, but visible, will show as a status flag on the right side of the block in Ladder. This can be used for status flags like DN or ER.
### Input, Output, and InOut Required and Visible Settings Effects

<table>
<thead>
<tr>
<th>Type</th>
<th>Required</th>
<th>Visible</th>
<th>Ladder Value</th>
<th>Ladder Argument</th>
<th>FBD Must Connect</th>
<th>FBD Argument</th>
<th>FBD Change Visibility</th>
<th>ST Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N/A</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Input</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N/A</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Input</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/A</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Output</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N/A</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Output</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N/A</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Output</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/A</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>InOut</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N/A</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

- If you have a Parameter for which the user must specify a tag as its source for input or its destination as output, and you don't want this to be optional, set the Parameter as Required. Any required Parameters are automatically set to Visible.

- The visible setting is always set to visible for InOut Parameters. All InOut Parameters are required.

---

#### Create Local Tags

Use the Definition Editor to create the Local Tags for your instructions. Local Tags contain data that will be used by your Add-On Instruction but which you do not want exposed to the user of your instruction. Local Tags will not appear in the data structure for an Add-On Instruction because they are hidden members. You can access local tag values from an HMI by specifying the name of the local tag as a member in an Add-On Instruction type tag. For example, aMotorStarter_Tag.CheckAuxContact.

Follow these steps to define the Local Tags for your instruction.

1. In the Controller Organizer, right-click an instruction and select Open Definition.

2. Click the Local Tags tab and enter a name for a new tag and select a data type.

You cannot use these data types for Local Tags - ALARM_ANALOG, ALARM_DIGITAL, MESSAGE, or any motion data types, for example Axis or MOTION_GROUP. To use these type of tags in your...
3. Set the default values.

Default values are loaded from the Add-On Instruction definition into the tag of the Add-On Instruction data type when it is created or any time a new tag is added in the Instruction Definition Editor to the Add-On Instruction definition.

**TIP**

Local tag values in your Add-On Instruction definition are the default values. Changing an existing local tag value will not affect any instances of your Add-On Instruction that already exist.

One way to be sure a value is applied to all instances of your instruction is to assign a literal value to the tag in your code for your Add-On Instruction.

4. Set the display style.

5. Write a description, a maximum of 120 characters.

6. Repeat for additional Local Tags.
The following are examples of the Local Tags tab of the Add-On Instruction definition.

**Motor Starter Local Tags**

![Motor Starter Local Tags](image)

**Deadtime and Lag Simulation Local Tags Tab**

![Deadtime and Lag Simulation Local Tags Tab](image)

**Motor Starter Tag Editor**

You can also add and edit Parameters and Local Tags using the tag editor as shown below.

![Motor Starter Tag Editor](image)
Enter the Logic of the Add-On Instruction

Follow these steps to enter logic into your project.

1. Expand the Add-On Instruction.

2. Expand an instruction and double-click the Logic routine to open.

3. Edit your logic using the available language editors.

Add-On Instructions Execution Considerations

An Add-On Instruction is executed just like any other routine belonging to a particular task. Because another task can preempt a task containing an Add-On Instruction being executed, it is not guaranteed that an Add-On Instruction will always be executed to its completion. You can use the User Interrupt Disable / Enable (UID/UIE) instructions to block a task switch if you want to be sure the Add-On Instruction executes to completion before switching to another task.

Optimizing Performance Considerations

The performance depends on the structuring, configuration, and the amount of code in an Add-On Instruction. You can pass large amounts of data through a structure using an InOut Parameter. The size of data referenced by an InOut Parameter does not impact scan time and there is no difference between passing a a user-defined type tag or an atomic tag since it is passed by reference.

When a rung condition is false, any calls to an Add-On Instruction are still processed even though the Logic routine is not executed. The scan time can be affected when many instances of an Add-On Instruction are executed false. Be sure to provide instructions in your documentation if an Add-On Instruction can be skipped when rung condition is false.
Defining the Operation of an Instruction in Different Scan Modes

To provide Add-On Instructions with the same flexibility as built-in instructions, optional Scan mode routines can be configured allowing you to fully define the behavior of the instruction. The Scan mode routines do not initially exist for an Add-On Instructions as they are optional and must be created by the user depending upon the requirements of the instruction.

Like all built-in instructions in the controller, Add-On Instructions support the following four controller Scan modes:

- **True**: Logic is scanned as the result of a true rung condition or the EnableIn Parameter is set True.
- **False**: Logic is scanned as the result of a false rung condition or the EnableIn Parameter is set False. Instructions in the controller may or may not have logic that executes only when that instruction is scanned false.
- **Prescan**: Occurs when the controller either powers up in Run mode or transitions from Program to Run. Instructions in the controller may or may not have logic that executes only when that instruction is executed in Prescan mode.
- **Postscan**: Occurs as a result of an Action in an SFC routine becoming inactive if SFCs are configured for Automatic Reset. Instructions in the controller may or may not have logic that executes only when that instruction is executed in Postscan mode.

The default behavior for executing an Add-On Instruction with no optional scan routines created may be sufficient for the intended operation of the instruction. If you do not define an optional Scan Mode the following default behavior of an Add-On Instruction occurs:

- **True**: Executes the main Logic routine of the Add-On Instruction.
- **False**: Does not execute any logic for the Add-On Instruction and does not write any outputs. Input Parameters are passed values from their arguments.
- **Prescan**: Executes the main Logic routine of the Add-On Instruction in Prescan mode. Any required input and output Parameters' values are passed.
- **Postscan**: Executes the main Logic routine of the Add-On Instruction in Postscan mode.

For each Scan mode, you can define a routine that is programmed specifically for that Scan mode and can be configured to execute in that mode.

- **True**: The main Logic routine for the Add-On Instruction executes (not optional).
- **False**: The EnableIn False routine executes normally in place of the main Logic when a scan false of the instruction occurs. Any required (or wired in FBD) input and output Parameters' values are passed.
- **Prescan**: The Prescan routine executes normally after a prescan execution of the main Logic routine. Any required input and output Parameters' values are passed.
Postscan - The Postscan routine executes normally after a postscan execution of the main Logic routine.

### Enabling Scan Modes

The Scan Modes tab in the Instruction Definition Editor lets you create and enable execution of the routines for the three Scan modes: Prescan, Postscan, and EnableInFalse.

### Prescan Routine

When the controller transitions from Program mode to Run mode or when the controller powers up in Run mode, all logic within the controller is executed in Prescan mode. During this scan, each instruction may initialize itself and some initialize any tags it may reference. For most instructions, Prescan mode is synonymous with scanning false. For example, an OTE instruction clears its output bit when executed during Prescan. For others, special initialization may be done, such as an ONS instruction setting its storage bit during Prescan. During Prescan mode, all instructions evaluate false so conditional logic does not execute.

The optional Prescan routine for an Add-On Instruction provides a way for an Add-On Instruction to define additional behavior for Prescan mode. When a Prescan routine is defined and enabled, the Prescan routine executes normally after the primary Logic routine executes in Prescan mode. This is useful when it is desired to initialize tag values to some known or predefined state prior to execution. For example, setting a PID instruction to Manual mode with a 0% output prior to its first execution or to initialize some coefficient values in your Add-On Instruction.

**TIP**

When an Add-On Instruction executes in Prescan mode, any required Parameters have their data passed.

- Values are passed to Input Parameters from their arguments in the instruction call.
- Values are passed out of Output Parameters to their arguments defined in the instruction call.

These values are passed even when the rung condition is false in Ladder Diagram or when the instruction call is in a false conditional statement in Structured Text. When Function Block Diagram routines execute, the data values are copied to all wired inputs and from all wired outputs, whether the Parameters are required or not.
Follow these steps to create a Prescan routine.

1. In the Controller Organizer, right-click an instruction and select Open Definition.

2. Click the Scan Modes tab.

3. Click New for Prescan Routine.

![New Scan Mode Routine dialog](image)

The New Scan Mode Routine dialog displays.

4. Select the type of programming language: Ladder Diagram, Function Block, or Structured Text.

5. Write a description of the Prescan behavior.

6. Click OK to create the routine and return to the Scan Modes tab.

7. Define if the prescan routine executes (or not) with the checkbox.

![Prescan routine settings](image)

The Prescan routine can now be edited like any other routine.
Postscan Routine

Postscan mode only occurs for logic in an SFC Action when the Action becomes inactive and the SFC language is configured for Automatic Reset (which is not the default option for SFC). When an SFC Action becomes inactive, then the logic in the Action is executed one more time in Postscan mode. This mode is similar to Prescan in that most instructions simply execute as if they have a false condition. It is possible for an instruction to have different behavior during Postscan than it has during Prescan.

When an Add-On Instruction is called by logic in an SFC Action or a call resides in a routine called by a JSR from an SFC Action, and the Automatic Reset option is set, the Add-On Instruction executes in Postscan mode. The primary Logic routine of the Add-On Instruction executes in Postscan mode. Then if it is defined and enabled, the Postscan routine for the Add-On Instruction executes. This could be useful in resetting internal states, status values, or de-energizing instruction outputs automatically when the action is finished.

Follow these steps to create a postscan routine.

1. In the Controller Organizer, right-click an instruction and select Open Definition.
2. Click the Scan Modes tab.
3. Click New for Postscan Routine.
The New Scan Mode Routine dialog displays.

4. Select the type of programming language: Ladder Diagram, Function Block, or Structured Text.

5. Write a description of the postscan behavior.

6. Click OK to create the routine and return to the Scan Modes tab.

7. Define if the postscan routine executes (or not) with the checkbox.

The Postscan routine can now be edited like any other routine.
EnableInFalse Routine

When defined and enabled for an Add-On Instruction, the EnableInFalse routine executes when the rung condition is false or if the EnableIn Parameter of the Add-On Instruction is false (0). This is useful primarily for scan false logic, when used as an output instruction in a Ladder routine. A common use of scan false is the setting of OTE’s to the de-energized state when the preceding rung conditions are false. An Add-On Instruction can use the EnableInFalse capability to let you to define behavior for the False conditions.

When the Add-On Instruction is executed in the false condition and has an EnableInFalse routine defined and enabled, any required Parameters have their data passed.

- Values are passed to Input Parameters from their arguments in the instruction call.
- Values are passed out of Output Parameters to their arguments defined in the instruction call.

If the EnableInFalse routine is not enabled, the only action performed for the Add-On Instruction in the false condition is that the values are passed to any required Input Parameters in Ladder logic.

Follow these steps to create an EnableInFalse routine.

1. Right-click the instruction and select Open Definition.

2. Click the Scan Modes tab.
3. Click New on EnableInFalse routine.

The New Scan Mode Routine dialog displays.

4. Select the Type pull-down list to select the language that you want to write the routine in Ladder, Function Block, or Structured Text.

5. Write a description of the EnableInFalse behavior.

6. Click OK to add an EnableInFalse routine to the Add-On Instruction definition.

7. Define if EnableIn False routine executes (or not) by checking Execute EnableInFalse routine.

The EnableInFalse routine can now be edited like any other routine.

### Using the EnableIn and EnableOut Parameters

The EnableIn and EnableOut Parameters that appear by default in every Add-On Instruction have behaviors that conform to the three language environments: Ladder Diagram, Function Block Diagram, and Structured Text.

In order to execute the primary Logic routine in any of the language environments, the EnableIn Parameter must be True (1). In general, the EnableIn Parameter should not be referenced by the primary Logic routine within the instruction definition. The EnableOut parameter will, by default, follow the state of the EnableIn parameter but can be overridden by user logic to force the state of this Parameter.

**TIP**

If EnableIn is false, then EnableOut cannot be made true in an EnableIn False routine.
If the EnableIn parameter of the instruction is False (0), the Logic routine is not executed and the EnableOut parameter is set False (0). If an EnableInFalse routine is included in the instruction definition and it is enabled, the EnableInFalse routine will be executed.

**EnableIn Parameter and Ladder Diagram**

In the Ladder diagram environment, the EnableIn parameter reflects the rung state on entry to the instruction. If the rung state preceding the instruction is True (1), the EnableIn parameter will be True and the primary Logic routine of the instruction will be executed. Likewise, if the rung state preceding the instruction is False (0), the EnableIn parameter will be False and the primary Logic routine will not be executed.

**TIP**
An instruction with only input Parameters, except EnableOut, is treated as an input instruction in a Ladder diagram. The EnableOut parameter is used for the rung condition.

**EnableIn Parameter and Function Blocks**

In the Function Block environment, the EnableIn Parameter can be manipulated by the user through its pin connection. If no connection is made, the EnableIn Parameter is set True (1) when the instruction begins to execute and the primary Logic routine of the instruction will be executed. If a wired connection to this Parameter is False (0), the primary Logic routine of the instruction will not execute. Another reference writing to the EnableIn Parameter, such as a Ladder Diagram rung or a Structured Text assignment, will have no influence on the state of this Parameter. Only a wired connection to this Parameter’s input pin can force it to be False (0).

**EnableIn Parameter and Structured Text**

In the Structured Text environment, the EnableIn Parameter is always set True (1) by default. The user cannot influence the state of the EnableIn Parameter in a Structured Text call to the instruction. Since EnableIn is always True (1) in Structured Text, the EnableInFalse routine will never execute for an instruction call in Structured Text.
You need to test and troubleshoot the logic of an instruction to get it working.

**TIP**
When a fault occurs in an Add-On Instruction routine, a fault log is created that contains extended information useful for troubleshooting.

**Testing the Add-On Instruction**

Before You Test

Before you start to test an Add-On Instruction, do the following.

1. Open a project to debug offline.

**TIP**
Add-On Instruction definitions can only be created or modified when offline. You can add, delete, or modify tag arguments using online editing.

2. Add the Add-On Instruction to the project, if it is not already there.

**Test the Flow**

1. Add a call to the instruction in a routine in the open project.

2. Assign any arguments to required Parameters for your call.

3. Download the project.

**Monitor Logic with Data Context Views**

You can simplify the online monitoring and troubleshooting of your Add-On Instruction by using Data Context views. The Data Context selector lets you select a specific call to the Add-On Instruction that defines the calling instance and arguments whose values are used to visualize the logic for the Add-On Instruction.

**TIP**
When troubleshooting an Add-On Instruction, use a non-arrayed instance tag for the call to the instruction. This lets you monitor and troubleshoot the instruction's Logic routine with a data context. Variable indexed arrays cannot be used to monitor the logic inside an Add-On Instruction.
1. Go into Run mode.

2. Right-click the instruction call and select Open Instruction Logic.

   The Logic routine opens with animated logic for the specific calling instance.

3. Monitor the Logic routine with data context.

   You can also show the Add-On Instruction logic animated with other calling instances by using the Data Context menu located on the upper right of the logic dialog.

   You can also monitor any defined Scan mode routines this way by opening the routine and selecting the calling instance in the Data Context.
Chapter 1  Designing Add-On Instructions

Tips for Validating the Add-On Instruction in Different Scan Modes

The most straightforward method to verify Scan mode operation is to execute the instruction first with the Scan mode routine disabled, then again with it enabled. Then you can determine whether the Scan mode routine performed as expected.

Verifying Individual Scan Modes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>This is simply the execution of the main Logic routine.</td>
</tr>
<tr>
<td>False</td>
<td>In a Ladder Logic target routine, this entails placing an XIC before an instance of the instruction and evaluating instruction results when the XIC is false.</td>
</tr>
<tr>
<td></td>
<td>In a Function Block target routine, this entails executing an instance of the instruction with the EnableIn Parameter set to zero(0).</td>
</tr>
<tr>
<td>Prescan</td>
<td>Place the controller in Program mode, then place it in Run mode.</td>
</tr>
<tr>
<td>Postscan</td>
<td>With the controller configured for SFC Automatic Reset, place an instance of the instruction into the Action of an SFC. Run the SFC such that this Action is executed and the SFC proceeds beyond the step that is associated with this Action.</td>
</tr>
</tbody>
</table>

Storing Your Instructions

There are two ways to store a group of instruction definitions together. The first way is to store your Add-On Instructions in a project file.

1. Identify what instructions you want to store.

2. Place them in a project file called something like MyInstructions.ACD.

3. Open other projects in additional instances of RSLogix5000 software and use copy and paste or drag and drop to move a copy of the instruction definitions from one project to another.

   If any of these instruction definitions reference the same Add-On Instruction or User-Defined Type, there is only one shared copy in the project file. When an Add-On Instruction is copied to another project, it also copies any definition it references to the target project.

The second way is to store your Add-On Instruction definition in an .L5X export file.
When you select to Export an Add-On Instruction, that Add-On Instruction definition is saved to the export file. Optionally, you can include any nested Add-On Instruction definitions or User-Defined Data Types that are referenced by the exported Add-On Instruction.

In deciding how to manage your Add-On Instruction definitions in export files, you need to consider your goals in storing the definitions.

**Do You Want to Store Many Add-On Instructions That Share a Set of Common Add-On Instructions or User-Defined Types in a Common Location?**

If so, you may want to choose to export each Add-On Instruction and User-Defined Types to separate files without including references by using these steps.

1. Select an Add-On Instruction in the Controller Organizer.

2. Right-click the instruction and select Export Add-On Instruction.

3. Choose the common location to store the L5X file.

4. Type a name for the file.

5. Uncheck the Include referenced Add-On Instructions and User-Defined Types checkbox.

6. Click Export.

7. Follow the above steps to individually export the other shared Add-On Instructions and User-Defined Types.
Using export in this way lets you manage the shared Add-On Instruction and User-Defined Type definitions independently of the definitions that reference them. One advantage of this is the ability to update the shared component without having to regenerate all the export files for the definitions that reference it. That is, it is only stored in one file instead of in every file whose definition references it. This helps with the maintenance of the definitions as you only have to update one export file.

When Add-On Instructions have been exported this way, to use them in a project you must import any definition that it references before the referencing one can import successfully. To do this, work bottom up by importing nested Add-On Instructions or referenced User-Defined Types, and then the Add-On Instructions that reference them.

**Do You Want to Distribute an Add-On Instruction Definition as One File or to Manage Each Add-On Instruction as a Stand-alone Definition?**

If so, you may want to export the instruction definition and any referenced Add-On Instructions or User-Defined Types into one export file by using these steps.

1. Right-click Add-On Instruction in Controller Organizer and select Export Add-On Instruction.

2. Choose the location to store the L5X file.

3. Type a name for the file.
4. Check the Include referenced Add-On Instructions and User-Defined Types checkbox.

5. Click Export.

This exports the selected Add-On Instruction definition and all the referenced definitions into the same export file. This file can be used to distribute an Add-On Instruction definition. When the exported Add-On Instruction is imported into the project, the referenced definitions are imported as well in one step.

Updating an Add-On Instruction to a Newer Revision

When you need to update an instruction, you must be working offline. If you need a newer revision of an Add-On Instruction, import from an L5X file or copy it from an existing project.

Follow these steps to update an Add-On Instruction to a newer revision.

**IMPORTANT** Be careful when updating an Add-On Instruction. You need to ensure that your existing code will work correctly with the new version of the instruction.

1. Right-click the Add-On Instruction folder and choose Import Add-On Instruction.
2. Select the file with the Add-On Instruction definition and click Import.

![Image showing Import Add-On Instruction dialog]

3. Review the Import Configuration dialog.

It shows two definitions because the Conveyor_Control references Motor_Startor and both definitions are in the import file.

![Image showing Import Configuration dialog]

The icon in the row header indicates that the instruction definition collides with one in the project.

For any instruction definition being imported that collides with one in the project, there are two options for the Add-On Instructions, Overwrite or Use Existing.

4. Select a row and click the Compare to Existing button to see the Collision dialog.

Use the Collision dialog to see the differences and where the Add-On Instruction is used.
5. Click on Compare to Existing to open the Collision dialog.

The Collision tab shows where the existing Add-On Instruction is used.

The Compare tab shows what is different between the definitions, in this case, the revision information.

**TIP**
The Compare dialog only compares metadata for each instruction definition, such as description, revision or edited date. For effective revision control, enter a detailed revision note.
6. Select Use Existing, to leave the definition in the project as is, or Select Overwrite to replace the Add-On Instruction in your project with the definition that you are importing.

7. Click OK on the Import Configuration dialog to complete the operation.

**Defining Source Protection for an Add-On Instruction**

You can apply source protection to protect your intellectual property or prevent unintended edits of validated source. Source protecting your Add-On Instructions protects your intellectual property and avoids unwanted changes.

**Applying Source Protection**

You can use the Source Protection capability to limit a user of your Add-On Instruction to read-only access or to allow no access to the internal logic or Local Tags used by the instruction.

You can protect their use and modification with a source key file when you distribute them. This lets you stop unwanted changes to your instruction as well as protects your intellectual property.
Sometimes you will want to protect the source definition of an Add-On Instruction from the view of a user. This may be due to the proprietary nature of the algorithms or for strict revision control.

- **Source Protected**

  Users without the source key cannot view any routine or Local Tags, or make any edits to the Add-On Instruction.

- **Source Protected with Viewable Option**

  Users without the source key can view all components of the Add-On Instruction including its logic and Local Tags, but are prevented from making any edits to the instruction.

### Enable the Source Protection Feature

You may find that Source Protection is unavailable and not listed in your menus. To enable the Source Protection feature, enable it during the RSLogix 5000 System Install or use the tool RS5KSsrcPtc.exe on your installation CD.

If it is the first time the Source Protection has been configured, a dialog displays asking to configure a source key file location.

1. Enter the location of (or browse to) where the source key is kept, it may or may not exist yet.

2. Click OK.

### Apply Source Protection

To source protect your Add-On Instruction follow these steps.

1. In RSLogix 5000 select Tools -> Security -> Configure Source Protection, to access the source configuration dialog.
2. Click Specify to identify the location for the sk.dat Source Key File.

3. Click OK in both dialogs to accept.

4. Click Yes to confirm creation of the sk.dat Key File.

5. Select the component you want to protect.

6. Click Protect.
Designing Add-On Instructions

Chapter 1

The Source Key Entry dialog displays. This is where you enter the individual source key for this component. These source keys follow the conventions for routine source keys.

7. Expand the Add-On Instructions folder to view the instructions available to apply source protection.

The components include all Routines and Add-On Instruction definitions in the project.

8. Select the Add-On Instruction you wish to protect and click Protect.

9. Enter a name for the source key for this component or use an existing one.

10. If you wish to allow users to be able to see the logic and Local Tags, check the box Allow viewing of component(s).

The Source Protection Configuration should now resemble the following dialog.

IMPORTANT Before you click Close, you must first either click Clear or Disable Ability to Configure Source Protection or remove the sk.dat file from the PC so that the source key is no longer present, if you wish to observe the source protection settings.
11. Click Close.

![Source Protection Configuration](image)

12. Save the project.

13. Close RSLogix 5000 software.

**Observe Source Protection**

If you want to observe how source protection works, do the following steps:

1. Rename the sk.dat file to some other name.

2. Start RSLogix 5000 software and open the project.

   There is now no valid key file available to this RSLogix 5000 project.

   **TIP** When the Source Key is available, the Add-On Instruction will behave the same as if it were not source protected.

3. Review how the instruction appears in the Add-On Instructions folder.

   Now the protected Add-On Instruction definition's routines and tag folder are not shown if fully protected, and the definition cannot be edited.
Creating Instruction Help

Custom instruction help is generated automatically as you are creating your Add-On Instructions and as you add information in the description fields. You can also provide additional help documentation by entering it on the Help tab of the Add-On Instruction Definition Editor. The instruction help is available in the instruction browser and from any call to the instruction in a language editor by pressing F1.

Custom instruction help makes it easier for users to get the help they need for using the instructions.

Define the Documentation for Add-On Instructions

RSLogix 5000 software automatically builds help for your Add-On Instructions using the instruction’s description, revision note, and Parameter descriptions. By creating meaningful descriptions, you can help the users of your instruction. In addition, you can add your own custom text to the help by using the extended description field.

Write Clear Descriptions

When writing your descriptions keep the following in mind.

- Use short sentences and simple language
- Be brief and direct when you write
- Offer simple examples
- Proof read your field entries.
This is an example of the Extended Description Text field in the Help tab of the Instruction Definitions Editor. This area lets you create directions on how to use your instruction and troubleshooting information. The Instruction Help Preview window shows how your text will look as generated instruction help.

When you are entering your text into the Extended Description Text field you can use returns and tabs in the field to format the text, and if you copy and paste text into the field tabs are preserved.
Document an Add-On Instruction

Follow these steps to create custom help for an instruction.

1. Right-click an Add-On Instruction and select Open Definition.

2. On the General tab, enter a description and a revision note for the Add-On Instruction to explain the purpose of the instruction.

3. Click the Parameters tab and enter a meaningful description for each Parameter.

4. Right-click each routine located below the Add-On Instruction in the Controller Organizer and select Properties.

5. Enter description for execution of each routine:
   a. For the Logic routine, describe execution of the instruction when EnableIn is true.
   b. For the EnableInFalse routine (if one exists), describe actions that will take place when EnableIn is false, such as any outputs that get cleared.
   c. For the Prescan routine (if one exists), briefly describe actions that will take place during Prescan, such as initialization of any Parameters.
   d. For the Postscan routine (if one exists), briefly describe actions that will take place during Postscan, such as initialization of any Parameters resetting any internal state of the instruction.

6. Click the Help tab of the Add-On Instruction definition and enter additional information in the Extended Description field.

   The extended description can include the following information.
   
   · Additional Parameter information
   · Description of how the instruction executes
   · Change history notes

7. Review the Help format in the preview window.
This is an example of the RSLogix 5000 software generated help for the instruction. This information is gathered from the definition descriptions that you complete when defining an instruction.
Language Switching

With RSLogix 5000 software, version 17, you have the option to display project documentation, such as tag descriptions and rung comments for any supported localized language. You can store project documentation for multiple languages in a single project file rather than in language-specific project files. You define all the localized languages that the project will support and set the current, default, and optional custom localized language. The software uses the default language if the current language’s content is blank for a particular component of the project. However, you can use a custom language to tailor documentation to a specific type of project file user.

Enter the localized descriptions in your RSLogix 5000 project, either when programming in that language or by using the import/export utility to translate the documentation off-line and then import it back into the project. Once you enable language switching in RSLogix 5000 software, you can dynamically switch between languages as you use the software.

Project documentation that supports multiple translations within a project includes:

- Component descriptions in tags, routines, programs, user-defined data types, and Add-On Instructions.
- Equipment phases.
- Trends.
- Controllers.
- Alarm Messages (in ALARM_ANALOG and ALARM_DIGITAL configuration).
- Tasks.
- Property descriptions for modules in the Controller Organizer.
- Rung comments, SFC text boxes, and FBD text boxes.

For more information on enabling a project to support multiple translations of project documentation, see the online help.
Motor Starter Instruction Example

The Motor_Starter Add-On Instruction starts and stops a motor.

If the stop pushbutton = off and the start pushbutton = on then

The conveyor gets the command to run.

The instruction seals in the command, so the conveyor keeps running even after you release the start pushbutton.

If the stop pushbutton = on then the conveyor stops.

The following three diagrams show the Motor Starter instruction called in different programming languages.

Motor Starter Example Definition Editor General Tab

Motor Starter Example Definition Editor Parameter Tab
Motor Starter Example Ladder Logic

- **In the Stop button is closed, the motor gets the command to turn on when you press the Start button. The motor runs even after you release the Start button. The motor stops when you press the Stop button.**

- **The motor starts if the run command is on. The motor also starts if the way in is on and there isn't a fault.**

- **If the Fault is is greater than 0, run on CheckAuxContact. This lets the instruction use the auxiliary contact of the motor.**

- **If CheckAuxContact is on, the only reason for the state of the auxiliary contact. The fault signal on the auxiliary contact doesn’t enable the instructions that it the motor fault.**

- **To clear the fault, turn on the inner run on the fault.”**
Chapter 1  Designing Add-On Instructions

Motor Starter Ladder Diagram

Motor Starter Function Block Diagram

Motor Starter Structured Text

Motor_Starter(Motor_Starter_ST, Stop_PB, Start_PB, Motor_Out_ST);
Simulation Instruction Example

The Simulation_DT_1st Add-On Instruction adds a dead time and a first-order lag to an input variable.

Simulation Example Definition Editor General Tab

Simulation Example Definition Editor Parameter Tab
In this example, the instruction simulates a closed-loop control.

**Ladder Diagram Configuration**

The Simulation_DT_1st instruction reads the control variable from the PID instruction. The PID instruction reads the SimOutput Parameter of the Simulation_DT_1st instruction.
Function Block Diagram Configuration

The PIDE instruction sends the control variable to the Simulation_DT_1st instruction. The Simulation_DT_1st instruction calculates an output and sends it to the PIDE instruction as the process variable.

Structured Text Configuration

The Simulation_DT_1st instruction reads the control variable from the PIDE instruction and calculates an output. The output goes to the process variable of the PIDE instruction.
Notes:
Using an Add-On Instruction

Introduction

Add-On Instructions are used in your routines like any built-in instructions. You add calls to your instruction and then define the arguments for any Parameters.

The Add-On Instruction can be used in any one of the Ladder Diagram, Function Block or Structured Text languages (including Structured Text within Sequential Function Chart actions). The appearance of the instruction will conform to the language in which it is placed.

The Add-On Instructions in the project can be accessed from any of the normal instruction selection tools.

The instruction toolbar has an Add-On tab that lists all of the currently available Add-On Instructions in the project.

Follow these instructions to access the Add (language) Element dialog.

1. Press Alt + Insert keys anywhere in the editor or by right-clicking on the logic in the Editor.
2. Locate the Add-On Instruction you want to add to your routine.

3. Select the Add-On Instruction and click OK.

Use [Instruction Help] to display the instruction help for any instruction in the browser.
Follow this procedure when you want to use an Add-On Instruction in one of your routines.

1. Open the Add-On Instruction folder in the Controller Organizer and view the listed instructions.

   If the instruction you want to use is not listed, you need to do one of the following:

   • Create the instruction in your project.
   • Copy and paste an instruction into your project.
   • Get the file for an exported instruction definition and then import the instruction into your current project.

2. Open the routine that will use the instruction.

3. Click the Add-On tab on the instruction toolbar.

4. Click the desired Add-On Instruction, for example Simulation, and drag the instruction from the toolbar into the routine.

5. Define arguments for each Parameter on the instruction call.
The instruction appears as follows in each of the languages.

**Ladder Diagram**

<table>
<thead>
<tr>
<th>Parameter With Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single question mark</td>
<td>This is a required Parameter. Enter a tag.</td>
</tr>
<tr>
<td>Single and double question marks</td>
<td>This is a required Parameter. Enter a tag.</td>
</tr>
<tr>
<td>Double question marks</td>
<td>This is not a required Parameter. You can either:</td>
</tr>
<tr>
<td></td>
<td>• leave as is and use the default value.</td>
</tr>
<tr>
<td></td>
<td>• if it’s an input value, enter a different value.</td>
</tr>
</tbody>
</table>

**Function Block Diagram**
Chapter 2   Using an Add-On Instruction

The instruction expects arguments for required Parameters as listed in the instruction tooltip.

**Structured Text**

```
Simulation_DT_1st( )

Simulation instruction which includes a ...
Simulation_DT_1st(Simulation_DT_1st, SimInput DA1)
```

The instruction expects arguments for required Parameters as listed in the instruction tooltip.

**TIP** For help with an instruction, select the instruction and then press [F1]. In Structured Text, make sure the cursor is in the blue instruction name.

### Programming Tips when Using an Add-On Instruction

This table describes programming tips for you to reference when using Add-On Instructions.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Help</td>
<td>Use the instruction help to determine how to use the instruction in your code.</td>
</tr>
<tr>
<td>Ladder Rungs</td>
<td>In a Ladder rung, consider if the instruction should be executed on a false rung condition. It may improve scan time to not execute it.</td>
</tr>
</tbody>
</table>
Programmatically Accessing a Parameter

Follow these procedures for any language when you want to access an Add-On Instruction Parameter that isn't available on the instruction face by default.

The following procedures demonstrate how to use the Jog parameter of the Motor Starter Add-On Instructions.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Name</th>
<th>Data Type</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>InOut</td>
<td>BOOL</td>
<td>Input</td>
<td>Jog command for the motor.</td>
</tr>
<tr>
<td></td>
<td>Jog</td>
<td>BOOL</td>
<td>Input</td>
<td>Jog command for the motor.</td>
</tr>
<tr>
<td></td>
<td>Start</td>
<td>BOOL</td>
<td>Input</td>
<td>Start command for the motor.</td>
</tr>
<tr>
<td></td>
<td>Stop</td>
<td>BOOL</td>
<td>Input</td>
<td>Stop command for the motor.</td>
</tr>
</tbody>
</table>

Using the Jog Command in Ladder Diagram

The first rung sets the Jog bit of Motor_Starter_LD = Jog_PB.
Use another instruction, an assignment, or an expression to read or write to the tag name of the Parameter. Use this format for the tag name of the Parameter.

\[ \text{Add}_\text{On}_{-}\text{Tag}.\text{Parameter} \]

<table>
<thead>
<tr>
<th>Where</th>
<th>Is</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Add}<em>\text{On}</em>{-}\text{Tag}</td>
<td>An instance tag defined by the Add On data type.</td>
</tr>
<tr>
<td>\text{Parameter}</td>
<td>Name of the Parameter.</td>
</tr>
</tbody>
</table>
Using the Jog Command In Function Block Diagram

Any Parameter can be made visible or not except those defined as required. If the Parameter is required, you will see it checked in the Properties dialog.

1. Click Properties for the instruction.

2. Check the Vis box of the Jog Parameter to use it in your diagram.

3. Click OK.

4. Wire to the pin for the Parameter.
Chapter 2  Using an Add-On Instruction

Using the Jog Command in Structured Text

The assignment sets the Jog bit of Motor_Starter_ST = Jog_PB.

Motor_Starter(Motor_Starter_ST, Stop_PB, Start_PB, Motor_Out_ST);

Motor_Starter_ST . Jog

Motor_Starter tag dot Jog Parameter

Monitoring the Value of a Parameter

Follow this procedure when you want to see or change a Parameter value of an Add-On Instruction.

1. Open the Properties of the Instruction based on what language you are using.
   - For either a Function Block or Ladder Diagram, click Properties for the instruction.
   - For Structured Text, right-click the instruction name and choose Properties.

2. Monitor the value of the Parameters and change any if needed.

3. Type a new value for each Parameter as needed.

4. Click Apply and when finished, OK.
View Logic and Monitor with Data Context

Follow this procedure when you want to view the logic of an Add-On Instruction and monitor data values with the logic.

1. Right-click the instruction call in any routine.

2. Select Open Instruction Logic.
The Language Editor opens with the Add-On Instruction's Logic routine and with data values from the instruction call.

As you view the logic you can:
- identify the instruction call whose tags are being used for data.
- see the logic as it executes (when offline).
- see Parameter and Local Tag values.
- change local tag and Parameter values for the data instance selected.

To edit the logic of the Add-On Instruction, you must select the instruction <definition> in Data Context.

You can’t edit logic online or when logic is in the context of an instruction call, or if the instruction is source protected.
Is the Add-On Instruction Source Protected?

An Add-On Instruction may be source protected so you cannot view the logic. Follow these steps to see if an Add-On Instruction is source protected.

1. Select the Add-On Instruction in the Controller Organizer.

   The Add-On Instruction cannot be expanded when fully protected.

2. Look in the Quick View pane for Source Protection.

   If the Source Protection attribute isn’t listed, then the instruction isn’t protected.
Copying an Add-On Instruction Definition

Copy an Add-On Instruction into your project when it exists in another RSLogix 5000 project. After you copy the definition, you can use the instruction as is or rename it, modify it and then use it in your programs.

1. Open the RSLogix 5000 project that contains the Add-On Instruction definition.

2. Find the definition in the Add-On Instructions folder.

3. Right-click the definition and choose Copy.

4. Go to your other project where you want to use the copied definition.

5. Right-click the Add-On Instructions folder and choose Paste.

**IMPORTANT** The software will add a numeric index after the Add-On instruction in case the instruction’s name already exists. For example: Motor_Starter1.
Importing an Add-On Instruction Definition

You can import a definition for an Add-On Instruction that was exported from another RSLogix 5000 project. Once the project has the definition, you can use the instruction in your programs.

1. Right-click the Add-On Instructions folder and select Import Add-On Instruction.

2. Locate the file containing the instruction.

3. Select the file and click Import.

4. The instruction appears in the Controller Organizer Add-On Instruction folder.
Exporting an Add-On Instruction Definition

Follow these steps to export an Add-On Instruction.

1. Right-click Add-On Instruction in Controller Organizer and select Export Add-On Instruction.

2. Choose a location for the L5X file.

3. Type a name for the file.

4. Decide to include references or not.

5. Click Export.
Notes:
Rockwell Automation provides technical information on the Web to assist you in using its 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 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 product up and running.

<table>
<thead>
<tr>
<th>Location</th>
<th>Support Number</th>
</tr>
</thead>
</table>
| United States          | 1.440.646.3434
|                        | 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 Automation tests all of its 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, follow these procedures.

<table>
<thead>
<tr>
<th>Location</th>
<th>Process Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>United States</td>
<td>Contact your distributor. You must provide a Customer Support case number (call the phone number above to obtain one) to your distributor in order to complete the return process.</td>
</tr>
<tr>
<td>Outside United States</td>
<td>Please contact your local Rockwell Automation representative for the return procedure.</td>
</tr>
</tbody>
</table>

www.rockwellautomation.com

Power, Control and Information Solutions Headquarters

Americas: Rockwell Automation, 1201 South Second Street, Milwaukee, WI 53204-2496 USA, Tel: (1) 414.382.2000, Fax: (1) 414.382.4444
Europe/Middle East/Africa: Rockwell Automation, Vorstlaan/Boulevard du Souverain 56, 1170 Brussels, Belgium, Tel: (32) 2 663 0600, Fax: (32) 2 663 0640
Asia Pacific: Rockwell Automation, Level 14, Core F, Cyberport 5, 100 Cyberport Road, Hong Kong, Tel: (852) 2887 4788, Fax: (852) 2508 1846

Publication 1756-PM010B-EN-P - July 2008
Supercedes Publication 1756-PM010A-EN-P - July 2007
Copyright © 2008 Rockwell Automation, Inc. All rights reserved. Printed in the U.S.A.