Logix 5000 Controllers Add-On Instructions

1756 ControlLogix, 1756 GuardLogix, 1769 CompactLogix, 1769 Compact GuardLogix, 1789 SoftLogix, 5069 CompactLogix, Emulate 5570

Rockwell Automation

Programming Manual

Original Instructions

Important User Information

Read this document and the documents listed in the additional resources section about installation, configuration, and operation of this equipment before you install, configure, operate, or maintain this product. Users are required to familiarize themselves with installation and wiring instructions in addition to requirements of all applicable codes, laws, and standards.

Activities including installation, adjustments, putting into service, use, assembly, disassembly, and maintenance are required to be carried out by suitably trained personnel in accordance with applicable code of practice.

If this equipment is used in a manner not specified by the manufacturer, the protection provided by the equipment may be impaired.

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.

Throughout this manual, when necessary, 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.

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, and recognize the consequence.

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

These labels may also be on or inside the equipment to provide specific precautions.

SHOCK HAZARD: Labels may be on or inside the equipment, for example, a drive or motor, to alert people that dangerous voltage may be present.

BURN HAZARD: Labels may be on or inside the equipment, for example, a drive or motor, to alert people that surfaces may reach dangerous temperatures.

ARC FLASH HAZARD: Labels may be on or inside the equipment, for example, a motor control center, to alert people to potential Arc Flash. Arc Flash will cause severe injury or death. Wear proper Personal Protective Equipment (PPE). Follow ALL Regulatory requirements for safe work practices and for Personal Protective Equipment (PPE).

The following icon may appear in the text of this document.

Identifies information that is useful and can help to make a process easier to do or easier to understand.

Rockwell Automation recognizes that some of the terms that are currently used in our industry and in this publication are not in alignment with the movement toward inclusive language in technology. We are proactively collaborating with industry peers to find alternatives to such terms and making changes to our products and content. Please excuse the use of such terms in our content while we implement these changes.

Summary of changes

This manual includes new and updated information. Use these reference tables to locate changed information.

Grammatical and editorial style changes are not included in this summary.

Global changes

None in this release.

New or enhanced features

This table contains a list of topics changed in this version, the reason for the change, and a link to the topic that contains the changed information.

Change	Topic
Updated the descriptions for the LastEditDate, SafetySignatureID, and SignatureID attributes.	Use GSV and SSV instructions on page 24

Preface

This manual shows how to use Add-On Instructions, which are custom instructions that you design and create, for the Logix Designer application.

If you design, program, or troubleshoot safety applications that use GuardLogix controllers, refer to the GuardLogix Safety Application Instruction Set Safety Reference Manual, publication 1756-RM095.

This manual is one of a set of related manuals that show common procedures for programming and operating Logix 5000 controllers.

For a complete list of common procedures manuals, refer to the Logix 5000 Controllers Common Procedures Programming Manual, publication 1756-PM001.

The term Logix 5000 controller refers to any controller based on the Logix 5000 operating system. Rockwell Automation recognizes that some of the terms that are currently used in our industry and in this publication are not in alignment with the movement toward inclusive language in technology. We are proactively collaborating with industry peers to find alternatives to such terms and making changes to our products and content. Please excuse the use of such terms in our content while we implement these changes.

Studio 5000 environment

The Studio 5000 Automation Engineering & Design Environment® combines engineering and design elements into a common environment. The first element is the Studio 5000 Logix Designer® application. The Logix Designer application is the rebranding of RSLogix 5000® software and will continue to be the product to program Logix 5000™ controllers for discrete, process, batch, motion, safety, and drive-based solutions.

The Studio 5000® environment is the foundation for the future of Rockwell Automation® engineering design tools and capabilities. The Studio 5000 environment is the one place for design engineers to develop all elements of their control system.

Understanding terminology

This table defines some of the terms used in this manual when describing how parameters and arguments are used in Add-On Instructions.

Term	Definition

Argument	An argument is assigned to a parameter of an Add-On Instruction instance. An argument contains the specification of the data used by an instruction in a user program. An argument can contain the following: • A simple tag (for example, L101) • A literal value (for example, 5) • A tag structure reference (for example, Recipe.Temperature) • A direct array reference (for example, Buffer[1]) • An indirect array reference (for example, Buffer[Index+1]) • A combination (for example, Buffer[Index+1].Delay)
Parameter	Parameters are created in the Add-On Instruction definition. When an Add-On Instruction is used in application code, arguments must be assigned to each required parameter of the Add-On Instruction.
InOut parameter	An InOut parameter defines data that can be used as both input and output data during the execution of the instruction. Because InOut parameters are always passed by reference, their values can change from external sources during the execution of the Add-On Instruction.
Input parameter	For an Add-On Instruction, an Input parameter defines the data that is passed by value into the executing instruction. Because Input parameters are always passed by value, their values cannot change from external sources during the execution of the Add-On Instruction.
Output parameter	For an Add-On Instruction, an Output parameter defines the data that is produced as a direct result of executing the instruction. Because Output parameters are always passed by value, their values cannot change from external sources during the execution of the Add-On Instruction.
Passed by reference	When an argument is passed to a parameter by reference, the logic directly reads or writes the value that the tag uses in controller memory. Because the Add-On Instruction is acting on the same tag memory as the argument, other code or HMI interaction that changes the argument's value can change the value while the Add-On Instruction is executing.
Passed by value	When an argument is passed to a parameter by value, the value is copied in or out of the parameter when the Add-On Instruction executes. The value of the argument does not change from external code or HMI interaction outside of the Add-On Instruction itself.
Module reference parameter	A module reference parameter is an InOut parameter of the MODULE data type that points to the Module Object of a hardware module. You can use module reference parameters in both Add-on Instruction logic and program logic. Since the module reference parameter is passed by reference, it can access and modify attributes in a hardware module from an Add-On Instruction.

Additional resources

These documents contain additional information concerning related Rockwell Automation products.

Resource	Description
Industrial Automation Wiring and Grounding Guidelines, publication, 1770-4.1	Provides general guidelines for installing a Rockwell Automation industrial system.
Logix 5000™ Controllers Design Considerations, publication 1756-RM094.	Provides information to help design and plan Logix 5000 systems.
Rockwell Automation product certifications	Provides declarations of conformity, certificates, and other certification details.

View or download publications at https://www.rockwellautomation.com/en-us/support/documentation/literature-library.html. To order paper copies of technical documentation, contact a local Rockwell Automation distributor or sales representative.

Legal notices

Rockwell Automation publishes legal notices, such as privacy policies, license agreements, trademark disclosures, and other terms and conditions on the Legal Notices page of the Rockwell Automation website.

Software and Cloud Services Agreement

Review the Rockwell Automation Software and Cloud Services Agreement here.

Open Source Software Licenses

The software included in this product contains copyrighted software that is licensed under one or more open source licenses.

You can view a full list of all open source software used in this product and their corresponding licenses at this URL:

Studio 5000 Logix Designer Open Source Attribution List

You may obtain Corresponding Source code for open source packages included in this product from their respective project web site(s). Alternatively, you may obtain complete Corresponding Source code by contacting Rockwell Automation via the **Contact** form on the Rockwell Automation website: http://www.rockwellautomation.com/global/about-us/contact/contact.page. Please include "Open Source" as part of the request text.

Designing Add-On Instructions

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.

About Add-On Instructions

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. The following table lists the benefits of using Add-On Instructions.

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.
Export and import an Add-On Instruction	• You can export an Add-On Instruction to an .L5X file that can then be imported into another project. You can also copy and paste between projects.
Simplify maintenance	• You can simplify code maintenance because Add-On Instruction logic, monitored in the Logix Designer application, animates with tag values relative to that specific instance of the Add-On Instruction.
Track revisions, view change history, and confirm instruction functionality	• You can add an instruction signature to your Add-On Instruction, which generates a unique identifier and prevents the instruction from being edited without resulting in a change to the signature.

Use Add-On-Instructions across multiple projects. You can define the instructions, 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 Logix Designer application. They appear on the instruction toolbar and in the instruction browser for easy access, just like built-in Logix Designer application instructions.

Like standard Add-On Instructions, safety Add-On Instructions let you encapsulate commonly-used safety logic into a single instruction, making it modular and easier to reuse. In addition to the instruction signature used for high-integrity Add-On Instructions, safety Add-On Instructions feature a SIL 3 safety instruction signature for use in safety-related functions up to and including SIL 3.

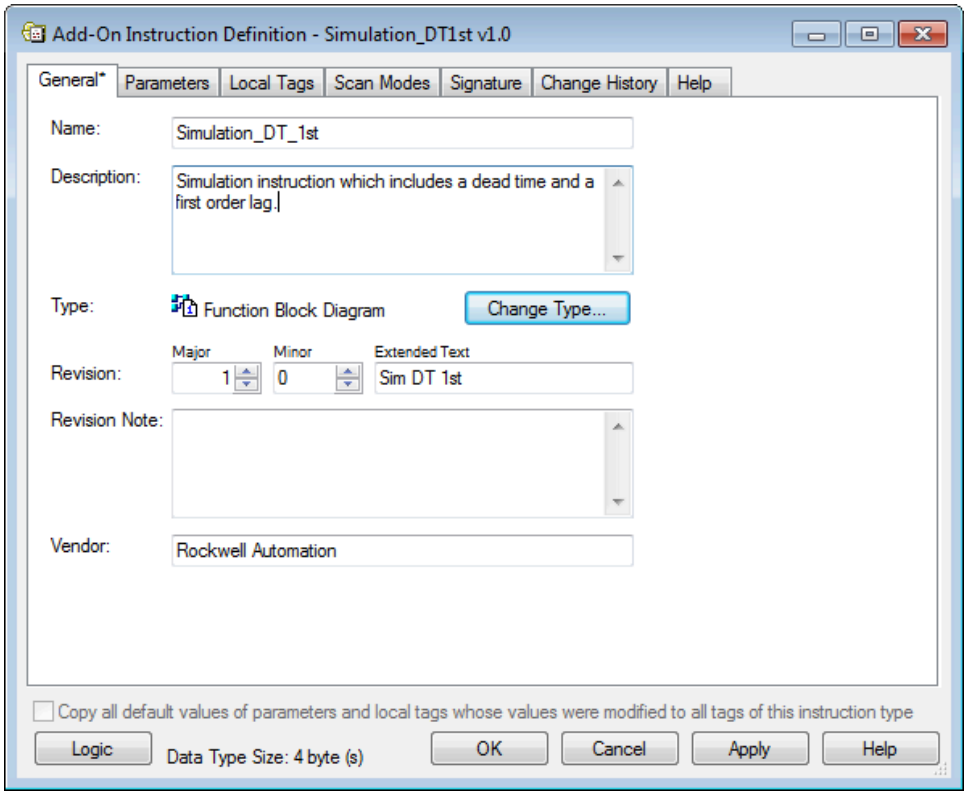
Refer to the safety reference manual for your controller, listed in the Additional resources on page 9, for details on certifying safety Add-On Instructions and using them in SIL 3 safety applications.

Components of an Add-On Instruction

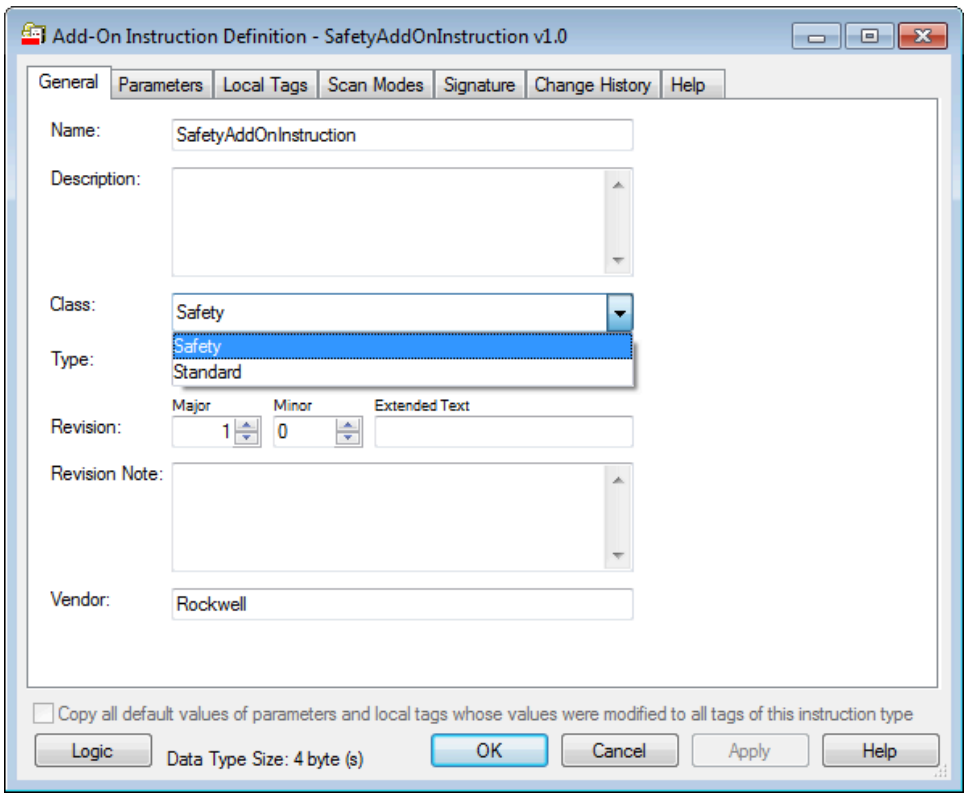
Add-On Instructions are made up of the following parts.

General information

The **General** tab contains the information you enter when you first create the instruction. You can use this tab to update that 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 for defining how it is used and when it is updated.



Class information for safety controller projects appears on the **General** tab as well. The class can be standard or safety. Safety Add-On Instructions must meet requirements specific to safety applications. See Safety Add-On Instructions on page 18 for more information.



Parameters

- What data needs to be passed to the instruction?
- What information needs to be accessible outside of the instruction?
- Do alias parameters need to be defined for data from local tags that needs to be accessible from outside the Add-On Instruction?
- How does the parameters display? The order of the parameters defines the appearance of instruction.
- Which parameters should be required or visible?

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.
- Do you want to create an alias parameter to provide outside access to a local tag?

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.

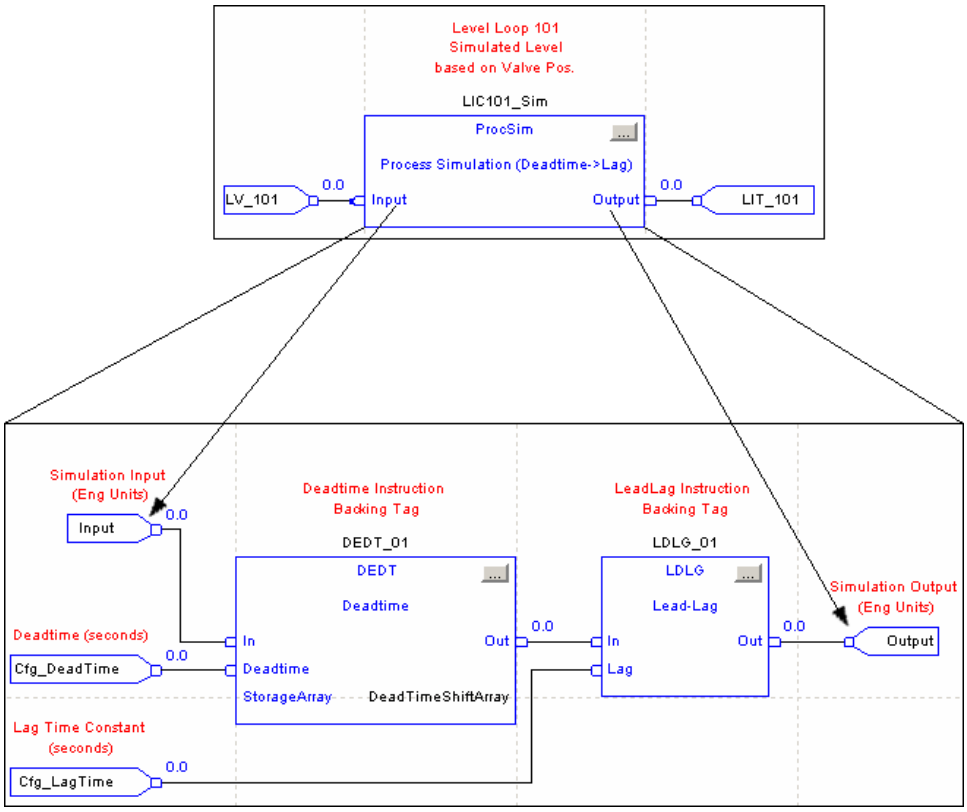
The Data Type for a Local Tag may not be:

- A multi-dimensional array or an object type, which includes all Motion types, MSG, ALARM_ANALOG, and ALARM_DIGITAL.
- A data type used only for InOut parameters (MODULE).

The **Data Type** field is unavailable for members of a Local Tag.

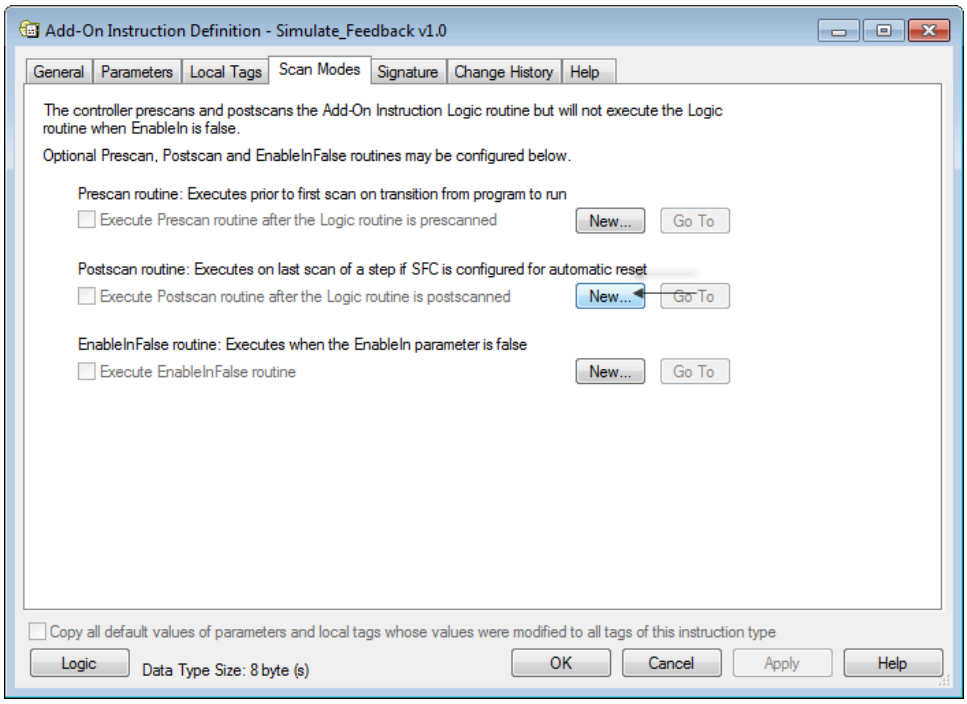
Logic routine

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. The following image is the interface of an Add-On Instruction and its primary logic routine that defines what the instruction does.



Optional Scan Modes routines

You can define additional routines for scan mode behavior.



Instruction signature

The instruction signature, available for both standard and safety controllers, lets you quickly determine if the Add-On Instruction has been modified. Each Add-On Instruction has its own instruction signature on the Add-On Instruction definition. The instruction signature is required when an Add-On Instruction is used in SIL 3 safety-related functions, and may be required for regulated industries. Use it when your application calls for a higher level of integrity.

Once generated, the instruction signature seals the Add-On Instruction, preventing it from being edited until the signature is removed. This includes rung comments, tag descriptions, and any instruction documentation that was created. When an instruction is sealed, you can perform only these actions:

- Copy the instruction signature
- Create or copy a signature history entry
- Create instances of the Add-On Instruction
- Download the instruction
- Remove the instruction signature
- Print reports

The instruction signature does not prevent referenced Add-On Instructions or User-defined Data Types from being modified. Changes to the parameters of a referenced Add-On Instruction or

to the members of a referenced User-defined Data Type can cause the instruction signature to become invalid. These changes include:

- Adding, deleting, or moving parameters, local tags, or members in referenced User-defined Data Types.
- Changing the name, data type, or display style of parameters, local tags, or members in referenced User-defined Data Types.

If you want to enable project documentation or source protection on an Add-On Instruction to be sealed with an instruction signature, you need to import the translated information or apply source protection before generating the signature. You must have the source key to generate a signature or to create a signature history entry for a source-protected Add-On Instruction that has an instruction signature.

See Defining Source Protection for an Add-On Instruction on page 53 for more information on source protecting your Add-On Instruction.

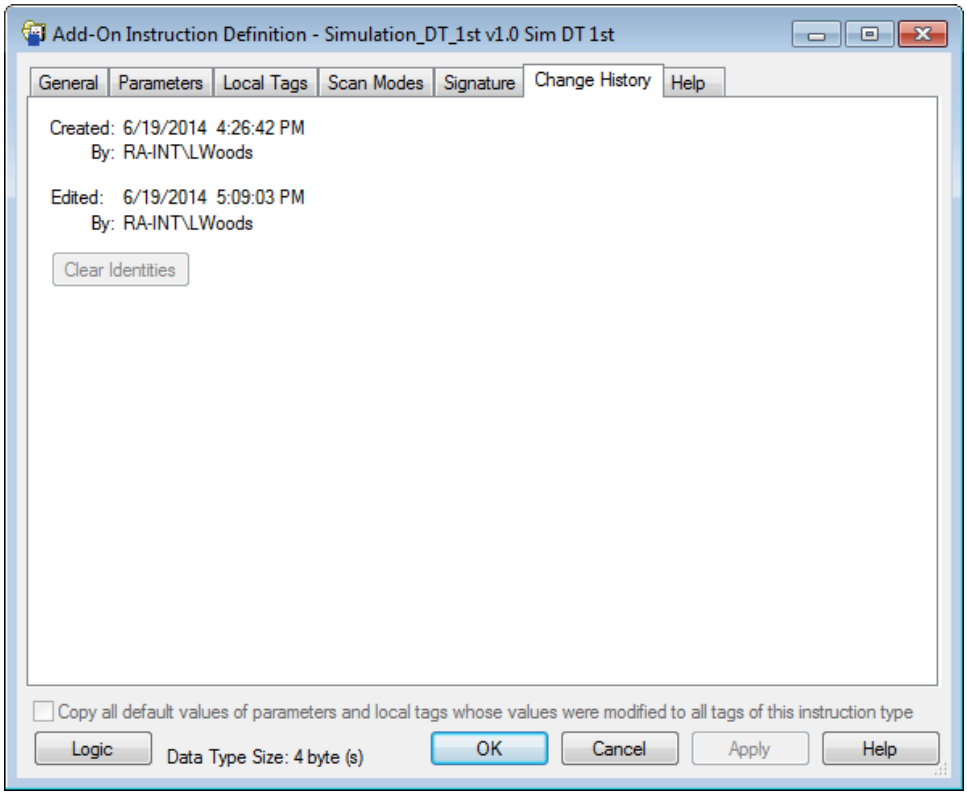
The instruction signature is not guaranteed to be maintained when migrating between major revisions of the Logix Designer application.

Signature history

The Signature history provides a record of signatures for future reference. A signature history entry consists of the name of the user, the instruction signature, the timestamp value, and a user-defined description. Up to six history entries can be stored. If a seventh entry is made, the oldest entry is automatically deleted.

Change History

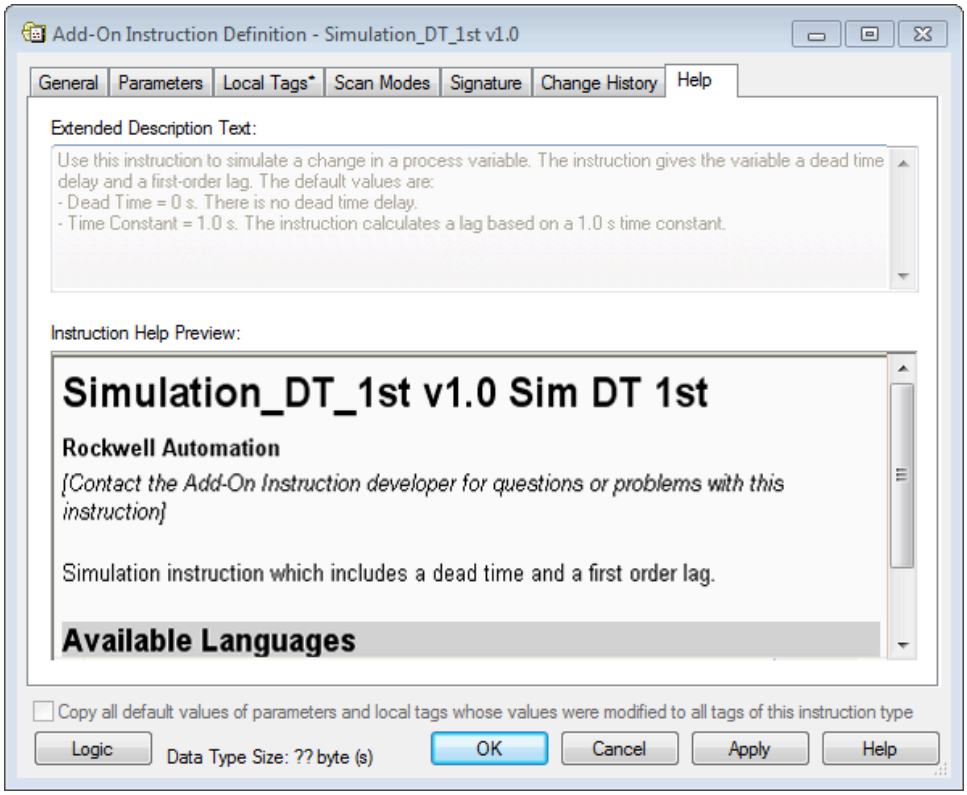
The **Change History** tab displays the creation and latest edit information that is tracked by the software. The **By** fields show who made the change based on the Windows user name at the time of the change.



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. This content reflects updates as Parameters or other attributes are changed in the definition.

The **Instruction Help Preview** shows how your instruction will appear in the various languages, based on parameters defined as required or visible.



Considerations for Add-On Instructions

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, do it 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 uses only 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.

Safety Add-On Instructions

Safety Add-On Instructions are used in the safety task of GuardLogix safety controllers. Create a safety Add-On Instruction if you need to use your instruction in a safety application. Safety Add-On Instructions are subject to a number of restrictions. These restrictions, enforced by Logix

Designer application and all GuardLogix controllers, are listed here for informational purposes only.

- They may use only safety-approved instructions and data types.
- All parameters and local tags used in a safety Add-On Instruction must also be safety class.
- Safety Add-On Instructions use ladder diagram logic only and can be called in safety routines only, which are currently restricted to ladder logic.
- Safety Add-On Instructions may be referenced by other safety Add-On Instructions, but not by standard Add-On Instructions.
- Safety Add-On instructions cannot be created, edited, or imported when a safety project is safety-locked or has a safety task signature.

Refer to the the safety reference manual for your controller, listed in the Additional resources on page 9, for information on how to certify a safety Add-On Instruction as well as details on requirements for safety applications, the safety task signature, and a list of approved instructions and data types.

Instruction signature

The instruction signature, available for both standard and safety controllers, lets you quickly determine if the Add-On Instruction has been modified. Each Add-On Instruction has its own instruction signature on the Add-On Instruction definition. The instruction signature is required when an Add-On Instruction is used in SIL 3 safety-related functions, and may be required for regulated industries. Use it when your application calls for a higher level of integrity.

Once generated, the instruction signature seals the Add-On Instruction, preventing it from being edited until the signature is removed. This includes rung comments, tag descriptions, and any instruction documentation that was created. When an instruction is sealed, you can perform only these actions:

- Copy the instruction signature
- Create or copy a signature history entry
- Create instances of the Add-On Instruction
- Download the instruction
- Remove the instruction signature
- Print reports

The instruction signature does not prevent referenced Add-On Instructions or User-defined Data Types from being modified. Changes to the parameters of a referenced Add-On Instruction or to the members of a referenced User-defined Data Type can cause the instruction signature to become invalid. These changes include:

- Adding, deleting, or moving parameters, local tags, or members in referenced User-defined Data Types.
- Changing the name, data type, or display style of parameters, local tags, or members in referenced User-defined Data Types.

If you want to enable project documentation or source protection on an Add-On Instruction to be sealed with an instruction signature, you need to import the translated information or apply source protection before generating the signature. You must have the source key to generate a

signature or to create a signature history entry for a source-protected Add-On Instruction that has an instruction signature.

See Defining Source Protection for an Add-On Instruction on page 53 for more information on source protecting your Add-On Instruction.

The instruction signature is not guaranteed to be maintained when migrating between major revisions of the Logix Designer application.

Safety instruction signature

When a sealed safety Add-On Instruction is downloaded for the first time, a SIL 3 Safety Instruction Signature is automatically generated.

The Safety Instruction Signature is a number that identifies the execution characteristics of the safety Add-On Instruction (AOI). The Safety Instruction Signature is different than the ID, which consists of a number and time stamp which helps determine if the instruction has been modified.

The Safety Instruction Signature takes into account a variety of factors that affect execution characteristics of the Safety Instruction, such as firmware and software version, and technology considerations such as compilers and hardware platform. Therefore, the Safety Instruction Signature is not guaranteed to be maintained when migrating between major revisions of Logix Designer.

The Safety Instruction Signature is computed by the controller and may change upon download.

For details on how to certify a safety Add-On Instruction, refer to the safety reference manual for your controller.

Programming languages

- What language do you want to use to program your instruction?

  The primary logic of your instruction will consist of a single routine of code. Determine which software programming language to use based on the use and type of application. Safety Add-On Instructions are restricted to Ladder Diagram.

- If execution time and memory usage are critical factors, refer to the Logix5000 Controllers Execution Time and Memory Use Reference Manual, publication 1756-RM087.

Transitional instructions

Some instructions execute (or retrigger) only when rung-condition-in toggles from false to true. These are transitional-ladder diagram 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, thus the transitional instruction does not detect the transition to the false state. Extra conditional logic is 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.

The EnableInFalse routine can be used to provide the conditioning required to retrigger 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.

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.

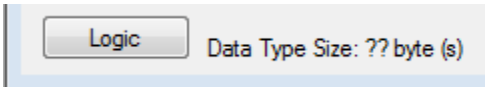
Logix Designer versions 27 and earlier do not set a limit on the number of InOut parameters. However, limit extended properties (.@Min and .@Max syntax) should not be defined on an InOut parameter of an Add-On Instruction and should not be used in Add-On Instruction definition logic or the logic does not verify.

Logix Designer version 28 limits the number of InOut parameters for Add-On Instructions to 40.

Logix Designer versions 29 and later limit the number of InOut parameters for Add-On Instructions to 64 and limit the Add-On Instruction nesting levels to 16. Rockwell recommends limiting the level of nesting to eight levels to reduce complexity and make the instruction easier to manage.

Limits cannot be accessed inside Add-On Instruction logic.

The maximum data instance supported (which includes Inputs, Outputs, and local tags) is two megabytes. The data type size is displayed on the bottom of the **Parameters** and **Local Tags** tab in the Add-On Instruction Definition.



Runtime editing

Add-On Instructions can only be edited offline. If the intended functionality needs 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 up to 16 levels deep. However, Rockwell recommends limiting the level of nesting to eight levels to reduce complexity and make the instruction easier to manage.

Add-On Instructions cannot call other routines through a JSR instruction. You must use nested instructions if you need complex functionality consisting of multiple routines.

To nest Add-On Instructions, both the nested instruction and the instruction that calls it must be of the same class type or the calling instruction will not verify. That is, standard Add-On Instructions may call only standard Add-On Instructions and safety Add-On Instructions may call only safety Add-On Instructions.

Routines versus Add-On Instructions

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

Aspect	Main Routine	Subroutine	Add-On Instruction
Accessibility	N/A	Within program (multiple copies, one for each program)	Anywhere in controller (single copy for the entire project)
Parameters	N/A	Pass by value	Pass by value with Input and Output parameters
Numeric parameters	N/A	No conversion, user must manage	Automatic data type conversion for Input and Output parameters
Parameters data types	N/A	Atomic, arrays, structures	Atomic for any parameter Arrays and structures must be InOut parameters
Parameter checking	N/A	None, user must manage	Verification checks that correct type of argument has been provided for a parameter
Data encapsulation	N/A	All data at program or controller scope (accessible to anything)	Local data is isolated (only accessible within instruction)
Monitor/debug	In-line code with its data	Mixed data from multiple calls, which complicates debugging	Single calling instance data, which simplifies debugging
Supported programming languages	FBD, LD, SFC, ST	FBD, LD, SFC, ST	FBD, LD, ST
Callable from	N/A	FBD, LD, SFC, ST	FBD, LD, SFC through ST, ST
Protection	Locked and view only	Locked and view only	Locked and view only
Documentation	Routine, rung, textbox, line	Routine, rung, textbox, line	Instruction, revision information, vendor, rung, textbox, line, extended help
Execution performance	Fastest	JSR/SBR/RTN instructions add overhead All data is copied Indexed reference impact	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
Memory use	Most used	Very compact	Compact call requires more memory than a subroutine call All references need an additional word
Edit	Online/offline	Online/offline	Offline only
Import/export	Entire routine, including tags and instruction definitions to L5X	Entire routine, including tags and instruction definitions to L5X	Full instruction definition including routines and tags to L5X
Instruction signature	N/A	N/A	32-bit signature value seals the instruction to prevent modification and provide high integrity

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 uses 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 through this instruction-defined tag within the normal tag-scoping rules.

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.

You can access a local tag through an alias parameter.

See Creating an alias parameter for a local tag on page 27.

Unavailable instructions within Add-On Instructions

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

Unavailable Instruction	Description
BRK	Break
EOT	End of Transition
EVENT	Event Task Trigger
FOR	For (For/Next Loop)

IOT	Immediate Output
JSR	Jump to Subroutine
JXR	Jump to External Routine
MAOC	Motion Arm Output Cam
PATT	Attach to Equipment Phase
PCLF	Equipment Phase Clear Failure
PCMD	Equipment Phase Command
PDET	Detach from Equipment Phase
POVR	Equipment Phase Override Command
RET	Return
SBR	Subroutine
SFP	SFC Pause
SFR	SFC Reset

Safety application instructions, such as Safety Mat (SMAT), may be used in safety Add-On Instructions only. For detailed information on safety application instructions, refer to the GuardLogix Safety Application Instruction Set Safety Reference Manual, publication 1756-RM095.

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

- ALMA (Analog Alarm)
- ALMD (Digital Alarm)
- All Motion Instructions
- MSG (Message)

Use GSV and SSV instructions

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

- AddOnInstructionDefinition

  GSV-only. SSV instructions will not verify. Also, the classes that represent programming components (Task, Program, Routine, AddOnInstructionDefinition) support only 'this' as the Instance Name.

- Axis
- Controller
- Controller Device

- CoordinateSystem
- CST

GSV-only. SSV instructions will not verify.

- DF1
- FaultLog
- HardwareStatus

GSV-only. SSV instructions will not verify.

- Message
- Module
- MotionGroup
- Program

The classes that represent programming components (Task, Program, Routine, AddOnInstructionDefinition) support only 'this' as the Instance Name.

- Redundancy
- Routine
- Safety
- Serial Port
- Task
- TimeSynchronize
- WallClockTime

When you enter a GSV or SSV instruction, Logix Designer application displays the object classes, object names, and attribute names for each instruction. This table lists the attributes for the AddOnInstructionDefinition class.

Attribute Name	Data Type	Attribute Description
MajorRevision	DINT	Major revision number of the Add-On Instruction
MinorRevision	DINT	Minor revision number of the Add-On Instruction
Name	String	Name of the Add-On Instruction
RevisionExtendedText	String	Text describing the revision of the Add-On Instruction
Vendor	String	Vendor that created the Add-On Instruction

LastEditDate	LINT	Coordinated Universal Time (UTC) value of the last time the AOI was edited. The LastEditDate is a 64-bit µS value referenced from 0000 hours, January 1, 1970.
SignatureID	DINT	Calculated by software based on the AOI definition.
SafetySignatureID	DINT	Calculated by firmware based on the AOI compiled logic. Used only for a Safety AOI.

For more information on using GSV and SSV instructions, refer to the Logix Controllers Instructions Reference Manual, publication 1756-RM009.

Considerations when creating parameters

Consider the following information when you are creating parameters.

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.

Aspect	By Value (Input or Output)	By Reference (InOut)
Value	Synchronous—the argument's value does not change during Add-On Instruction execution.	Asynchronous—the argument'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.
Performance	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.	Parameters access argument tags directly by reference, which leads to faster execution of instruction calls.
Memory usage	Most amount.	Least amount.
Parameter data types supported	Atomic (SINT, DINT, INT, REAL, BOOL).	Atomic, arrays, and structures.

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.

Creating an alias parameter for a local tag

Alias parameters simplify connecting local tags to an Input or Output tag that is commonly used in the Add-On Instruction's application without requiring that manual code be created to make the association. Aliases can be used to define an Input or Output parameter with direct access to a local tag or its member. Changing the value of an alias parameter changes the data of the local tag or local tag member it represents and vice versa.

Alias parameters are subject to these restrictions:

- Alias parameters must be either Input or Output parameters.
- You can only create an alias parameter for a local tag or its member.
- Only one Input and one Output parameter may be mapped to the same local tag or the same member of a local tag.
- Only BOOL, SINT, INT, DINT, and REAL data types may be used.
- Alias parameters may not be constants.
- The External Access type of an alias parameter matches the External Access type of the local tag to which it is mapped.

  For information on constants and External Access, see Data access control on page 29.

Using a single dimension array as an InOut parameter

The InOut parameter can be defined to be a single dimension array. When specifying the size of this array, consider that the user of your array can either:

- Pass an array tag that is the same size as your definition.
- Pass an array tag that is larger than your definition.

When developing your logic, use the Size instruction to determine the actual size of the referenced array to accommodate this flexibility.

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, assume you have defined 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.

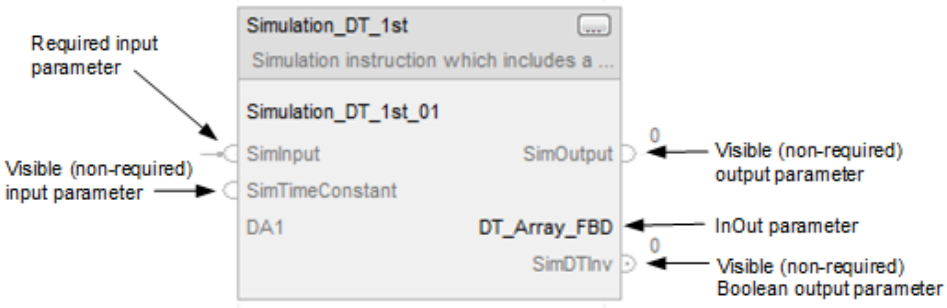
Determining which parameters to make visible or required

To help be sure that specific data is passed into the Add-On Instruction, you can use required parameters. A required parameter must be passed each argument 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 do not usually need to pass the parameter value out to an argument, but you do want to display its value prominently for troubleshooting.

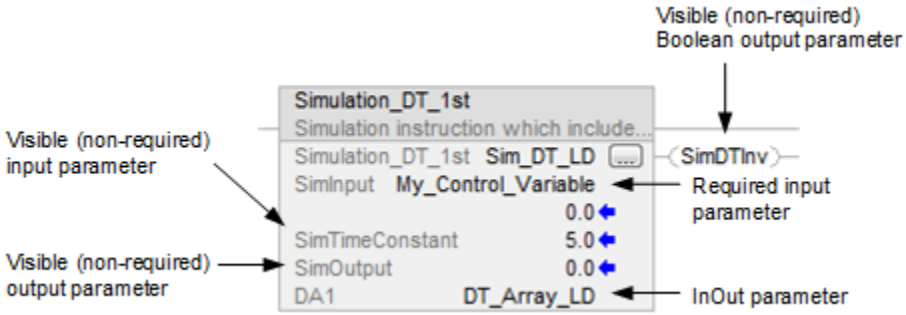
Required parameters are always visible, and InOut parameters are always required and visible. All Input and Output parameters, regardless of being marked as required or visible, can be programmatically accessed as a member of the instruction's tag.

The following picture shows a Simulation instruction in the function block editor.



If you want a pin that is displayed in Function Block, but wiring to it is optional, set it as Visible.

The following picture shows a Simulation instruction in the ladder editor.



- 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.

This table explains the effects of the Required and Visible parameter settings on the display of the instructions.

Parameter Type	Is the Parameter Required?	Is the Parameter Visible?	Ladder Diagram		Function Block Diagram			Structured Text
			Does the Value display?	Does the Argument display?	Do You Need to Connect the Parameter?	Does the Argument display?	Can You Change the Visibility Setting Within the Function Block?	Does the Argument display?
Input	Y	Y	Y	Y	Y	N/A	N	Y

Parameter Type	Is the Parameter Required?	Is the Parameter Visible?	Ladder Diagram		Function Block Diagram			Structured Text
Input	N	Y	Y	N	N	N/A	Y	N
Input	N	N	N	N	N	N/A	Y	N
Output	Y	Y	Y	Y	Y	N/A	Y	Y
Output	N	Y	Y	N	N	N/A	Y	N
Output	N	N	N	N	N	N/A	Y	N
InOut	Y	Y	N	Y	N/A	Y	N	Y

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 do not 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.

When you are using your Add-On Instructions, the Visible setting may be overridden in Function Block Diagram routines if the parameter is not required or already wired. Overriding the visibility at the instruction call does not affect this definition configuration.

Using standard and safety tags

When creating a safety Add-On Instruction, follow these guidelines for standard and safety tags:

- Standard tags may not be used as Input, Output, or InOut parameters of a safety Add-On Instruction.
- Safety tags may be used as Input parameters for standard Add-On Instructions.

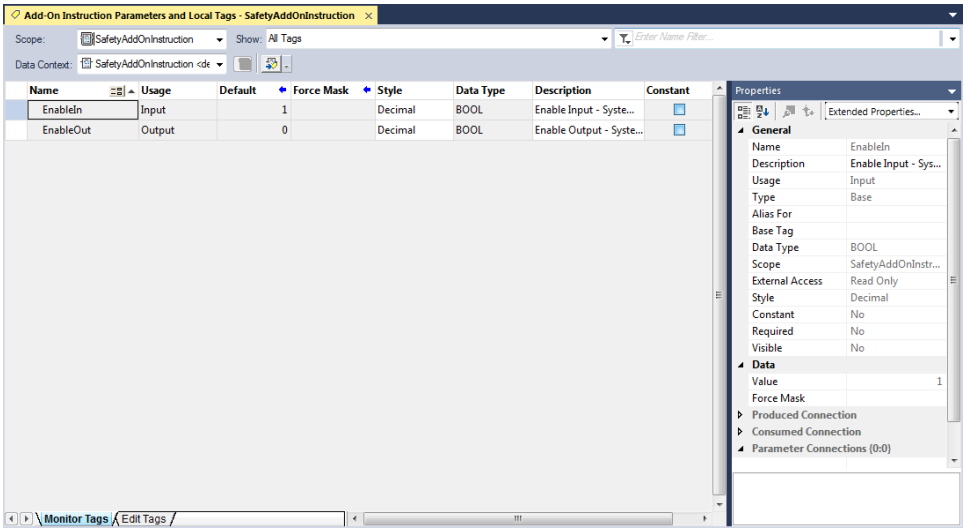
Data access control

In the Logix Designer application, versions 18 and later, you can prevent programmatic modification of InOut parameters by designating them as constants. You can also configure the type of access to allow external devices, such as an HMI, to have to your tag and parameter data. You can control access to tag data changes with Logix Designer application by configuring FactoryTalk security.

Constant values

InOut parameters may be designated as constant value tags to prevent their data from being modified by controller logic. If the logic of an Add-On Instruction contains a write operation to

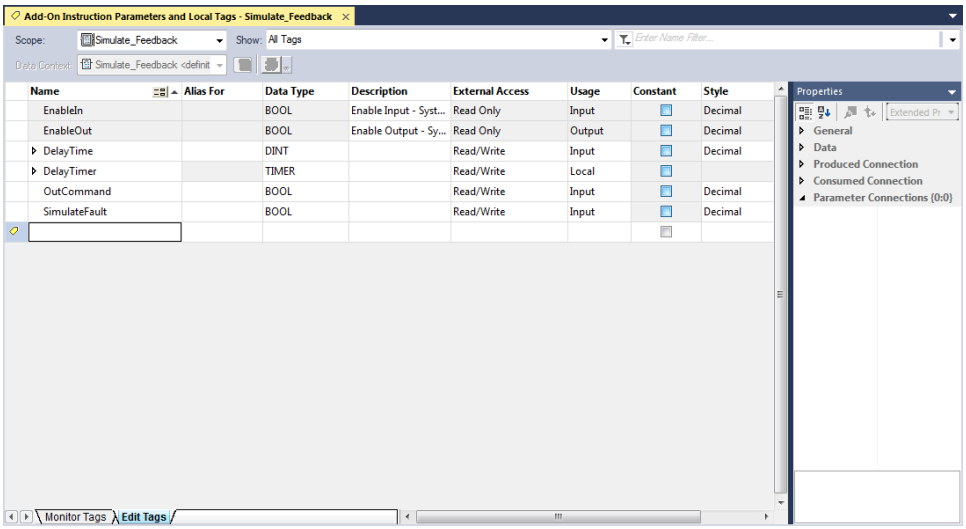
a constant value parameter, the Add-On Instruction does not verify in the Add-On Instruction definition context.



External Access

External Access defines the level of access that is allowed for external devices, such as an HMI, to see or change tag values.

Add-On Instruction Parameters and Tags	External Access Options
Local tag	Read/Write
Input parameter	Read Only
Output parameter	None
EnableIn parameter	Read Only
EnableOut parameter	
InOut parameter	N/A



Planning your Add-On Instruction design

Take time to plan your instruction design. Advance planning can identify issues that need to be addressed. 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 creating your Add-On 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?
- Do you need to higher level of integrity on your Add-On Instruction?

  If so, you can generate an instruction signature as a means to verify that your Add-On Instruction has not been modified.
- Do you need to use safety application instructions and certify your safety Add-On Instruction to SIL-3 integrity?

  For details on how to certify a safety Add-On Instruction, refer to the safety reference manual for your controller, listed in the Additional resources on page 9.

Parameters

- What data needs to be passed to the instruction?
- What information needs to be accessible outside of the instruction?
- Do alias parameters need to be defined for data from local tags that needs to be accessible from outside the Add-On Instruction?
- How does the parameters display? The order of the parameters defines the appearance of instruction.
- Which parameters should be required or visible?

Naming conventions

The instruction name is to be used as the mnemonic for your instruction. Although the name can be up to 40 characters long, you typically want to use shorter, more manageable names.

Source protection

- What type of source protection needs to be defined, if any?
- Who has access to the source key?
- Will you need to manage source protection and an instruction signature?

  Source protection can be used to provide read-only access of the Add-On Instruction or to completely lock or hide the Add-On Instruction and local tags.

  Source protection must be applied prior to generating an instruction signature.

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.
- Do you want to create an alias parameter to provide outside access to a local tag?

Programming languages

- What language do you want to use to program your instruction?

  The primary logic of your instruction will consist of a single routine of code. Determine which software programming language to use based on the use and type of application. Safety Add-On Instructions are restricted to Ladder Diagram.
- 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.
- In what language do Scan mode routines need to 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.

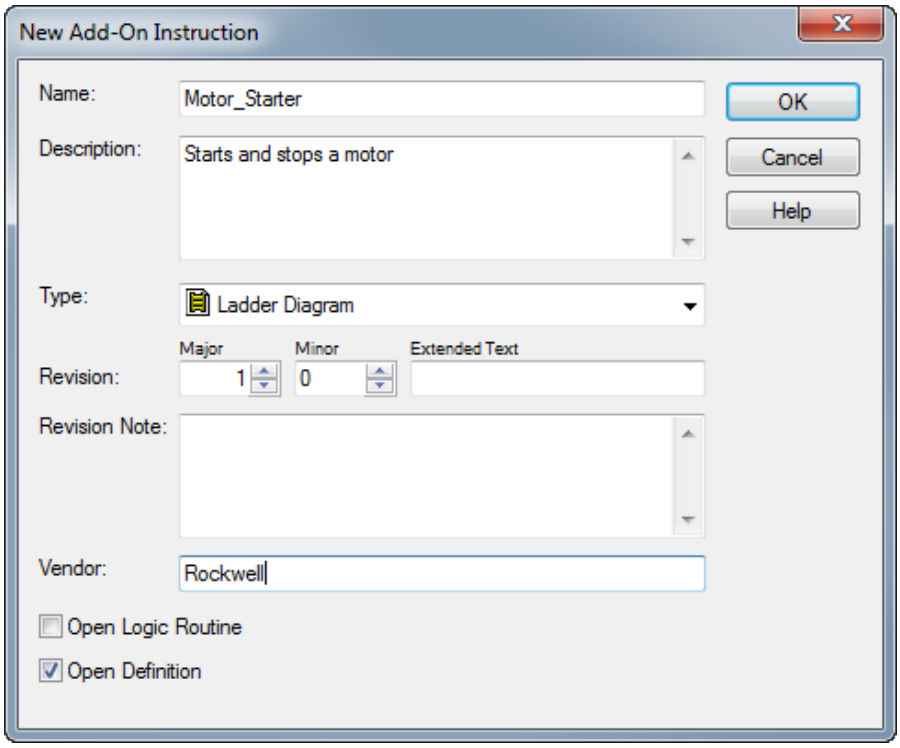
Defining Add-On Instructions

Create an Add-On Instruction

Use the **New Add-On Instruction** dialog to create Add-On Instructions.

To create a New Add-On Instruction

1. Open a new or existing project.
2. Right-click the **Add-On Instructions** folder in the **Controller Organizer** and select **New Add-On Instruction**.



3. In the **Name** box, type a unique name for the new instruction.

   The name can be up to 40 characters long. It must start with a letter or underscore and must contain only letters, numbers, or underscores. The name must not match the name of a built-in instruction or an existing Add-On Instructions.
4. In **Description** box, type a description for the new instruction, maximum 512 characters.
5. For safety projects, in the **Class** box, select either a **Safety** or **Standard**.

   The **Class** field is available on the **Add-On Instructions** dialog box for safety controller projects.
6. In the **Type** box, select a programming language for Add-On Instruction logic.

   The language Type defaults to Ladder Diagram for safety Add-On Instructions.
7. In **Revision** box, assign a Revision level for the instruction.
8. (Optional) In the **Revision Note** box, type a Revision note.

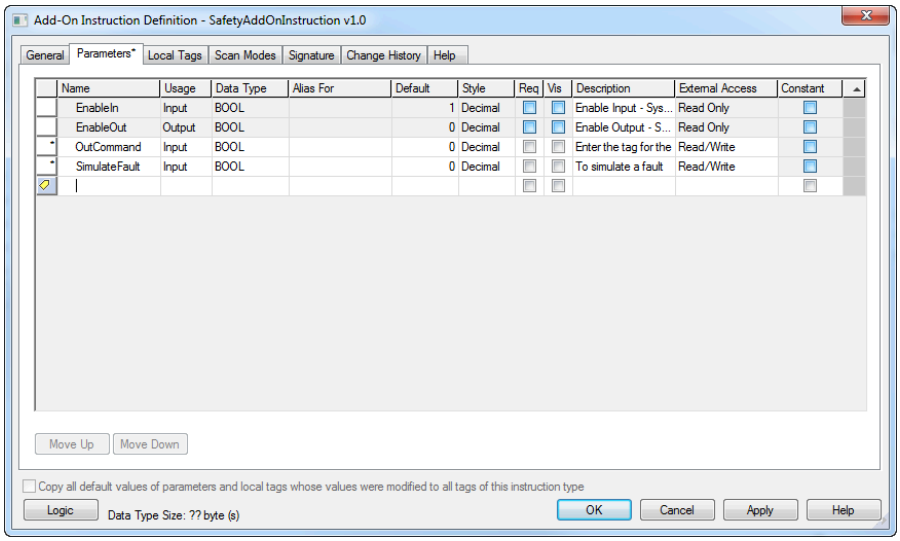
9. (Optional) In the **Vendor** box, add information about the Vendor.
10. Click **OK** to create the instruction.

Create a parameter

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

To create a parameter

1. In the **Controller Organizer**, right-click an **Add-On Instruction** and select **Open Definition**.
2. On the **Parameters** tab, in the blank Name box, type a name for a parameter.



3. In the **Usage** box, select **Input**, **Output**, or **InOut**.

> An instruction with only Input parameters, except EnableOut, is treated as an input instruction in a Ladder diagram and is displayed left-justified. The EnableOut parameter is used for the rung-out condition.

4. In the **Data Type** list, choose the type based on the parameter usage:
   - An Input parameter is a passed by value into the Add-On Instruction and must be a SINT, INT, DINT, REAL, or BOOL data type.
   - An Output parameter is a passed by value out of the Add-On Instruction and must be a SINT, INT, DINT, REAL, or BOOL data type.
   - An InOut parameter is a passed by reference into the Add-On Instruction and can be any data type including structures and array. Module reference parameters must be InOut parameters with the MODULE data type (see Creating a module reference parameter on page 36).
5. If this parameter is intended as an alias for an existing local tag, select the **Alias For** check box to select the local tag or its member.

> You can also designate a parameter as an alias for a local tag by using the Tag Editor. See Edit Parameters and Local Tags on page 38.

6. In the **Default** values list, 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 to the Add-On Instruction.

   If you want to update existing invocations of the instruction to the new default values, select the **Copy all default values of parameters and local tags whose values were modified to all tags of this instruction type** check box at the bottom of the Add-On Instruction Definition Editor. For details on copying default values, see Copying Parameter or Local Tag Default Values on page 40.

7. In the **Style** list, set the display style.
8. In the **Req** and **Vis** lists, select the check box to make the parameter required or visible, as desired.

   See Determining which parameters to make visible or required on page 27. If you decide to make the parameter required, it will also be visible.

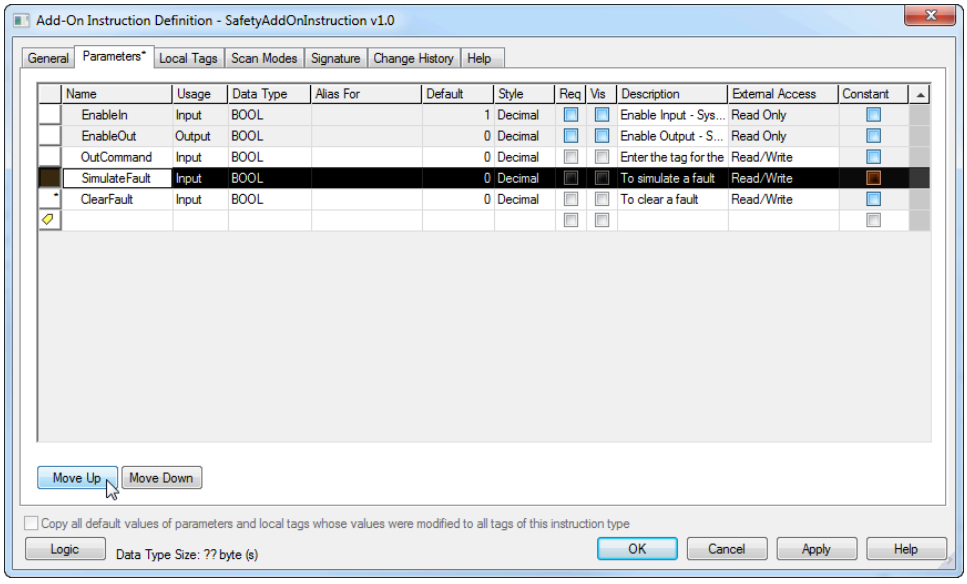
9. In the **Description** list, type a description, maximum 512 characters.

   This description appears in the instruction's help.

10. In the **External Access** list, select a type for Input or Output parameters; **Read/Write**, **Read Only**, **None**.
11. In the **Constant** list, select InOut parameters check box you want to designate as constant values.
12. Repeat for additional parameters.

    You can also create parameters by using the Tag Editor, **New Parameter or Local Tag** dialog box, or by right-clicking a tag name in the logic of your routine.

The order that you create the parameters is how they appear in the data type and on the instruction face. To rearrange the order of the **Parameter** tab on the Add-On Instruction Definition Editor, select the parameter row and click **Move Up** or **Move Down**.



Create a module reference parameter

A module reference parameter is an InOut parameter of the MODULE data type that you use to access and modify attributes in a hardware module from within the Add-On Instruction. For information on using a module reference parameter, see Referencing a hardware module on page 74. You can use the module reference parameter in two ways:

- In a GSV or SSV instruction, or an Add-on Instruction, you can use the module reference parameter as the Instance Name or Add-on Instruction parameter.
- In an Add-on Instruction, or in a GSV or SSV instruction, you can pass the module reference parameter into the InOut parameter of another nested Add-on Instruction.

There are several limitations on module reference parameters:

- Module references parameters can only be InOut parameters with the MODULE data type.
- You can use a module reference parameter only in standard programs and Add-on Instructions, not in Safety programs or Safety Add-on Instructions.
- Program parameters that reference a module must connect to a module, and cannot reference other module reference parameters.
- Module reference parameters must be program or Add-on Instruction scope, not controller scope.

You cannot create a module reference tag. You can only reference modules using an InOut parameter of the MODULE data type.

To create a module reference parameter

1. In the **Controller Organizer**, right-click an **Add-On Instruction** and select **Open Definition**.
2. On the **Parameters** tab, in the blank **Name** box, type a name for a parameter.
3. In the **Usage** box, select **InOut**.
4. In the **Data Type** list, select the **MODULE** type. This data type is for the Module object, and contains the following information:
   - Entry Status
   - Fault Code
   - Fault Info
   - FW Supervisor Status
   - Force Status
   - INSTANCE
   - LED Status
   - Mode
   - Path

   For more information on the Module object, search the Logix Designer online help.
5. In the **Description** list, type a description, maximum 512 characters.

This description appears in the instruction's help.

6. In the **Constant** list, select InOut parameters check box you want to designate as constant values.

You can also create parameters by using the Tag Editor, **New Parameter or Local Tag** dialog box, or by right-clicking a tag name in the logic of your routine.

For more information on using parameters in programs, see Logix5000 Controllers Program Parameters Programming Manual, publication 1756-PM021.

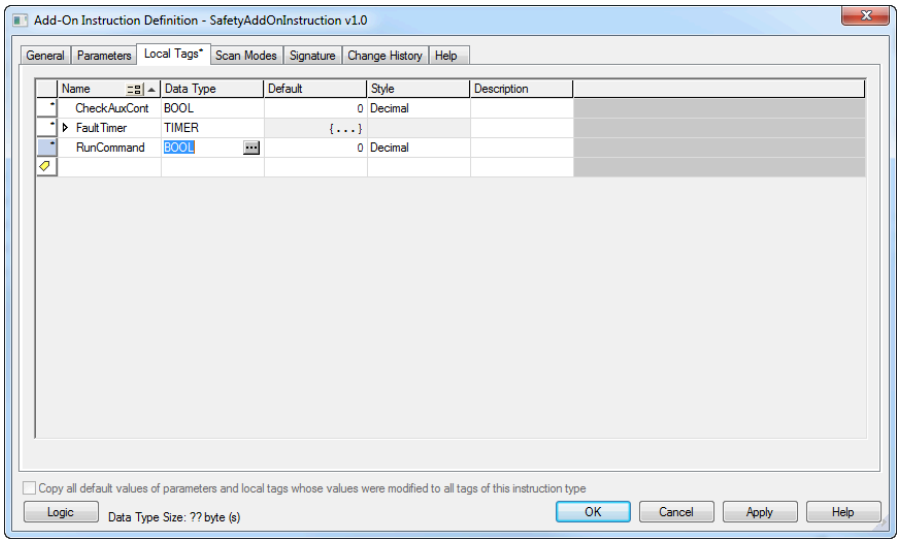
Create local tags

Use the Add-On Instruction Definition Editor to create the local tags for your instructions. Local tags contain data that will be used by your Add-On Instruction but that you do not want exposed to the user of your instruction. Local tags do 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, the Motor_Starter v1.0 instruction, shown in step 2, has a tag called 'CheckAuxContact'. This tag can be referenced by an HMI through 'instancetag.CheckAuxContact', where instancetag is the tag used to call the instruction.

To create local tags

1. In the **Controller Organizer**, right-click an instruction and select **Open Definition**.
2. On the **Local Tags** tab, in the blank **Name** box field, type a name for a new tag.
3. In the **Data Type** list, select a data type from the **Select Data Type** dialog box.



You cannot use these data types for local tags - ALARM_ANALOG, ALARM_DIGITAL, MESSAGE, MODULE, or any Motion data types, for example Axis or MOTION_GROUP. To use these type of tags in your Add-On Instruction, define an InOut Parameter. Local tags also are limited to single dimension arrays, the same as User-Defined Data Types.

Refer to the safety reference manual for your controller, listed in the Additional resources on page 9, for a list of data types supported for safety instructions.

4. In the **Default** list, 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 to the Add-On Instruction.

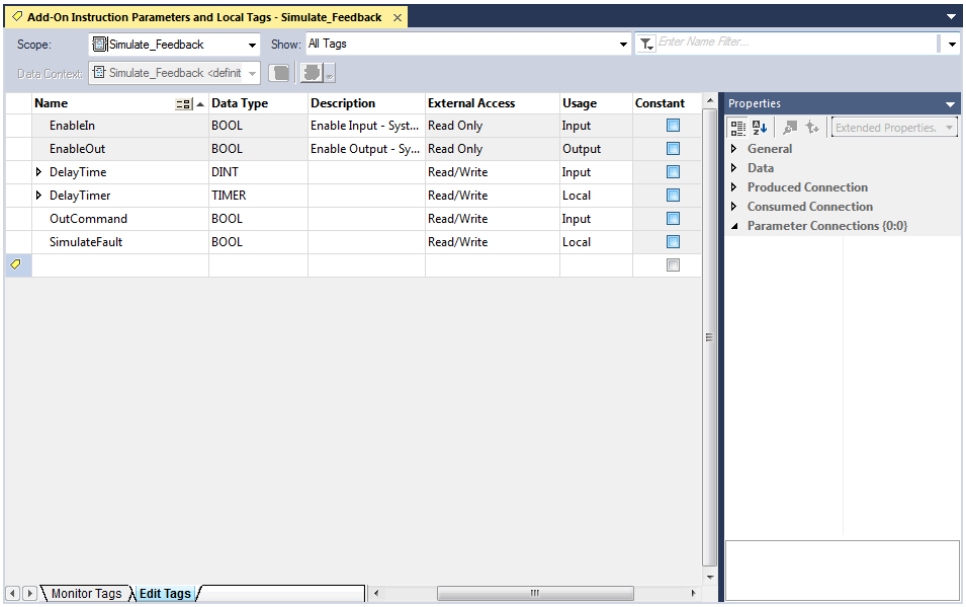
   Select the **Copy all default values of parameters and local tags whose values were modified to all tags of this instruction type** check box at the bottom of the Add-On Instruction Definition Editor if you want to update existing invocations of the instruction to the new default values. For details on copying default values, see Copying Parameter or Local Tag Default Values on page 40.

5. In the **Style** list, set the display style.
6. In the **Description** list, type a description, a maximum of 512 characters.
7. Repeat for additional local tags.

   When you create a local tag from the Local Tags tab, the **External Access**setting defaults to **None**. You can edit the External Access setting by using the Tag Editor. See Edit Parameters and Local Tags on page 38.

Editing parameters and local tags

You can also add and edit parameters and local tags on the **Edit Tags** tab.



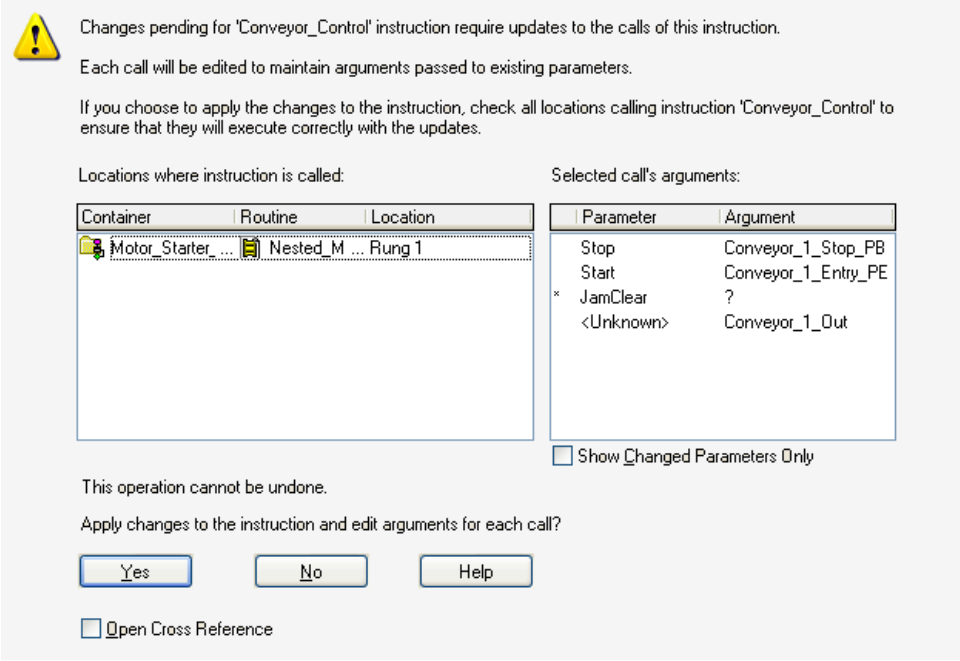
Updates to arguments following parameter edits

If you edit an Add-On Instruction by adding, deleting, renaming, reordering, or changing the status or usage type of one or more parameters, RSLogix 5000 software, version 18 and later, automatically updates the arguments on calls to the instruction.

ATTENTION: Source-protected routines and other source-protected Add-On Instructions that use the edited Add-On Instruction are not automatically updated if the source key is unavailable. The Add-On Instruction or routine may still verify, but the resulting operation may not be as intended.

It is your responsibility to know where Add-On Instructions are used in logic when you make edits to existing Add-On Instructions.

A confirmation dialog box shows you the impacts of the edits and lets you review the pending changes before confirming or rejecting them.



- An asterisk identifies parameters with changes pending.
- Existing arguments are reset to the parameters they were originally associated with.
- Newly added parameters are inserted with a '?' in the argument field, except for Structured Text, where the field is blank.
- Unknown parameters are created for arguments where associated parameters have been deleted.

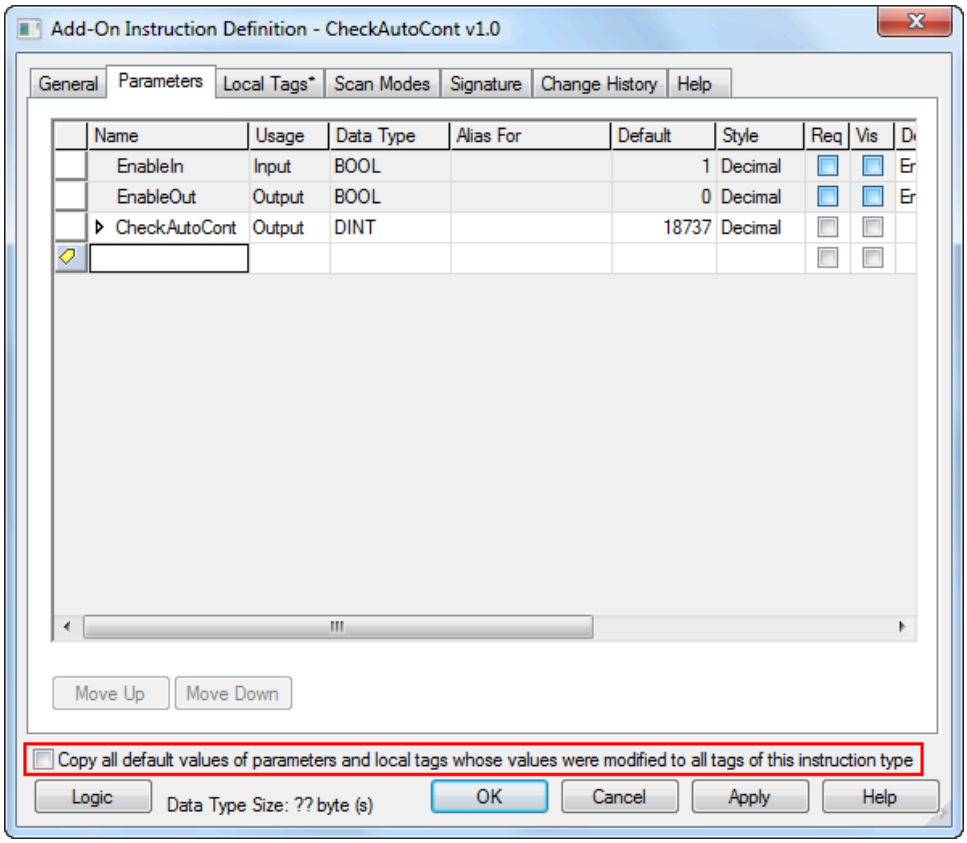
To accomplish this update, Logix Designer application tracks the changes made to the Add-On Instruction parameters from the original instruction to the final version. In contrast, the import and paste processes compare only parameter names to associate arguments with parameters. Therefore, if two different parameters have the same name, but different operational definitions, importing or pasting may impact the behavior of the instruction.

Copy parameter or local tag default values

In RSLogix 5000 software, version 18 or later, you can copy either parameter or local tag default values to all tags of the Add-On Instruction data type or just to specific tags. You can do so only when you are offline.

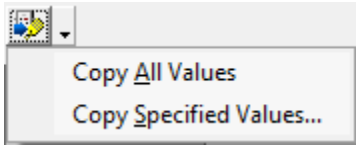
ATTENTION: Values cannot be modified when the instance tags are part of a source-protected Add-On Instruction or you do not have sufficient permission to make edits.

If you change the default values of a parameter or local tag by using the **Add-On Instruction Definition** editor, you can copy the modified values to all of the tags of the Add-On Instruction data type by selecting the **Copy all default values of parameters and local tags whose values were modified to all tags of this instruction type** check box.

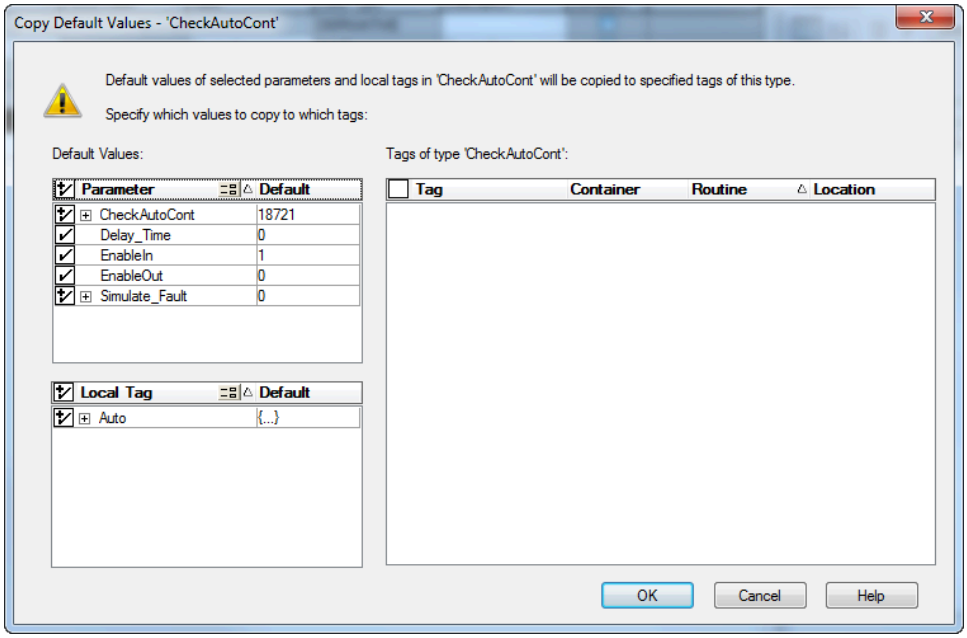


You can also click the copy default values icon to copy default values to all tags the Add-On Instruction data type. The icon appears on the watch pane (as a context menu), data monitor, and logic editor when the Data Context is the Add-On Instruction's definition.

If you want to select which specific tags and values to copy, click the pull-down arrow of the copy default values icon and select **Copy Specified Values**.



The **Copy Default Values** dialog box shows the current default values for the parameters and local tags, and the instance tags where the Add-On Instruction is used or referenced.



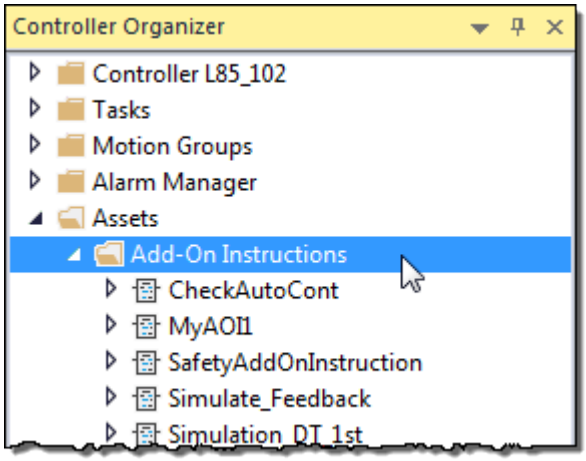
Select the check boxes to select which values to copy to which tags, and click **OK**.

Creating logic for the Add-On instruction

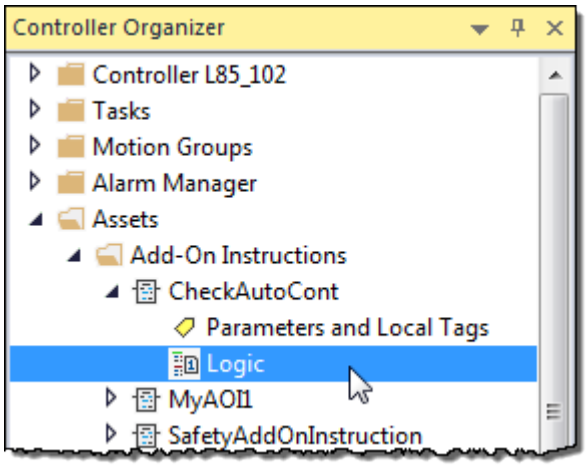
Use a logic editor to create logic for an Add-On Instruction.

To create logic for an Add-On Instruction

1. In the **Controller Organizer**, expand the **Assets** folder and the **Add-On Instructions** folder.



2. Expand the instruction and double-click the logic routine to open it.



3. Edit your logic using the available language editors.

Execution considerations for Add-On Instructions

An Add-On Instruction is executed just like any other routine belonging to a particular program. Because another task can preempt a program containing an Add-On Instruction that is being executed, an Add-On Instruction may be interrupted prior to completing its execution.

In standard programs, 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 uninterrupted before switching to another task.

UID and UIE instructions are not supported in the safety task of GuardLogix projects.

Optimizing performance

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 by 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 user-defined type tag or an atomic tag because 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 the rung condition is false.

Defining operation in different scan modes

To provide Add-On Instructions with the same flexibility as built-in instructions, optional Scan mode routines can be configured to fully define the behavior of the instruction. Scan mode routines do not initially exist for Add-On Instructions. You can create them depending upon the requirements of your instruction.

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

Scan Mode	Description
True	The instruction is scanned as the result of a true rung condition or the EnableIn parameter is set True.
False	The instruction 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(1)	Occurs as a result of an Action in a Sequential Function Chart (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.

[1]Postscan mode routines cannot be created for safety Add-On Instructions because safety instructions do not support SFC.

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.

Scan Mode	Result
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.

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.

Scan Mode	Result
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.

Create a 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 instructions also initialize any tags they 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 mode. For others, special initialization may be done, such as an ONS instruction setting its storage bit during Prescan mode. 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 you want 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.

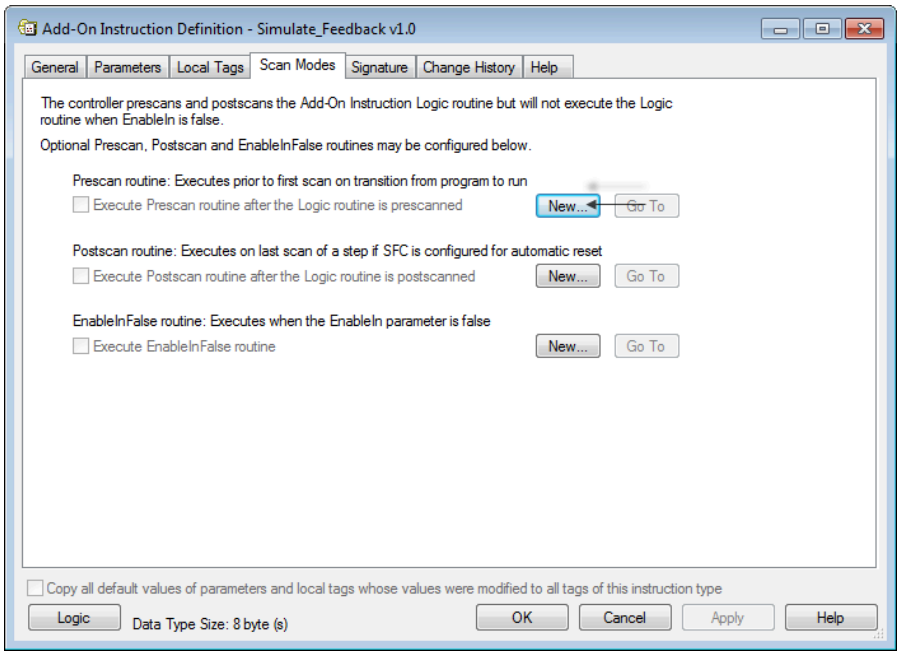
When an Add-On Instruction executes in Prescan mode, any required parameters have their data passed.

- 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.

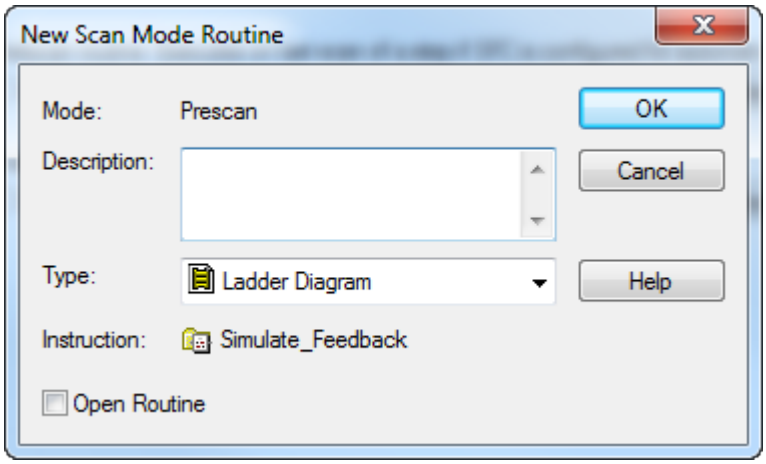
These values are passed even when the rung condition is false in Ladder Diagram or when the instruction call is in a false conditional statement in Structured Text. When Function Block Diagram routines execute, the data values are copied to all wired inputs and from all wired outputs, whether or not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

To create a Prescan routine

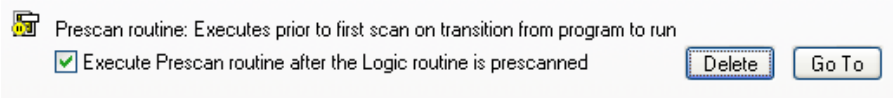
1. In the **Controller Organizer**, right-click an instruction and select **Open Definition**.
2. Click the **Scan Modes** tab.
3. Click **New** for **Prescan routine**.



4. On the **New Scan Mode Routine** dialog box, from the **Type** list, select the type of programming language; **Ladder Diagram**, **Function Block**, or **Structured Text**.



5. In the **Description** box, type 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) by checking or clearing **Execute Prescan routine after the logic routine is prescanned** check box.



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

Create a postscan routine

Postscan mode occurs only for logic in a Sequential Function Chart (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 mode than it has during Prescan mode.

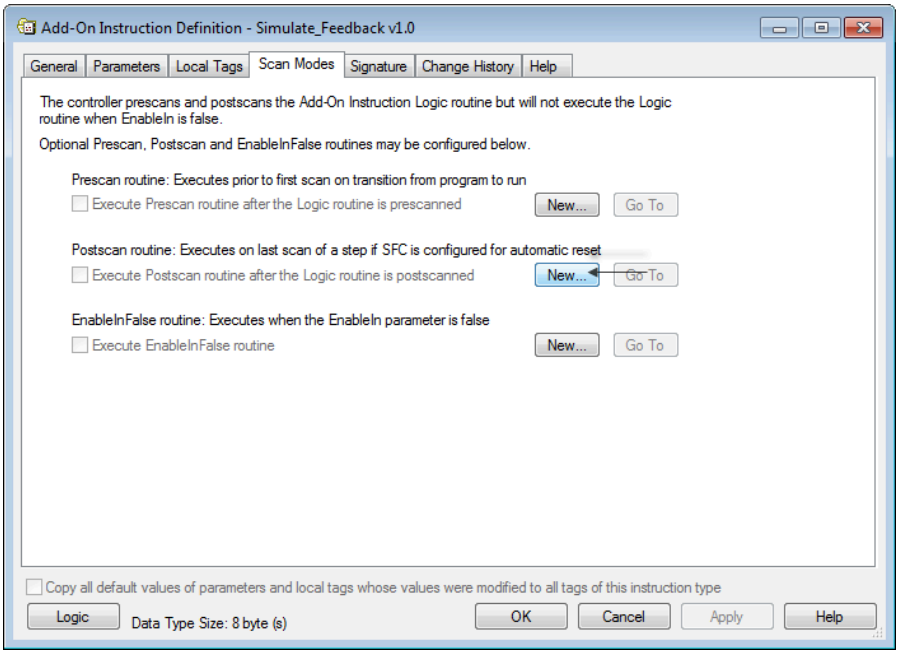
When an Add-On Instruction is called by logic in an SFC Action or a call resides in a routine called by a JSR from an SFC Action, and the Automatic Reset option is set, the Add-On Instruction executes in Postscan mode. The primary logic routine of the Add-On Instruction executes in Postscan mode. Then 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.

Because safety Add-On Instructions cannot be called from an SFC Action, this option is disabled for safety Add-On Instructions.

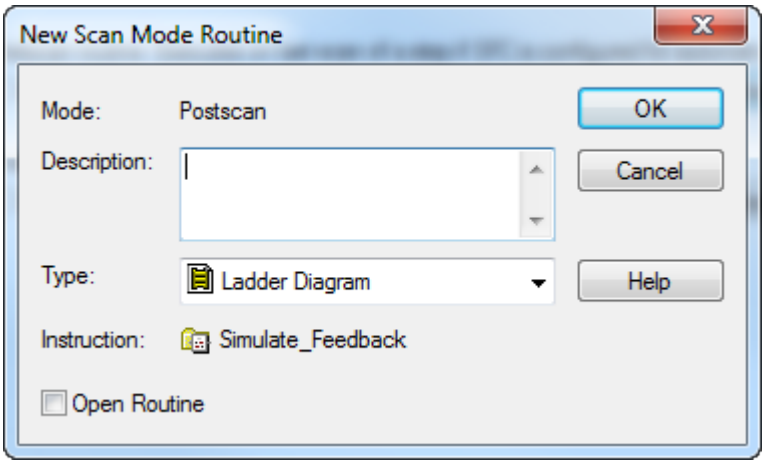
To create a postscan routine

1. In the **Controller Organizer**, right-click an instruction and select **Open Definition**.
2. Click the **Scan Modes** tab.

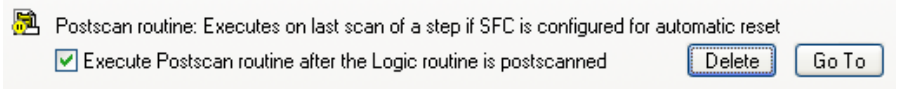
3. Click **New** for **Postscan Routine**.



4. On the **New Scan Mode Routine** dialog box, from the **Type** list, select the type of programming language; **Ladder Diagram**, **Function Block**, or **Structured Text**.



5. In the **Description** box, type 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) by checking or clearing **Execute Postscan routine after the logic routine is postscanned**.



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

Create an 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 OTEs to the de-energized state when the preceding rung conditions are false. An Add-On Instruction can use the EnableInFalse capability to let you 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.

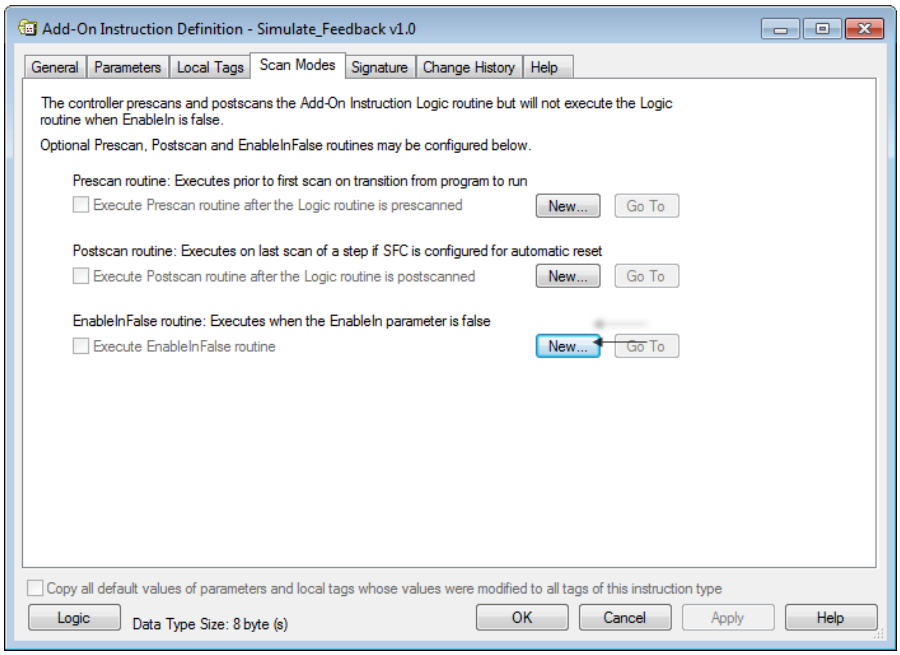
- Values are passed to Input parameters from their arguments in the instruction call.
- Values are passed out of Output parameters from their arguments in the instruction call.

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.

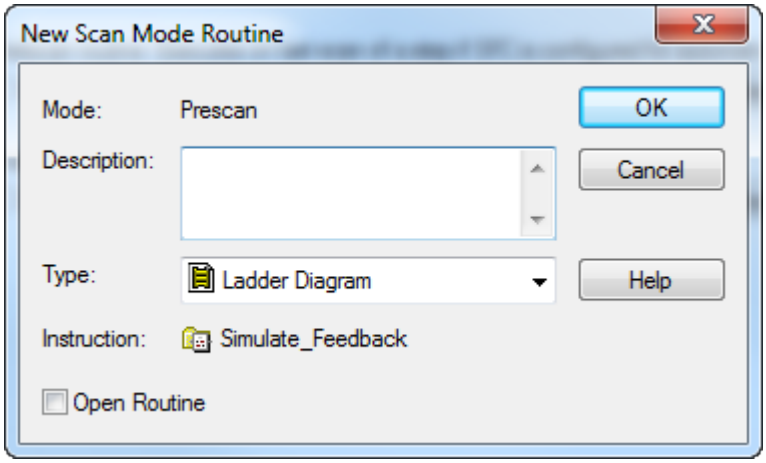
Follow these steps to create an EnableInFalse routine. For more information on other scan mode instructions, see Prescan routine on page 44 and Postscan routine on page 46.

To create an EnableInFalse routine

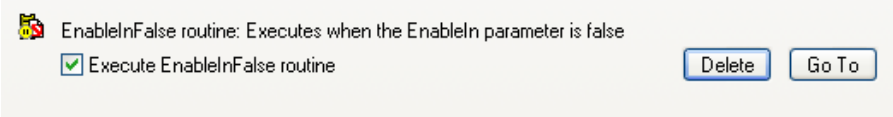
1. In the **Controller Organizer**, right-click an instruction and select **Open Definition**.
2. Click the **Scan Modes** tab.
3. Click **New** on **EnableInFalse routine**.



4. On the **New Scan Mode Routine** dialog box, from the **Type** list, select the type of programming language; **Ladder Diagram**, **Function Block**, or **Structured Text**.



5. In the **Description** box, type the EnableInFalse behavior.
6. Click **OK** to create the routine and return to the **Scan Modes** tab.
7. Define if EnableIn False routine executes (or not) by checking or clearing 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.

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.

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 diagrams

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.

An instruction with only Input parameters, except EnableOut, is treated as an input instruction (left-justified) in a Ladder Diagram. The EnableOut parameter is used for the rung-out 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. Because EnableIn is always True (1) in structured text, the EnableInFalse routine will never execute for an instruction call in structured text.

Change the class of an Add-On Instruction

You can change the class of a safety Add-On Instruction so that it can be used in a standard task or standard controller.

To change the class of an Add-On Instruction

- You can change the class in a safety project if the instruction does not have an instruction signature, you are offline, the application does not have a safety task signature, and is not safety-locked.
- You can also change the class from standard to safety so that the Add-On Instruction can be used in the safety task.

Changing the class of an Add-On Instruction results in the same class change being applied to the routines, parameters, and local tags of the Add-On Instruction. The change does not affect nested Add-On Instructions or existing instances of the Add-On Instruction.

If any parameters or tags become unverified due to the change of class, they are identified on the **Parameters** and **Local Tags** tabs of the Add-On Instruction Editor.

If any of the restrictions for safety Add-On Instructions are violated by changing the class from standard to safety, one of the following errors is displayed and the change does not succeed:

- Routines must be of Ladder Diagram type.
- Safety Add-On Instructions do not support the Postscan routine.
- One or more parameters or local tags have an invalid data type for a safety Add-On Instruction.

You must edit the parameter, tag, or routine types before the class change can be made.

If the safety controller project contains safety Add-On Instructions, you must remove them from the project or change their class to standard before changing to a standard controller type.

Testing the Add-On Instruction

You need to test and troubleshoot the logic of an instruction to get it working.

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

Prepare to test an Add-On Instruction

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

To prepare to test an Add-On Instruction

1. Open a project to debug offline.

   Add-On Instructions can only be created or modified when offline. You can add, delete, or modify tag arguments in calls to Add-On Instructions while editing online, but you cannot edit arguments inside the Add-On Instruction while online.

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

Test the flow

Follow these steps to test the flow of an Add-On Instruction.

To 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

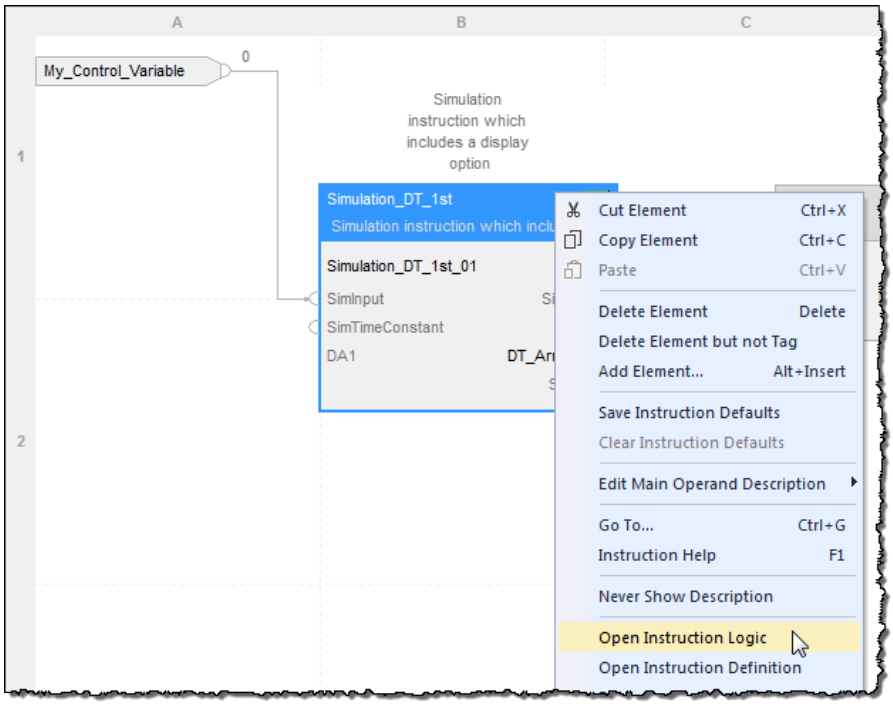
You can simplify the online monitoring and troubleshooting of your Add-On Instruction by using Data Context views. The Data Context selector lets you select a specific call to the Add-On Instruction that defines the calling instance and arguments whose values are used to visualize the logic for the Add-On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

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.

To monitor logic with data context views

Follow these steps to monitor the logic.

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.

Verifying individual 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.

Instruction	Description
True	This is simply the execution of the main logic routine.
False	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. In a Function Block target routine, this entails executing an instance of the instruction with the EnableIn parameter set to zero (0).
Prescan	Place the controller in Program mode, then place it in Run mode.
Postscan	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.

Source protection for an Add-On Instruction

You can apply source protection to your Add-On Instruction to protect your intellectual property or prevent unintended edits of a validated source.
Cannot modify the Source Protection settings if the Add-On Instruction is sealed. To source protect and seal an Add-On Instruction, apply the source protection settings before sealing.

Source protection limits user access to your Add-On Instruction or blocks access to the internal logic or local tags used by the instruction. You can protect Add-On Instructions using Source Key protection or License protection. You can also apply Execution Protection to source-protected components to allow execution only on controllers with a specific execution license.

Source Key protection:

- Protects components using existing source keys.

	Tip: You can optionally allow source-protected components to be available in a read-only format on a system that does not have the source key required for access.
	Tip: Apply Source Key protection before generating an instruction signature for your Add-On Instruction definition. You will need the source key to create a signature history entry. When source protection is enabled, you can still copy the instruction signature or signature history, if they exist.

License-based protection:

- Protects components with specific licenses.

License-Based Source Protection is not supported on Sequential Function Chart routines in version 30 of the Logix Designer application.

- Execution Protection is an extension of License-Based Source Protection. You can apply Execution Protection to limit the execution of routines and Add-On Instructions, including equipment phase state routines, to controllers that contain a specific execution license.
- When you protect a component with License-Based Source Protection, you can also lock it.
  When locking a component, the routine's logic is compiled into executable code and encrypted. The code is decrypted by the controller when it is ready for execution. As a result, sharing project files containing locked components with users without licenses to use the locked components is possible. Those users can use unprotected parts of the project, upload and download the project file, and copy and paste locked components into other project files. However, if a component is protected using the **Protect with**

controller key and specific license option, executing the project requires an SD card with the correct execution license.

Execution Protection and component locking is supported only on CompactLogix 5380, CompactLogix 5480, and ControlLogix 5580 controllers in version 30 of the Logix Designer application.

Enable the source protection feature

If source protection is unavailable and not listed in your menus, follow the instructions for enabling Source Protection in the *Logix5000 Controllers Security Programming Manual*, publication 1756-PM016.

The Security Programming manual provides detailed instructions for configuring Source Protection for routines and Add-On Instructions.

Generating an Add-On Instruction signature

The **Signature** tab on the Add-On Instruction Definition Editor lets you manage the instruction signature, create signature history entries, and view the safety instruction signature, if it exists. Instruction signatures are applied to the definition of the Add-On Instruction. All instances of that Add-On Instruction are sealed when the signature is applied.

Generate, remove, or copy an instruction signature

Use this procedure to generate, remove, or copy an instruction signature

To generate, remove, or copy an instruction signature

1. On the **Signature** tab in the **Add-On Instruction Definition Editor**, click **Generate** to create an instruction signature or **Remove** to delete the instruction signature.

   You must be offline to generate or remove an instruction signature. Both actions change the **Last Edited Date**.

IMPORTANT: If you remove an instruction signature when the Add-On Instruction also has a safety instruction signature, the safety instruction signature is also deleted.

1. Click **Copy** to copy the instruction signature and the safety instruction signature, if it exists, to the clipboard to facilitate record-keeping.

IMPORTANT: If an invalid instruction signature is detected during verification, an error message indicates that the signature is invalid. You must remove the instruction signature, review the Add-On Instruction, and generate a new instruction signature.

The instruction signature is not guaranteed to be maintained when migrating between major revisions of the Logix Designer application.

Create a signature history entry

The signature history provides a record of signatures for future reference. A signature history entry consists of the name of the user, the instruction signature, the timestamp value, and a user-defined description. You can only create a signature history if an instruction signature exists and you are offline. Creating a signature history changes the Last Edited Date, which becomes the timestamp shown in the history entry. Up to six history entries may be stored.

To create a signature history entry

1. On the **Signature** tab on the Add-On Instruction Definition Editor, click **Add to History**.
2. In the **Create History Entry** description box, type up to 512 characters long, for the entry.
3. Click **OK**.

To facilitate record-keeping, you can copy the entire signature history to the clipboard by selecting all the rows in the signature history and choosing **Copy** from the **Edit** menu. The data is copied in tab separated value (TSV) format.

To delete the signature history, click **Clear Signature History**. You must be offline to delete the Signature History.

Generate a Safety Instruction Signature

When a sealed safety Add-On Instruction is downloaded for the first time, a SIL 3 safety instruction signature is automatically generated. Once created, the safety instruction signature is compared at every download.

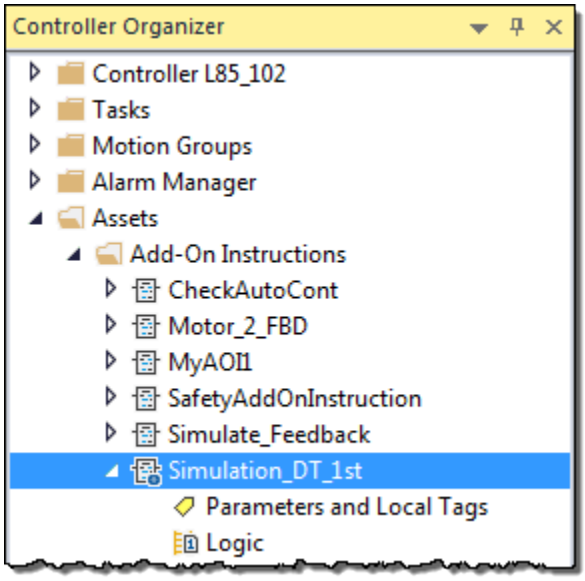
If Logix Designer application detects an invalid safety instruction signature value, it generates a new safety instruction signature value in the offline project and displays a warning indicating that the safety instruction signature was changed. The safety instruction signature is deleted if the instruction signature is removed.

IMPORTANT: After testing the safety Add-On Instruction and verifying its functionality, you must record the instruction signature, the safety instruction signature and the timestamp value. Recording these values will help you determine if the instruction functionality has changed.

Refer to the safety reference manual for your controller, listed in the Additional resources on page 9, for details on safety application requirements.

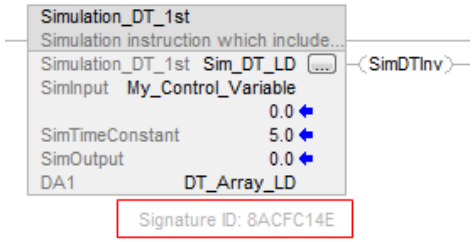
View and print the instruction signature

When the instruction signature has been generated, Logix Designer application displays the instruction with the blue seal icon in the **Controller Organizer**, on the Add-On Instruction title bar, and in the Logic Editor.

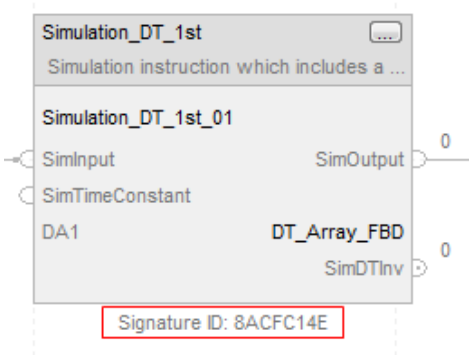


When an instruction is sealed, the instruction signature is displayed on the faceplate of the instruction in the Ladder Diagram Editor and the Function Block Diagram Editor.

Ladder editor



Function Block Diagram editor



To view and print the instruction signature

- Turn off the display of the instruction signature in the **Workstation Options** dialog box of the Logix Designer application.
- View the instruction signature and the safety instruction signature on the **Quick View** pane of the **Controller Organizer** and on the **Signature** tab of the Instruction Definition Editor.

  The Add-On Instruction name, revision, instruction signature, safety instruction signature, and timestamp are printed on the Add-On Instruction Signature Listing report.

The first 32 bits of an AOI's signature (Safety ID) as it appears in the **Quick View Pane** and in the signature report are comparable to the AOI's signature as it appears on the **AOI Properties > Signature** tab.

- Include the instruction signature, safety instruction signature, and signature history on the Add-On Instruction report by clicking **Print Options** on the **Generate Report** dialog box.

Create an alarm definition

Use tag-based alarms and alarm definitions to notify users of conditions that they might need to respond to, such as temperature over-limit, excessive current, or a motor failure. A tag-based alarm is similar to an instruction-based alarm (ALMA and ALMD instructions) in that it monitors a tag value to determine the alarm condition. However, a tag-based alarm is not part of the logic program and does not increase the scan time for a project.

An alarm definition is associated with an Add-On Instruction or a defined data type. When a tag is created using a data type or an Add-On Instruction that has alarm definitions, alarm conditions are created automatically based on the alarm definitions.

Tag-based alarms and alarm definitions are supported only on CompactLogix 5380, CompactLogix 5480, and ControlLogix 5580 controllers.

To create an alarm definition:

1. On the **Controller Organizer**, right-click the **Alarms** folder and select **New Alarm Definition**.

   You can also right-click a scalar tag or parameter of an Add-On Instruction and select **Add Alarm Definition**.
2. In the **Name** box, enter a name for the alarm definition.
3. In the **Input** box, add an input tag.
4. To enable all instances of the alarm, select the **Required to be used and evaluated** check box. All new alarms definitions are disabled by default.
5. Use the tabs on the **New Alarm Definition** dialog box to configure additional settings:
   - **General** - Used to configure the name, input tag, trigger condition, timing, severity, message, associated tags, and shelving settings.
   - **Class/Group** - Used to identify the class and group name for the alarm definition.

- **Advanced** - Used to assign additional settings to the alarm definition, such as an FactoryTalk View command to run in response to an alarm, latch state, acknowledgment requirement, alarm set inclusion, and use of the alarm.

Alarm definitions associated with the tag are not included when an Add-On Instruction tag is copied and pasted in a project.

After you copy and paste an Add-On Instruction tag, open the **Alarm Definition list** and copy and paste the alarm definition for the tag. Follow these steps:

1. In the **Controller Organizer**, right-click **Alarms** and select **Edit Alarm Definitions**.
2. Right-click the alarm definition for the Add-On Instruction tag and select **Copy**.
3. Right-click again and select **Paste**. The alarm definition is pasted into the list with **_000** added to the alarm name.
4. Double-click the copy of the alarm definition to open the **Alarm Definition Properties** dialog box.
5. In the **Input** box, change the input tag to the Add-On Instruction tag that you copied and pasted.

Access attributes from Add-On Instruction alarm sets

The alarms contained in an Add-On Instruction definition, a structured tag of an Add-On Instruction definition, or an array tag of an Add-On Instruction definition can be referenced as an alarm set. Use these alarm set attributes as operands in logic.

When you reference an attribute from an individual alarm, you insert the owner of the alarm in the operand syntax. Similarly, when you reference an attribute from an Add-On Instruction alarm set, you insert the alarm set container (the AOI definition, AOI structured tag, or AOI array tag) in the operand syntax.

To access attributes from an Add-On Instruction alarm set

1. In the logic editor, create an instruction.
2. On the instruction operand that accesses an Add-On Instruction alarm set attribute, enter the following syntax:
   - The container for the alarm set.
   - @AlarmSet
   - The attribute to access.

   To access an attribute of an alarm set in a container that is also the Add-On Instruction in which you are entering the logic, enter THIS, followed by a period. For alarm definitions associated with a nested Add-On Instruction, the alarm definition attributes can be accessed programmatically through the nested Add-On Instruction.

   The following table lists example syntax.

To access:	Alarm set container	Syntax
Attribute of an alarm set in a container that is also the AOI in which you are entering the logic	MyAOI	THIS.@Alarms.FailToOpen.InAlarm

To access:	Alarm set container	Syntax
Attribute of an alarm set in a container that is an AOI definition	MyTank	MyTank.@Alarms.FailToClose.AckRequired
Attribute of an alarm set in a container that is an AOI array tag	MyTank[3]	MyTank[3].@Alarms.FailToOpen.AckRequired
Attribute of an alarm set in a container that is an AOI structured tag	MyTank.MyValve	MyTank.MyValve.@Alarms.FailToClose.AckRequired

In this version of the Logix Designer application, the **@Alarms** and **@AlarmSet** syntax is not supported in the following instructions:

- CMP
- CPT
- FAL
- FSC

The following example shows how inserting an **MOV** instruction allows the **@Alarms** and **@AlarmSet** syntax to work with CMP, CPT, FAL, and FSC instructions.

Unsupported expression:

CPT(Tag1, RightValve.ValveTimer.@Alarms.TM_ACC_1.Severity + Tag2)

Supported expression:

MOV(RightValve.ValveTimer.@Alarms.TM_ACC_1.Severity, MyIntermediateTag)

CPT(Tag1, MyIntermediateTag + Tag2)

In this version of the Logix Designer application, in the Structured Text editor, the **@Alarms** and **@AlarmSet** syntax is supported only in simple arithmetic expressions, such as a + b. The following example shows how inserting an additional step allows creation of more complex arithmetic expressions.

Unsupported expression	Alternative
cTag1 := cTag1.@Alarms.new.Severity + ::THIS.@AlarmSet.DisabledCount + 1;	cTag1 := cTag1.@Alarms.new.Severity + ::THIS.@AlarmSet.DisabledCount; cTag1 := cTag1 + 1;

Creating instruction help

Custom instruction help is generated automatically as you are creating your Add-On Instructions. Logix Designer application automatically builds help for your Add-On Instructions by 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. You can provide additional help documentation by entering it on the **Help** tab on 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.

Write clear descriptions

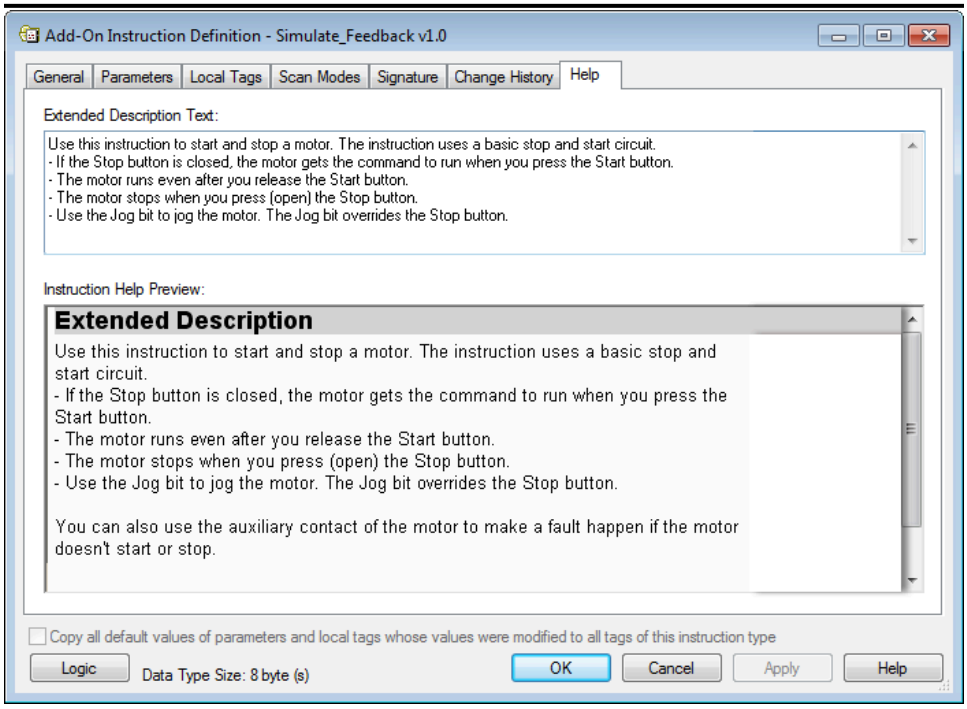
When writing your descriptions keep the following in mind.

To write clear descriptions

- Use short sentences and simple language.
- Be brief and direct when you write.
- Include simple examples.
- Proofread your entries.

This is an example of the **Extended Description Text** field in the **Help** tab on the Add-On Instruction Definition Editor. This area lets you create directions on how to use and troubleshoot your instruction. The **Instruction Help Preview** window shows how your text will look as generated instruction help.

When you are typing 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.

To document an Add-On Instruction

1. Right-click an **Add-On Instruction** and select **Open Definition**.
2. On the **General** tab, in the **Extended Description Text** box, type a description and a revision note for the Add-On Instruction to explain the purpose of the instruction.
3. Click the **Parameters** tab in the **Description** box, type 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. In the **Description** box, type a 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 the Prescan routine, such as initialization of any parameters.

d. For the Postscan routine (if one exists), briefly describe actions that will take place during the Postscan routine, such as initialization of any parameters resetting any internal state of the instruction.

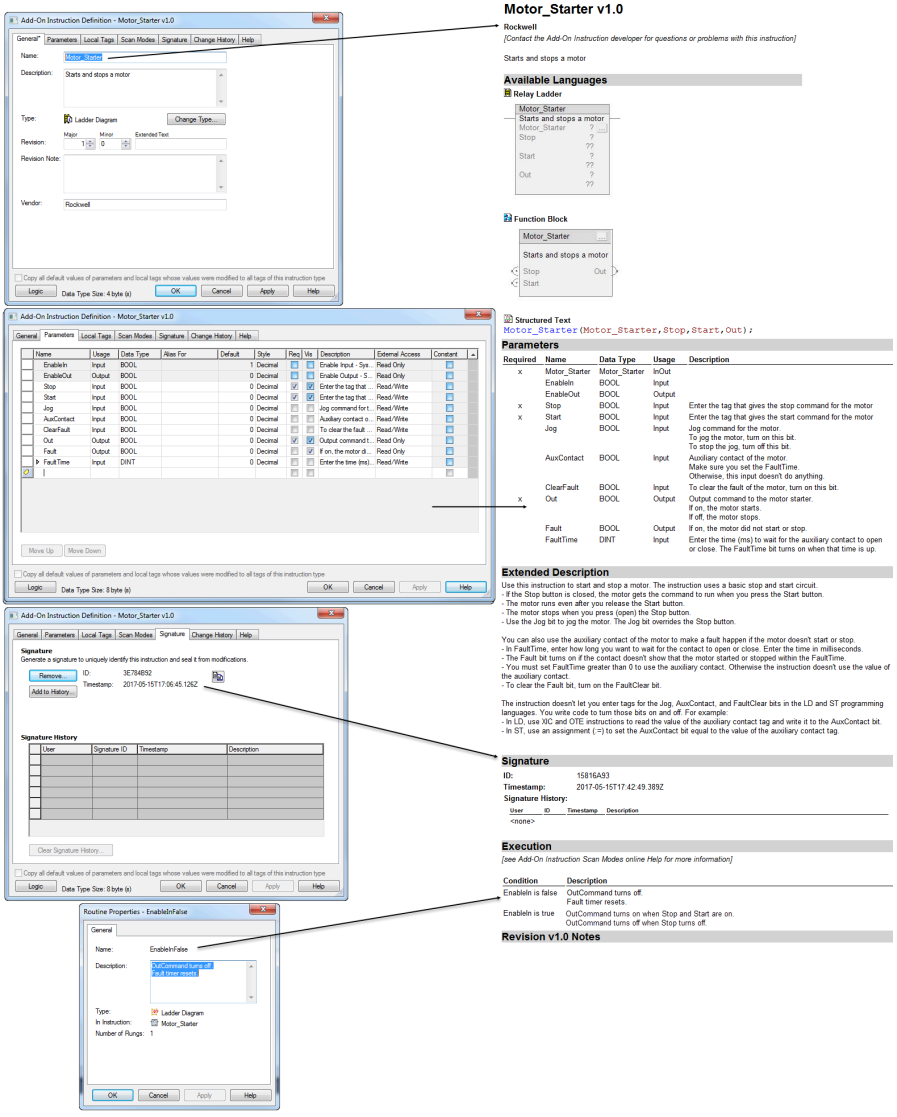
1. Click the **Help** tab on the Add-On Instruction Definition Editor and type additional information in the **Extended Description** field.

   The extended description can include the following information:

   - Additional parameter information
   - Description of how the instruction executes
   - Change history notes

2. Review the Help format in the preview window.

This is an example of the Logix Designer application generated help for the instruction. This information is gathered from the definition descriptions that you complete when defining an instruction.



Project documentation

With RS Logix 5000 software, version 17 and later, you have the option to display project documentation, such as tag descriptions and rung comments in 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 project, either when programming in that language or by using the import/export utility to translate the documentation offline and then import it back into the project. Once you enable project documentation in application, you can dynamically switch between languages as you use the software.

Project documentation that supports multiple translations includes these variables:

- Component descriptions in tags, routines, programs, equipment phases, user-defined data types, and Add-On Instructions
- Engineering units and state identifiers added to tags, user-defined data types, or Add-On Instructions
- Trends
- Controllers
- Alarm Messages (in configuration of ALARM_ANALOG and ALARM_DIGITAL tags)
- Tasks
- Property descriptions for module in the **Controller Organizer**
- Rung comments, SFC text boxes, and FBD text boxes

If you want to allow project documentation on an Add-On Instruction that is sealed with an instruction signature, you must enter the localized documentation into your Add-On Instruction before generating the signature. Because the signature history is created after the instruction signature is generated, the signature history is not translatable. If the translated information already exists when you generate the Add-On Instruction signature, you can switch the language while keeping the signature intact because the switch does not alter the instruction definition, it only changes the language that is displayed.

For more information on enabling a project to support multiple translations of project documentation, refer to the online help.

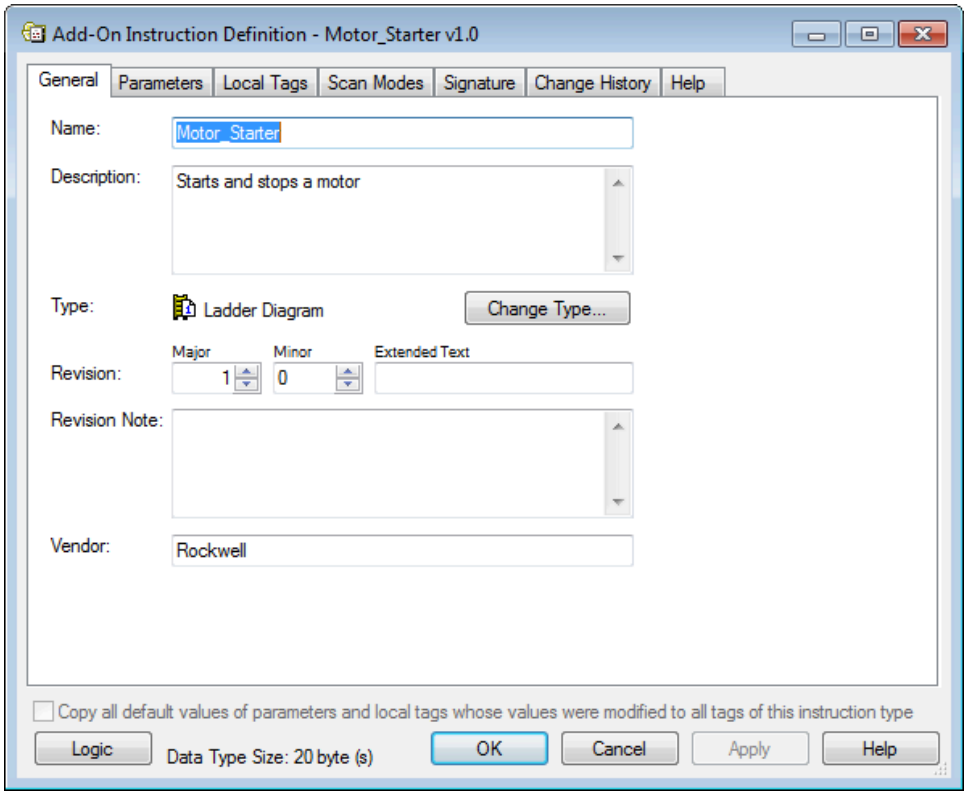
Motor starter instruction example

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

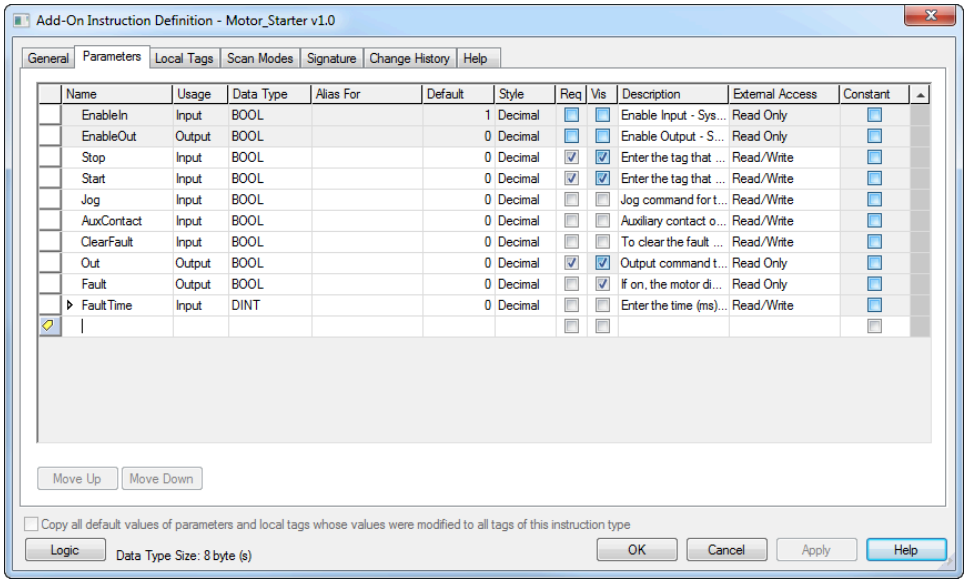
If the stop pushbutton is closed and the start pushbutton is pressed then:

- The motor gets the command to run.
- The instruction seals in the command, so the motor keeps running even after you release the start pushbutton.

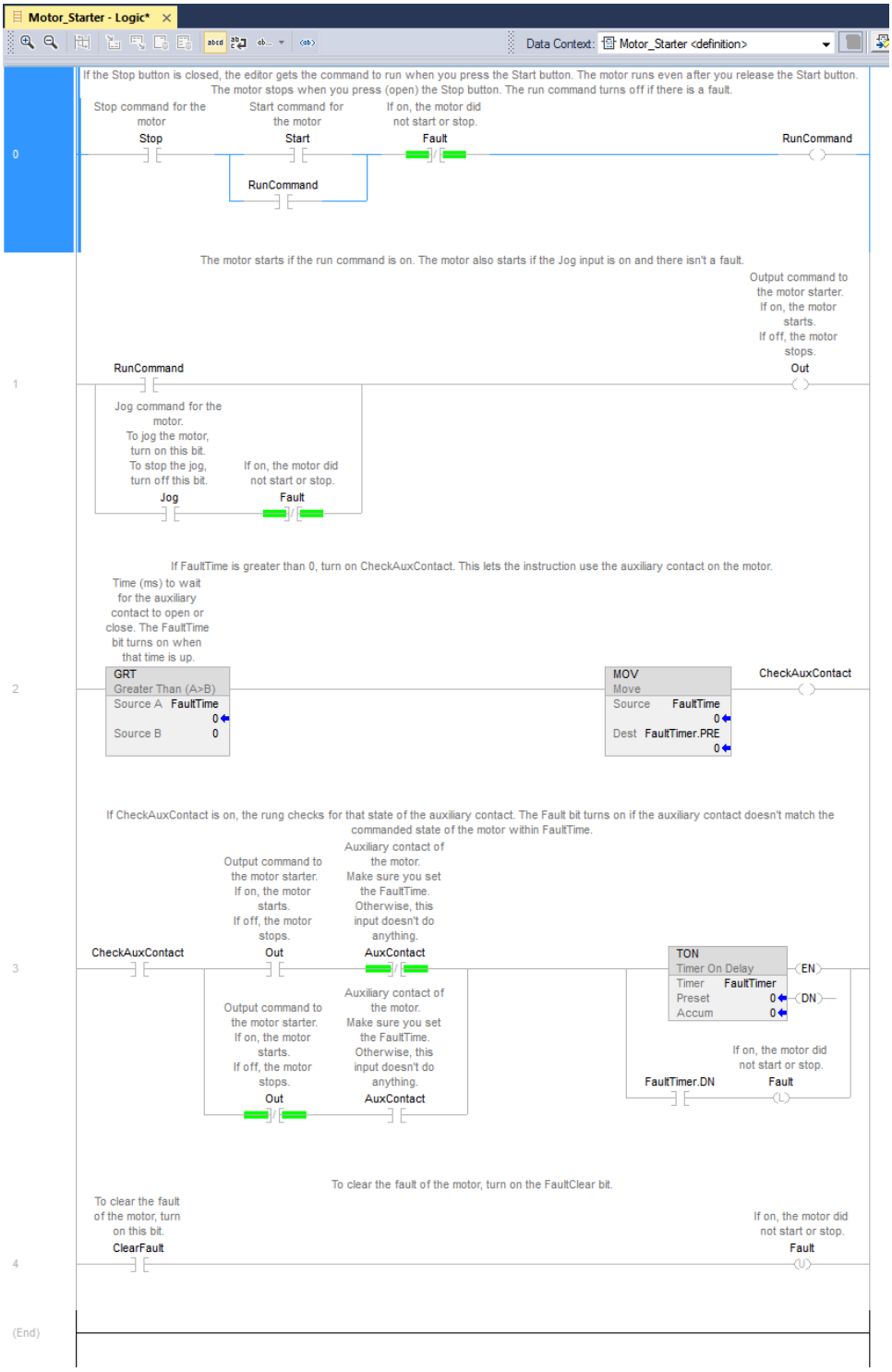
If the stop pushbutton is pressed (opened), then the motor stops. The following screen capture shows the **General** tab for the Motor Starter Add-On Instruction.



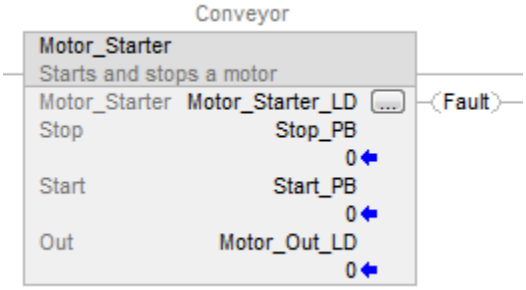
The following screen capture shows the **Parameters** tab for the Motor Starter Example Definition Editor.



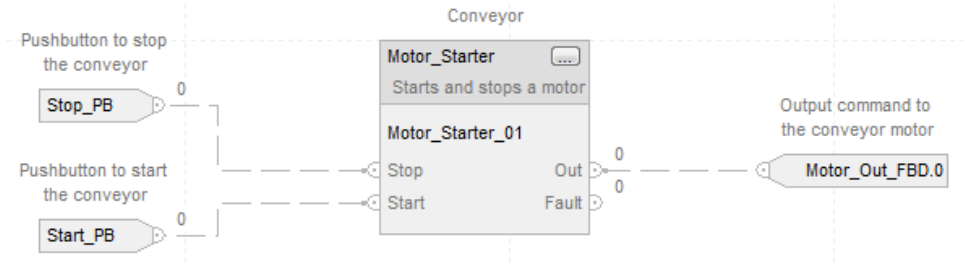
The following screen capture shows the Motor Starter Example ladder logic.



The following diagrams show the Motor Starter instruction called in three different programming languages. First is Motor Starter Ladder Logic.



Here is the Motor Starter Function Block Diagram.

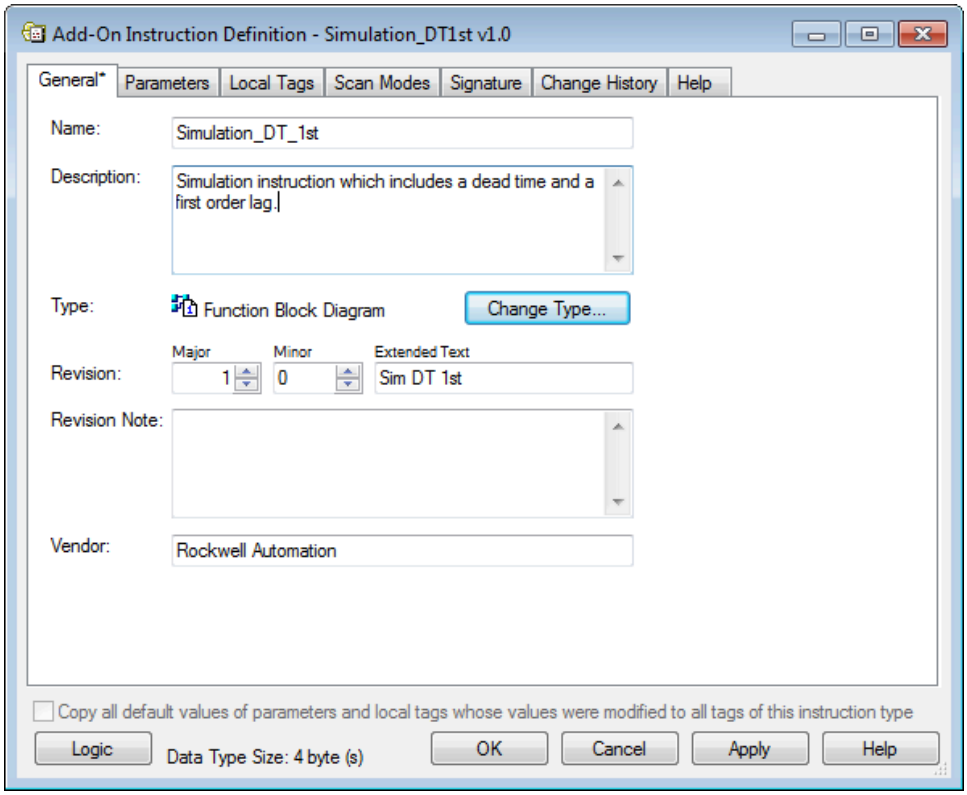


Here is the Motor Starter Structured Text.

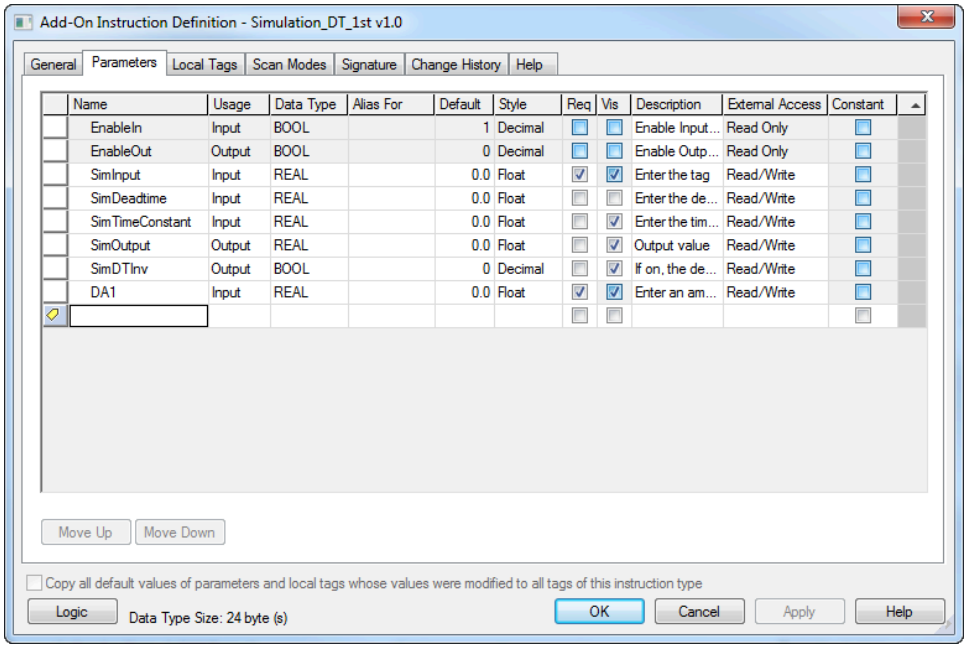
```
Motor_Starter(Motor_Starter_ST, Stop_PB, Start_PB, Motor_Out_ST);
```

Simulation instruction example

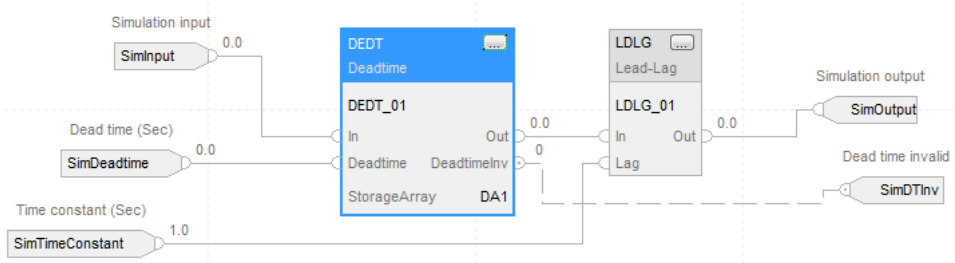
The Simulation_DT_1st Add-On Instruction adds a dead time and a first-order lag to an input variable. The following screen capture shows the **General** tab for the Simulation Example Definition Editor.



The following screen capture shows the **Parameter** tab for the Simulation Example Definition Editor.



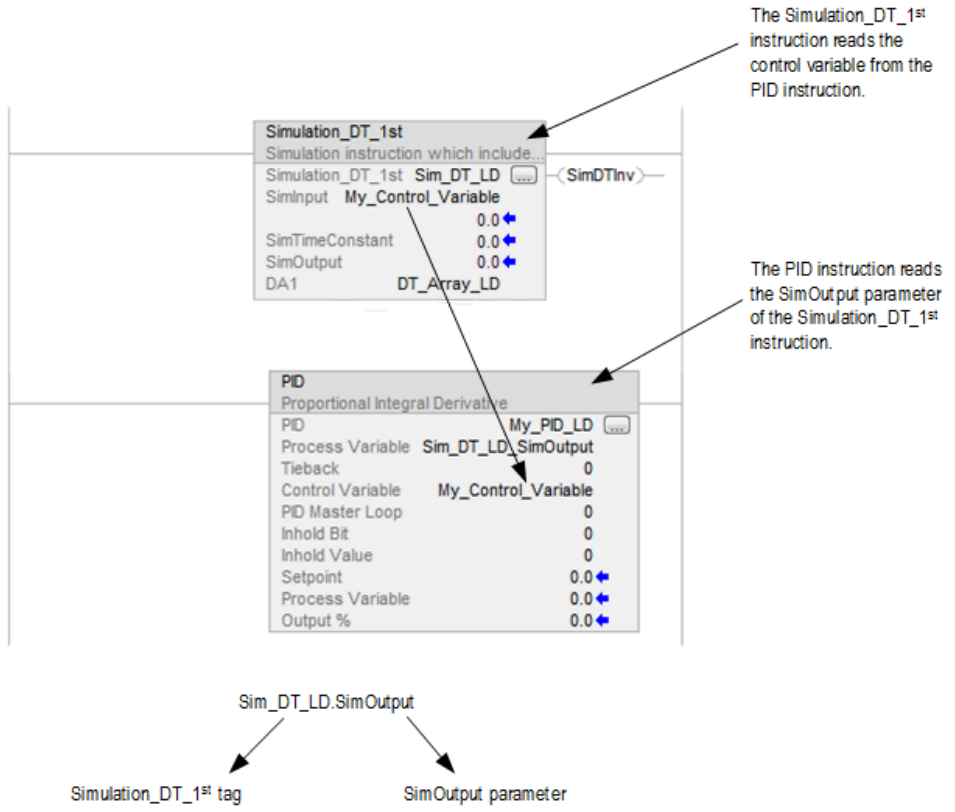
The following image shows the Simulation example logic.



Ladder diagram configuration

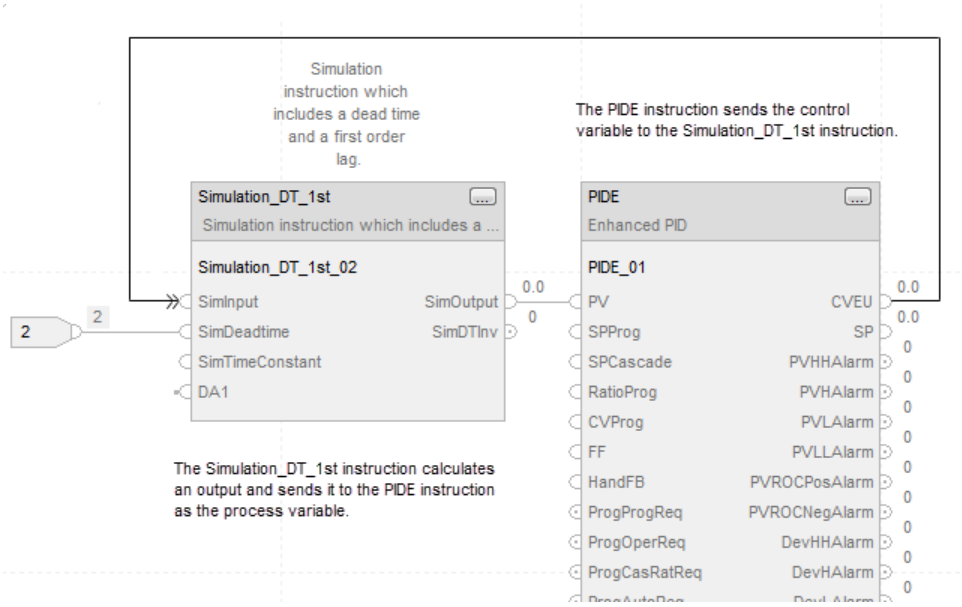
In this example, the instruction simulates a deadtime and lag (first order) process.

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.

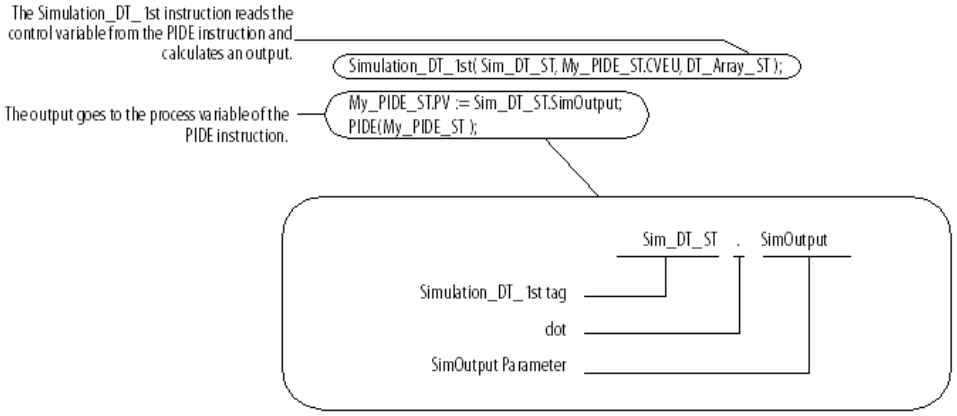


Function block diagram configuration

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



Using Add-On Instructions

Introduction

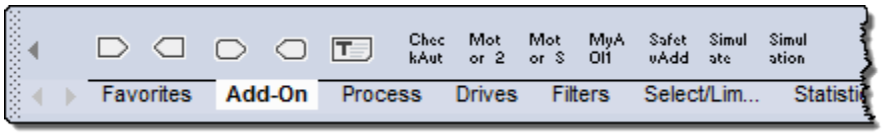
Add-On Instructions are used in your routines like any built-in instructions. You add calls to your instruction and then define the arguments for any parameters.

Access Add-On Instructions

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 conforms to the language in which it is placed.

To access Add-On Instructions

- Access them 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.



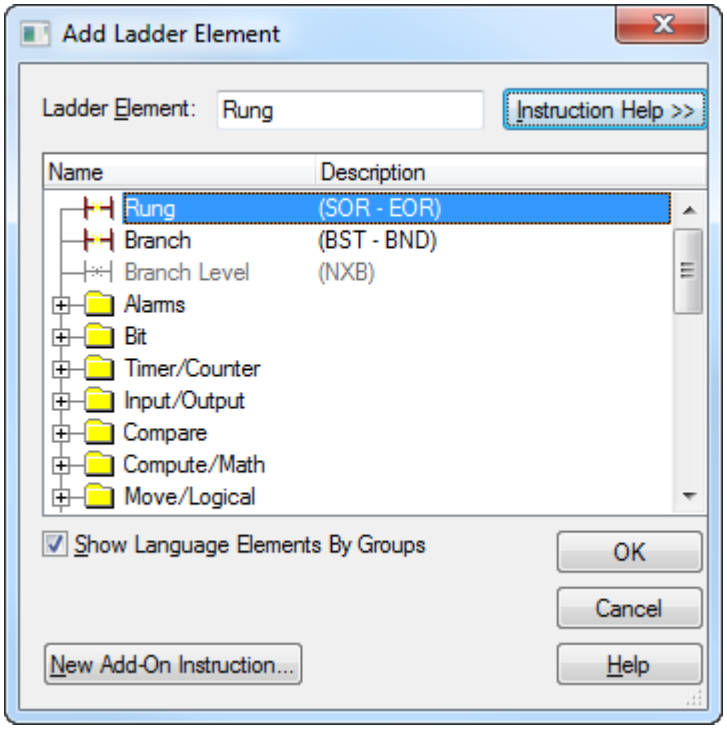
IMPORTANT: Safety Add-On Instructions can be used only in safety routines, which are currently restricted to ladder logic. Safety Add-On Instructions are shown in the **Language Element** toolbar only when the routine is a safety routine.

Use the Add Ladder Element dialog box

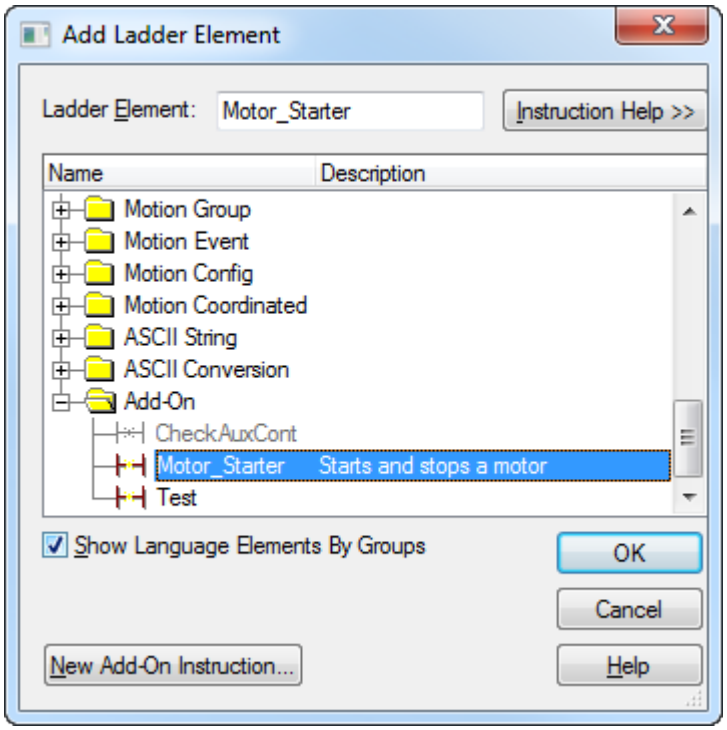
Use the **Add (language) Element** dialog to add elements to add an element to logic.

Use the Add (language) Element dialog

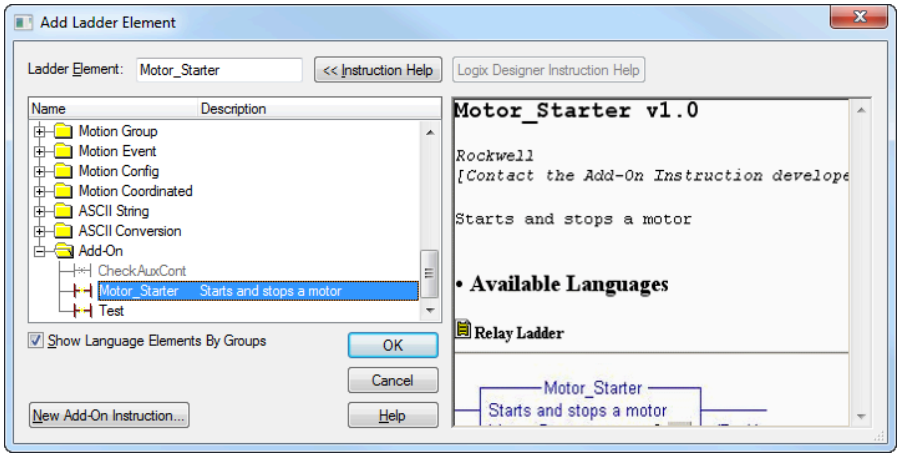
1. Press Alt + Insert anywhere in the editor or right-click the logic in the Editor and select **Add Element**.



2. From the **Element** list, select the Add-On Instruction you want to add to your routine.



3. Click **Instruction Help** to display the instruction help for any instruction in the browser.



4. Click **OK**.

Including an Add-On Instruction in a routine

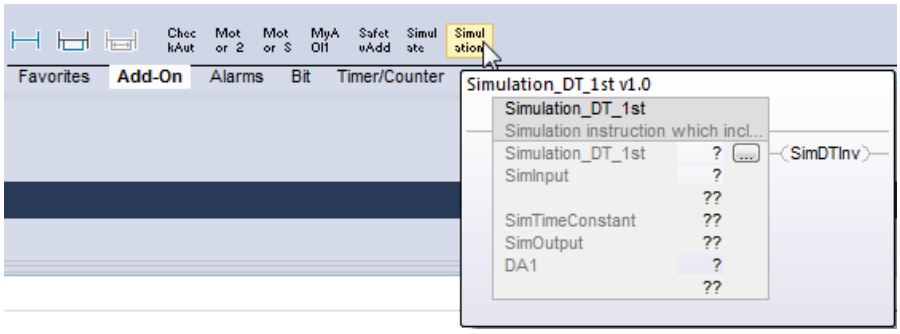
Follow this procedure when you want to use an Add-On Instruction in one of your routines.

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:

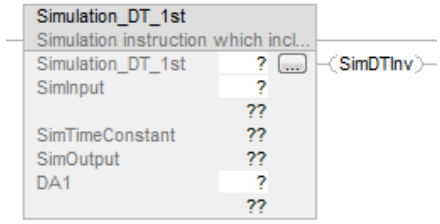
   - Create the instruction in your project.
   - Copy and paste an instruction into your project.
   - Get the file for an exported instruction definition and then import the instruction into your current project.

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 to insert it in the editor. Hover the cursor on an instruction to see a preview of it.

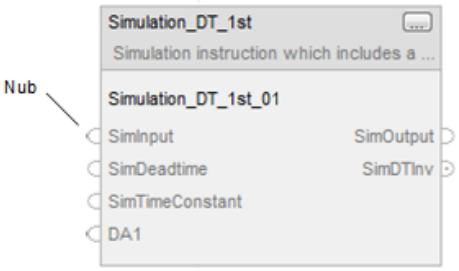


5. Define arguments for each Parameter on the instruction call.

   The instruction appears as follows in each of the languages.

Ladder Diagram:	

Parameter With	Description
Single question mark	This is a required InOut parameter. Enter a tag.
Single and double question marks	This is a required Input or Output parameter. Enter a tag.
Double question marks	This is not a required parameter. You can either: • Leave as is and use the default value. • Enter a different value if it's an Input parameter.

Function Block Diagram:	

Item	Description
Nub on the end of pin	This is a required Input or Output parameter. You must wire the pin to an IREF, OREF, connector, or another block to verify.
Single question mark	This is a required InOut parameter. Enter a tag.
No nub on the end of pin	This is not a required parameter. You can either: • leave as is and use the default value. • enter a different value if it's an Input parameter.

NOTE: The instruction expects arguments for required parameters as listed in the instruction tooltip.

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.

Track an Add-On Instruction

Use component tracking to determine whether tracked components have been changed. The Logix Designer application creates an overall tracked value to indicate the current state of tracked components.

Tracked components and their current states appear in the **Tracked Components** dialog box, which is accessible on the **Controller Properties dialog box - Security** tab. The recommended limit on the number of Add-On Instructions that can be tracked is 100. If this limit is exceeded, there might be a noticeable impact on performance in the Logix Designer application.

The FactoryTalk Security permission **Add-On Instruction: Modify** controls a user's ability to change the tracking status for an Add-On Instruction.

- Component tracking is supported only on CompactLogix 5370, ControlLogix 5570, Compact GuardLogix 5370, and GuardLogix 5570 controllersin version 30 of the Logix Designer application.
- To optimize performance, configure component tracking so that the tracked state value is calculated on demand rather than at regular intervals.

To track an Add-On Instruction

1. In the **Controller Organizer**, highlight the component to track.
2. Right-click and select **Include in tracking group**.
3. To stop tracking a component, right-click and select **Include in tracking group** again.

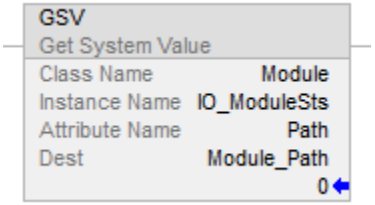
Reference a hardware module

A module reference parameter is an InOut parameter of the MODULE data type that points to the Module Object of a hardware module. You can use module reference parameters in both Add-on Instruction logic and program logic. Since a module reference parameter is passed by reference, it can access and modify attributes in a hardware module from an Add-On Instruction. Follow this procedure to use a module reference parameter from within the Add-On Instruction. This example shows how to retrieve the communication path for a hardware module.

To reference a hardware module

1. Create the module reference parameter in the Add-On Instruction. See Creating a module reference parameter on page 36.
2. Create a SINT tag in the Add-On Instruction to hold the module communication path.
3. Add a GSV instruction in the Add-On Instruction, using the programming language you chose for the Add-On Instruction. The GSV instruction allows you to retrieve module information.
4. In the GSV instruction, choose the following values to retrieve the communication path to the module.

Attribute	Value
Class Name	Module
Instance Name	The module reference parameter you created in the Add-On Instruction (IO_ModuleSts below)
Attribute Name	Path. This is the communication path to the module.
Dest	The tag to hold the module path (Module_Path below)

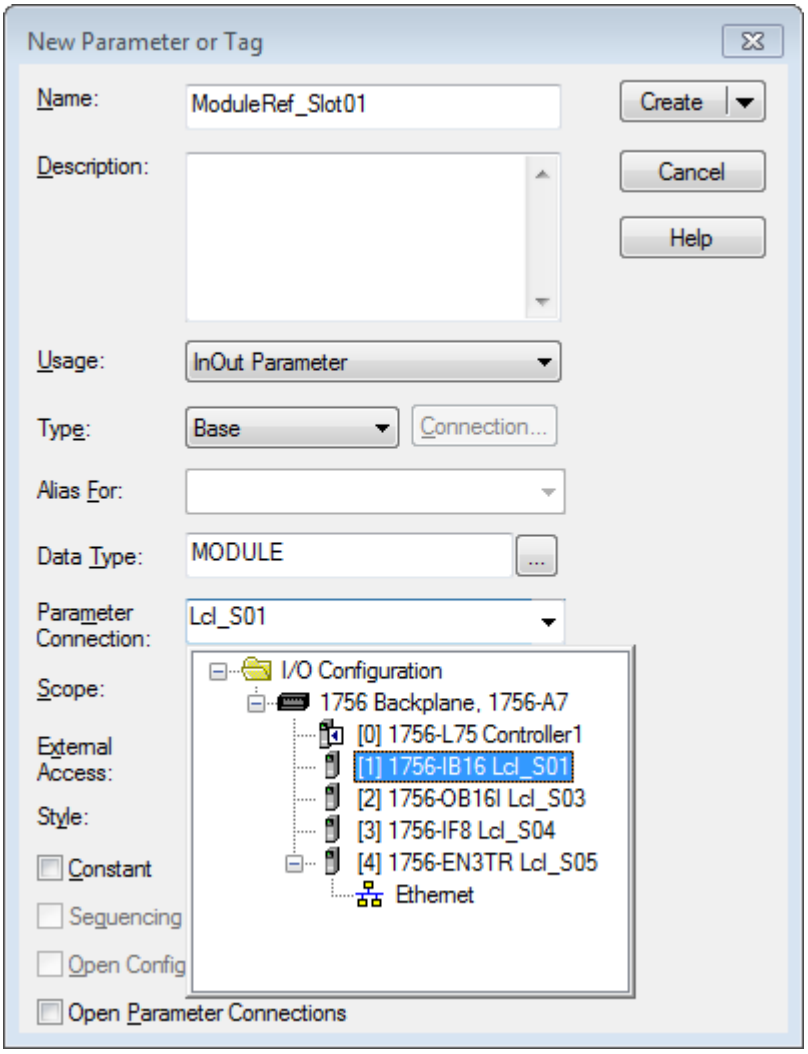


1. In the routine that includes the Add-On Instruction, create another module reference parameter.
   a. In the routine, right-click **Parameters and Local Tags** and then click **New Parameter**.
   b. Enter the following values in the dialog box.

Attribute	Value
Name	ModuleRef_Slot01
Usage	InOut parameter

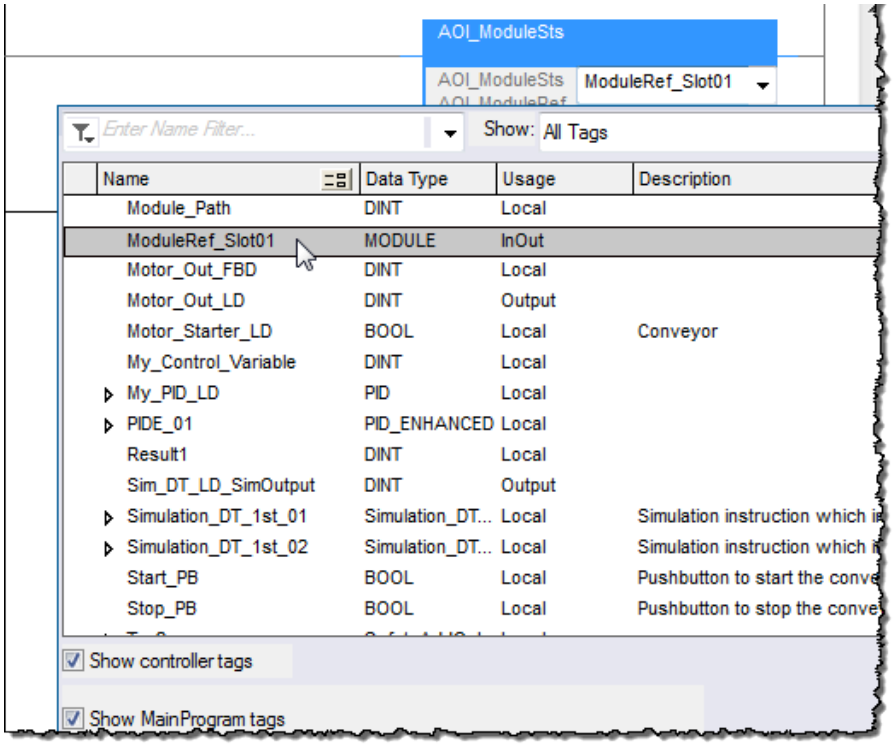
Data Type	MODULE

c. Click the down arrow in the **Parameter Connections** box, and choose a hardware module. This is the module that the parameter references.



2. Add the Add-On Instruction to the routine.

3. Connect the routine's module reference parameter to the Add-On Instruction's module reference parameter. Double-click the question mark next to the Add-On Instructions module reference parameter, then click the down arrow and choose the module reference parameter from the program.



You can now access the attributes associated with the Module Object from within the Add-On Instruction.

Tips for using an Add-On Instruction

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

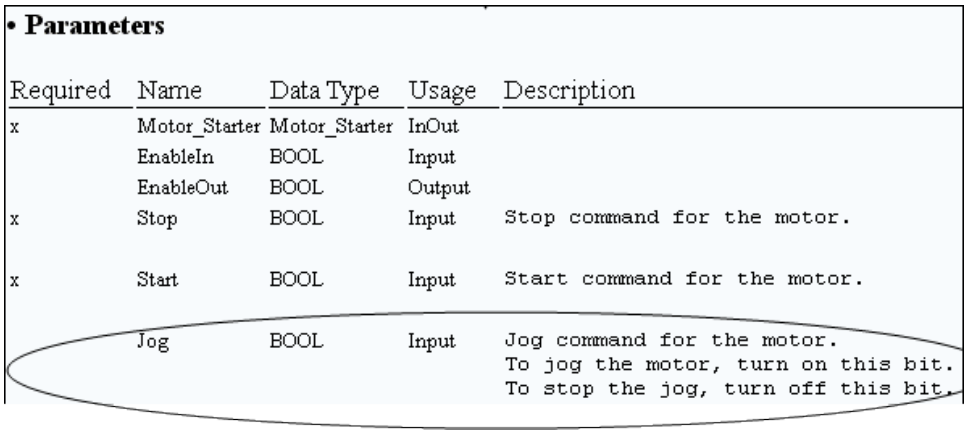
Topic	Description
Instruction Help	Use the instruction help to determine how to use the instruction in your code.
Ladder Rungs	In a ladder rung, consider if the instruction should be executed on a false rung condition. It may improve scan time to not execute it.
Data Types	A data type defined with the Add-On Instruction is used for the tag that provides context for the execution from your code. A tag must be defined of this Add-On Instruction-defined type on the call to the instruction.
Indexed Tag	You can use an indirect array indexed tag for the Instruction instance. One drawback is that you cannot

	monitor the Add-On Instruction by using this as a data context.
Passing Data	• Input and Output parameters are passed by value. • InOut parameters are passed by reference.

Programmatically access 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.

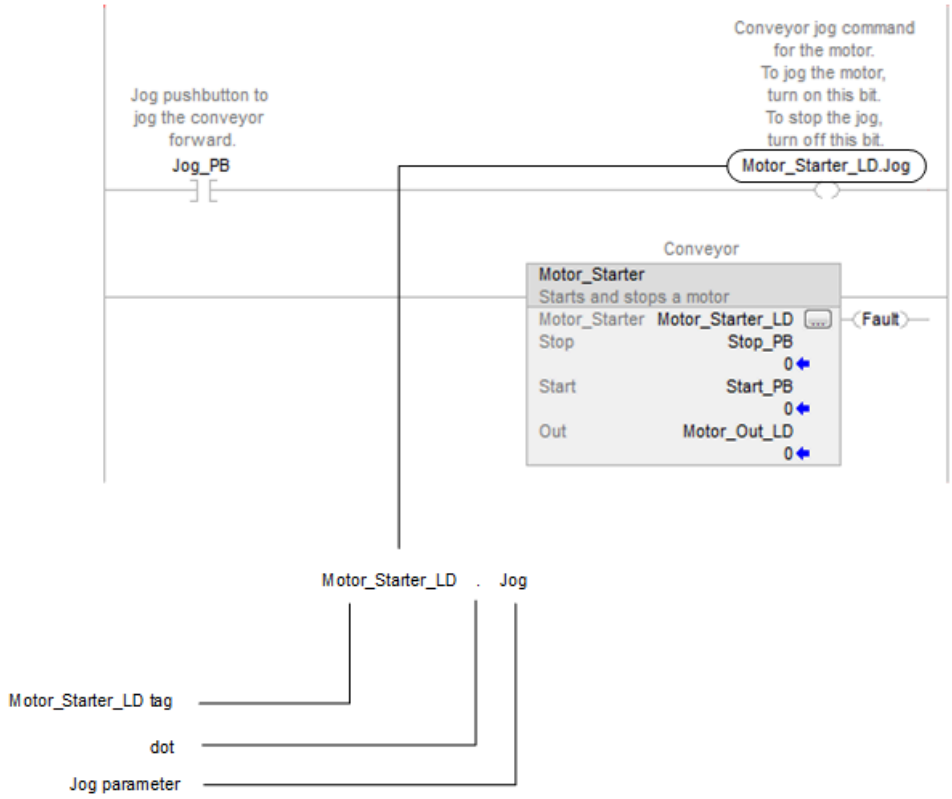


• **Parameters**

Required	Name	Data Type	Usage	Description
x	Motor_Starter	Motor_Starter	InOut	
	EnableIn	BOOL	Input	
	EnableOut	BOOL	Output	
x	Stop	BOOL	Input	Stop command for the motor.
x	Start	BOOL	Input	Start command for the motor.
	Jog	BOOL	Input	Jog command for the motor. To jog the motor, turn on this bit. To stop the jog, turn off this bit.

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:

Add_On_Tag.Parameter

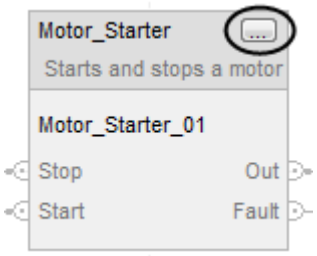
Where	Is
Add_On_Tag	An instance tag defined by the Add On data type.
Parameter	Name of the parameter.

Use the Jog command in a function block diagram

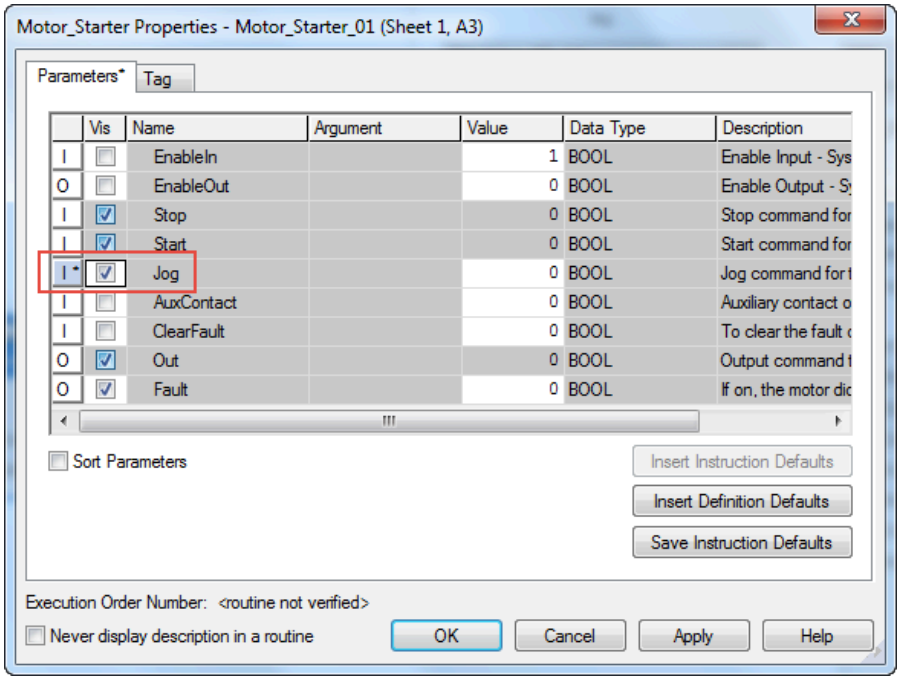
Any parameter can be made visible or invisible except those defined as required. Required parameters are always visible. If the parameter is required, you will see it checked in the **Properties** dialog box.

To use the Jog command in a function block diagram

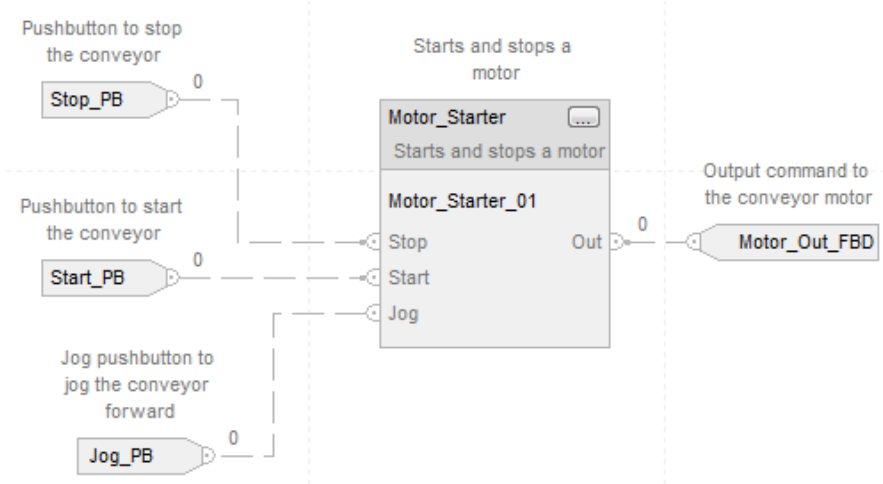
1. Click **Properties** for the instruction.



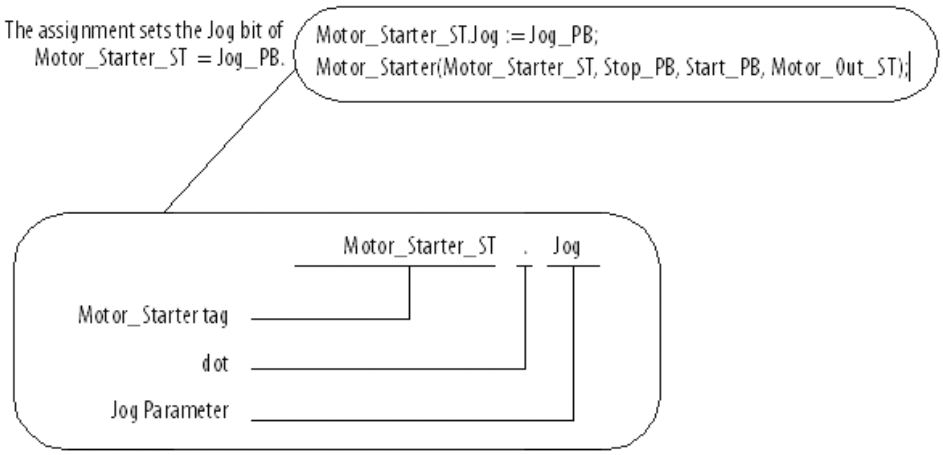
2. Select the **Vis** check box of the Jog parameter to use it in your diagram.



3. Click **OK**.
4. Wire to the pin for the parameter.



Using the Jog command in structured text



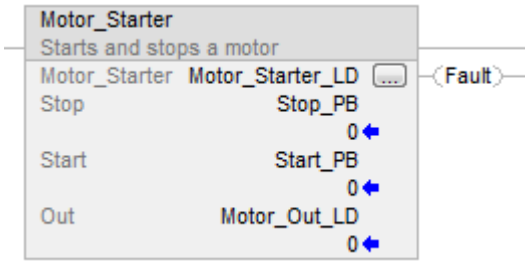
Monitor the value of a parameter

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

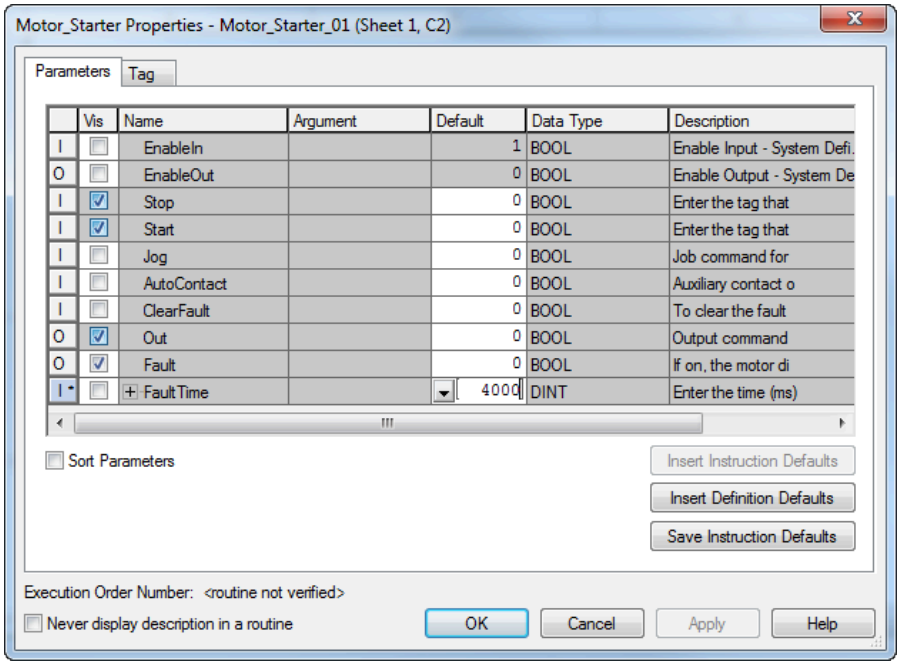
To monitor the value of a parameter

1. Open the **Properties** of the instruction based on what language you are using.

a. For either a Function Block or Ladder Diagram, click **Properties** ... for the instruction.



b. For Structured Text, right-click the instruction name and select **Properties**.



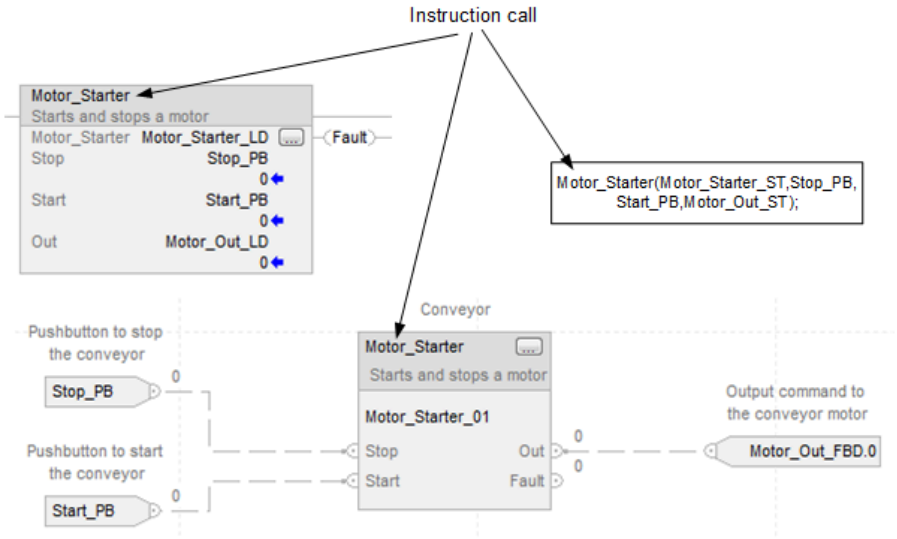
1. Monitor the value of the parameters and change any if needed.
2. Type a new value for each parameter as needed.
3. Click **Apply** and when finished, click **OK**.

View logic and monitor with data context

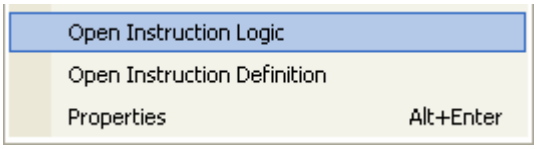
Follow this procedure when you want to view the logic of an Add-On Instruction and monitor data values with the logic.

To view logic and monitor with data context

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



2. Select **Open Instruction Logic**.

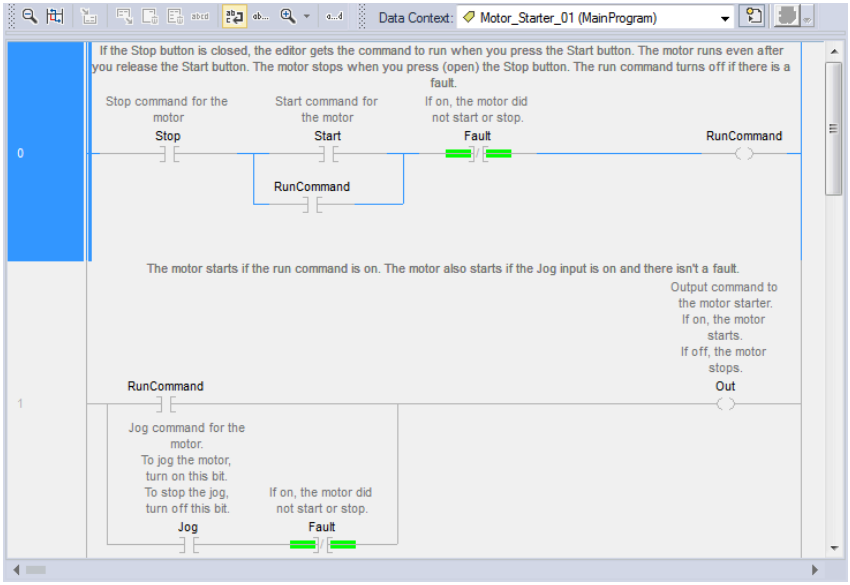


The Language Editor opens with the Add-On Instruction's logic routine and with data values from the instruction call.

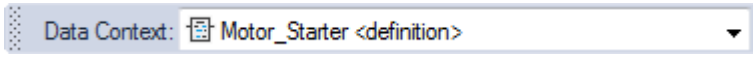
As you view the logic you can:

- Identify the instruction call whose tags are being used for data.
- See the logic as it executes (when online).

- See Parameter and Local Tag values.
- Change local tag and parameter values for the data instance selected.



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



You can't edit the instruction logic:

- Online
- When the logic is in the context of an instruction call
- If the instruction is source-protected
- If the instruction is sealed with an instruction signature

Determine if the Add-On Instruction is 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.

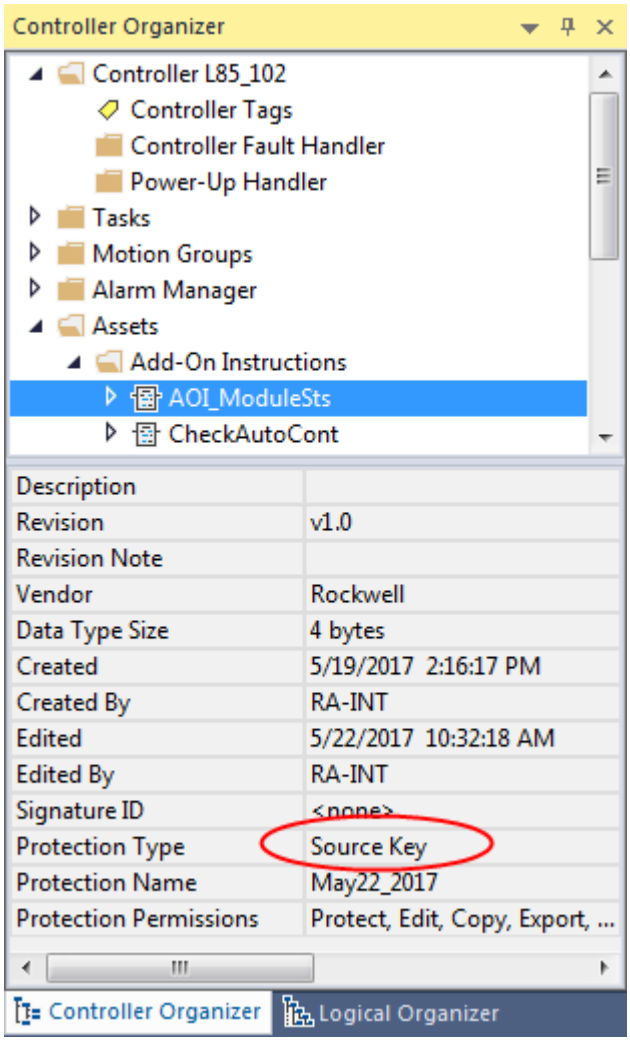
To determine if an Add-On Instruction is source protected

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 **Protection Type**.

   The **Protection Type** field indicates if the Add-On Instruction is protected by a license or a source key. If the **Protection Type** attribute is not listed, then the instruction is not protected.



Copy an Add-On Instruction

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

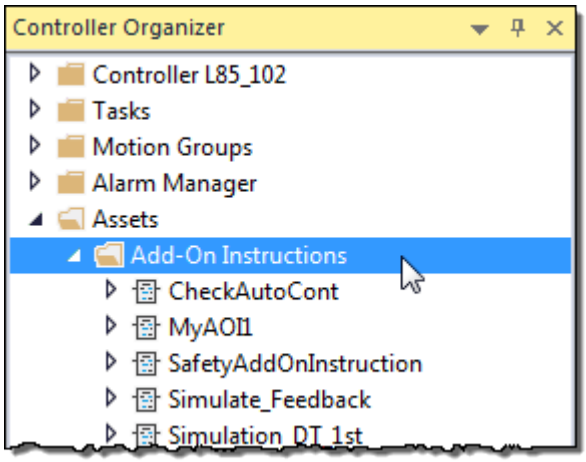
IMPORTANT: Use caution when copying and pasting components between different versions of Logix Designer programming application. Logix Designer application only supports pasting to the same version or newer version of Logix Designer application. Pasting to an earlier version of Logix Designer application is not supported. When pasting to an earlier version, the paste action may succeed, but the results may not be as expected.

When copying and pasting Add-On Instructions, consider these guidelines:

- You cannot paste a safety Add-On Instruction into a standard routine.
- You cannot paste a safety Add-On Instruction into a safety project that has been safety-locked or one that has a safety task signature.
- You cannot copy and paste a safety Add-On Instruction while online.

To copy an Add-On Instruction

1. Open the Logix Designer project that contains the Add-On Instruction.
2. Find the instruction in the **Add-On Instructions** folder.



3. Right-click the instruction and select **Copy**.
4. Go to the other project where you want to paste the instruction.
5. Right-click the **Add-On Instructions** folder and select **Paste**.

Store Add-On Instructions

There are two ways to store a group of Add-On Instructions together. One is to save your Add-On Instructions in a project file. Another is to create an L5X export file, as described in Chapter 4 on page 87.

To store instructions by saving them in a project file

1. Identify the instructions to store.
2. Place them in a project file with a distinctive name, such as MyInstructions.ACD.
3. Open other projects in additional instances of the Logix Designer application and use copy and paste or drag and drop to move a copy of the instruction from MyInstructions.ACD to another project.

   If any of these instructions reference the same Add-On Instruction or User-Defined Data 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 instruction it references to the target project.

Importing and Exporting Add-On Instructions

Create an export file

When you export an Add-On Instruction, the exported Add-On Instruction includes all of its parameters, local tags, and routines. These will be imported with the Add-On Instruction automatically.

To create an export file

- Optionally, you can include any nested Add-On Instructions or User-Defined Data Types that are referenced by the exported Add-On Instruction. Referenced Add-On Instructions and data types are exported to the L5X file if you select **Include all referenced Add-On Instructions and User-Defined Types** during the export.
- Add-On Instruction definition references may also be exported when a program, routine, set of rungs, or User-Defined Data Type is exported.

If an Add-On Instruction uses Message (MSG) instruction and InOut parameters of type MESSAGE, you may wish to export a rung containing the Add-On Instruction to include the MESSAGE tags. This captures the message configuration data, such as type and path.

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

If	Then
You want to store many Add-On Instructions that share a set of common Add-On Instructions or User-Defined Data Types in a common location	Export to separate files as described in Exporting to separate files on page 88.
You want to distribute an Add-On Instruction as one file	Export to a single file as described in Exporting to a single file on page 89.
You want to manage each Add-On Instruction as a standalone instruction	
You want to preserve the instruction signature on your Add-On Instruction	

Add-On Instructions with instruction signatures are encrypted upon export to prevent modifications to the export file.

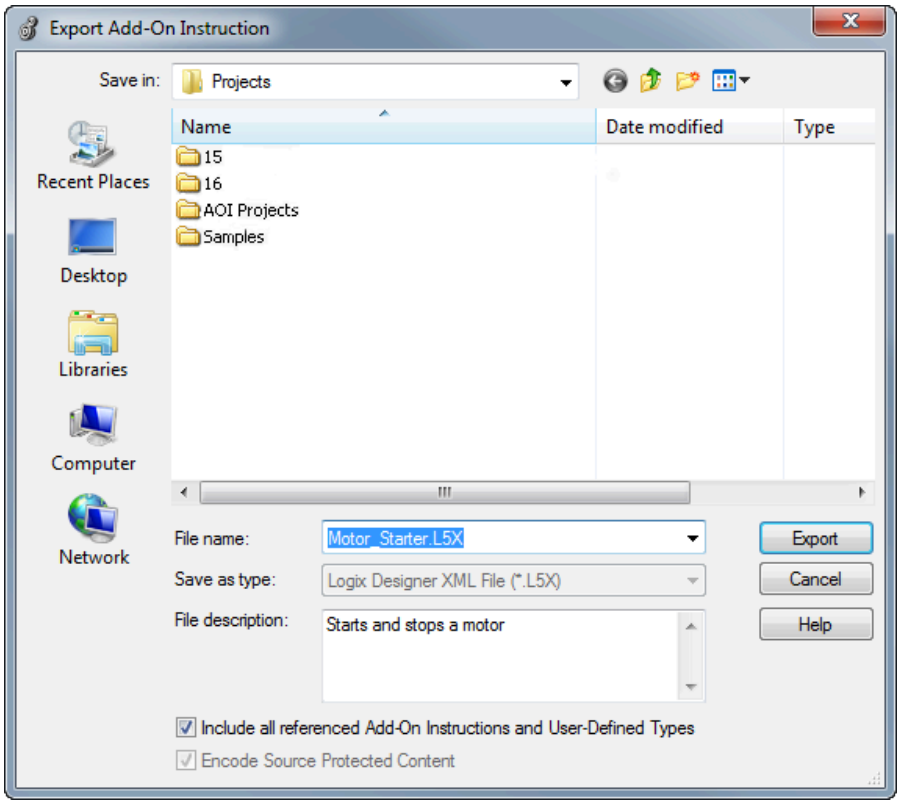
A License-protected Add-On Instruction is written to the export file in an encoded format unless the user's license contains Export permission. To export in non-encrypted text, the license must contain the Export permission, and when saving the export file, the user must deselect the **Encode Source Protected Content** option.

Export to separate files

If you want to store many Add-On Instructions that share a set of common Add-On Instructions or User-Defined Data Types in a common location, you may want to export each Add-On Instruction and User-Defined Data Types to separate files without including references.

To export to separate files

1. In the **Controller Organizer**, right-click the **Add-On Instruction** and select **Export Add-On Instruction**.
2. In the **Save In** box, select the common location to store the L5X file.



3. In the **File** name box, type a name for the file.
4. Clear the **Include referenced Add-On Instructions and User-Defined Types** check box.
5. Click **Export**.
6. Follow the above steps to individually export the other shared Add-On Instructions and User-Defined Data Types.

   Using export in this way lets you manage the shared Add-On Instruction and User-Defined Data Types independently of the Add-On Instructions 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 instructions that reference it. That is, it is only stored in one file instead of in every file whose instruction references it. This can help with the maintenance of the instructions as you only have to update one export file.

   To use Add-On Instructions that have been exported in a separate file, without references, you must first import any User-Defined Data Types of Add-On Instructions that the exported instruction references before the import of the referencing instruction

can be successful. To do this, work from the bottom up. Import the lowest-level User-Defined Data Types and any User-Defined Data Types that reference them.

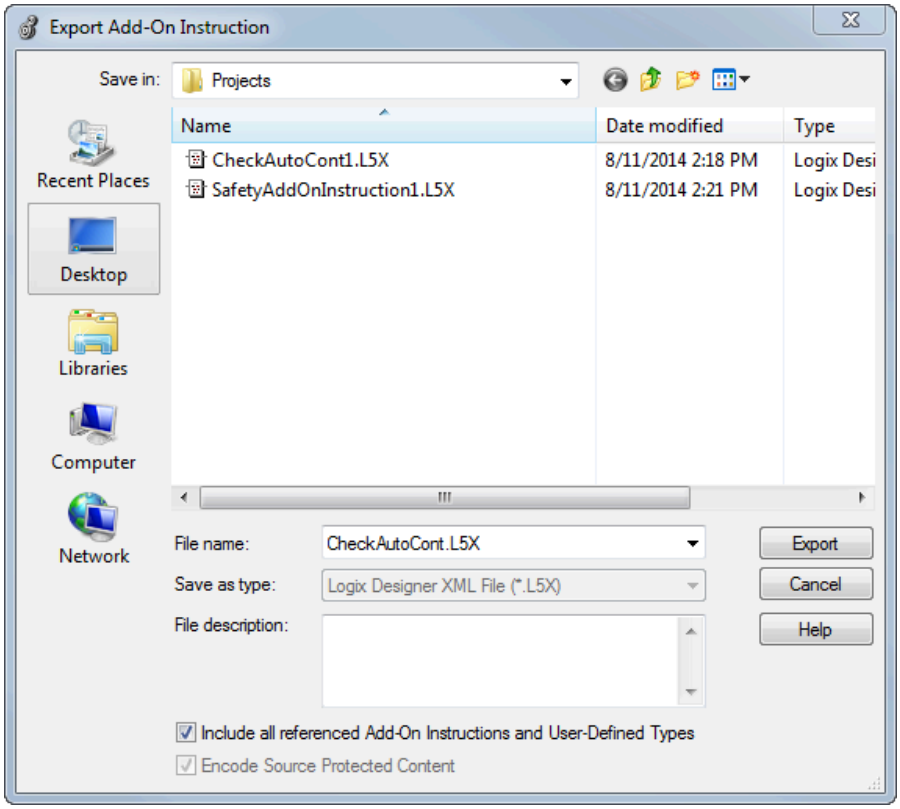
Then, import the lowest-level Add-On Instructions, followed by any Add-On Instructions that reference those low-level Add-On Instructions. Once all of the items it references are in place, the import of the Add-On Instruction will succeed.

Export to a single file

If you manage each Add-On Instruction as a standalone, you might want to export the instruction and any referenced Add-On Instructions or User-Defined Data Types into one export file. By including any referenced Add-On Instructions or User-Defined Data Types, you also make it easier to preserve the instruction signature of an Add-On Instruction.

To export to a single file and include any referenced items

1. In the **Controller Organizer**, right-click the **Add-On Instruction** and select **Export Add-On Instruction**.
2. In the **Save In** box, select the common location to store the L5X file.



3. In the **File name** box, type a name for the file.
4. Select the **Include referenced Add-On Instructions and User-Defined Types** check box.
5. Click **Export**.

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

Importing an Add-On Instruction

You can import an Add-On Instruction that was exported from another Logix Designer project. When importing an Add-On Instruction, the parameters, local tags, and routines are imported as part of the Add-On Instruction. Once the project has the Add-On Instruction, you can use it in your programs.

Import considerations

ATTENTION: Editing an L5K or L5X File:

The **EditedDate** attribute of an Add-On Instruction must be updated if the Add-On Instruction is modified by editing an L5K or L5X file. If Logix Designer application detects edits to the Add-On Instruction, but the **EditedDate** attribute is the same, the Add-On Instruction will not be imported.

When importing Add-On Instructions directly or as references, consider the following guidelines:

Topic	Consideration
Tag Data	Imported tags that reference an Add-On Instruction in the import file may be affected if the Add-On Instruction is not imported as well. In this case, the imported tag's data may be converted if the existing Add-On Instruction's data structure is different and tag data may be lost. If an existing Add-On Instruction is overwritten, project tag data may be converted if the Add-On Instruction's data structure is different and tag data may be lost. See Import configuration on page 92 for more information.
Logic	Imported logic that references the Add-On Instruction in the import file may be affected if the Add-On Instruction is not imported. If an existing Add-On Instruction is used for the imported logic reference and the parameter list of the Add-On Instruction in the project is different, the project may not verify or it may verify but not work as expected. If an existing Add-On Instruction is overwritten, logic in the project that references the Add-On Instruction may be affected. The project may not verify or may verify but not work as expected. See Import configuration on page 92 for more information.
Add-On Instructions While Online	An Add-On Instruction cannot be overwritten during import while online with the controller, though a new Add-On Instruction may be created while online.
License-protected Add-On Instructions	A License-protected Add-On Instruction is written to the export file in an encoded format unless the

	user's license contains Export permission. To export in non-encrypted text, the license must contain the Export permission, and when saving the export file, the user must deselect the **Encode Source Protected Content** option.
Final Name Change	If the Final Name of an Add-On Instruction is modified during import configuration, the edit date of the imported Add-On Instruction will be updated. In addition, all logic, tags, User-Defined Data Types, and other Add-On Instructions in the import file that reference the Add-On Instruction will be updated to reference the new name. As a result, the edit date of any Add-On Instruction that references the Add-On Instruction will be updated. Add-On Instructions that have been sealed with an instruction signature cannot be renamed during import.
User-Defined Data Types	Add-On Instructions cannot overwrite User-Defined Data Types. Add-On Instructions and User-Defined Data Types must have unique names.
Instruction Signature	If you import an Add-On Instruction with an instruction signature into a project where referenced Add-On Instructions or User-Defined Data Types are not available, you may need to remove the signature. You can overwrite an Add-On Instruction that has an instruction signature by importing a different Add-On Instruction with the same name into an existing routine. Add-On Instructions that have been sealed with an instruction signature cannot be renamed during import.
Safety Add-On Instructions	You cannot import a safety Add-On Instruction into a standard task. You cannot import a safety Add-On Instruction into a safety project that has been safety-locked or one that has a safety task signature. You cannot import a safety Add-On Instruction while online. Class, instruction signature, signature history, and safety instruction signature, if it exists, remain intact when an Add-On Instruction with an instruction signature is imported.

IMPORTANT: Importing an Add-On Instruction created in version 18 or later of Logix Designer software into an older project that does not support Add-On Instruction signatures causes the Add-On Instruction to lose attribute data and the instruction may no longer verify.

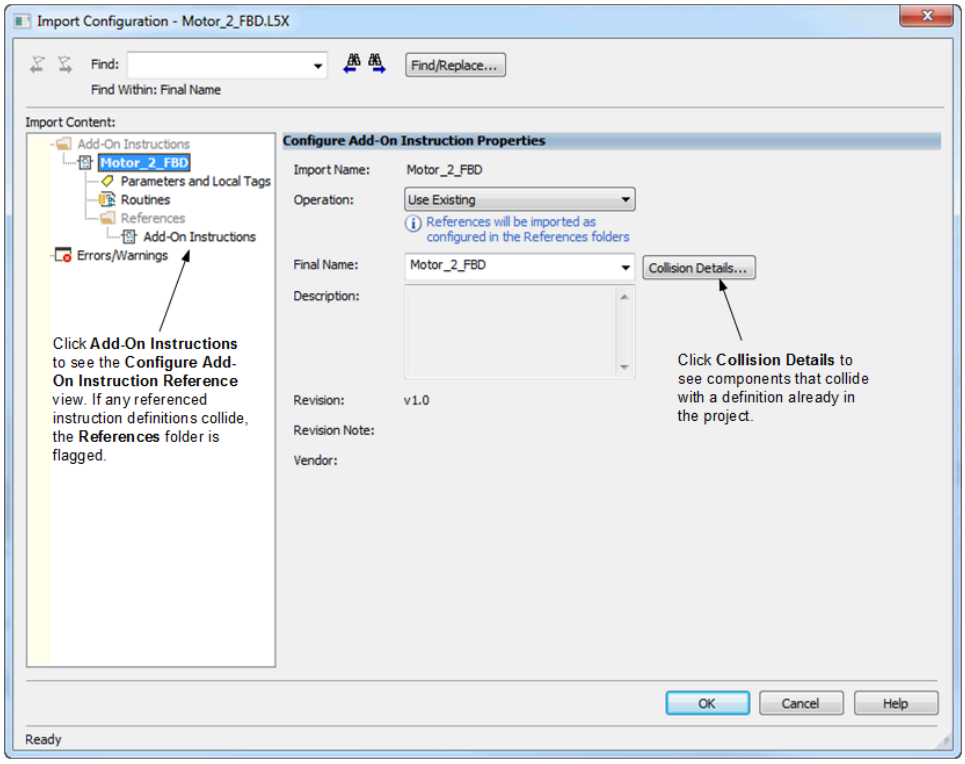
Import configuration

When you select a file to import, the **Import Configuration** dialog box lets you select how the Add-On Instruction and referenced components are imported.

If there are no issues, you can simply click **OK** to complete the import.

If your Add-On Instruction collides with one already in the project, you can:

- Rename it, by typing a new, unique name in the **Final Name** list.
- Select **Overwrite** from the **Operation** list.
- Select **Use Existing** from the **Operation** list.



You can only rename an Add-On Instruction if it has not been sealed with an instruction signature.

To rename an Add-On Instruction that has been source-protected, you need the source key or the required license.

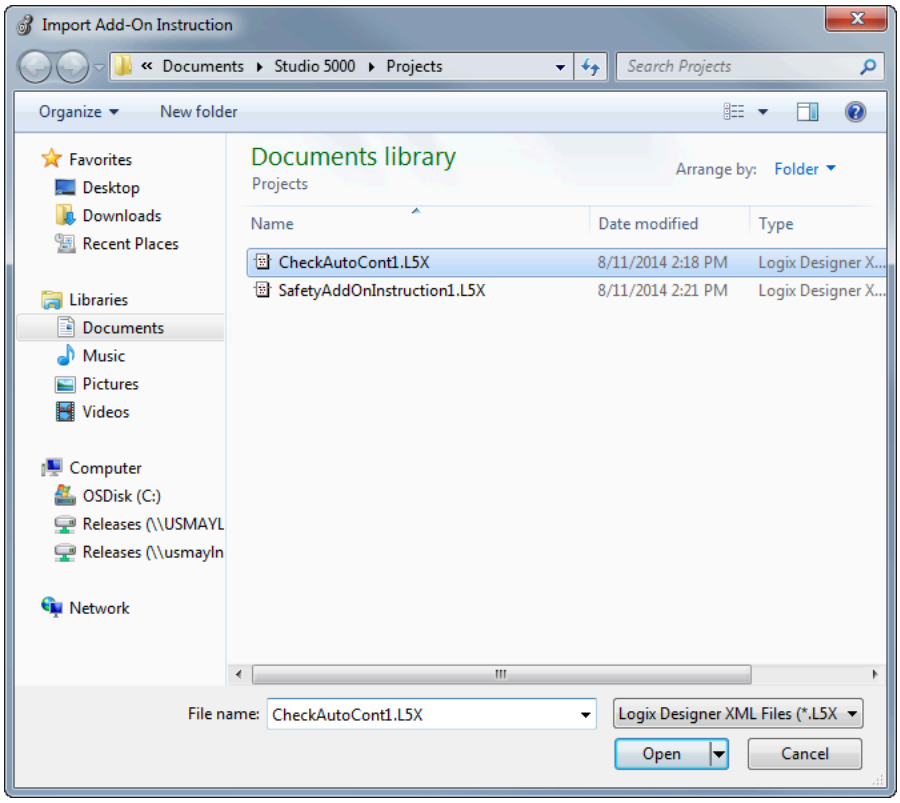
The **Collision Details** button displays the **Property Compare** tab, which shows the differences between the two instructions, and the **Project References** tab, which shows where the existing Add-On Instruction is used.

Update an Add-On Instruction to a newer revision through import

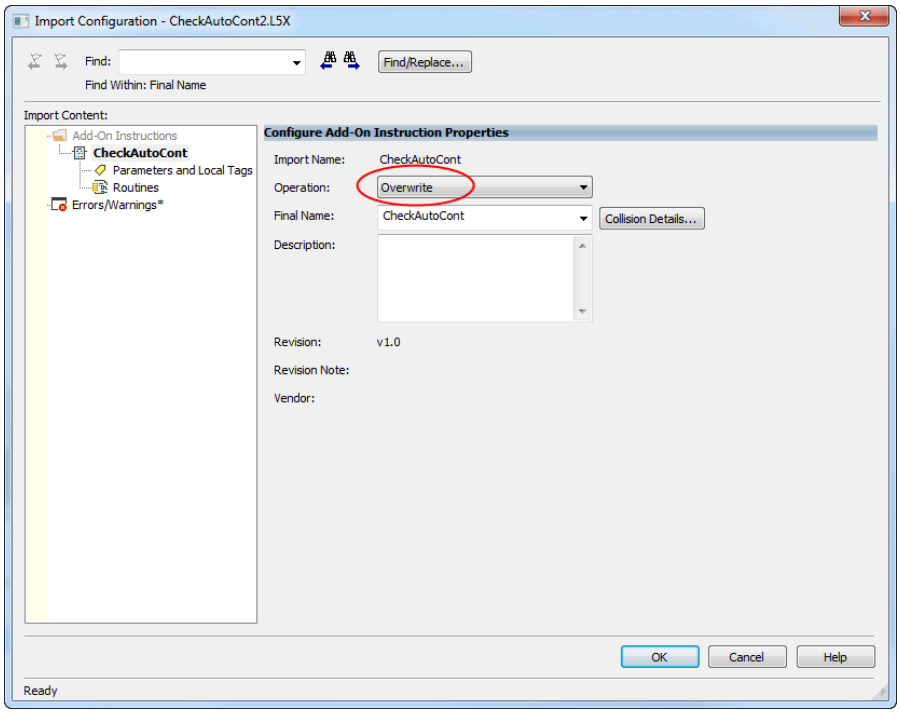
When you need to update an instruction to a newer revision, you can import it from an L5X file or copy it from an existing project. You must be offline to update an Add-On Instruction.

To update an Add-On Instruction to a newer revision through import

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

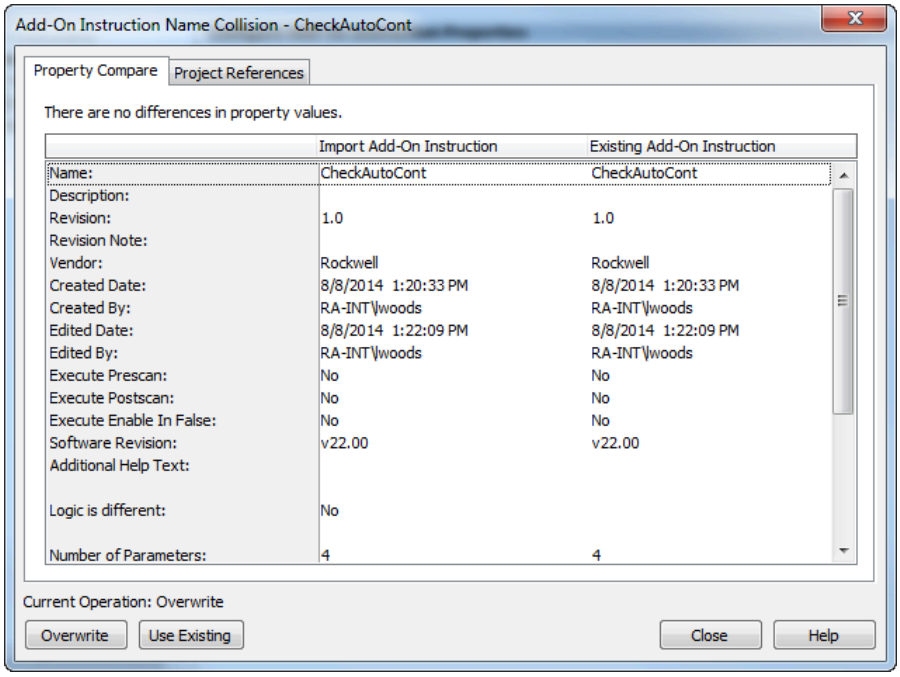


3. Review the **Import Configuration** dialog box, and from the **Operations** list, click **Overwrite**.



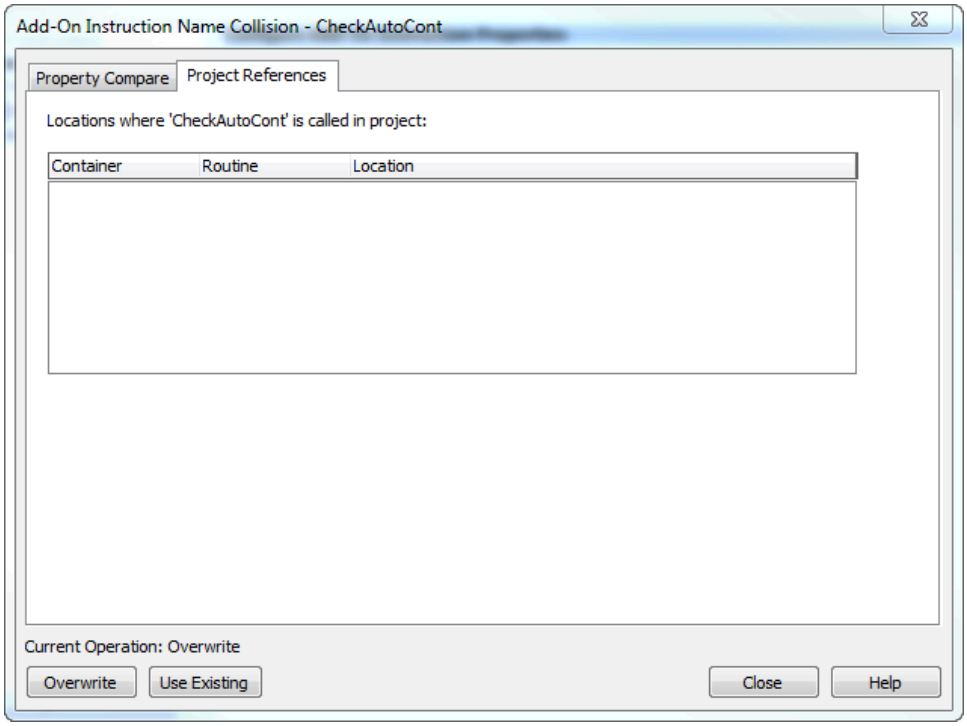
4. Click **Collision Details** to see any differences in the Add-On Instructions and to view where the Add-On Instruction is used.

   The **Property Compare** tab shows the differences between the instructions, in this case, the **Revision**, **Edited Date**, and **Software Revision**.



The **Compare** dialog box only compares extended properties for each instruction definition, such as description, revision, or edited date. For effective revision control, enter a detailed revision note.

The **Project References** tab shows where the existing Add-On Instruction is used.



IMPORTANT: Check each routine where your Add-On Instruction is used to make sure that your existing program code will work correctly with the new version of the instruction.

For more information on updates to arguments, see Updates to arguments following parameter edits on page 39.

1. Click **Close** and then **OK** to complete the operation.

Rockwell Automation Support

Use these resources to access support information.

Technical Support Center	Find help with how-to videos, FAQs, chat, user forums, and product notification updates.	rok.auto/support
Local Technical Support Phone Numbers	Locate the telephone number for your country.	rok.auto/phonesupport
Technical Documentation Center	Quickly access and download technical specifications, installation instructions, and user manuals.	rok.auto/techdocs
Literature Library	Find installation instructions, manuals, brochures, and technical data publications.	rok.auto/literature
Product Compatibility and Download Center (PCDC)	Get help determining how products interact, check features and capabilities, and find associated firmware.	rok.auto/pcdc

Documentation Feedback

Your comments help us serve your documentation needs better. If you have any suggestions on how to improve our content, complete the form at rok.auto/docfeedback.

Waste Electrical and Electronic Equipment (WEEE)

At the end of life, this equipment should be collected separately from any unsorted municipal waste.

Rockwell Automation maintains current product environmental information on its website at rok.auto/pec.

Rockwell Otomasyon Ticaret A.Ş. Kar Plaza İş Merkezi E Blok Kat:6 34752 İçerenköy, İstanbul, Tel: +90 (216) 5698400 EEE Yönetmeliğine Uygundur

Connect with us.

rockwellautomation.com — expanding **human possibility**®

AMERICAS: Rockwell Automation, 1201 South Second Street, Milwaukee, WI 53204-2496 USA, Tel: (1) 414.382.2000
EUROPE/MIDDLE EAST/AFRICA: Rockwell Automation NV, Pegasus Park, De Kleetlaan 12a, 1831 Diegem, Belgium, Tel: (32) 2663 0600
ASIA PACIFIC: Rockwell Automation SEA Pte Ltd, 2 Corporation Road, #04-05, Main Lobby, Corporation Place, Singapore 618494, Tel: (65) 6510 6608
UNITED KINGDOM: Rockwell Automation Ltd., Pitfield, Kiln Farm, Milton Keynes, MK11 3DR, United Kingdom, Tel: (44)(1908) 838-800

Publication 1756-PM010N-EN-P - September 2025
Supersedes Publication 1756-PM010M-EN-P - September 2024