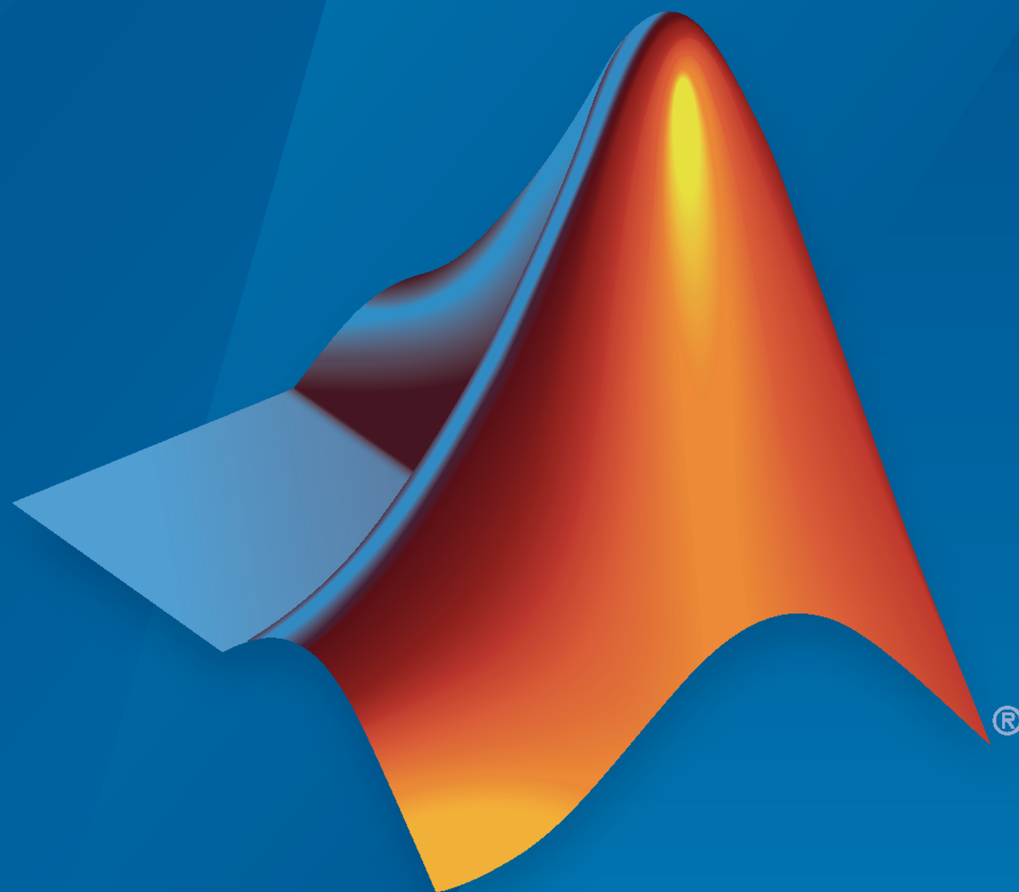
Stateflow®

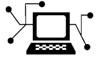
Reference



MATLAB® & SIMULINK®

R2023a

How to Contact MathWorks



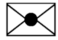
Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000



The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Stateflow® Reference

© COPYRIGHT 2006–2023 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

March 2006	Online only	New for Version 6.4 (Release 2006a)
September 2006	Online only	Revised for Version 6.5 (Release R2006b)
September 2007	Online only	Rereleased for Version 7.0 (Release 2007b)
March 2008	Online only	Revised for Version 7.1 (Release 2008a)
October 2008	Online only	Revised for Version 7.2 (Release 2008b)
March 2009	Online only	Rereleased for Version 7.3 (Release 2009a)
September 2009	Online only	Revised for Version 7.4 (Release 2009b)
March 2010	Online only	Rereleased for Version 7.5 (Release 2010a)
September 2010	Online only	Rereleased for Version 7.6 (Release 2010b)
April 2011	Online only	Rereleased for Version 7.7 (Release 2011a)
September 2011	Online only	Rereleased for Version 7.8 (Release 2011b)
March 2012	Online only	Revised for Version 7.9 (Release 2012a)
September 2012	Online only	Revised for Version 8.0 (Release 2012b)
March 2013	Online only	Revised for Version 8.1 (Release 2013a)
September 2013	Online only	Revised for Version 8.2 (Release 2013b)
March 2014	Online only	Revised for Version 8.3 (Release 2014a)
October 2014	Online only	Revised for Version 8.4 (Release 2014b)
March 2015	Online only	Revised for Version 8.5 (Release 2015a)
September 2015	Online only	Revised for Version 8.6 (Release 2015b)
October 2015	Online only	Rereleased for Version 8.5.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 8.7 (Release 2016a)
September 2016	Online only	Revised for Version 8.8 (Release 2016b)
March 2017	Online only	Revised for Version 8.9 (Release 2017a)
September 2017	Online only	Revised for Version 9.0 (Release 2017b)
March 2018	Online only	Revised for Version 9.1 (Release 2018a)
September 2018	Online only	Revised for Version 9.2 (Release 2018b)
March 2019	Online only	Revised for Version 10.0 (Release 2019a)
September 2019	Online only	Revised for Version 10.1 (Release 2019b)
March 2020	Online only	Revised for Version 10.2 (Release 2020a)
September 2020	Online only	Revised for Version 10.3 (Release 2020b)
March 2021	Online only	Revised for Version 10.4 (Release 2021a)
September 2021	Online only	Revised for Version 10.5 (Release 2021b)
March 2022	Online only	Revised for Version 10.6 (Release 2022a)
September 2022	Online only	Revised for Version 10.7 (Release 2022b)
March 2023	Online only	Revised for Version 10.8 (Release 2023a)

Contents

1 Blocks

2 Functions

3 Operators

4 Objects

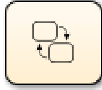
5 Object Functions

6 Tools

Blocks

Chart

Implement control logic with finite state machine



Libraries:
Stateflow

Description

The Chart block is a graphical representation of a finite state machine based on a state transition diagram. In a Stateflow chart, states and transitions form the basic building blocks of a sequential logic system. States correspond to operating modes and transitions represent pathways between states. For more information, see "Model Finite State Machines by Using Stateflow Charts".

To implement control logic, Stateflow charts can use MATLAB® or C as the action language. For more information, see "Differences Between MATLAB and C as Action Language Syntax".

Chart properties specify how your Stateflow chart interfaces with the Simulink® model. You can modify chart properties in the **Property Inspector**, the Model Explorer, or the Chart properties dialog box. For more information, see "Specify Properties for Stateflow Charts". Alternatively, you can modify chart properties programmatically by using `Stateflow.Chart` objects. For more information about the Stateflow programmatic interface, see "Overview of the Stateflow API".

Ports

Input

Port_1 — Input port
scalar | vector | matrix

When you create input data in the **Symbols** pane, Stateflow creates input ports. The input data that you create has a corresponding input port that appears once you create data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus` | `string`

Output

Port_1 — Output port
scalar | vector | matrix

When you create output data in the **Symbols** pane, Stateflow creates output ports. The output data that you create has a corresponding output port that appears once you create data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus` | `string`

Parameters

Parameters on the Code Generation tab require Simulink Coder™ or Embedded Coder®.

Main

Show port labels — Select how to display port labels
`FromPortIcon` (default) | `none` | `FromPortBlockName` | `SignalName`

Select how to display port labels on the Chart block icon.

`none`

Do not display port labels.

`FromPortIcon`

If the corresponding port icon displays a signal name, display the signal name on the Chart block. Otherwise, display the port block name.

`FromPortBlockName`

Display the name of the corresponding port block on the Chart block.

`SignalName`

If a signal name exists, display the name of the signal connected to the port on the Chart block. Otherwise, display the name of the corresponding port block.

Programmatic Use
Parameter: `ShowPortLabels`
Type: string scalar or character vector
Value: `"none""FromPortIcon"` | `"FromPortBlockName"` | `"SignalName"`
Default: `"FromPortIcon"`

Read/Write permissions — Select access to contents of chart
`ReadWrite` (default) | `ReadOnly` | `NoReadOrWrite`

Control user access to the contents of the chart.

`ReadWrite`

Enable opening and modification of chart contents.

`ReadOnly`

Enable opening but not modification of the chart. If the chart resides in a block library, you can create and open links to the chart and can make and modify local copies of the chart but you cannot change the permissions or modify the contents of the original library instance.

`NoReadOrWrite`

Disable opening or modification of chart. If the chart resides in a library, you can create links to the chart in a model but you cannot open, modify, change permissions, or create local copies of the chart.

Programmatic Use
Parameter: `Permissions`
Type: string scalar or character vector
Value: `"ReadWrite"` | `"ReadOnly"` | `"NoReadOrWrite"`
Default: `"ReadWrite"`

Minimize algebraic loop occurrences — Control elimination of algebraic loops
off (default) | on

off
: Do not try to eliminate any artificial algebraic loops that include the atomic subchart.

on
: Try to eliminate any artificial algebraic loops that include the atomic subchart.

Programmatic Use
Parameter: `MinAlgLoopOccurrences`
Type: string scalar or character vector
Value: `"off"` | `"on"`
Default: `"off"`

Sample time — Specify time interval
`-1` (default) | `[Ts 0]`

Specify whether all blocks in this chart must run at the same rate or can run at different rates.

- If the blocks in the chart can run at different rates, specify the chart sample time as inherited (`-1`).
- If all blocks must run at the same rate, specify the sample time corresponding to this rate as the value of the **Sample time** parameter.
- If any of the blocks in the chart specify a different sample time (other than `-1` or `inf`), Simulink displays an error message when you update or simulate the model. For example, suppose all the blocks in the chart must run `5` times a second. To ensure this time, specify the sample time of the chart as `0.2`. In this example, if any of the blocks in the chart specify a sample time other than `0.2`, `-1`, or `inf`, Simulink displays an error when you update or simulate the model.

`-1`
: Specify inherited sample time. If the blocks in the chart can run at different rates, use this sample time.

`[Ts 0]`
: Specify periodic sample time.

Programmatic Use
Parameter: `SystemSampleTime`
Type: string scalar or character vector
Value: `"-1"` | `"[Ts 0]"`
Default: `"-1"`

Code Generation

Function packaging — Select code format
`Auto` (default) | `Inline` | `Nonreusable function` | `Reusable function`

Select the generated code format for an atomic (nonvirtual) subchart.

`Auto`
: Simulink Coder chooses the optimal format for your system based on the type and number of instances of the chart that exist in the model.

`Inline`

Simulink Coder inlines the chart unconditionally.

`Nonreusable function`

Simulink Coder explicitly generates a separate function in a separate file. Charts with this setting generate functions that might have arguments depending on the “Function interface” (Simulink) parameter setting. You can name the generated function and file using parameters “Function name” (Simulink) and “File name (no extension)” (Simulink). These functions are not reentrant.

`Reusable function`

Simulink Coder generates a function with arguments that allows reuse of chart code when a model includes multiple instances of the chart.

This option generates a function with arguments that allows chart code to be reused in the generated code of a model reference hierarchy that includes multiple instances of a chart across referenced models. In this case, the chart must be in a library.

Tips

- When you want multiple instances of a chart represented as one reusable function, you can designate each one of them as `Auto` or as `Reusable function`. It is best to use one because using both creates two reusable functions, one for each designation. The outcomes of these choices differ only when reuse is not possible. Selecting `Auto` does not allow for control of the function or file name for the chart code.
- The `Reusable function` and `Auto` options both determine whether multiple instances of a chart exist and the code can be reused. The options behave differently when it is impossible to reuse the code. In this case, `Auto` yields inlined code, or if circumstances prohibit inlining, separate functions for each chart instance.
- If you select the `Reusable function` while your generated code is under source control, set **File name options** to `Use subsystem name`, `Use function name`, or `User specified`. Otherwise, the names of your code files change whenever you modify your model, which prevents source control on your files.

Dependency

- This parameter requires Simulink Coder.
- Setting this parameter to `Nonreusable function` or `Reusable function` enables the following parameters:
  - **Function name options**
  - **File name options**
  - Memory section for initialize/terminate functions (requires Embedded Coder and an ERT-based system target file)
  - Memory section for execution functions (requires Embedded Coder and an ERT-based system target file)
- Setting this parameter to `Nonreusable function` enables **Function with separate data** (requires a license for Embedded Coder and an ERT-based system target file).

Programmatic Use

Parameter: `RTWSystemCode`
Type: string scalar or character vector
Value: `"Auto"` | `"Inline"` | `"Nonreusable function"` | `"Reusable function"`

Default: `"Auto"`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder™ provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

Active State Output

To generate an output port in the HDL code that shows the active state, in the Properties window of the chart, select **Create output for monitoring**. The output is an enumerated data type. See "Simplify Stateflow Charts by Incorporating Active State Output".

Registered Output

To insert an output register that delays the chart output by a simulation cycle, use the OutputPipeline (HDL Coder) block property.

HDL Block Properties

ClockDrivenOutput	Enable clock-driven outputs to prevent combinatorial logic from driving the output and to allow an immediate output update when the clock signal and state change. The default is `off`. When you set **ClockDrivenOutput** to `on`, HDL Coder adds an output register that updates when the state updates. The final output variable is then assigned a value from the clock-drive register. This option is available only for Moore charts.
ConstMultiplierOptimization	Canonical signed digit (CSD) or factored CSD optimization. The default is `none`. See also "ConstMultiplierOptimization" (HDL Coder).
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
DistributedPipelining	Pipeline register distribution, or register retiming. The default is `inherit`. See also "DistributedPipelining" (HDL Coder).

InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
InstantiateFunctions	Generate a VHDL® `entity` or Verilog® `module` for each function. The default is `off`. See also "InstantiateFunctions" (HDL Coder).
LoopOptimization	Unroll, stream, or do not optimize loops. The default is `none`. See also "LoopOptimization" (HDL Coder).
MapPersistentVarsToRAM	Map persistent arrays to RAM. The default is `off`. See also "MapPersistentVarsToRAM" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).
ResetType	Suppress reset logic generation. The default is `default`, which generates reset logic. See also "ResetType" (HDL Coder).
SharingFactor	Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also "Resource Sharing" (HDL Coder).
VariablesToPipeline	**Warning** `VariablesToPipeline` is not recommended. Use `coder.hdl.pipeline` instead. Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.

Complex Data Support

This block supports code generation for complex signals.

Restrictions

To learn about restrictions of using charts, see "Introduction to Stateflow HDL Code Generation" (HDL Coder).

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Blocks
State Transition Table | Truth Table

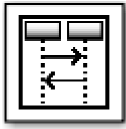
Objects
`Stateflow.Chart`

Topics
"Construct and Run a Stateflow Chart"

“Model Finite State Machines by Using Stateflow Charts”
“Specify Properties for Stateflow Charts”
“Differences Between MATLAB and C as Action Language Syntax”

Sequence Viewer

Display messages, events, states, transitions, and functions between blocks during simulation



Libraries:
Simulink / Messages & Events
Simulink Test
SimEvents
Stateflow

Description

The Sequence Viewer block displays messages, events, states, transitions, and functions between certain blocks during simulation. The blocks that you can display are called lifeline blocks and include:

- Subsystems
- Referenced models
- Blocks that contain messages, such as Stateflow charts
- Blocks that call functions or generate events, such as Function Caller, Function-Call Generator, and MATLAB Function blocks
- Blocks that contain functions, such as Function-Call Subsystem and Simulink Function blocks

To see states, transitions, and events for lifeline blocks in a referenced model, you must have a Sequence Viewer block in the referenced model. Without a Sequence Viewer block in the referenced model, you can see only messages and functions for lifeline blocks in the referenced model.

Note The Sequence Viewer block does not display function calls generated by MATLAB Function blocks and S-functions.

Parameters

Time Precision for Variable Step — Digits for time increment precision
3 (default) | scalar

Number of digits for time increment precision. When using a variable step solver, change this parameter to adjust the time precision for the sequence viewer. By default the block supports `3` digits of precision.

Suppose the block displays two events that occur at times `0.1215` and `0.1219`. Displaying these two events precisely requires `4` digits of precision. If the precision is `3`, then the block displays two events at time `0.121`.

Programmatic Use
Block Parameter: `VariableStepTimePrecision`
Type: string scalar or character vector
Values: `"3"` | scalar

Default: `"3"`

History — Maximum number of previous events to display
`5000` (default) | scalar

Total number of events before the last event to display.

For example, if **History** is `5` and there are `10` events in your simulation, then the block displays `6` events, including the last event and the five events prior the last event. Earlier events are not displayed. The time ruler is greyed to indicate the time between the beginning of the simulation and the time of the first displayed event.

Each send, receive, drop, or function call event is counted as one event, even if they occur at the same simulation time.

Programmatic Use
Block Parameter: `History`
Type: string scalar or character vector
Values: `"1000"` | scalar
Default: `"1000"`

Block Characteristics

Data Types	`Boolean` | `bus` | `double` | `enumerated` | `fixed point` | `integer` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`yes`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

Version History

Introduced in R2015b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block can be used for visualizing message transitions during simulation, but is not included in the generated code.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block displays messages during simulation when used in subsystems that generate HDL code, but it is not included in the hardware implementation.

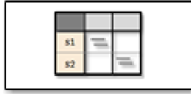
See Also

Tools
Sequence Viewer

Topics
"Use the Sequence Viewer to Visualize Messages, Events, and Entities"

State Transition Table

Represent modal logic in tabular format



Libraries:
Stateflow

Description

The State Transition Table block represents a finite state machine for sequential modal logic in tabular format. Instead of drawing states and transitions in a Stateflow chart, you can use a state transition table to model a state machine in a concise, compact format that requires minimal maintenance of graphical objects. For more information, see “Use State Transition Tables to Express Sequential Logic in Tabular Form”.

To implement control logic, State Transition Table blocks can use MATLAB or C as the action language. For more information, see “Differences Between MATLAB and C as Action Language Syntax”.

State Transition Table block properties specify how your state transition table interfaces with the Simulink model. You can modify these properties in the **Property Inspector**, the Model Explorer, or the State Transition Table properties dialog box. For more information, see “Specify Properties for State Transition Tables”. Alternatively, you can specify state transition table properties programmatically by using `Stateflow.StateTransitionTableChart` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Ports

Input

Port_1 — Input port
scalar | vector | matrix

When you create input data in the **Symbols** pane, Stateflow creates input ports. The input data that you create has a corresponding input port that appears once you create data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus` | `string`

Output

Port_1 — Output port
scalar | vector | matrix

When you create output data in the **Symbols** pane, Stateflow creates output ports. The output data that you create has a corresponding output port that appears once you create data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus` | `string`

Parameters

Parameters on the Code Generation tab require Simulink Coder or Embedded Coder.

Main

Show port labels — Select how to display port labels
`FromPortIcon` (default) | `none` | `FromPortBlockName` | `SignalName`

Select how to display port labels on the State Transition Table block icon.

`none`

Do not display port labels.

`FromPortIcon`

If the corresponding port icon displays a signal name, display the signal name on the State Transition Table block. Otherwise, display the port block name.

`FromPortBlockName`

Display the name of the corresponding port block on the State Transition Table block.

`SignalName`

If a signal name exists, display the name of the signal connected to the port on the State Transition Table block. Otherwise, display the name of the corresponding port block.

Programmatic Use
Parameter: `ShowPortLabels`
Type: string scalar or character vector
Value: `"none""FromPortIcon"` | `"FromPortBlockName"` | `"SignalName"`
Default: `"FromPortIcon"`

Read/Write permissions — Select access to contents of chart
`ReadWrite` (default) | `ReadOnly` | `NoReadOrWrite`

Control user access to the contents of the chart.

`ReadWrite`

Enable opening and modification of chart contents.

`ReadOnly`

Enable opening but not modification of the chart. If the chart resides in a block library, you can create and open links to the chart and can make and modify local copies of the chart but you cannot change the permissions or modify the contents of the original library instance.

`NoReadOrWrite`

Disable opening or modification of chart. If the chart resides in a library, you can create links to the chart in a model but you cannot open, modify, change permissions, or create local copies of the chart.

Programmatic Use
Parameter: `Permissions`
Type: string scalar or character vector
Value: `"ReadWrite"` | `"ReadOnly"` | `"NoReadOrWrite"`
Default: `"ReadWrite"`

Minimize algebraic loop occurrences — Control elimination of algebraic loops
off (default) | on

off
: Do not try to eliminate any artificial algebraic loops that include the atomic subchart.

on
: Try to eliminate any artificial algebraic loops that include the atomic subchart.

Programmatic Use
Parameter: `MinAlgLoopOccurrences`
Type: string scalar or character vector
Value: `"off"` | `"on"`
Default: `"off"`

Sample time — Specify time interval
`-1` (default) | `[Ts 0]`

Specify whether all blocks in this chart must run at the same rate or can run at different rates.

- If the blocks in the chart can run at different rates, specify the chart sample time as inherited (`-1`).
- If all blocks must run at the same rate, specify the sample time corresponding to this rate as the value of the **Sample time** parameter.
- If any of the blocks in the chart specify a different sample time (other than `-1` or `inf`), Simulink displays an error message when you update or simulate the model. For example, suppose all the blocks in the chart must run `5` times a second. To ensure this time, specify the sample time of the chart as `0.2`. In this example, if any of the blocks in the chart specify a sample time other than `0.2`, `-1`, or `inf`, Simulink displays an error when you update or simulate the model.

`-1`
: Specify inherited sample time. If the blocks in the chart can run at different rates, use this sample time.

`[Ts 0]`
: Specify periodic sample time.

Programmatic Use
Parameter: `SystemSampleTime`
Type: string scalar or character vector
Value: `"-1"` | `"[Ts 0]"`
Default: `"-1"`

Code Generation

Function packaging — Select code format
`Auto` (default) | `Inline` | `Nonreusable function` | `Reusable function`

Select the generated code format for an atomic (nonvirtual) subchart.

`Auto`
: Simulink Coder chooses the optimal format for your system based on the type and number of instances of the chart that exist in the model.

`Inline`

Simulink Coder inlines the chart unconditionally.

`Nonreusable function`

Simulink Coder explicitly generates a separate function in a separate file. State transition tables with this setting generate functions that might have arguments depending on the "Function interface" (Simulink) parameter setting. You can name the generated function and file using parameters "Function name" (Simulink) and "File name (no extension)" (Simulink). These functions are not reentrant.

`Reusable function`

Simulink Coder generates a function with arguments that allows reuse of chart code when a model includes multiple instances of the chart.

This option generates a function with arguments that allows chart code to be reused in the generated code of a model reference hierarchy that includes multiple instances of a chart across referenced models. In this case, the chart must be in a library.

Tips

- When you want multiple instances of a chart represented as one reusable function, you can designate each one of them as `Auto` or as `Reusable function`. It is best to use one because using both creates two reusable functions, one for each designation. The outcomes of these choices differ only when reuse is not possible. Selecting `Auto` does not allow for control of the function or file name for the chart code.
- The `Reusable function` and `Auto` options both determine whether multiple instances of a chart exist and the code can be reused. The options behave differently when it is impossible to reuse the code. In this case, `Auto` yields inlined code, or if circumstances prohibit inlining, separate functions for each chart instance.
- If you select the `Reusable function` while your generated code is under source control, set **File name options** to `Use subsystem name`, `Use function name`, or `User specified`. Otherwise, the names of your code files change whenever you modify your model, which prevents source control on your files.

Dependency

- This parameter requires Simulink Coder.
- Setting this parameter to `Nonreusable function` or `Reusable function` enables the following parameters:
  - **Function name options**
  - **File name options**
  - Memory section for initialize/terminate functions (requires Embedded Coder and an ERT-based system target file)
  - Memory section for execution functions (requires Embedded Coder and an ERT-based system target file)
- Setting this parameter to `Nonreusable function` enables **Function with separate data** (requires a license for Embedded Coder and an ERT-based system target file).

Programmatic Use

Parameter: `RTWSystemCode`
Type: string scalar or character vector

Value: `"Auto"` | `"Inline"` | `"Nonreusable function"` | `"Reusable function"`
Default: `"Auto"`

Version History

Introduced in R2012b

R2022b: New keyboard shortcut

Behavior changed in R2022b

The keyboard shortcut to append a transition column to a state transition table is now **Ctrl+K**. In previous releases, the shortcut was **Ctrl+M**.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

Tunable Parameters

You can use a tunable parameter in a State Transition Table intended for HDL code generation. For details, see “Generate DUT Ports for Tunable Parameters” (HDL Coder).

HDL Architecture

This block has one default HDL architecture.

Active State Output

To generate an output port in the HDL code that shows the active state, in the Properties window of the chart, select **Create output for monitoring**. The output is an enumerated data type. See “Simplify Stateflow Charts by Incorporating Active State Output”.

HDL Block Properties

ConstMultiplierOptimization	Canonical signed digit (CSD) or factored CSD optimization. The default is `none`. See also “ConstMultiplierOptimization” (HDL Coder).
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
DistributedPipelining	Pipeline register distribution, or register retiming. The default is `inherit`. See also “DistributedPipelining” (HDL Coder).

InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
InstantiateFunctions	Generate a VHDL `entity` or Verilog `module` for each function. The default is `off`. See also “InstantiateFunctions” (HDL Coder).
LoopOptimization	Unroll, stream, or do not optimize loops. The default is `none`. See also “LoopOptimization” (HDL Coder).
MapPersistentVarsToRAM	Map persistent arrays to RAM. The default is `off`. See also “MapPersistentVarsToRAM” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).
ResetType	Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType” (HDL Coder).
SharingFactor	Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing” (HDL Coder).
VariablesToPipeline	**Warning** `VariablesToPipeline` is not recommended. Use `coder.hdl.pipeline` instead. Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

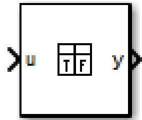
Blocks
Chart | Truth Table

Objects
`Stateflow.StateTransitionTableChart`

Topics
“Use State Transition Tables to Express Sequential Logic in Tabular Form”
“Inspect the Design of State Transition Tables”
“Specify Properties for Stateflow Charts”

Truth Table

Represent logical decision-making behavior with conditions, decisions, and actions



Libraries:
Stateflow

Description

The Truth Table block implements combinatorial logic design in a tabular format. You can use truth table blocks to model decision making for fault detection and management and mode switching. For more information, see “Use Truth Tables to Model Combinatorial Logic”.

To implement control logic, Truth Table blocks use MATLAB as the action language.

Truth Table block properties specify how your truth table interfaces with the Simulink model. You can modify these properties in the **Property Inspector**, the Model Explorer, or the Truth Table properties dialog box. For more information, see “Specify Properties for Truth Table Blocks”. Alternatively, you can modify Truth Table block properties programmatically by using `Stateflow.TruthTableChart` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Ports

Input

u — Input port
scalar | vector | matrix

When you create input data in the **Symbols** pane, Stateflow creates input ports. The input data that you create has a corresponding input port that appears once you create data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus` | `string`

Output

y — Output port
scalar | vector | matrix

When you create output data in the **Symbols** pane, Stateflow creates output ports. The output data that you create has a corresponding output port that appears once you create data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus` | `string`

Parameters

Parameters on the Code Generation tab require Simulink Coder or Embedded Coder.

Main

Show port labels — Select how to display port labels
FromPortIcon (default) | none | FromPortBlockName | SignalName

Select how to display port labels on the Truth Table block icon.

`none`

Do not display port labels.

`FromPortIcon`

If the corresponding port icon displays a signal name, display the signal name on the Truth Table block. Otherwise, display the port block name.

`FromPortBlockName`

Display the name of the corresponding port block on the Truth Table block.

`SignalName`

If a signal name exists, display the name of the signal connected to the port on the Truth Table block. Otherwise, display the name of the corresponding port block.

Programmatic Use
Parameter: `ShowPortLabels`
Type: string scalar or character vector
Value: `"none""FromPortIcon"` | `"FromPortBlockName"` | `"SignalName"`
Default: `"FromPortIcon"`

Read/Write permissions — Select access to contents of chart
ReadWrite (default) | ReadOnly | NoReadOrWrite

Control user access to the contents of the chart.

`ReadWrite`

Enable opening and modification of chart contents.

`ReadOnly`

Enable opening but not modification of the chart. If the chart resides in a block library, you can create and open links to the chart and can make and modify local copies of the chart but you cannot change the permissions or modify the contents of the original library instance.

`NoReadOrWrite`

Disable opening or modification of chart. If the chart resides in a library, you can create links to the chart in a model but you cannot open, modify, change permissions, or create local copies of the chart.

Programmatic Use
Parameter: `Permissions`
Type: string scalar or character vector
Value: `"ReadWrite"` | `"ReadOnly"` | `"NoReadOrWrite"`
Default: `"ReadWrite"`

Minimize algebraic loop occurrences — Control elimination of algebraic loops
off (default) | on

off
: Do not try to eliminate any artificial algebraic loops that include the atomic subchart.

on
: Try to eliminate any artificial algebraic loops that include the atomic subchart.

Programmatic Use
Parameter: `MinAlgLoopOccurrences`
Type: string scalar or character vector
Value: `"off"` | `"on"`
Default: `"off"`

Sample time — Specify time interval
`-1` (default) | `[Ts 0]`

Specify whether all blocks in this chart must run at the same rate or can run at different rates.

- If the blocks in the chart can run at different rates, specify the chart sample time as inherited (`-1`).
- If all blocks must run at the same rate, specify the sample time corresponding to this rate as the value of the **Sample time** parameter.
- If any of the blocks in the chart specify a different sample time (other than `-1` or `inf`), Simulink displays an error message when you update or simulate the model. For example, suppose all the blocks in the chart must run `5` times a second. To ensure this time, specify the sample time of the chart as `0.2`. In this example, if any of the blocks in the chart specify a sample time other than `0.2`, `-1`, or `inf`, Simulink displays an error when you update or simulate the model.

`-1`
: Specify inherited sample time. If the blocks in the chart can run at different rates, use this sample time.

`[Ts 0]`
: Specify periodic sample time.

Programmatic Use
Parameter: `SystemSampleTime`
Type: string scalar or character vector
Value: `"-1"` | `"[Ts 0]"`
Default: `"-1"`

Code Generation

Function packaging — Select code format
`Auto` (default) | `Inline` | `Nonreusable function` | `Reusable function`

Select the generated code format for an atomic (nonvirtual) subchart.

`Auto`
: Simulink Coder chooses the optimal format for your system based on the type and number of instances of the chart that exist in the model.

`Inline`

Simulink Coder inlines the chart unconditionally.

`Nonreusable function`

Simulink Coder explicitly generates a separate function in a separate file. Truth table blocks with this setting generate functions that might have arguments depending on the "Function interface" (Simulink) parameter setting. You can name the generated function and file using parameters "Function name" (Simulink) and "File name (no extension)" (Simulink). These functions are not reentrant.

`Reusable function`

Simulink Coder generates a function with arguments that allows reuse of chart code when a model includes multiple instances of the chart.

This option generates a function with arguments that allows chart code to be reused in the generated code of a model reference hierarchy that includes multiple instances of a chart across referenced models. In this case, the chart must be in a library.

Tips

- When you want multiple instances of a chart represented as one reusable function, you can designate each one of them as `Auto` or as `Reusable function`. It is best to use one because using both creates two reusable functions, one for each designation. The outcomes of these choices differ only when reuse is not possible. Selecting `Auto` does not allow for control of the function or file name for the chart code.
- The `Reusable function` and `Auto` options both determine whether multiple instances of a chart exist and the code can be reused. The options behave differently when it is impossible to reuse the code. In this case, `Auto` yields inlined code, or if circumstances prohibit inlining, separate functions for each chart instance.
- If you select the `Reusable function` while your generated code is under source control, set **File name options** to `Use subsystem name`, `Use function name`, or `User specified`. Otherwise, the names of your code files change whenever you modify your model, which prevents source control on your files.

Dependency

- This parameter requires Simulink Coder.
- Setting this parameter to `Nonreusable function` or `Reusable function` enables the following parameters:
  - **Function name options**
  - **File name options**
  - Memory section for initialize/terminate functions (requires Embedded Coder and an ERT-based system target file)
  - Memory section for execution functions (requires Embedded Coder and an ERT-based system target file)
- Setting this parameter to `Nonreusable function` enables **Function with separate data** (requires a license for Embedded Coder and an ERT-based system target file).

Programmatic Use

Parameter: `RTWSystemCode`
Type: string scalar or character vector

Value: `"Auto"` | `"Inline"` | `"Nonreusable function"` | `"Reusable function"`
Default: `"Auto"`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

Tunable Parameters

You can use a tunable parameter in a Truth Table block intended for HDL code generation. For details, see “Generate DUT Ports for Tunable Parameters” (HDL Coder).

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

ConstMultiplierOptimization	Canonical signed digit (CSD) or factored CSD optimization. The default is `none`. See also “ConstMultiplierOptimization” (HDL Coder).
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
DistributedPipelining	Pipeline register distribution, or register retiming. The default is `inherit`. See also “DistributedPipelining” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
InstantiateFunctions	Generate a VHDL `entity` or Verilog `module` for each function. The default is `off`. See also “InstantiateFunctions” (HDL Coder).
LoopOptimization	Unroll, stream, or do not optimize loops. The default is `none`. See also “LoopOptimization” (HDL Coder).
MapPersistentVarsToRAM	Map persistent arrays to RAM. The default is `off`. See also “MapPersistentVarsToRAM” (HDL Coder).

OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).
ResetType	Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType” (HDL Coder).
SharingFactor	Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing” (HDL Coder).
VariablesToPipeline	**Warning** `VariablesToPipeline` is not recommended. Use `coder.hdl.pipeline` instead. Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Blocks
Chart | State Transition Table

Objects
`Stateflow.TruthTableChart`

Topics
“Use Truth Tables to Model Combinatorial Logic”
“Program a Truth Table”
“Specify Properties for Stateflow Charts”

Functions

sfclipboard

Clipboard object

Syntax

```
clipboard = sfclipboard
```

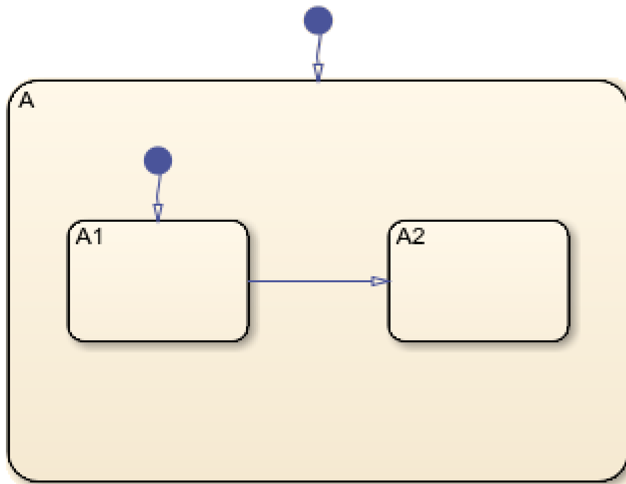
Description

`clipboard = sfclipboard` returns the `Stateflow.Clipboard` object. Use the `Clipboard` object to copy and paste objects within the same chart, between charts in the same Simulink model, or between charts in different models.

Examples

Copy and Paste by Grouping

Group a state and copy its contents to the chart. When you group a state, box, or graphical function, you can copy and paste all the objects contained in the grouped object, as well as all the relationships among these objects. This method is the simplest way of copying and pasting objects programmatically. If a state is not grouped, copying the state does not copy any of its contents.



Open the model and access the `Stateflow.Chart` object for the chart.

```
open_system("sfHierarchyAPIExample")
ch = find(sfroot,"-isa","Stateflow.Chart");
```

Find the `Stateflow.State` object named `A`.

```
sA = find(ch,"-isa","Stateflow.State",Name="A");
```

Group state `A` and its contents by setting the `IsGrouped` property for `sA` to `true`. Save the previous setting of this property so you can revert to it later.

```
prevGrouping = sA.IsGrouped;
sA.IsGrouped = true;
```

Change the name of the state to `Copy_of_A`. Save the previous name so you can revert to it later.

```
prevName = sA.Name;
newName = "Copy_of_"+prevName;
sA.Name = newName;
```

Access the clipboard object.

```
cb = sfclipboard;
```

Copy the grouped state to the clipboard.

```
copy(cb,sA);
```

Restore the state properties to their original settings.

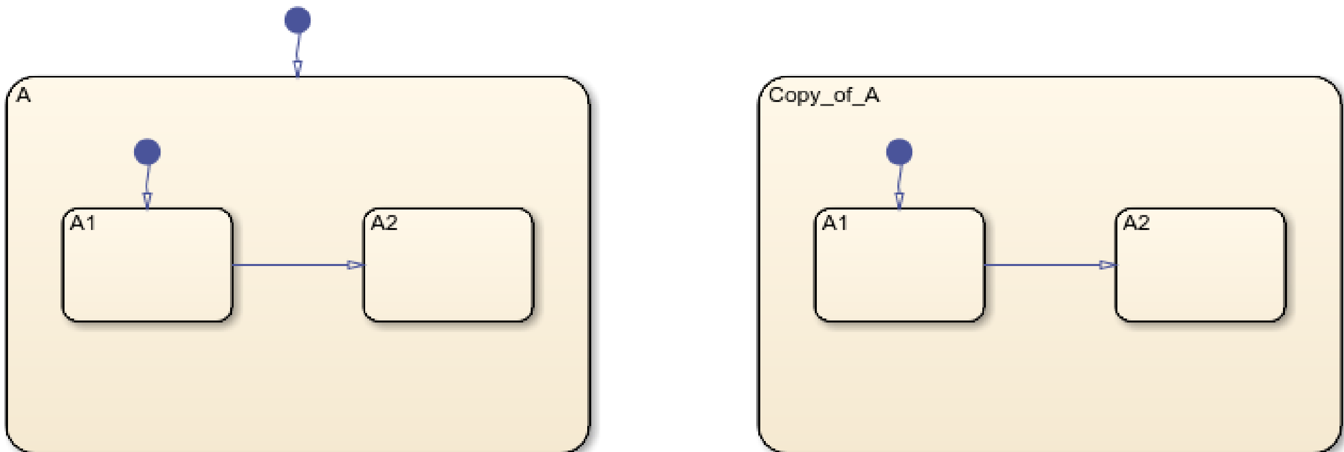
```
sA.IsGrouped = prevGrouping;
sA.Name = prevName;
```

Paste a copy of the objects from the clipboard to the chart.

```
pasteTo(cb,ch);
```

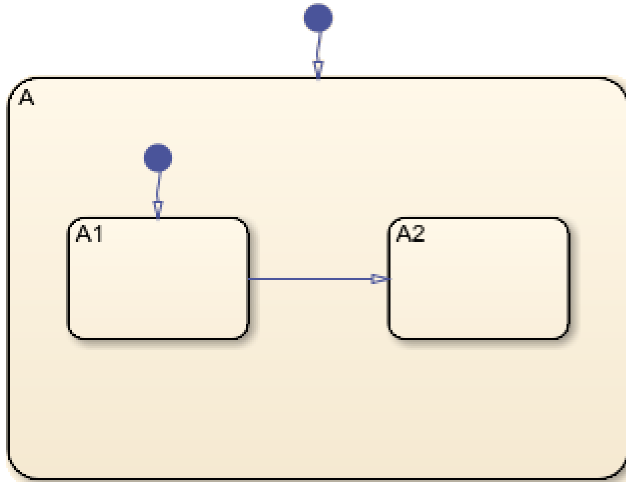
Adjust the state properties of the new state.

```
sNew = find(ch,"-isa","Stateflow.State",Name=newName);
sNew.Position = sA.Position + [400 0 0 0];
sNew.IsGrouped = prevGrouping;
```



Copy and Paste Array of Objects

Copy states A1 and A2, along with the transition between them, to a new state in the chart. To preserve transition connections and containment relationships between objects, copy all the connected objects at once.



Open the model and access the Stateflow.Chart object for the chart.

```
open_system("sfHierarchyAPIExample")
ch = find(sfroot,"-isa","Stateflow.Chart");
```

Find the Stateflow.State object named A.

```
sA = find(ch,"-isa","Stateflow.State",Name="A");
```

Add a new state called B. To enable pasting of other objects inside B, convert the new state to a subchart.

```
sB = Stateflow.State(ch);
sB.Name = "B";
sB.Position = sA.Position + [400 0 0 0];
sB.IsSubchart = true;
```

Create an array called objArray that contains the states and transitions in state A. Use the function setdiff to remove state A from the array of objects to copy.

```
objArrayS = find(sA,"-isa","Stateflow.State");
objArrayS = setdiff(objArrayS,sA);
objArrayT = find(sA,"-isa","Stateflow.Transition");
objArray = [objArrayS objArrayT];
```

Access the clipboard object.

```
cb = sfclipboard;
```

Copy the objects in objArray and paste them in subchart B.

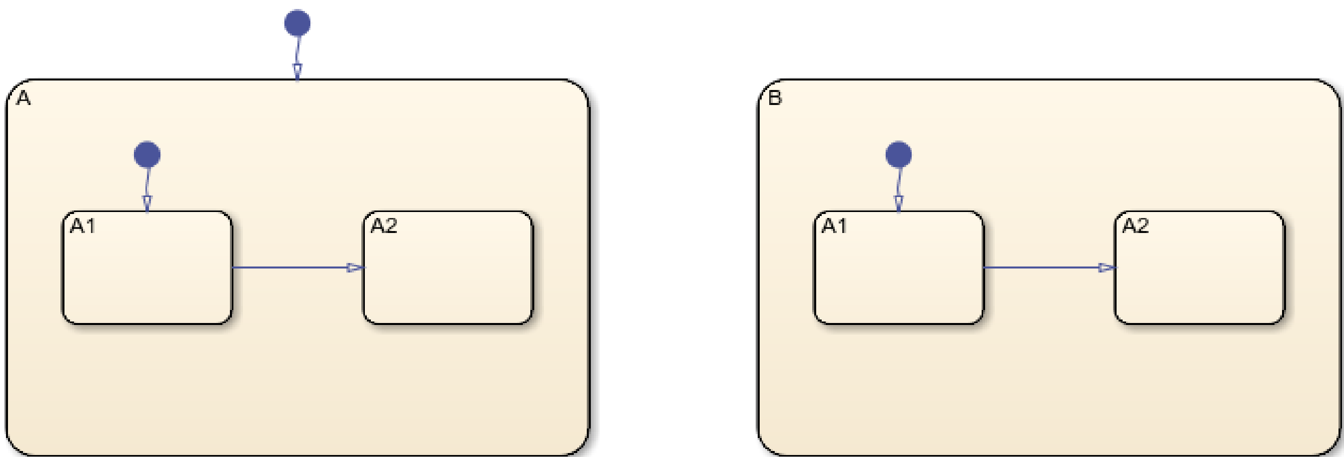
```
copy(cb,objArray);
pasteTo(cb,sB);
```

Revert B to a state.

```
sB.IsSubchart = false;
sB.IsGrouped = false;
```

Reposition the states and transitions in B.

```
newStates = find(sB,"-isa","Stateflow.State");
newStates = setdiff(newStates,sB);

newTransitions = find(sB,"-isa","Stateflow.Transition");
newOClocks = get(newTransitions,{"SourceOClock","DestinationOClock"});

for i = 1:numel(newStates)
    newStates(i).Position = newStates(i).Position + [25 35 0 0];
end
set(newTransitions,{"SourceOClock","DestinationOClock"},newOClocks);
```



Version History

Introduced before R2006a

See Also

Functions
copy | find | pasteTo | setdiff

Objects
Stateflow.Clipboard | Stateflow.State

Topics
"Overview of the Stateflow API"

sfclose

Close Stateflow chart

Syntax

```
sfclose
sfclose all
sfclose chartName
sfclose( ___ )
```

Description

`sfclose` closes the chart that was opened or modified most recently. Closing a chart in a Simulink model also closes the model.

`sfclose all` closes all open charts.

`sfclose chartName` closes all open charts called `chartName`.

`sfclose( ___ )` enables you to specify the input arguments in the previous syntaxes by using variables or strings. For example, you can enter `sfclose(var)` where `var` is a variable set to `"My Chart"` or `"all"`.

Examples

Close Current Chart

Close the Stateflow chart that was opened or modified most recently.

```
sfclose
```

Close All Open Charts

Close all open Stateflow charts.

```
sfclose all
```

Close Specified Chart

Close all open Stateflow charts called `MyChart`.

```
sfclose MyChart
```

Specify Chart Name Using a Variable

Close the open Stateflow chart specified by the variable `chart`.

```
chart = "My Chart";
sfclose(chart)
```

Input Arguments

`chartName` — Name of chart

string scalar | character vector

Name of Stateflow chart to close, specified as a string scalar or character vector. If the name of the chart includes spaces, enclose the chart name in quotes. To specify the name of the chart using a variable or a string, call `sfclose` with its input argument enclosed in parentheses.

Example: `sfclose MyChart`

Example: `sfclose("My Chart")`

Data Types: `char` | `string`

Version History

Introduced in R2006a

See Also

`sfnew` | `sfopen` | `sflib` | `sfsave`

sfdebugger

Open Breakpoints and Watch window

Syntax

`sfdebugger`

Description

`sfdebugger` opens the Stateflow Breakpoints and Watch window. In this window, you can manage the breakpoints in a chart and view the current data and message values while the simulation is paused at a breakpoint.

- To see a list of all of the breakpoints and their associated conditions, select the **Breakpoints** tab. For more information, see "Manage Breakpoints Through the Breakpoints and Watch Window".
- To inspect data and message values, select the **Watch** tab. For more information, see "View Data in the Breakpoints and Watch Window".

Version History

Introduced in R2006a

See Also

`sfnew` | `sfopen` | `sflib` | `sfexplr`

Topics

"Set Breakpoints to Debug Charts"
"Inspect and Modify Data and Messages While Debugging"

sfexplr

Open Model Explorer

Syntax

```
sfexplr
```

Description

`sfexplr` opens the Model Explorer. If the Model Explorer is already open, but not visible, then `sfexplr` brings it to the foreground. A Simulink model does not need to be open.

Version History

Introduced in R2006a

See Also

Tools
Model Explorer

Functions
`sfnew` | `sfopen` | `sflib` | `sfdebugger`

Topics
"Use the Model Explorer with Stateflow Objects"

sfgco

Selected objects in chart

Syntax

```
objects = sfgco
```

Description

`objects = sfgco` returns a handle or vector of handles to the most recently selected Stateflow objects. If more than one chart is open, the function searches the last chart with which you interacted that is still open.

Examples

Zoom in on Selected State

In the Stateflow Editor, select a state by clicking on it.

Access the `Stateflow.State` object.

```
myState = sfgco;
```

Zoom in on the selected state.

```
fitToView(myState)
```

Display Names of Selected States

In the Stateflow Editor, simultaneously select several states by clicking each state while pressing the **Shift** key.

Access the `Stateflow.State` objects.

```
myStates = sfgco;
```

Display the names of the selected states.

```
get(mystates,"Name")
```

Output Arguments

`objects` — Selected graphical objects
handle | vector of handles

Selected graphical objects, returned as a handle or vector of handles to Stateflow API objects. This table describes the format and content of the output of the function, depending on your selection.

Value	Description
Empty matrix	You have not opened or edited any charts.
Handle to the chart most recently clicked	You clicked in a chart, but did not select any objects.
Handle to the selected object	You selected one object in a chart.
Vector of handles to the selected objects	You selected multiple objects in a chart.
Vector of handles to the most recently selected objects in the most recently selected chart	You selected multiple objects in multiple charts.

Version History

Introduced before R2006a

See Also

Functions
`find` | `fitToView`

Objects
`Stateflow.State`

Topics
"Overview of the Stateflow API"
"Access Objects in Your Stateflow Chart"
"Create Charts by Using the Stateflow API"

sfhelp

Open Stateflow documentation in Help browser

Syntax

```
sfhelp
```

Description

`sfhelp` opens the Stateflow documentation in the MATLAB Help browser.

Version History

Introduced before R2006a

See Also

`doc` | `docsearch` | `help` | `lookfor`

sflib

Open Stateflow block library

Syntax

```
sflib
```

Description

`sflib` opens the Stateflow block library. From this library, you can drag Stateflow charts, State Transition Table blocks, Truth Table blocks, and Sequence Viewer blocks into Simulink models.

Version History

Introduced in R2006a

See Also

Blocks
Chart | State Transition Table | Truth Table | Sequence Viewer

Functions
`sfnew` | `sfopen` | `sfdebugger` | `sfexplr`

sfnew

Create Simulink model that contains an empty Stateflow block

Syntax

```
sfnew
sfnew chartType
sfnew modelName
sfnew chartType modelName
sfnew( ___ )
```

Description

`sfnew` creates an untitled Simulink model that contains an empty Stateflow chart.

`sfnew chartType` creates an untitled model that contains an empty block of type `chartType`.

`sfnew modelName` creates a model called `modelName` that contains an empty chart.

`sfnew chartType modelName` creates a model called `modelName` that contains an empty block of type `chartType`.

`sfnew( ___ )` enables you to specify the input arguments in the previous syntaxes by using variables or strings. For example, you can enter `sfnew(var1,var2)` where `var1` is a variable set to `"-C"` and `var2` is a variable set to `"MyModel"`.

Examples

Create Untitled Model with Chart

Create an untitled model that contains an empty Stateflow chart that uses the default action language for new charts.

```
sfnew
```

For more information, see “Modify the Action Language for a Chart”.

Create Untitled Model with Truth Table

Create an untitled model called `MyModel` that contains an empty Truth Table block.

```
sfnew -TT
```

Create Named Model with Chart

Create a model called `MyModel` that contains an empty Stateflow chart that uses MATLAB as the action language.

```
sfnew MyModel
```

Create Named Model with Moore Chart

Create a model called `MyModel` that contains an empty Stateflow chart that uses Moore semantics.

```
sfnew -Moore MyModel
```

Specify Chart Type Using a Variable

Create an untitled model that contains an empty Stateflow chart of the type specified by the variable `type`.

```
type = "-C";
sfnew(type)
```

Input Arguments

`chartType` — Type of block
`-MATLAB` (default) | `-M` | `-C` | `-Mealy` | `-Moore` | `-STT` | `-TT`

Type of Stateflow block to add to empty model, specified as one of these options:

- `-MATLAB` or `-M` — Chart that uses MATLAB as the action language
- `-C` — Chart that uses C as the action language
- `-Mealy` — Chart that supports Mealy machine semantics
- `-Moore` — Chart that supports Moore machine semantics
- `-STT` — State Transition Table
- `-TT` — Truth Table

To specify the type of the block using a variable or a string, call `sfnew` with its input arguments enclosed in parentheses.

Example: `sfnew -MATLAB`

Example: `sfnew("-MATLAB")`

`modelName` — Name of model
string scalar | character vector

Name of the Simulink model, specified as a string scalar or character vector. To specify the name of the model using a variable or a string, call `sfnew` with its input arguments enclosed in parentheses.

Example: `sfnew MyModel`

Example: `sfnew("MyModel")`

Data Types: `char` | `string`

Tips

- The default action language for new charts is MATLAB. To change the default action language, use the `sfpref` function. For example, to change the default action language to C, enter:

  ```
  sfpref(ActionLanguage="C")
  ```

  For more information, see “Modify the Action Language for a Chart”.
- To create a standalone chart that you can execute as a MATLAB object, use the `edit` function. For example, in the MATLAB Command Window, enter:

  ```
  edit chart.sfx
  ```

  For more information, see “Create Stateflow Charts for Execution as MATLAB Objects”.

Version History

Introduced before R2006a

See Also

Blocks
Chart | State Transition Table | Truth Table

Functions
`sfopen` | `sfclose` | `sflib` | `sfpref`

Topics
“Differences Between MATLAB and C as Action Language Syntax”
“Overview of Mealy and Moore Machines”
“Use Truth Tables to Model Combinatorial Logic”
“Use State Transition Tables to Express Sequential Logic in Tabular Form”

sfopen

Open Simulink model

Syntax

```
sfopen
```

Description

`sfopen` prompts you to select a Simulink model file and opens the model.

Tips

To open a standalone chart in MATLAB, use the `edit` function. For example, in the MATLAB Command Window, enter:

```
edit chart.sfx
```

For more information, see “Create Stateflow Charts for Execution as MATLAB Objects”.

Version History

Introduced in R2006a

See Also

`sfnew` | `sflib` | `sfclose`

sfpref

Set preferences for Stateflow charts

Syntax

```
allSettings = sfpref
```

```
setting = sfpref(preference)
```

```
setting = sfpref(ActionLanguage=actionLanguage)
setting = sfpref(PatternWizardCustomDir=customPatternFolder)
setting = sfpref(EnableLabelAutoCorrectionForMAL=autoCorrection)
setting = sfpref(ShowTransitionLabelOwner=transitionLabelLines)
```

Description

`allSettings = sfpref` returns the Stateflow preferences and settings.

`setting = sfpref(preference)` returns the setting for the specified preference.

`setting = sfpref(ActionLanguage=actionLanguage)` sets the default action language used by new Stateflow charts and state transition tables. For more information, see “Change the Default Action Language”.

`setting = sfpref(PatternWizardCustomDir=customPatternFolder)` sets the custom pattern folder used by the Pattern Wizard. For more information, see “Save Custom Flow Chart Patterns”.

`setting = sfpref(EnableLabelAutoCorrectionForMAL=autoCorrection)` enables or disables automatic correction of common C constructs in Stateflow charts that use MATLAB as the action language. For more information, see “Auto Correction When Using MATLAB as the Action Language”.

`setting = sfpref(ShowTransitionLabelOwner=transitionLabelLines)` enables or disables indicator lines between transitions and associated labels.

Examples

Display Settings for All Preferences

Display the settings for all Stateflow preferences.

```
sfpref
```

```
ans = 

  struct with fields:

                              ActionLanguage: 'MATLAB'
             EnableLabelAutoCorrectionForMAL: 1
```

```
           PatternWizardCustomDir: ''
         ShowTransitionLabelOwner: 0
```

Display Default Action Language

Display the default action language used by new Stateflow charts and state transition tables.

```
sfpref("ActionLanguage")
```

```
ans =

    'MATLAB'
```

Change Default Action Language

Change the default action language used by new Stateflow charts and state transition tables to C.

```
sfpref(ActionLanguage="C")
```

```
ans =

    'C'
```

Set Custom Pattern Folder

Set the custom pattern folder used by the Pattern Wizard to `C:\patterns`.

```
sfpref(PatternWizardCustomDir=fullfile("C:","patterns"))
```

```
ans =

    'C:\patterns'
```

Disable Automatic Correction of C Constructs

Disable automatic correction of common C constructs in charts that use MATLAB as the action language.

```
sfpref(EnableLabelAutoCorrectionForMAL=false)
```

```
ans =

    0
```

Display Indicator Lines Between Transitions and Labels

Display an indicator line between every transition and the associated label.

```
sfpref(ShowTransitionLabelOwner=true)
```

```
ans =

    1
```

Input Arguments

`preference` — Stateflow preference to return
`"ActionLanguage"` | `"PatternWizardCustomDir"` | `"EnableLabelAutoCorrectionForMAL"` | `"ShowTransitionLabelOwner"`

Stateflow preference to return, specified as one of these values:

- `"ActionLanguage"` — Default action language used by new Stateflow charts and state transition tables
- `"PatternWizardCustomDir"` — Custom pattern folder used by the Pattern Wizard
- `"EnableLabelAutoCorrectionForMAL"` — Whether Stateflow charts that use MATLAB as the action language automatically correct common C constructs
- `"ShowTransitionLabelOwner"` — Whether indicator lines appear between transitions and associated labels

`actionLanguage` — Default action language
`"MATLAB"` (default) | `"C"`

Default action language used by new Stateflow charts and state transition tables, specified as `"MATLAB"` or `"C"`.

`customPatternFolder` — Custom pattern folder
`""` (default) | string scalar | character vector

Custom pattern folder used by the Pattern Wizard, specified as a string scalar or character vector.

Data Types: `string` | `char`

`autoCorrection` — Whether to enable automatic correction of C constructs
`true` or `1` (default) | `false` or `0`

Whether to enable automatic correction of common C constructs in Stateflow charts that use MATLAB as the action language, specified as a numeric or logical `1` (`true`) or `0` (`false`).

`transitionLabelLines` — Whether to display indicator lines between transitions and labels
`false` or `0` (default) | `true` or `1`

Whether to display the indicator lines between transitions and associated labels, specified as a numeric or logical `1` (`true`) or `0` (`false`), where:

- `true` — Displays indicator lines for every transition
- `false` — Displays indicator lines only when label ownership is unclear

Output Arguments

`allSettings` — Settings for all preferences
structure

Settings for all Stateflow preferences, returned as a structure with these fields:

- `ActionLanguage` — Default action language used by new Stateflow charts and state transition tables, returned as `'MATLAB'` or `'C'`
- `PatternWizardCustomDir` — Custom pattern folder used by the Pattern Wizard, returned as a character vector
- `EnableLabelAutoCorrectionForMAL` — Whether Stateflow charts that use MATLAB as the action language automatically correct common C constructs, returned as `1` or `0` of data type `double`
- `ShowTransitionLabelOwner` — Whether indicator lines appear between transitions and associated labels, returned as `1` or `0` of data type `double`

`setting` — Setting for specified preference
any data type, depending on the preference

Setting for specified preference, returned in the format determined by the preference:

- `ActionLanguage` — `'MATLAB'` or `'C'`
- `PatternWizardCustomDir` — character vector
- `EnableLabelAutoCorrectionForMAL` — `1` or `0` of data type `double`
- `ShowTransitionLabelOwner` — `1` or `0` of data type `double`

Data Types: `char` | `double`

Version History

Introduced before R2006a

See Also

`sfnew` | `sfclose` | `sflib` | `sfopen` | `stateflow`

Topics

"Modify the Action Language for a Chart"
"Create Flow Charts by Using Pattern Wizard"

sfprint

Print Stateflow charts

Syntax

```
sfprint
sfprint(source)
sfprint(source,format)
sfprint(source,format,destination)
sfprint(source,format,destination,wholeChart)
```

Description

`sfprint` prints the current chart to the default printer.

`sfprint(source)` prints all charts specified by `source` to the default printer.

`sfprint(source,format)` prints charts by using the specified `format` to output files. Each output file name matches the name of the chart and the file extension matches the `format`.

`sfprint(source,format,destination)` prints charts to the specified `destination`.

`sfprint(source,format,destination,wholeChart)` specifies whether to print the complete or current view of the charts.

Examples

Print open chart

```
sfprint
```

Prints the current chart to the default printer.

Print all charts specified in path

```
sfprint("sf_car/shift_logic");
```

Prints the chart with the path `sf_car/shift_logic` to the default printer.

Print chart specified in path to a JPG file format.

```
sfprint("sf_car/shift_logic","jpg")
```

Prints the chart `sf_car/shift_logic` in JPG format to the file `sf_car_shift_logic.jpg`.

Print chart in TIFF format to the clipboard.

```
sfprint(gcs,"tiff","clipboard")
```

Prints the chart in the current system to the clipboard in TIFF format.

Print the current view of a chart.

```
sfprint("sf_car/shift_logic","png","file",0)
```

Prints the current view of `sf_car/shift_logic` in a PNG format to the file `sf_car_shift_logic.png`.

Input Arguments

`source` — Source of charts to print

string scalar | character vector | vector of string scalars | cell array of character vectors

Source of charts to print, specified as a string scalar or character vector that contains the path of a chart, model, subsystem, or block. To specify multiple paths, use a vector of string scalars or cell array of character vectors. To specify the current block or system of the model, use `gcb` or `gcs`.

Example: `sfprint(gcs)` prints all the charts in the current system to the default printer.

Example: `sfprint("sf_pool/Pool")` prints the chart `sf_pool/Pool` to the default printer.

Data Types: `string` | `char`

`format` — Output format

`"bitmap"` | `"jpg"` | `"meta"` | `"pdf"` | `"png"` | `"svg"` | `"tiff"`

Output format of the printed charts specified as one of these values:

- `"bitmap"` — Save the chart image to the clipboard as a bitmap (for Windows® operating systems only).
- `"jpg"` — Generate a JPEG file.
- `"meta"` — Save the chart image to the clipboard as an enhanced metafile (for Windows operating systems only).
- `"pdf"` — Generate a PDF file.
- `"png"` — Generate a PNG file.
- `"svg"` — Generate an SVG file.
- `"tiff"` — Generate a TIFF file

Example: `sfprint("sf_car/shift_logic","jpg")` prints the chart `sf_car/shift_logic` to a JPEG file named `sf_car_shift_logic.jpg` in the current folder.

Example: `sfprint("sf_bounce/BouncingBall","meta","myImage")` prints the chart `sf_bounce/BouncingBall` as an enhanced metafile named `myImage.emf` in the current folder.

`destination` — Destination for printed charts

`"file"` (default) | `"clipboard"` | `"printer"` | `"promptForFile"` | string scalar | character vector

Destination for printed charts, specified as one of these values:

- `"file"` — Send output to a file with the default name *`chart_name.file_extension`*, where the file name is the name of the chart and the file extension that matches the output format.
- `"clipboard"` — Copy output to the clipboard.
- `"printer"` — Send output to the default printer. Use only with `"ps"` or `"eps"` formats.
- `"promptForFile"` — Prompt for path and file name.

Alternatively, you can specify the name of the output file by using a string scalar or character vector.

Example: `sfprint("sf_car/shift_logic","png","myFile")` prints the chart `sf_car/shift_logic` to a PNG file named `myFile.png` in the current folder.

Example: `sfprint(gcb,"pdf","promptForFile")` prints all charts in the current block of the model in PDF format. A dialog box prompts you for the path and name of the output file for each chart.

Data Types: `string` | `char`

`wholeChart` — Whether to print complete charts
`true` or 1 (default) | `false` or 0

Whether to print the complete charts, specified as a numeric or logical 1 (`true`) or 0 (`false`).

- `true` — Print the complete views of the specified charts.
- `false` — Print the current views of the specified charts.

Example: `sfprint(gcs,"png","file",0)` prints the current view of the charts in the current system in PNG format using default file names.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Version History

Introduced before R2006a

See Also

`gcb` | `gcs` | `sfhelp` | `sfnew` | `sfsave` | `stateflow`

sfroot

Root of Stateflow hierarchy

Syntax

`root = sfroot`

Description

`root = sfroot` returns the `Simulink.Root` object at the top level of the Stateflow hierarchy of objects. Use the `Root` object to access all other API objects in your charts. For more information, see “Access Objects in Your Stateflow Chart”.

Examples

Zoom in on State in Chart

Open a Simulink model called `myModel`. Suppose that the model contains a Stateflow chart with a state named `A`.

```
open_system("myModel")
```

Find the state named `A`.

```
st = find(sfroot,"-isa","Stateflow.State",Name="A");
```

Zoom in on the state in the Stateflow Editor.

```
fitToView(st);
```

Version History

Introduced before R2006a

See Also

Functions
`find` | `fitToView` | `open_system`

Objects
`Stateflow.State`

Topics
“Overview of the Stateflow API”
“Access Objects in Your Stateflow Chart”
“Create Charts by Using the Stateflow API”

sfsave

Save Simulink model

Syntax

```
sfsave
sfsave modelName
sfsave modelName newModelName
sfsave(___)
```

Description

`sfsave` saves the current model in the current folder. The current folder must be writable.

`sfsave modelName` saves the specified model in the current folder. The specified model must be open and the current folder must be writable.

`sfsave modelName newModelName` saves the specified model using a new model name. The specified model must be open and the current folder must be writable.

`sfsave(___)` enables you to specify the input arguments in the previous syntaxes by using variables or strings. For example, you can enter `sfsave(var1,var2)` where `var1` is a variable set to `"OldModel"` and `var2` is a variable set to `"NewModel"`.

Examples

Save Current Model

Save the model that is currently open.

```
sfsave
```

Save Specified Model

Save an open model called `MyModel`.

```
sfsave MyModel
```

Rename Model

Save a model called `MyModel` using the name `MyNewModel`.

```
sfsave MyModel MyNewModel
```

Specify Model Name Using a Variable

Save the model whose name is specified by the variable `model`.

```
model = "MyModel";
sfsave(model)
```

Input Arguments

`modelName` — Name of model
string scalar | character vector

Name of the Simulink model, specified as a string scalar or character vector. To specify the name of the model using a variable or a string, call `sfnew` with its input arguments enclosed in parentheses.

Example: `sfsave MyModel`

Example: `sfsave("MyModel")`

Data Types: `char` | `string`

`newModelName` — Name of new model
string scalar | character vector

Name of the new Simulink model, specified as a string scalar or character vector. To specify the name of the new model using a variable or a string, call `sfnew` with its input arguments enclosed in parentheses.

Example: `sfsave MyModel MyNewModel`

Example: `sfsave("MyModel","MyNewModel")`

Data Types: `char` | `string`

Version History

Introduced before R2006a

See Also

`sfnew` | `sfopen` | `sfclose`

Topics

"Create Charts by Using the Stateflow API"
"Create Charts by Using a MATLAB Script"

Simulink.sdi.compareRuns

Package: `Simulink.sdi`

Compare data in two simulation runs

Syntax

```
diffResult = Simulink.sdi.compareRuns(runID1,runID2)
diffResult = Simulink.sdi.compareRuns(runID1,runID2,Name=Value)
```

Description

`diffResult = Simulink.sdi.compareRuns(runID1,runID2)` compares the data in the runs that correspond to `runID1` and `runID2` and returns the result in the `Simulink.sdi.DiffRunResult` object `diffResult`. For more information about the comparison algorithm, see “How the Simulation Data Inspector Compares Data” (Simulink).

`diffResult = Simulink.sdi.compareRuns(runID1,runID2,Name=Value)` compares the simulation runs that correspond to `runID1` and `runID2` using the options specified by one or more name-value arguments. For more information about comparison options, see “How the Simulation Data Inspector Compares Data” (Simulink).

Examples

Compare Runs with Global Tolerance

You can specify global tolerance values to use when comparing two simulation runs. Global tolerance values are applied to all signals within the run. This example shows how to specify global tolerance values for a run comparison and how to analyze and save the comparison results.

First, load the session file that contains the data to compare. The session file contains data for four simulations of an aircraft longitudinal controller. This example compares data from two runs that use different input filter time constants.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

To access the run data to compare, use the `Simulink.sdi.getAllRunIDs` (Simulink) function to get the run IDs that correspond to the last two simulation runs.

```
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);
```

Use the `Simulink.sdi.compareRuns` (Simulink) function to compare the runs. Specify a global relative tolerance value of `0.2` and a global time tolerance value of `0.5`.

```
runResult = Simulink.sdi.compareRuns(runID1,runID2,'reltol',0.2,'timetol',0.5);
```

Check the `Summary` property of the returned `Simulink.sdi.DiffRunResult` object to see whether signals were within the tolerance values or out of tolerance.

```
runResult.Summary
```

```
ans = struct with fields:
       OutOfTolerance: 0
      WithinTolerance: 3
            Unaligned: 0
        UnitsMismatch: 0
                Empty: 0
             Canceled: 0
          EmptySynced: 0
     DataTypeMismatch: 0
         TimeMismatch: 0
    StartStopMismatch: 0
          Unsupported: 0
```

All three signal comparison results fell within the specified global tolerance.

You can save the comparison results to an MLDATX file using the `saveResult` (Simulink) function.

```
saveResult(runResult,'InputFilterComparison');
```

Analyze Simulation Data Using Signal Tolerances

You can programmatically specify signal tolerance values to use in comparisons performed using the Simulation Data Inspector. In this example, you compare data collected by simulating a model of an aircraft longitudinal flight control system. Each simulation uses a different value for the input filter time constant and logs the input and output signals. You analyze the effect of the time constant change by comparing results using the Simulation Data Inspector and signal tolerances.

First, load the session file that contains the simulation data.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

The session file contains four runs. In this example, you compare data from the first two runs in the file. Access the `Simulink.sdi.Run` objects for the first two runs loaded from the file.

```
runIDs = Simulink.sdi.getAllRunIDs;
runIDTs1 = runIDs(end-3);
runIDTs2 = runIDs(end-2);
```

Now, compare the two runs without specifying any tolerances.

```
noTolDiffResult = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
```

Use the `getResultByIndex` function to access the comparison results for the `q` and `alpha` signals.

```
qResult = getResultByIndex(noTolDiffResult,1);
alphaResult = getResultByIndex(noTolDiffResult,2);
```

Check the `Status` of each signal result to see whether the comparison result fell within our out of tolerance.

```
qResult.Status
```

```
ans = 
  ComparisonSignalStatus enumeration
```

```
    OutOfTolerance
```

```
alphaResult.Status
```

```
ans = 
  ComparisonSignalStatus enumeration

    OutOfTolerance
```

The comparison used a value of `0` for all tolerances, so the `OutOfTolerance` result means the signals are not identical.

You can further analyze the effect of the time constant by specifying tolerance values for the signals. Specify the tolerances by setting the properties for the `Simulink.sdi.Signal` objects that correspond to the signals being compared. Comparisons use tolerances specified for the baseline signals. This example specifies a time tolerance and an absolute tolerance.

To specify a tolerance, first access the `Signal` objects from the baseline run.

```
runTs1 = Simulink.sdi.getRun(runIDTs1);
qSig = getSignalsByName(runTs1,'q, rad/sec');
alphaSig = getSignalsByName(runTs1,'alpha, rad');
```

Specify an absolute tolerance of `0.1` and a time tolerance of `0.6` for the `q` signal using the `AbsTol` and `TimeTol` properties.

```
qSig.AbsTol = 0.1;
qSig.TimeTol = 0.6;
```

Specify an absolute tolerance of `0.2` and a time tolerance of `0.8` for the `alpha` signal.

```
alphaSig.AbsTol = 0.2;
alphaSig.TimeTol = 0.8;
```

Compare the results again. Access the results from the comparison and check the `Status` property for each signal.

```
tolDiffResult = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
qResult2 = getResultByIndex(tolDiffResult,1);
alphaResult2 = getResultByIndex(tolDiffResult,2);
```

```
qResult2.Status
```

```
ans = 
  ComparisonSignalStatus enumeration

    WithinTolerance
```

```
alphaResult2.Status
```

```
ans = 
  ComparisonSignalStatus enumeration

    WithinTolerance
```

Configure Comparisons to Check Metadata

You can use the Simulink.sdi.compareRuns function to compare signal data and metadata, including data type and start and stop times. A single comparison may check for mismatches in one or more pieces of metadata. When you check for mismatches in signal metadata, the Summary property of the Simulink.sdi.DiffRunResult object may differ from a basic comparison because the Status property for a Simulink.sdi.DiffSignalResult object can indicate the metadata mismatch. You can configure comparisons using the Simulink.sdi.compareRuns function for imported data and for data logged from a simulation.

This example configures a comparison of runs created from workspace data three ways to show how the Summary of the DiffSignalResult object can provide specific information about signal mismatches.

Create Workspace Data

The Simulink.sdi.compareRuns function compares time series data. Create data for a sine wave to use as the baseline signal, using the timeseries format. Give the timeseries the name Wave Data.

```
time = 0:0.1:20;
sig1vals = sin(2*pi/5*time);
sig1_ts = timeseries(sig1vals,time);
sig1_ts.Name = 'Wave Data';
```

Create a second sine wave to compare against the baseline signal. Use a slightly different time vector and attenuate the signal so the two signals are not identical. Cast the signal data to the single data type. Also name this timeseries object Wave Data. The Simulation Data Inspector comparison algorithm will align these signals for comparison using the name.

```
time2 = 0:0.1:22;
sig2vals = single(0.98*sin(2*pi/5*time2));
sig2_ts = timeseries(sig2vals,time2);
sig2_ts.Name = 'Wave Data';
```

Create and Compare Runs in the Simulation Data Inspector

The Simulink.sdi.compareRuns function compares data contained in Simulink.sdi.Run objects. Use the Simulink.sdi.createRun function to create runs in the Simulation Data Inspector for the data. The Simulink.sdi.createRun function returns the run ID for each created run.

```
runID1 = Simulink.sdi.createRun('Baseline Run','vars',sig1_ts);
runID2 = Simulink.sdi.createRun('Compare to Run','vars',sig2_ts);
```

You can use the Simulink.sdi.compareRuns function to compare the runs. The comparison algorithm converts the signal data to the double data type and synchronizes the signal data before computing the difference signal.

```
basic_DRR = Simulink.sdi.compareRuns(runID1,runID2);
```

Check the Summary property of the returned Simulink.sdi.DiffRunResult object to see the result of the comparison.

```
basic_DRR.Summary

ans = struct with fields:
        OutOfTolerance: 1
       WithinTolerance: 0
             Unaligned: 0
         UnitsMismatch: 0
                 Empty: 0
              Canceled: 0
           EmptySynced: 0
      DataTypeMismatch: 0
          TimeMismatch: 0
     StartStopMismatch: 0
           Unsupported: 0
```

The difference between the signals is out of tolerance.

Compare Runs and Check for Data Type Match

Depending on your system requirements, you may want the data types for signals you compare to match. You can use the `Simulink.sdi.compareRuns` function to configure the comparison algorithm to check for and report data type mismatches.

```
dataType_DRR = Simulink.sdi.compareRuns(runID1,runID2,'DataType','MustMatch');
dataType_DRR.Summary

ans = struct with fields:
        OutOfTolerance: 0
       WithinTolerance: 0
             Unaligned: 0
         UnitsMismatch: 0
                 Empty: 0
              Canceled: 0
           EmptySynced: 0
      DataTypeMismatch: 1
          TimeMismatch: 0
     StartStopMismatch: 0
           Unsupported: 0
```

The result of the signal comparison is now `DataTypeMismatch` because the data for the baseline signal is `double` data type, while the data for the signal compared to the baseline is `single` data type.

Compare Runs and Check for Start and Stop Time Match

You can use the `Simulink.sdi.compareRuns` function to configure the comparison algorithm to check whether the aligned signals have the same start and stop times.

```
startStop_DRR = Simulink.sdi.compareRuns(runID1,runID2,'StartStop','MustMatch');
startStop_DRR.Summary

ans = struct with fields:
        OutOfTolerance: 0
       WithinTolerance: 0
             Unaligned: 0
         UnitsMismatch: 0
```

```
             Empty: 0
          Canceled: 0
       EmptySynced: 0
  DataTypeMismatch: 0
      TimeMismatch: 0
 StartStopMismatch: 1
       Unsupported: 0
```

The signal comparison result is now `StartStopMismatch` because the signals created in the workspace have different stop times.

Compare Runs with Alignment Criteria

When you compare runs using the Simulation Data Inspector, you can specify alignment criteria that determine how signals are paired with each other for comparison. This example compares data from simulations of a model of an aircraft longitudinal control system. The simulations used a square wave input. The first simulation used an input filter time constant of `0.1s` and the second simulation used an input filter time constant of `0.5s`.

First, load the simulation data from the session file that contains the data for this example.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

The session file contains data for four simulations. This example compares data from the first two runs. Access the run IDs for the first two runs loaded from the session file.

```
runIDs = Simulink.sdi.getAllRunIDs;
runIDTs1 = runIDs(end-3);
runIDTs2 = runIDs(end-2);
```

Before running the comparison, define how you want the Simulation Data Inspector to align the signals between the runs. This example aligns signals by their name, then by their block path, and then by their Simulink identifier.

```
alignMethods = [Simulink.sdi.AlignType.SignalName
               Simulink.sdi.AlignType.BlockPath
               Simulink.sdi.AlignType.SID];
```

Compare the simulation data in your two runs, using the alignment criteria you specified. The comparison uses a small time tolerance to account for the effect of differences in the step size used by the solver on the transition of the square wave input.

```
diffResults = Simulink.sdi.compareRuns(runIDTs1,runIDTs2,'align',alignMethods,...
    'timetol',0.005);
```

You can use the `getResultByIndex` function to access the comparison results for the aligned signals in the runs you compared. You can use the `Count` property of the `Simulink.sdi.DiffRunResult` object to set up a `for` loop to check the `Status` property for each `Simulink.sdi.DiffSignalResult` object.

```
numComparisons = diffResults.count;

for k = 1:numComparisons
    resultAtIdx = getResultByIndex(diffResults,k);
```

```
    sigID1 = resultAtIdx.signalID1;
    sigID2 = resultAtIdx.signalID2;

    sig1 = Simulink.sdi.getSignal(sigID1);
    sig2 = Simulink.sdi.getSignal(sigID2);

    displayStr = 'Signals %s and %s: %s \n';
    fprintf(displayStr,sig1.Name,sig2.Name,resultAtIdx.Status);
end
```

```
Signals q, rad/sec and q, rad/sec: OutOfTolerance
Signals alpha, rad and alpha, rad: OutOfTolerance
Signals Stick and Stick: WithinTolerance
```

Input Arguments

`runID1` — Baseline run identifier
integer

Numeric identifier for the baseline run in the comparison, specified as a run ID that corresponds to a run in the Simulation Data Inspector. The Simulation Data Inspector assigns run IDs when runs are created. You can get the run ID for a run by using the `ID` property of the `Simulink.sdi.Run` object, the `Simulink.sdi.getAllRunIDs` function, or the `Simulink.sdi.getRunIDByIndex` function.

`runID2` — Identifier for run to compare
integer

Numeric identifier for the run to compare, specified as a run ID that corresponds to a run in the Simulation Data Inspector. The Simulation Data Inspector assigns run IDs when runs are created. You can get the run ID for a run by using the `ID` property of the `Simulink.sdi.Run` object, the `Simulink.sdi.getAllRunIDs` function, or the `Simulink.sdi.getRunIDByIndex` function.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `AbsTol=x,Align=alignOpts`

`Align` — Signal alignment options
`Simulink.sdi.AlignType` scalar | `Simulink.sdi.AlignType` vector

Signal alignment options, specified as a `Simulink.sdi.AlignType` scalar or vector. The `Simulink.sdi.AlignType` enumeration includes a value for each option available for pairing each signal in the baseline run with a signal in the comparison run. You can specify one or more alignment options for the comparison. To use more than one alignment option, specify an array. When you specify multiple alignment options, the Simulation Data Inspector aligns signals first by the option in the first element of the array, then by the option in the second element array, and so on. For more information, see “Signal Alignment” (Simulink).

Value	Aligns By
`Simulink.sdi.AlignType.BlockPath`	Path to the source block for the signal
`Simulink.sdi.AlignType.SID`	Automatically assigned Simulink identifier
`Simulink.sdi.AlignType.SignalName`	Signal name
`Simulink.sdi.AlignType.DataSource`	Path of the variable in the MATLAB workspace

Example: `[Simulink.sdi.AlignType.SignalName,Simulink.sdi.AlignType.BlockPath]` specifies signal alignment by signal name and then by block path.

`AbsTol` — Global absolute tolerance for comparison
`0` (default) | positive-valued scalar

Global absolute tolerance for comparison, specified as a positive-valued scalar.

Global tolerances apply to all signals in the run comparison. To use a different tolerance value for a signal in the comparison, specify the tolerance you want to use on the `Simulink.sdi.Signal` object in the baseline run and set the `OverrideGlobalTol` property for that signal to `true`.

For more information about how tolerances are used in comparisons, see “Tolerance Specification” (Simulink).

Example: `0.5`

Data Types: `double`

`RelTol` — Global relative tolerance for comparison
`0` (default) | positive-valued scalar

Global relative tolerance for comparison, specified as a positive-valued scalar. The relative tolerance is expressed as a fractional multiplier. For example, `0.1` specifies a 10 percent tolerance.

Global tolerances apply to all signals in the run comparison. To use a different tolerance value for a signal in the comparison, specify the tolerance you want to use on the `Simulink.sdi.Signal` object in the baseline run and set the `OverrideGlobalTol` property for that signal to `true`.

For more information about how tolerances are used in comparisons, see “Tolerance Specification” (Simulink).

Example: `0.1`

Data Types: `double`

`TimeTol` — Global time tolerance for comparison
`0` (default) | positive-valued scalar

Global time tolerance for comparison, specified as a positive-valued scalar, using units of seconds.

Global tolerances apply to all signals in the run comparison. To use a different tolerance value for a signal in the comparison, specify the tolerance you want to use on the `Simulink.sdi.Signal` object in the baseline run and set the `OverrideGlobalTol` property for that signal to `true`.

For more information about tolerances in the Simulation Data Inspector, see “Tolerance Specification” (Simulink).

Example: `0.2`

Data Types: `double`

`DataType` — Comparison sensitivity to signal data types
`"MustMatch"`

Comparison sensitivity to signal data types, specified as `"MustMatch"`. Specify `DataType="MustMatch"` when you want the comparison to be sensitive to numeric data type mismatches in compared signals.

When signal data types do not match, the `Status` property of the `Simulink.sdi.DiffSignalResult` object for the result is set to `DataTypeMismatch`.

The `Simulink.sdi.compareRuns` function compares the data types for aligned signals before synchronizing and comparing the signal data. When you do not specify this name-value argument, the comparison checks data types only to detect a comparison between string and numeric data. For a comparison between string and numeric data, results are not computed, and the status for the result is `DataTypeMismatch`. For aligned signals that have different numeric data types, the comparison computes results.

When you configure the comparison to stop on the first mismatch, a data type mismatch stops the comparison. A stopped comparison may not compute results for all signals.

`Time` — Comparison sensitivity to signal time vectors
`"MustMatch"`

Comparison sensitivity to signal time vectors, specified as `"MustMatch"`. Specify `Time="MustMatch"` when you want the comparison to be sensitive to mismatches in the time vectors of compared signals. When you specify this name-value argument, the algorithm compares the time vectors of aligned signals before synchronizing and comparing the signal data.

When the time vectors for signals do not match, the `Status` property of the `Simulink.sdi.DiffSignalResult` object for the result is set to `TimeMismatch`.

Comparisons are not sensitive to differences in signal time vectors unless you specify this name-value argument. For comparisons that are not sensitive to differences in the time vectors, the comparison algorithm synchronizes the signals prior to the comparison. For more information about how synchronization works, see “How the Simulation Data Inspector Compares Data” (Simulink).

When you specify that time vectors must match and configure the comparison to stop on the first mismatch, a time vector mismatch stops the comparison. A stopped comparison may not compute results for all signals.

`StartStop` — Comparison sensitivity to signal start and stop times
`"MustMatch"`

Comparison sensitivity to signal start and stop times, specified as `"MustMatch"`. Specify `StartStop="MustMatch"` when you want the comparison to be sensitive to mismatches in signal start and stop times. When you specify this name-value argument, the algorithm compares the start and stop times for aligned signals before synchronizing and comparing the signal data.

When the start times and stop times do not match, the `Status` property of the `Simulink.sdi.DiffSignalResult` object for the result is set to `StartStopMismatch`.

When you specify that start and stop times must match and configure the comparison to stop on the first mismatch, a start or stop time mismatch stops the comparison. A stopped comparison may not compute results for all signals.

`StopOnFirstMismatch` — Whether comparison stops on first detected mismatch
`"Metadata"` | `"Any"`

Whether comparison stops on first detected mismatch without comparing remaining signals, specified as `"Metadata"` or `"Any"`. A stopped comparison may not compute results for all signals, and can return a mismatched result more quickly.

- `"Metadata"` — A mismatch in metadata for aligned signals stops the comparison. Metadata comparisons happen before comparing signal data.

  The Simulation Data Inspector always aligns signals and compares signal units. When you configure the comparison to stop on the first mismatch, an unaligned signal or mismatched units always stop the comparison. You can specify additional name-value arguments to configure the comparison to check and stop on the first mismatch for additional metadata, such as signal data type, start and stop times, and time vectors.
- `"Any"` — A mismatch in metadata or signal data for aligned signals stops the comparison.

`ExpandChannels` — Whether to compute comparison results for each channel in multidimensional signals
`true` or `1` (default) | `false` or `0`

Whether to compute comparison results for each channel in multidimensional signals, specified as logical `true` (`1`) or `false` (`0`).

- `true` or `1` — Comparison expands multidimensional signals represented as a single signal with nonscalar sample values to a set of signals with scalar sample values and computes a comparison result for each signal.

  The representation of the multidimensional signal in the Simulation Data Inspector as a single signal with nonscalar sample values does not change.
- `false` or `0` — Comparison does not compute results for multidimensional signals represented as a single signal with nonscalar sample values.

Output Arguments

`diffResult` — Comparison results
`Simulink.sdi.DiffRunResult` object

Comparison results, returned as a `Simulink.sdi.DiffRunResult` object.

Limitations

The Simulation Data Inspector does not support comparing:

- Signals of data types `int64` or `uint64`.
- Variable-size signals.

Version History

Introduced in R2011b

See Also

Functions

`Simulink.sdi.compareSignals` | `Simulink.sdi.getRunIDByIndex` | `Simulink.sdi.getRunCount` | `getResultByIndex`

Objects

`Simulink.sdi.DiffRunResult` | `Simulink.sdi.DiffSignalResult`

Topics

"Inspect and Compare Data Programmatically" (Simulink)
"Compare Simulation Data" (Simulink)
"How the Simulation Data Inspector Compares Data" (Simulink)

stateflow

Open Stateflow block library and create Simulink model that contains an empty chart

Syntax

```
stateflow
```

Description

`stateflow` creates an untitled Simulink model that contains an empty Stateflow chart. The function also opens the Stateflow block library. From this library, you can drag Stateflow blocks into models.

Tips

- To only create a Simulink model that contains an empty Stateflow block, use the `sfnew` function.
- To only open the Stateflow block library, use the `sflib` function.
- To create a standalone chart that you can execute as a MATLAB object, open the Stateflow editor by using the `edit` function. For example, at the MATLAB Command Window, enter:

  ```
  edit chart.sfx
  ```

  For more information, see “Create Stateflow Charts for Execution as MATLAB Objects”.

Version History

Introduced before R2006a

R2019b: Opening Stateflow

Behavior change in future release

The behavior of the `stateflow` function will change in a future release. Use `sfnew` and `sflib` instead.

See Also

`edit` | `sflib` | `sfnew`

Stateflow.exportAsClass

Export MATLAB class for standalone chart

Syntax

```
Stateflow.exportAsClass(source)
Stateflow.exportAsClass(source,destination)
```

Description

`Stateflow.exportAsClass(source)` saves a standalone Stateflow chart as a MATLAB class file in the current folder. The saved file has the same name as the chart. For example, if `source` is `chart.sfx`, the function saves the MATLAB class in the file `chart.m`.

`Stateflow.exportAsClass(source,destination)` saves the chart as a MATLAB class file in the folder `destination`.

Note The MATLAB class produced by `Stateflow.exportAsClass` is intended for debugging purposes only, and not for production use or manual modification. For more information, see “Tips” on page 2-41.

Examples

Export Chart in Current Folder

Save Stateflow chart `chart.sfx` as the MATLAB class file `chart.m` in the current folder.

```
Stateflow.exportAsClass("chart.sfx");
```

Export Chart in Folder Specified by Path

Save Stateflow chart `chart.sfx`, which is located in folder `dir1`, as the MATLAB class file `chart.m` in the current folder.

```
Stateflow.exportAsClass(fullfile("dir1","chart.sfx"));
```

Export Chart to MATLAB Class in Another Folder

Save Stateflow chart `chart.sfx`, which is located in the current folder, as the MATLAB class file `chart.m` in the folder `dir2`.

```
Stateflow.exportAsClass("chart.sfx","dir2");
```

Input Arguments

`source` — Path and file name of standalone Stateflow chart
string scalar | character vector

Path and file name of a standalone chart, specified as a string scalar or character vector. You can use the absolute path from the root folder or the relative path from the current folder. Standalone charts have the extension `.sfx`.

Data Types: `char` | `string`

`destination` — Path of destination folder for MATLAB class file
string scalar | character vector

Path of the destination folder for the MATLAB class file, specified as a string scalar or character vector. You can use the absolute path from the root folder or the relative path from the current folder. If not specified, the function saves the MATLAB script file in the current folder.

Data Types: `char` | `string`

Tips

- Use the code produced by `Stateflow.exportAsClass` to debug run-time errors that are otherwise difficult to diagnose. For example, suppose that you encounter an error while executing a Stateflow chart that controls a MATLAB application. If you export the chart as a MATLAB class file, you can replace the chart with the class in your application and diagnose the error by using the MATLAB debugger.

  Note Error messages produced by the MATLAB class point to different line numbers than the corresponding error messages produced by the Stateflow chart.

- When you execute the MATLAB class produced by `Stateflow.exportAsClass`, the Stateflow Editor does not animate the original chart.

Version History

Introduced in R2019b

See Also

`fullfile`

Topics

"Create Stateflow Charts for Execution as MATLAB Objects"

Stateflow.exportToVersion

Export standalone chart for use in previous version of Stateflow

Syntax

exported_file = Stateflow.exportToVersion(source,file_name,version)

Description

exported_file = Stateflow.exportToVersion(source,file_name,version) exports the chart source to a file named file_name in a format that the specified previous Stateflow version can load. You can only export to R2019a and later releases.

Examples

Export Chart to an Earlier Version of MATLAB

To complete the export process, you need access to the versions of Stateflow from which and to which you are exporting.

Using the later version of Stateflow, convert the standalone chart chart.sfx.

```
edit chart.sfx
Stateflow.exportToVersion("chart","chart_19a.sfx","R2019a")
```

Using the earlier version of Stateflow, open and resave the exported chart.

```
edit chart_19a.sfx
sfsave chart_19a
```

Input Arguments

source — Chart to export
string scalar | character vector

Chart to export, specified as a string scalar or character vector, without any file extension. The chart must be open in the Stateflow Editor and have no unsaved changes.

Example: "chart"

Data Types: char | string

file_name — Exported file name
string scalar | character vector

Exported file name, specified as a string scalar or character vector. The exported file must not have the same name as the source chart.

Example: "chart_19a.sfx"

Data Types: char | string

`version` — MATLAB release name
"R2019a" | "R2019b" | "R2020a" | ...

MATLAB release name, specified as a string scalar or character vector. Release names are case sensitive. You can only export to R2019a and later releases.

Output Arguments

`exported_file` — Absolute path to exported file
character vector

Absolute path to exported file, returned as a character vector.

Tips

Attempting to execute an exported chart before resaving it will result in an error.

Version History

Introduced in R2020a

See Also

`edit` | `sfsave`

Topics
"Create Stateflow Charts for Execution as MATLAB Objects"

Stateflow.findMatchingPort

Identify matching entry or exit port

Syntax

```
matchingPort = Stateflow.findMatchingPort(port)
```

Description

`matchingPort = Stateflow.findMatchingPort(port)` returns a `Stateflow.Port` object that matches the specified port or junction.

Note Typically, `Stateflow.findMatchingPort` returns a single `Stateflow.Port` object. However, when an entry or exit junction is located in the top level of a linked atomic subchart, `Stateflow.findMatchingPort` returns an array that contains a separate `Stateflow.Port` object for each instance of the atomic subchart that is open.

Examples

Add Exit Port and Junction to Atomic Subchart

In an atomic subchart called `A`, add an exit port and an exit junction with the label `"exit"`.

Find the `Stateflow.AtomicSubchart` object that corresponds to the atomic subchart `A` in the chart `ch`.

```
atomicSubchart = find(ch,"-isa","Stateflow.AtomicSubchart",Name="A");
```

Add an exit junction to the atomic subchart. Use the `Subchart` property of the atomic subchart as the parent of the exit junction. Display the value of the `PortType` property of the exit junction.

```
exitJunction = Stateflow.Port(atomicSubchart.Subchart,"ExitJunction");
exitJunction.PortType
```

```
ans =

    'ExitJunction'
```

Set the label of the exit junction to `"exit"`.

```
exitJunction.labelString = "exit";
```

Find the `Stateflow.Port` object for the matching exit port. Display the value of the `PortType` property of the exit port.

```
exitPort = Stateflow.findMatchingPort(exitJunction);
exitPort.PortType
```

```
ans =

    'ExitPort'
```

Display the label of the exit port.

```
exitPort.labelString
```

```
ans =

    'exit'
```

Input Arguments

port — Port or junction
Stateflow.Port object

Port or junction, specified as a `Stateflow.Port` object.

Tips

- If you move an entry or exit junction to a different parent, Stateflow deletes the `Stateflow.Port` object for the matching port and creates a `Stateflow.Port` object on the new parent. To identify the new matching port, use the `Stateflow.findMatchingPort` function.

Version History

Introduced in R2021b

See Also

Functions
`find`

Objects
`Stateflow.AtomicSubchart` | `Stateflow.Port`

Topics
"Overview of the Stateflow API"
"Create Entry and Exit Connections Across State Boundaries"

Operators

after

Execute chart after event broadcast or specified time

Syntax

```
after(n,E)
after(n,tick)
after(n,time_unit)
```

Description

`after(n,E)` returns `true` if the event `E` has occurred at least `n` times since the associated state became active. Otherwise, the operator returns `false`.

`after(n,tick)` returns `true` if the chart has woken up at least `n` times since the associated state became active. Otherwise, the operator returns `false`.

The implicit event `tick` is not supported when a Stateflow chart in a Simulink model has input events.

`after(n,time_unit)` returns `true` if at least `n` units of time have elapsed since the associated state became active. Otherwise, the operator returns `false`.

In charts in a Simulink model, specify `time_unit` as seconds (`sec`), milliseconds (`msec`), or microseconds (`usec`). If you specify `n` as an expression, the chart adjusts the temporal delay as the expression changes value during the simulation.

In standalone charts in MATLAB, specify `n` with a value greater than or equal to `0.001` and `time_unit` as seconds (`sec`). The operator creates a MATLAB `timer` object that generates an implicit event to wake up the chart. MATLAB `timer` objects are limited to 1 millisecond precision. For more information, see “Events in Standalone Charts”.

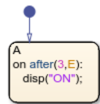
- The `timer` object is created when the chart finishes executing the `entry` actions of the associated state and its substates. If you specify `n` as an expression whose value changes during chart execution, the chart does not adjust the temporal delay of the `timer` object.
- The `timer` object starts running at the end of the chart step when the associated state becomes active. This step can include the execution of other parallel states in the chart.
- If the chart is processing another operation when it receives the implicit event from the `timer` object, the chart queues the event. When the current step is completed, the chart processes the event.
- If the state associated with the temporal logic operator becomes inactive before the chart processes the implicit event, the event does not wake up the chart.

Examples

Execute State Action on Event Broadcast

Display a status message when the chart processes a broadcast of the event E, starting on the third broadcast of E after the state became active.

```
on after(3,E):
   disp("ON");
```



Trigger Transition on Event Broadcast

Transition out of the associated state when the chart processes a broadcast of the event E, starting on the fifth broadcast of E after the state became active.

```
after(5,E)
```

Guard Transition with Temporal Condition

Transition out of the associated state if the state has been active for at least five broadcasts of the event E.

In charts in a Simulink model, enter:

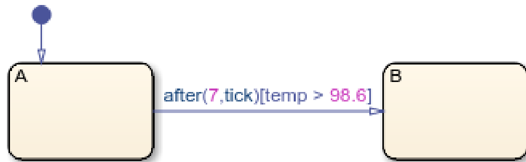
```
[after(5,E)]
```

Conditional notation for temporal logic operators is not supported in standalone charts in MATLAB.

Trigger Transition on Chart Execution

Transition out of the associated state when the chart wakes up for at least the seventh time since the state became active, but only if the variable `temp` is greater than 98.6.

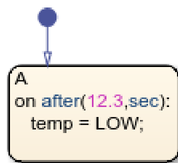
```
after(7,tick)[temp > 98.6]
```



Execute State Action After Specified Time

Set the `temp` variable to `LOW` every time that the chart wakes up, starting when the associated state is active for at least 12.3 seconds.

```
on after(12.3,sec):
   temp = LOW;
```



Tips

- You can use quotation marks to enclose the keywords `'tick'`, `'sec'`, `'msec'`, and `'usec'`. For example, `after(5,'tick')` is equivalent to `after(5,tick)`.
- The Stateflow chart resets the counter used by the `after` operator each time the associated state reactivates.
- The timing for absolute-time temporal logic operators depends on the type of Stateflow chart:

  - Charts in a Simulink model define absolute-time temporal logic in terms of simulation time.
  - Standalone charts in MATLAB define absolute-time temporal logic in terms of wall-clock time, which is limited to 1 millisecond precision.

  The difference in timing can affect the behavior of a chart. For example, suppose that this chart is executing the `during` action of state `A`.

  - In a Simulink model, the function call to `f` executes in a single time step and does not contribute to the simulation time. The transition from state `A` to state `B` occurs the first time the chart wakes up and state `A` has been active for at least 2 seconds. The value displayed by the `entry` action in state `B` depends only on the step size used by the Simulink solver.

- In a standalone chart, the function call to `f` can take several seconds of wall-clock time to complete. If the call lasts more than two seconds, the chart queues the implicit event associated with the `after` operator. The transition from state `A` to state `B` occurs when the function `f` finishes executing. The value displayed by the `entry` action in state `B` depends on the time the function call to `f` takes to complete.

Version History

Introduced in R2014b

See Also

`at` | `before` | `every` | `timer`

Topics

"Control Chart Execution by Using Temporal Logic"
"Use Events to Execute Charts"
"Control Chart Behavior by Using Implicit Events"

ascii2str

Convert array of type `uint8` to string

Syntax

```
str = ascii2str(A)
```

Description

`str = ascii2str(A)` converts ASCII values in array `A` of type `uint8` to a string.

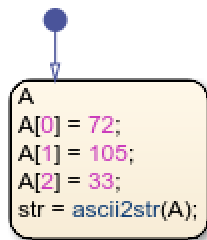
Note The operator `ascii2str` is supported only in Stateflow charts that use C as the action language.

Examples

Array of Type `uint8` to String

Return string `"Hi!"`.

```
A[0] = 72;
A[1] = 105;
A[2] = 33;
str = ascii2str(A);
```



Version History

Introduced in R2018b

See Also

`str2ascii`

Topics

“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

at

Execute chart at event broadcast or specified time

Syntax

```
at(n,E)
at(n,tick)
at(n,sec)
```

Description

`at(n,E)` returns `true` if the event `E` has occurred exactly `n` times since the associated state became active. Otherwise, the operator returns `false`.

`at(n,tick)` returns `true` if the chart has woken up exactly `n` times since the associated state became active. Otherwise, the operator returns `false`.

The implicit event `tick` is not supported when a Stateflow chart in a Simulink model has input events.

`at(n,sec)` returns `true` if exactly `n` seconds have elapsed since the associated state became active. Otherwise, the operator returns `false`.

In standalone charts in MATLAB, specify `n` with a value greater than or equal to `0.001`. The operator creates a MATLAB `timer` object that generates an implicit event to wake up the chart. MATLAB `timer` objects are limited to 1 millisecond precision. For more information, see "Events in Standalone Charts".

- The `timer` object is created when the chart finishes executing the `entry` actions of the associated state and its substates. If you specify `n` as an expression whose value changes during chart execution, the chart does not adjust the temporal delay of the `timer` object.
- The `timer` object starts running at the end of the chart step when the associated state becomes active. This step can include the execution of other parallel states in the chart.
- If the chart is processing another operation when it receives the implicit event from the `timer` object, the chart queues the event. When the current step is completed, the chart processes the event.
- If the state associated with the temporal logic operator becomes inactive before the chart processes the implicit event, the event does not wake up the chart.

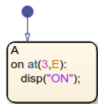
Note This syntax is supported only in standalone charts in MATLAB. For charts in Simulink models, use the `after` operator instead. For more information, see "Do Not Use at for Absolute-Time Temporal Logic in Charts in Simulink Models".

Examples

Execute State Action on Event Broadcast

Display a status message when the chart processes the third broadcast of the event E after the state became active.

```
on at(3,E):
   disp("ON");
```



Trigger Transition on Event Broadcast

Transition out of the associated state when the chart processes the fifth broadcast of the event E after the state became active.

```
at(5,E)
```

Guard Transition with Temporal Condition

Transition out of the associated state if the state has been active for exactly five broadcasts of the event E.

In charts in a Simulink model, enter:

```
[at(5,E)]
```

Conditional notation for temporal logic operators is not supported in standalone charts in MATLAB.

Trigger Transition on Chart Execution

Transition out of the associated state when the chart wakes up for the seventh time since the state became active, but only if the variable `temp` is greater than 98.6.

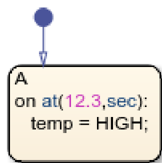
```
at(7,tick)[temp > 98.6]
```

Execute State Action at Specified Time

Set the `temp` variable to `HIGH` if the state has been active for exactly 12.3 seconds.

In standalone charts in MATLAB, enter:

```
on at(12.3,sec):
   temp = HIGH;
```



Using `every` as an absolute-time temporal logic operator is not supported in charts in Simulink models.

Tips

- You can use quotation marks to enclose the keywords `'tick'` and `'sec'`. For example, `at(5,'tick')` is equivalent to `at(5,tick)`.
- The Stateflow chart resets the counter used by the `at` operator each time the associated state reactivates.
- Standalone charts in MATLAB define absolute-time temporal logic in terms of wall-clock time, which is limited to 1 millisecond precision.

Version History

Introduced in R2014b

See Also

`after` | `before` | `every` | `timer`

Topics

"Control Chart Execution by Using Temporal Logic"
"Use Events to Execute Charts"
"Control Chart Behavior by Using Implicit Events"

before

Execute chart before event broadcast or specified time

Syntax

```
before(n,E)
before(n,tick)
before(n,time_unit)
```

Description

`before(n,E)` returns `true` if the event `E` has occurred fewer than `n` times since the associated state became active. Otherwise, the operator returns `false`.

`before(n,tick)` returns `true` if the chart has woken up fewer than `n` times since the associated state became active. Otherwise, the operator returns `false`.

The implicit event `tick` is not supported when a Stateflow chart in a Simulink model has input events.

`before(n,time_unit)` returns `true` if fewer than `n` units of time have elapsed since the associated state became active. Otherwise, the operator returns `false`.

Specify `time_unit` as seconds (`sec`), milliseconds (`msec`), or microseconds (`usec`). If you specify `n` as an expression, the chart adjusts the temporal delay as the expression changes value during the simulation.

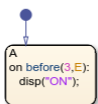
Note The temporal logic operator `before` is supported only in Stateflow charts in Simulink models.

Examples

Execute State Action on Event Broadcast

Display a status message when the chart processes the first and second broadcasts of the event `E` after the state became active.

```
on before(3,E):
   disp("ON");
```



Trigger Transition on Event Broadcast

Transition out of the associated state when the chart processes a broadcast of the event E, but only if the state has been active for fewer than five broadcasts of E.

```
before(5,E)
```

Guard Transition with Temporal Condition

Transition out of the associated state if the state has been active for fewer than five broadcasts of the event E.

```
[before(5,E)]
```

Trigger Transition on Chart Execution

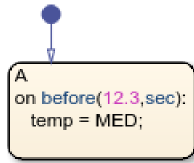
Transition out of the associated state when the chart wakes up, but only if the variable temp is greater than 98.6 and the chart has woken up fewer than seven times since the state became active.

```
before(7,tick)[temp > 98.6]
```

Execute State Action Before Specified Time

Set the temp variable to MED every time that the chart wakes up, but only if the associated state has been active for fewer 12.3 seconds.

```
on before(12.3,sec):
   temp = MED;
```



Tips

- You can use quotation marks to enclose the keywords `'tick'`, `'sec'`, `'msec'`, and `'usec'`. For example, `before(5,'tick')` is equivalent to `before(5,tick)`.
- The Stateflow chart resets the counter used by the `before` operator each time the associated state reactivates.

Version History

Introduced in R2014b

See Also

`after` | `at` | `every`

Topics

"Control Chart Execution by Using Temporal Logic"

blanks

Character array of spaces

Syntax

```
blanks(n)
```

Description

`blanks(n)` creates a `1`-by-`n` array of space characters. Use the return value in combination with the `string` operator to create a string of `n` spaces.

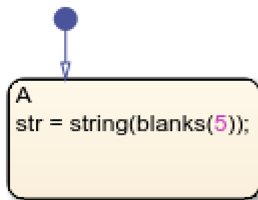
Note The operator `blanks` is not supported in Stateflow charts that use C as the action language.

Examples

Create String of Spaces

Create string that contains five spaces.

```
str = string(blanks(5));
```



Version History

Introduced in R2021b

See Also

`string`

Topics

"Manage Textual Information by Using Strings"
"Share String Data with Custom C Code"

boolean

Convert numeric value to Boolean value

Syntax

```
tf = boolean(X)
```

Description

`tf = boolean(X)` converts the numeric expression `X` into a Boolean value. If the expression evaluates to zero, `boolean` returns logical `0` (`false`). Otherwise, `boolean` returns logical `1` (`true`).

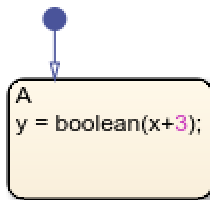
Note The operator `boolean` is supported only in Stateflow charts. In MATLAB, use `logical`.

Examples

Convert Numeric Expression to Boolean Value

Cast the expression `x+3` as a Boolean value and assign the value to the data `y`.

```
y = boolean(x+3);
```



Input Arguments

X — Numeric expression
scalar | vector | matrix | multidimensional array

Numeric expression, specified as a scalar, vector, matrix, or multidimensional array.

Example: `boolean(x+3)`

Version History

Introduced before R2006a

See Also

`logical`

Topics
"Type Cast Operations"

change, chg

Generate implicit event when data changes value

Syntax

```
change(data_name)
chg(data_name)
```

Description

`change(data_name)` generates an implicit local event when the chart sets the value of the variable `data_name`. If more than one data object has the same name, use dot notation to specify the name of the data object. For more information, see "Identify Data by Using Dot Notation".

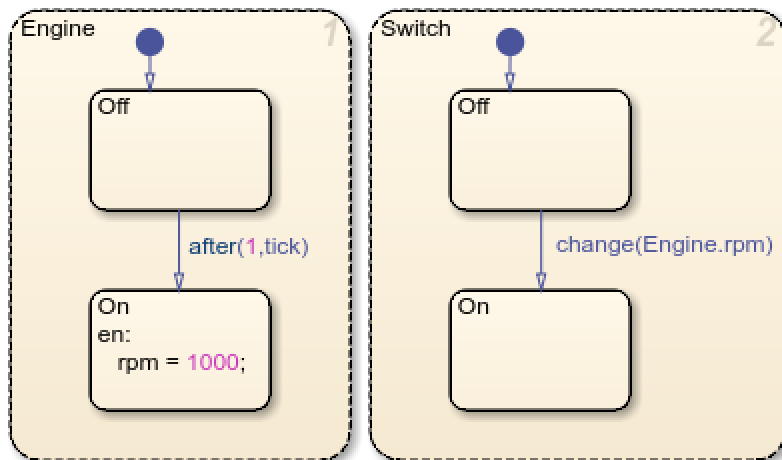
`chg(data_name)` is an alternative way to execute `change(data_name)`.

Examples

Implicit Event When Data Changes Value

Define an implicit local event when a state or transition action writes a value to the variable `Engine.rpm`.

```
change(Engine.rpm)
```



Tips

- The `change` operator is supported only in Stateflow charts in Simulink models.

Version History

Introduced before R2006a

See Also

`hasChanged` | `hasChangedFrom` | `hasChangedTo`

Topics

"Control Chart Behavior by Using Implicit Events"
"Use Events to Execute Charts"
"Detect Changes in Data and Expression Values"

contains

Determine if string contains substring

Syntax

```
tf = contains(str,substr)
tf = contains(str,substr,IgnoreCase=true)
```

Description

`tf = contains(str,substr)` returns `1` (`true`) if the string `str` contains the substring `substr`, and returns `0` (`false`) otherwise.

`tf = contains(str,substr,IgnoreCase=true)` checks if `str` contains `substr`, ignoring any differences in letter case.

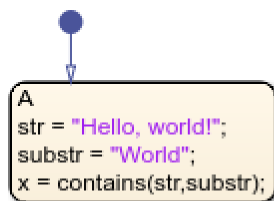
Note The `contains` operator is not supported in Stateflow charts that use C as the action language.

Examples

Determine if String Contains Substring

Return a value of `0` (`false`) because the string `"Hello, world!"` does not contain the substring `"World"`.

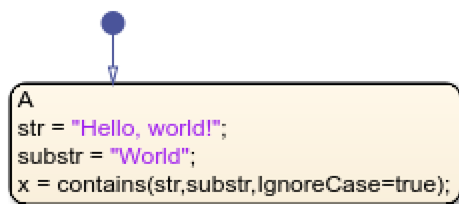
```
str = "Hello, world!";
substr = "World";
x = contains(str,substr);
```



Determine if String Contains Substring While Ignoring Case

Return a value of `1` (`true`) because the string `"Hello, world!"` contains the substring `"World"` when you ignore case.

```
str = "Hello, world!";
substr = "World";
x = contains(str,substr,IgnoreCase=true);
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`substr` — Substring
string scalar

Substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see "Access Bus Signals Through Stateflow Structures".

Version History

Introduced in R2021b

See Also

`startsWith` | `endsWith` | `strfind`

Topics
"Manage Textual Information by Using Strings"
"Share String Data with Custom C Code"

count

Number of chart executions during which condition is valid

Syntax

```
count(C)
```

Description

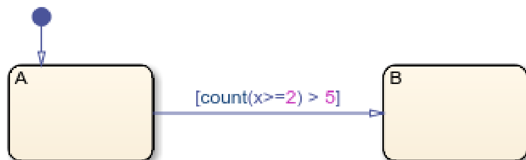
`count(C)` returns the number of times that the chart has woken up since the conditional expression `C` became `true` and the associated state became active.

Examples

Guard Transition with Temporal Condition

Transition out of the associated state when the variable `x` has been greater than or equal to 2 for longer than five chart executions.

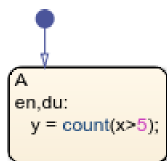
```
[count(x>=2) > 5]
```



Determine Number of Chart Executions

Store the number of chart executions since the variable `x` became greater than 5.

```
en,du:
   y = count(x>5);
```



Tips

- The Stateflow chart resets the value of the `count` operator if the conditional expression becomes `false` or if the associated state becomes inactive.

- When a chart in a Simulink model does not have input events, the value of `count` depends on the step size. Changing the solver or step size for the model affects the results produced by the `count` operator.
- To ensure that your Stateflow chart simulates without error, do not use count in these objects:
  - Continuous time charts
  - Graphical, MATLAB, or Simulink functions
  - Simulink based states
  - Transitions that can be reached from multiple states
  - Default transitions

Version History

Introduced in R2019a

See Also

`duration` | `elapsed` | `temporalCount`

Topics

"Control Chart Execution by Using Temporal Logic"

crossing

Detect rising or falling edge in data since last time step

Syntax

```
tf = crossing(expression)
```

Description

`tf = crossing(expression)` returns 1 (`true`) if:

- The previous value of `expression` was positive and its current value is zero or negative.
- The previous value of `expression` was zero and its current value is nonzero.
- The previous value of `expression` was negative and its current value is zero or positive.

Otherwise, the operator returns 0 (`false`). If `expression` changes value from positive to zero to negative or from negative to zero to positive at three consecutive time steps, the operator detects a single edge when the value of `expression` becomes zero.

The argument `expression`:

- Must be a scalar-valued expression
- Can combine chart input data, constants, nontunable parameters, continuous-time local data, and state data from Simulink based states
- Can include addition, subtraction, and multiplication of scalar variables, elements of a matrix, fields in a structure, or any valid combination of structure fields and matrix elements

Index elements of a matrix by using numbers or expressions that evaluate to a constant integer.

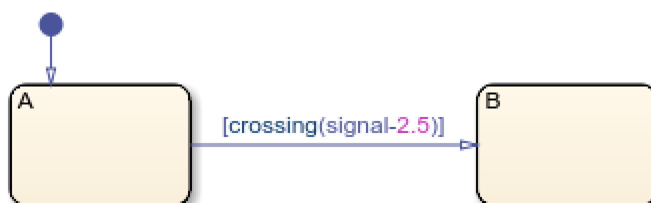
Note Edge detection is supported only in Stateflow charts in Simulink models.

Examples

Detect Signal Crossing Threshold

Transition out of state if the value of the input data `signal` crosses a threshold of 2.5.

```
[crossing(signal-2.5)]
```



The edge is detected when the value of the expression `signal-2.5` becomes zero or changes sign.

Tips

- The operator `crossing` imitates the behavior of a Trigger block with **Trigger Type** set to `either`.
- Edge detection for continuous-time local data and state data from Simulink based states is supported only in transition conditions.
- In atomic subcharts, map all input data that you use in edge detection expressions to input data or nontunable parameters in the main chart. Mapping these input data to output data, local data, or tunable parameters can result in undefined behavior.

Version History

Introduced in R2021b

See Also

`falling` | `rising` | Trigger

Topics

"Detect Changes in Data and Expression Values"
"Operations for Vectors and Matrices in Stateflow"
"Index and Assign Values to Stateflow Structures"

discard

Discard message

Syntax

```
discard(message_name)
```

Description

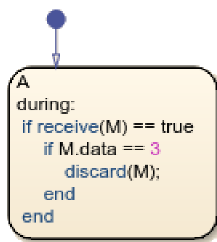
`discard(message_name)` discards a valid input or local message. After a chart discards a message, it can remove another message from the queue in the same time step. A chart cannot access the data of a discarded message.

Examples

Discard Message in State Action

Check the queue for message `M`. If a message is present, remove it from the queue. If the message has a data value equal to 3, discard the message.

```
during:
   if receive(M) == true
      if M.data == 3
         discard(M);
      end
   end
```



Version History

Introduced in R2015b

See Also

`receive`

Topics

"Control Message Activity in Stateflow Charts"

duration

Time during which condition is valid

Syntax

```
time = duration(condition)
time = duration(condition,time_unit)
```

Description

`time = duration(condition)` returns the length of time, in seconds, that `condition` stays `true` and the associated state became active.

`time = duration(condition,time_unit)` returns the length of time in the unit specified by `time_unit`.

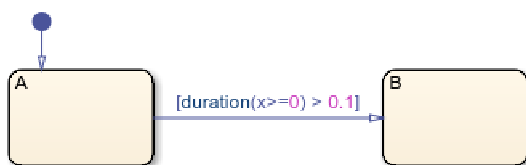
Note The temporal logic operator `duration` is not supported in standalone charts in MATLAB.

Examples

Guard Transition with Temporal Condition

Transition out of the state when the variable `x` has been greater than or equal to 0 for longer than 0.1 seconds.

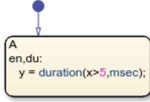
```
[duration(x>=0) > 0.1]
```



Determine Elapsed Time

Store the number of milliseconds since the variable `x` became greater than 5 and the state became active.

```
en,du:
   y = duration(x>5,msec);
```



Input Arguments

condition — Logical condition

true | false

Logical condition, specified as `true` or `false`. You can specify the value of `condition` by using an expression that evaluates to `true` or `false`. The operator evaluates `condition` at each time step.

`condition` does not support expressions that depend on local or output data.

Example: `duration(u)`

Example: `duration(u>=0)`

time_unit — Units of time

sec (default) | msec | usec

Units of time that `duration` returns, specified in seconds (`sec`), milliseconds (`msec`), or microseconds (`usec`).

Tips

- You can use quotation marks to enclose the keywords `'sec'`, `'msec'`, and `'usec'`. For example, `duration(x > 0,'sec')` is equivalent to `duration(x > 0,sec)`.
- The Stateflow chart resets the value of the `duration` operator if the conditional expression `C` becomes `false` or if the associated state becomes inactive.
- The `duration` operator does not support conditions that depend on local or output structures. For more information, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2017a

See Also

`count` | `elapsed` | `temporalCount`

Topics

“Control Chart Execution by Using Temporal Logic”
“Control Oscillations by Using the duration Operator”
“Reduce Transient Signals by Using Debouncing Logic”
“Implement an Automatic Transmission Gear System by Using the duration Operator”

elapsed, et

Time since state became active

Syntax

```
elapsed(sec)
et
```

Description

`elapsed(sec)` returns the length of time that has elapsed since the associated state became active.

`et` is an alternative way to execute `elapsed(sec)`.

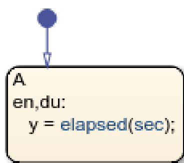
Note The expressions `elapsed(sec)` and `et` are equivalent to `temporalCount(sec)`.

Examples

Determine Time of State Activity

Store the number of seconds since the state became active.

```
en,du:
   y = elapsed(sec);
```



Display Elapsed Time

When the chart processes a broadcast of the event `E`, transition out of the associated state and display the elapsed time since the state became active.

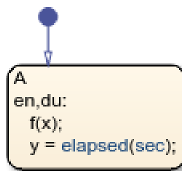
```
E{disp(et);}
```

Tips

- In state and transition actions, you can use quotation marks to enclose the keyword `'sec'`. For example, `elapsed('sec')` is equivalent to `elapsed(sec)`.
- The Stateflow chart resets the counter used by the `elapsed` operator each time the associated state reactivates.
- The timing for absolute-time temporal logic operators depends on the type of Stateflow chart:

  - Charts in a Simulink model define temporal logic in terms of simulation time.
  - Standalone charts in MATLAB define temporal logic in terms of wall-clock time.

  The difference in timing can affect the behavior of a chart. For example, suppose that this chart is executing the `entry` action of state `A`.



  - In a Simulink model, the function call to `f` executes in a single time step and does not contribute to the simulation time. After calling the function `f`, the chart assigns a value of zero to `y`.
  - In a standalone chart, the function call to `f` can take several seconds of wall-clock time to complete. After calling the function `f`, the chart assigns the nonzero time that has elapsed since state `A` became active to `y`.

Version History

Introduced in R2017a

See Also

`count` | `duration` | `temporalCount`

Topics

“Control Chart Execution by Using Temporal Logic”

endsWith

Determine if string ends with substring

Syntax

```
tf = endsWith(str,substr)
tf = endsWith(str,substr,IgnoreCase=true)
```

Description

`tf = endsWith(str,substr)` returns `1` (`true`) if the string `str` ends with the substring `substr`, and returns `0` (`false`) otherwise.

`tf = endsWith(str,substr,IgnoreCase=true)` checks if `str` ends with `substr`, ignoring any differences in letter case.

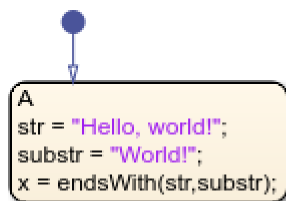
Note The `endsWith` operator is not supported in Stateflow charts that use C as the action language.

Examples

Determine if String Ends with Substring

Return a value of `0` (`false`) because the string `"Hello, world!"` does not end with the substring `"World!"`.

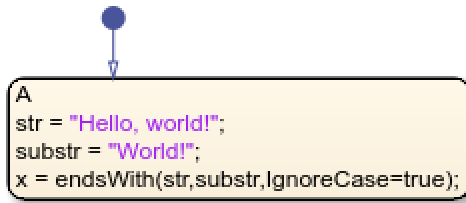
```
str = "Hello, world!";
substr = "World!";
x = endsWith(str,substr);
```



Determine if String Ends with Substring While Ignoring Case

Return a value of `1` (`true`) because the string `"Hello, world!"` ends with the substring `"World!"` when you ignore case.

```
str = "Hello, world!";
substr = "World!";
x = endsWith(str,substr,IgnoreCase=true);
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`substr` — Substring
string scalar

Substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`contains` | `startsWith` | `strfind`

Topics

“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

enter, en

Generate implicit event when state becomes active

Syntax

```
enter(state_name)
en(state_name)
```

Description

`enter(state_name)` generates an implicit local event when the chart execution enters the state `state_name`. If more than one state has the same name, use dot notation to specify the name of the state. For more information, see “State Name”.

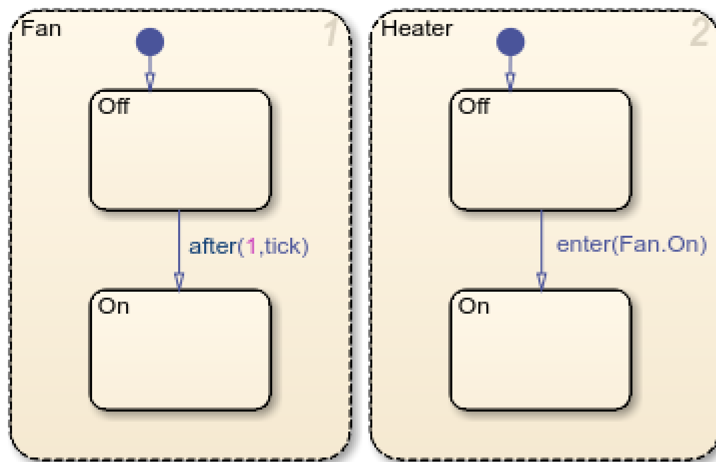
`en(state_name)` is an alternative way to execute `enter(state_name)`.

Examples

Implicit Event When State Becomes Active

Define an implicit local event when the chart execution enters the state `Fan.On`.

```
enter(Fan.On)
```



Tips

The `enter` operator is supported only in Stateflow charts in Simulink models.

Version History

Introduced before R2006a

See Also

`exit` | `in`

Topics

"Control Chart Behavior by Using Implicit Events"
"Use Events to Execute Charts"
"Check State Activity by Using the in Operator"

erase

Delete substrings within strings

Syntax

`newStr = erase(str,substr)`

Description

`newStr = erase(str,substr)` deletes instances of the substring `substr` that occur in the string `str`.

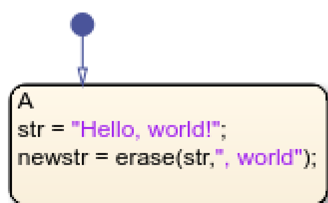
Note The `erase` operator is not supported in Stateflow charts that use C as the action language.

Examples

Erase Substring from a String

Delete a substring to form the string `"Hello!"`

```
str = "Hello, world!";
newstr = erase(str,", world");
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`substr` — Substring
string scalar

Substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`eraseBetween`

Topics

“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

eraseBetween

Delete substring between start and end points

Syntax

```
newStr = eraseBetween(str,startStr,endStr)
newStr = eraseBetween(str,startPos,endPos)
newStr = eraseBetween( ___ ,Boundaries=bounds)
```

Description

`newStr = eraseBetween(str,startStr,endStr)` deletes the substring in `str` between the substrings `startStr` and `endStr`. `eraseBetween` does not delete `startStr` and `endStr` themselves.

`newStr = eraseBetween(str,startPos,endPos)` deletes the substring in `str` between the character positions `startPos` and `endPos`, including the characters at those positions.

`newStr = eraseBetween( ___ ,Boundaries=bounds)` includes or excludes the boundaries specified in the previous syntaxes from the substrings that the operator deletes. Specify `bounds` as `"inclusive"` or `"exclusive"`.

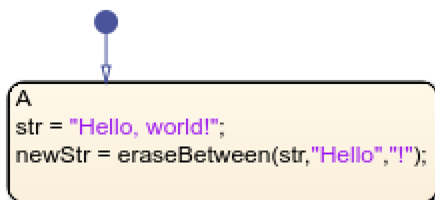
Note The `eraseBetween` operator is not supported in Stateflow charts that use C as the action language.

Examples

Erase Text Between Substrings

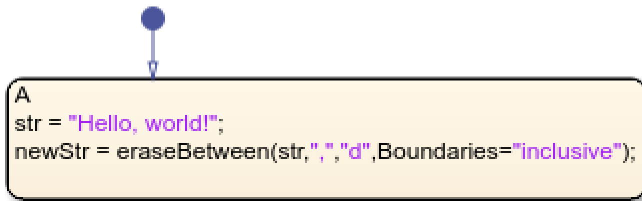
Delete a substring to form the string `"Hello!"`. By default, `eraseBetween` does not delete the boundary substrings.

```
str = "Hello, world!";
newStr = eraseBetween(str,"Hello","!");
```



Alternatively, use the option `Boundaries="inclusive"` to delete the boundary substrings.

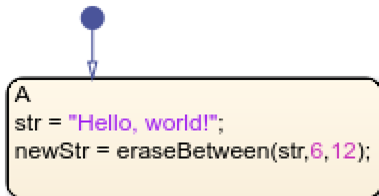
```
str = "Hello, world!";
newStr = eraseBetween(str,",","d",new,Boundaries="inclusive");
```



Erase Text Between Start and End Positions

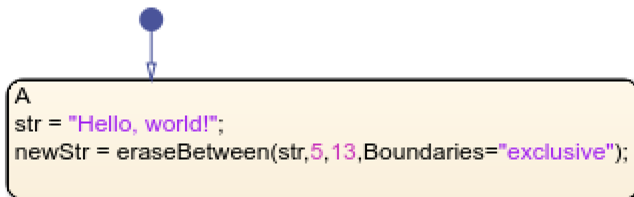
Delete a substring to form the string `"Hello!"`. By default, `eraseBetween` deletes the boundary characters.

```
str = "Hello, world!";
newStr = eraseBetween(str,6,12);
```



Alternatively, use the `Boundaries="exclusive"` option to prevent deletion of the boundary characters.

```
str = "Hello, world!";
newStr = eraseBetween(str,5,13,new,Boundaries="exclusive");
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`startStr` — Starting substring
string scalar

Staring substring, specified as a string scalar. Enclose literal strings with double quotes.

`endStr` — Ending substring
string scalar

Ending substring, specified as a string scalar. Enclose literal strings with double quotes.

`startPos` — Starting character position
positive integer

Starting character position, specified as a positive integer.

`endPos` — Ending character position
positive integer

Ending character position, specified as a positive integer.

`bounds` — Boundary type
`"inclusive"` | `"exclusive"`

Boundary type, specified as either `"inclusive"` or `"exclusive"`. When you set `bounds` to `"inclusive"`, `replaceBetween` erases the text between and including the boundaries. When you set `bounds` to `"exclusive"`, `replaceBetween` erases the text only between the boundaries.

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see "Access Bus Signals Through Stateflow Structures".

Version History

Introduced in R2021b

See Also

`erase` | `replaceBetween`

Topics
"Manage Textual Information by Using Strings"
"Share String Data with Custom C Code"

every

Execute chart at regular intervals

Syntax

```
every(n,E)
every(n,tick)
every(n,sec)
```

Description

`every(n,E)` returns `true` at every `n`th occurrence of the event `E` since the associated state became active. Otherwise, the operator returns `false`.

`every(n,tick)` returns `true` at every `n`th time that the chart wakes up since the associated state became active. Otherwise, the operator returns `false`.

The implicit event `tick` is not supported when a Stateflow chart in a Simulink model has input events.

`every(n,sec)` returns `true` every `n` seconds since the associated state became active. Otherwise, the operator returns `false`.

In standalone charts in MATLAB, specify `n` with a value greater than or equal to `0.001`. The operator creates a MATLAB `timer` object that generates an implicit event to wake up the chart. MATLAB `timer` objects are limited to 1 millisecond precision. For more information, see “Events in Standalone Charts”.

- The `timer` object is created when the chart finishes executing the `entry` actions of the associated state and its substates. For subsequent iterations, the `timer` object is reset when the chart finishes executing the `during` actions of the associated state and its substates. If you specify `n` as an expression whose value changes during chart execution, the chart adjusts the temporal delay only when the `timer` object is reset.
- The `timer` object starts running at the end of the chart step when the associated state becomes active. This step can include the execution of other parallel states in the chart.
- If the chart is processing another operation when it receives the implicit event from the `timer` object, the chart queues the event. When the current step is completed, the chart processes the event and resets the timer object for the next iteration.
- If the state associated with the temporal logic operator becomes inactive before the chart processes the implicit event, the event does not wake up the chart.

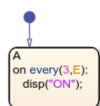
Note This syntax is supported only in standalone charts in MATLAB. In charts in Simulink models, use an outer self-loop transition with the `after` operator instead. For more information, see “Do Not Use every for Absolute-Time Temporal Logic in Charts in Simulink Models”.

Examples

Execute State Action on Event Broadcast

Display a status message when the chart processes every third broadcast of the event `E` after the state became active.

```
on every(3,E):
   disp("ON");
```



Trigger Transition on Event Broadcast

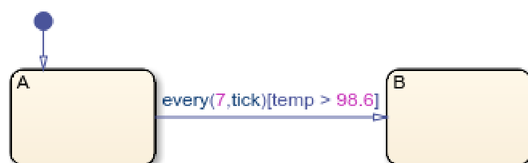
Transition out of the associated state when the chart processes every fifth broadcast of the event `E` after the state became active.

```
every(5,E)
```

Trigger Transition on Chart Execution

Transition out of the associated state every seventh `tick` event since the state became active, but only if the variable `temp` is greater than 98.6.

```
every(7,tick)[temp > 98.6]
```

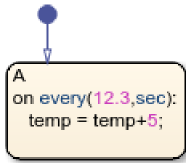


Execute State Action at Specified Time

Increment the `temp` variable by 5 every 12.3 seconds that the state is active.

In standalone charts in MATLAB, enter:

```
on every(12.3,sec):
   temp = temp+5;
```



Using `every` as an absolute-time temporal logic operator is not supported in charts in Simulink models.

Tips

- You can use quotation marks to enclose the keywords `'tick'` and `'sec'`. For example, `every(5,'tick')` is equivalent to `every(5,tick)`.
- The Stateflow chart resets the counter used by the `every` operator each time the associated state reactivates.
- Standalone charts in MATLAB define absolute-time temporal logic in terms of wall-clock time, which is limited to 1 millisecond precision.

Version History

Introduced in R2014b

See Also

`after` | `at` | `before` | `timer`

Topics

"Control Chart Execution by Using Temporal Logic"
"Use Events to Execute Charts"
"Control Chart Behavior by Using Implicit Events"

exit, ex

Generate implicit event when state becomes inactive

Syntax

```
exit(state_name)
ex(state_name)
```

Description

`exit(state_name)` generates an implicit local event when the chart execution exits the state `state_name`. If more than one state has the same name, use dot notation to specify the name of the state. For more information, see “State Name”.

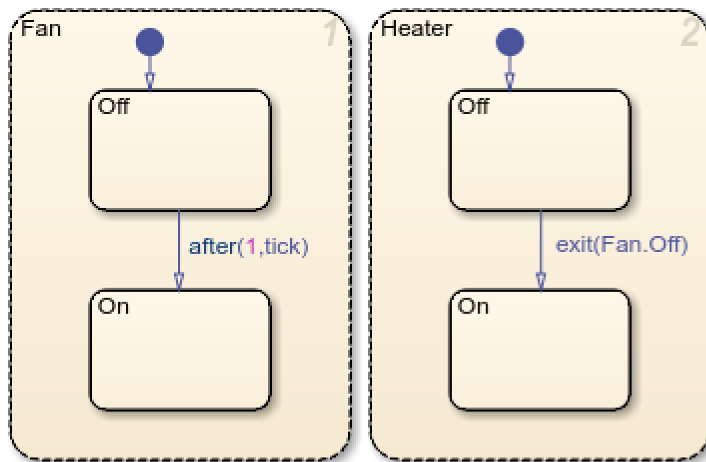
`ex(state_name)` is an alternative way to execute `exit(state_name)`.

Examples

Implicit Event When State Becomes Inactive

Define an implicit local event when the chart execution exits the state `Fan.Off`.

```
exit(Fan.Off)
```



Tips

The `exit` operator is supported only in Stateflow charts in Simulink models.

Version History

Introduced before R2006a

See Also

`enter` | `in`

Topics

“Control Chart Behavior by Using Implicit Events”
“Use Events to Execute Charts”
“Check State Activity by Using the in Operator”

extractAfter

Extract substring after position

Syntax

```
newStr = extractAfter(str,subStr)
newStr = extractAfter(str,pos)
```

Description

`newStr = extractAfter(str,subStr)` returns the substring of `str` that begins after the last occurrence of the substring `subStr`.

`newStr = extractAfter(str,pos)` returns the substring of `str` that begins after the character position `pos`.

Note The `extractAfter` operator is not supported in Stateflow charts that use C as the action language. For similar functionality, use `substr`.

Examples

Extract Text After Substring

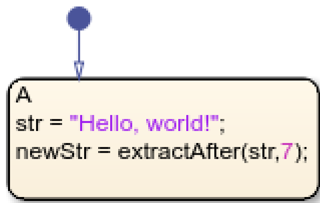
Extract substring `"world!"` from a longer string.

```
str = "Hello, world!";
newStr = extractAfter(str,"Hello, ");
```

Extract Text After Position

Extract substring `"world!"` from a longer string.

```
str = "Hello, world!";
newStr = extractAfter(str,7);
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`substr` — Substring
string scalar

Substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

pos — Character position
positive integer

Character position, specified as a positive integer.

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see "Access Bus Signals Through Stateflow Structures".

Version History

Introduced in R2021b

See Also

`extractBefore` | `substr` | `insertAfter`

Topics

"Manage Textual Information by Using Strings"
"Share String Data with Custom C Code"

extractBefore

Extract substring before position

Syntax

```
newStr = extractBefore(str,subStr)
newStr = extractBefore(str,pos)
```

Description

`newStr = extractBefore(str,subStr)` returns the substring of `str` that ends before the first occurrence of the substring `subStr`.

`newStr = extractBefore(str,pos)` returns the substring of `str` that ends before the character position `pos`.

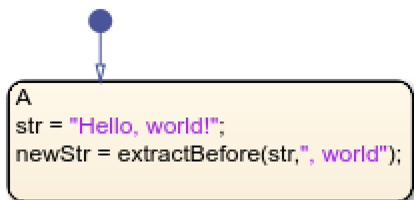
Note The `extractBefore` operator is not supported in Stateflow charts that use C as the action language. For similar functionality, use `substr`.

Examples

Select Text Before Substring

Extract substring `"Hello"` from a longer string.

```
str = "Hello, world!";
newStr = extractBefore(str,", world");
```



Select Text Before a Position

Extract substring `"Hello"` from a longer string.

```
str = "Hello, world!";
newstr = extractBefore(str,6);
```

Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`substr` — Substring
string scalar

Substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

pos — Character position
positive integer

Character position, specified as a positive integer.

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`extractAfter` | `substr` | `insertBefore`

Topics
“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

falling

Detect falling edge in data since last time step

Syntax

```
tf = falling(expression)
```

Description

`tf = falling(expression)` returns 1 (`true`) if:

- The previous value of `expression` was positive and its current value is zero or negative.
- The previous value of `expression` was zero and its current value is negative.

Otherwise, the operator returns 0 (`false`). If `expression` changes value from positive to zero to negative at three consecutive time steps, the operator detects a single edge when the value of `expression` becomes zero.

The argument `expression`:

- Must be a scalar-valued expression
- Can combine chart input data, constants, nontunable parameters, continuous-time local data, and state data from Simulink based states
- Can include addition, subtraction, and multiplication of scalar variables, elements of a matrix, fields in a structure, or any valid combination of structure fields and matrix elements

Index elements of a matrix by using numbers or expressions that evaluate to a constant integer.

Note Edge detection is supported only in Stateflow charts in Simulink models.

Examples

Detect Falling Edge

Transition out of state if the value of the input data `signal` falls below a threshold of 2.5.

```
[falling(signal-2.5)]
```

The falling edge is detected when the value of the expression `signal-2.5` becomes zero or negative.

Tips

- The operator `falling` imitates the behavior of a Trigger block with **Trigger Type** set to `falling`.
- Edge detection for continuous-time local data and state data from Simulink based states is supported only in transition conditions.
- In atomic subcharts, map all input data that you use in edge detection expressions to input data or nontunable parameters in the main chart. Mapping these input data to output data, local data, or tunable parameters can result in undefined behavior.

Version History

Introduced in R2021b

See Also

`crossing` | `rising` | Trigger

Topics

"Detect Changes in Data and Expression Values"
"Operations for Vectors and Matrices in Stateflow"
"Index and Assign Values to Stateflow Structures"

forward

Forward message

Syntax

```
forward(message_in_name,message_out_name)
```

Description

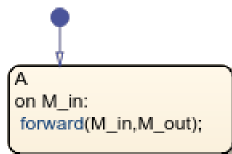
`forward(message_in_name,message_out_name)` forwards a valid input or local message to a local queue or an output port. After a chart forwards a message, it can remove another message from the queue in the same time step.

Examples

Forward an Input Message

Check the input queue for message `M_in`. If a message is present, remove the message from the queue and forward it to the output port `M_out`.

```
on M_in:
   forward(M_in,M_out);
```



Forward a Local Message

Check the local queue for message `M_local`. If a message is present, transition from state `A` to state `B`. Remove the message from the `M_local` message queue and forward it to the output port `M_out`.

```
M_local{forward(M_local,M_out)}
```

Version History

Introduced in R2015b

See Also

`receive`

Topics

"Control Message Activity in Stateflow Charts"

hasChanged

Detect change in data since last time step

Syntax

```
tf = hasChanged(data)
```

Description

`tf = hasChanged(data)` returns `1` (`true`) if the value of `data` at the beginning of the current time step is different from the value of `data` at the beginning of the previous time step. Otherwise, the operator returns `0` (`false`).

Examples

Detect Change in Matrix

Transition out of state if any element of the matrix `M` has changed value since the last time step or input event.

```
[hasChanged(M)]
```

Detect Change in Matrix Element

Transition out of state if the element in row 1 and column 3 of the matrix `M` has changed value since the last time step or input event.

In charts that use MATLAB as the action language, use:

```
[hasChanged(M(1,3))]
```

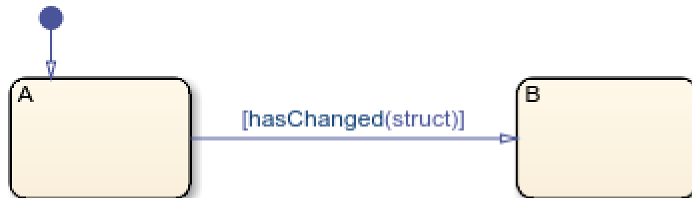
In charts that use C as the action language, use:

```
[hasChanged(M[0][2])]
```

Detect Change in Structure

Transition out of state if any field of the structure `struct` has changed value since the last time step or input event.

```
[hasChanged(struct)]
```



Detect Change in Structure Field

Transition out of state if the field `struct.field` has changed value since the last time step or input event.

```
[hasChanged(struct.field)]
```

Input Arguments

data — Data
scalar | matrix | structure | ...

Stateflow data, specified as a:

- Scalar
- Matrix or an element of a matrix
- Structure or a field in a structure
- Valid combination of structure fields or matrix elements

If `data` is a matrix, the operator returns `true` when it detects a change in one of the elements of `data`. You can also index elements of a matrix by using numbers or expressions that evaluate to an integer. See "Operations for Vectors and Matrices in Stateflow".

If `data` is a structure, the operator returns `true` when it detects a change in one of the fields of `data`. You can also index fields in a structure by using dot notation. See "Index and Assign Values to Stateflow Structures".

The argument `data` cannot be a nontrivial expression or a custom code variable.

Standalone charts in MATLAB do not support change detection on an element of a matrix or a field in a structure.

Tips

- If multiple input events occur in the same time step, the `hasChanged` operator can detect changes in data value between input events.
- If the chart writes to the data object but does not change the data value, the `hasChanged` operator returns `false`.
- The type of Stateflow chart determines the scope of the data supported by the change detection operators:
  - Standalone Stateflow charts in MATLAB: `Local` only
  - In Simulink models, charts that use MATLAB as the action language: `Input` only
  - In Simulink models, charts that use C as the action language: `Input`, `Output`, `Local`, or `Data Store Memory`
- In a standalone chart in MATLAB, a change detection operator can detect changes in data specified in a call to the `step` function because these changes occur before the start of the current time step. For example, if `x` is equal to zero, the expression `hasChanged(x)` returns `true` when you execute the chart `ch` with the command:

  ```
  step(ch,x=1);
  ```

  In contrast, a change detection operator cannot detect changes in data caused by assignments in state or transition actions in the same time step. Instead, the operator detects the change in value at the start of the next time step.
- In a chart in a Simulink model, if you enable the chart option **Initialize Outputs Every Time Chart Wakes Up**, using an output as the argument of the `hasChanged` operator always returns `false`. For more information, see "Initialize outputs every time chart wakes up".
- When row-major array layout is enabled in charts that use `hasChanged`, code generation produces an error. Before generating code in charts that use `hasChanged`, enable column-major array layout. See "Select Array Layout for Matrices in Generated Code".

Version History

Introduced in R2007a

See Also

`hasChangedFrom` | `hasChangedTo`

Topics

"Detect Changes in Data and Expression Values"
"Operations for Vectors and Matrices in Stateflow"
"Index and Assign Values to Stateflow Structures"
"Assign Values to All Elements of a Matrix"

hasChangedFrom

Detect change in data from specified value

Syntax

```
tf = hasChangedFrom(data,value)
```

Description

`tf = hasChangedFrom(data,value)` returns `1` (`true`) if the value of `data` is equal to `value` at the beginning of the previous time step and is a different value at the beginning of the current time step. Otherwise, the operator returns `0` (`false`).

Examples

Detect Change in Matrix

Transition out of state if the previous value of the matrix `M` was equal to `matrixValue` and any element of `M` has changed value since the last time step or input event.

```
[hasChangedFrom(M,matrixValue)]
```

Detect Change in Matrix Element

Transition out of state if the element in row 1 and column 3 of the matrix `M` has changed from the value 7 since the last time step or input event.

In charts that use MATLAB as the action language, use:

```
[hasChangedFrom(M(1,3),7)]
```

In charts that use C as the action language, use:

```
[hasChangedFrom(M[0][2],7)]
```

Detect Change in Structure

Transition out of state if the previous value of the structure `struct` was equal to `structValue` and any field of `struct` has changed value since the last time step or input event.

```
[hasChangedFrom(struct,structValue)]
```

Detect Change in Structure Field

Transition out of state if the field `struct.field` has changed from the value `5` since the last time step or input event.

```
[hasChangedFrom(struct.field,5)]
```

Input Arguments

data — Data
scalar | matrix | structure | ...

Stateflow data, specified as a:

- Scalar
- Matrix or an element of a matrix
- Structure or a field in a structure
- Valid combination of structure fields or matrix elements

If `data` is a matrix, the operator returns `true` when it detects a change in one of the elements of `data`. You can also index elements of a matrix by using numbers or expressions that evaluate to an integer. See “Operations for Vectors and Matrices in Stateflow”.

If `data` is a structure, the operator returns `true` when it detects a change in one of the fields of `data`. You can also index fields in a structure by using dot notation. See “Index and Assign Values to Stateflow Structures”.

The argument `data` cannot be a nontrivial expression or a custom code variable.

Standalone charts in MATLAB do not support change detection on an element of a matrix or a field in a structure.

`value` — Value of data at previous time step
scalar | matrix | structure

Value of the data at previous time step, specified as the same data type of `data`. `value` must be an expression that resolves to a value that is comparable with `data`:

- If `data` is a scalar, then `value` must resolve to a scalar.
- If `data` is a matrix, then `value` must resolve to a matrix with the same dimensions as `data`.

  Alternatively, in a chart that uses C as the action language, `value` can resolve to a scalar value. The chart uses scalar expansion to compare `data` to a matrix whose elements are all equal to the value specified by `value`. See “Assign Values to All Elements of a Matrix”.
- If `data` is a structure, then `value` must resolve to a structure whose field specification matches `data` exactly.

Tips

- If multiple input events occur in the same time step, the `hasChangedFrom` operator can detect changes in data value between input events.
- If the chart writes to the data object but does not change the data value, the `hasChangedFrom` operator returns `false`.
- The type of Stateflow chart determines the scope of the data supported by the change detection operators:
  - Standalone Stateflow charts in MATLAB: `Local` only
  - In Simulink models, charts that use MATLAB as the action language: `Input` only
  - In Simulink models, charts that use C as the action language: `Input`, `Output`, `Local`, or `Data Store Memory`
- In a standalone chart in MATLAB, a change detection operator can detect changes in data specified in a call to the `step` function because these changes occur before the start of the current time step. For example, if `x` is equal to zero, the expression `hasChangedFrom(x,0)` returns `true` when you execute the chart `ch` with the command:

```
step(ch,x=1);
```

In contrast, a change detection operator cannot detect changes in data caused by assignments in state or transition actions in the same time step. Instead, the operator detects the change in value at the start of the next time step.

- In a chart in a Simulink model, if you enable the chart option **Initialize Outputs Every Time Chart Wakes Up**, using an output as the argument of the `hasChanged` operator always returns `false`. For more information, see “Initialize outputs every time chart wakes up”.
- When row-major array layout is enabled in charts that use `hasChangedFrom`, code generation produces an error. Before generating code in charts that use `hasChangedFrom`, enable column-major array layout. See “Select Array Layout for Matrices in Generated Code”.

Version History

Introduced in R2007a

See Also

`hasChanged` | `hasChangedTo`

Topics

“Detect Changes in Data and Expression Values”
“Operations for Vectors and Matrices in Stateflow”
“Index and Assign Values to Stateflow Structures”
“Assign Values to All Elements of a Matrix”

hasChangedTo

Detect change in data to specified value

Syntax

`tf = hasChangedTo(data,value)`

Description

`tf = hasChangedTo(data,value)` returns `1` (`true`) if the value of `data` is not equal to `value` at the beginning of the previous time step and is equal to `value` at the beginning of the current time step. Otherwise, the operator returns `0` (`false`).

Examples

Detect Change in Matrix

Transition out of state if any element of `M` has changed value since the last time step or input event and the current value of the matrix `M` is equal to `matrixValue`.

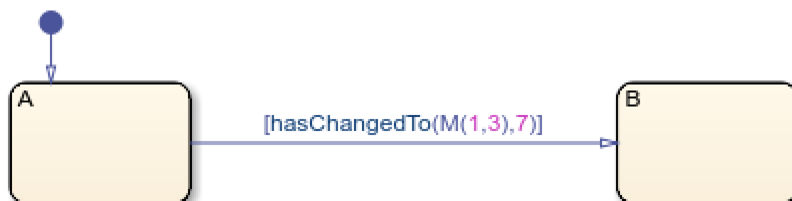
```
[hasChangedTo(M,matrixValue)]
```

Detect Change in Matrix Element

Transition out of state if the element in row 1 and column 3 of the matrix `M` has changed to the value 7 since the last time step or input event.

In charts that use MATLAB as the action language, use:

```
[hasChangedTo(M(1,3),7)]
```



In charts that use C as the action language, use:

```
[hasChangedTo(M[0][2],7)]
```

Detect Change in Structure

Transition out of state if any field of the structure `struct` has changed value since the last time step or input event and the current value of `struct` is equal to `structValue`.

```
[hasChangedTo(struct,structValue)]
```

Detect Change in Structure Field

Transition out of state if the field `struct.field` has changed to the value 5 since the last time step or input event.

```
[hasChangedTo(struct.field,5)]
```

Input Arguments

data — Data
scalar | matrix | structure | ...

Stateflow data, specified as a:

- Scalar
- Matrix or an element of a matrix
- Structure or a field in a structure
- Valid combination of structure fields or matrix elements

If `data` is a matrix, the operator returns `true` when it detects a change in one of the elements of `data`. You can also index elements of a matrix by using numbers or expressions that evaluate to an integer. See “Operations for Vectors and Matrices in Stateflow”.

If `data` is a structure, the operator returns `true` when it detects a change in one of the fields of `data`. You can also index fields in a structure by using dot notation. See “Index and Assign Values to Stateflow Structures”.

The argument `data` cannot be a nontrivial expression or a custom code variable.

Standalone charts in MATLAB do not support change detection on an element of a matrix or a field in a structure.

`value` — Value of data at current time step
scalar | matrix | structure

Value of the data at the current time step, specified as the same data type of `data`. `value` must be an expression that resolves to a value that is comparable with `data`:

- If `data` is a scalar, then `value` must resolve to a scalar.
- If `data` is a matrix, then `value` must resolve to a matrix with the same dimensions as `data`.

  Alternatively, in a chart that uses C as the action language, `value` can resolve to a scalar value. The chart uses scalar expansion to compare `data` to a matrix whose elements are all equal to the value specified by `value`. See “Assign Values to All Elements of a Matrix”.
- If `data` is a structure, then `value` must resolve to a structure whose field specification matches `data` exactly.

Tips

- If multiple input events occur in the same time step, the `hasChangedTo` operator can detect changes in data value between input events.
- If the chart writes to the data object but does not change the data value, the `hasChangedTo` operator returns `false`.
- The type of Stateflow chart determines the scope of the data supported by the change detection operators:
  - Standalone Stateflow charts in MATLAB: `Local` only
  - In Simulink models, charts that use MATLAB as the action language: `Input` only
  - In Simulink models, charts that use C as the action language: `Input`, `Output`, `Local`, or `Data Store Memory`
- In a standalone chart in MATLAB, a change detection operator can detect changes in data specified in a call to the `step` function because these changes occur before the start of the current time step. For example, if `x` is equal to zero, the expression `hasChangedTo(x,1)` returns `true` when you execute the chart `ch` with the command:

```
step(ch,x=1);
```

In contrast, a change detection operator cannot detect changes in data caused by assignments in state or transition actions in the same time step. Instead, the operator detects the change in value at the start of the next time step.

- In a chart in a Simulink model, if you enable the chart option **Initialize Outputs Every Time Chart Wakes Up**, using an output as the argument of the `hasChanged` operator always returns `false`. For more information, see “Initialize outputs every time chart wakes up”.
- When row-major array layout is enabled in charts that use `hasChangedTo`, code generation produces an error. Before generating code in charts that use `hasChangedTo`, enable column-major array layout. See “Select Array Layout for Matrices in Generated Code”.

Version History

Introduced in R2007a

See Also

`hasChanged` | `hasChangedFrom`

Topics

“Detect Changes in Data and Expression Values”
“Operations for Vectors and Matrices in Stateflow”
“Index and Assign Values to Stateflow Structures”
“Assign Values to All Elements of a Matrix”

in

Check state activity

Syntax

```
in(state_name)
```

Description

`in(state_name)` returns 1 (`true`) if the state `state_name` is active. Otherwise, the operator returns 0 (`false`).

Examples

Synchronize Substate Activity Between Parallel States

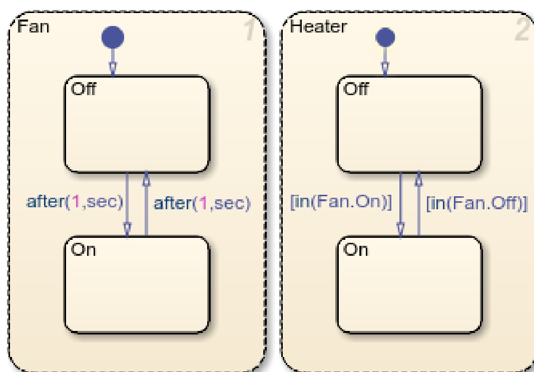
Check the substate activity in state `Fan` to keep the substates of state `Heater` synchronized.

When `Fan.On` becomes active, transition from `Heater.Off` to `Heater.On`.

```
[in(Fan.On)]
```

When `Fan.Off` becomes active, transition from `Heater.On` to `Heater.Off`.

```
[in(Fan.Off)]
```

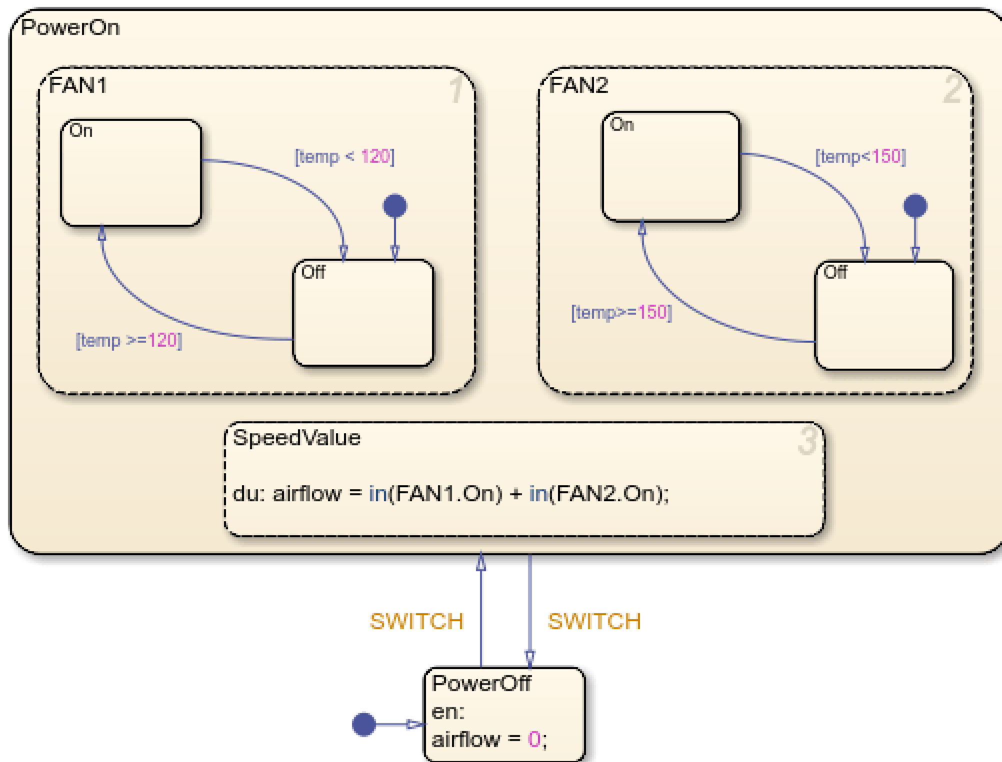


A change of active substate in `Fan` causes a corresponding change of active substate in `Heater`.

Find Number of Active Subcomponents

Set the value of `airflow` to the number of fans that are turned on.

```
airflow = in(FAN1.On) + in(FAN2.On);
```



Tips

To determine the state activity, a Stateflow chart performs a localized search of the state hierarchy. The chart does not perform an exhaustive search for all states and does not stop after finding the first match. To improve the chances of finding a unique search result:

- Use dot notation to qualify the name of the state.
- Give states unique names.
- Use states and boxes as enclosures to limit the scope of the path resolution search.

Additionally, a chart cannot use the `in` condition to trigger actions based on the activity of states in other charts.

For more information, see “Resolution of State Activity”.

Version History

Introduced before R2006a

See Also

`enter` | `exit`

Topics

“Check State Activity by Using the in Operator”

insertAfter

Insert string after substring

Syntax

```
newStr = insertAfter(str,subStr,new)
newStr = insertAfter(str,pos,new)
```

Description

`newStr = insertAfter(str,subStr,new)` inserts the string `new` into the string `str` after the substring `subStr`. `insertAfter` inserts `new` after every occurrence of `subStr`.

`newStr = insertAfter(str,pos,new)` inserts `new` into `str` after the character position `pos`.

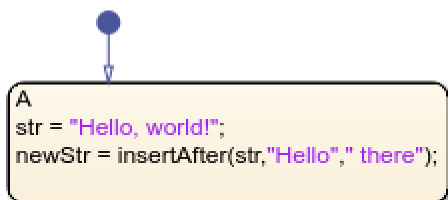
Note The `insertAfter` operator is not supported in Stateflow charts that use C as the action language.

Examples

Insert Text After Substring

Insert text to form the string `"Hello there, world!"`.

```
str = "Hello, world!";
newStr = insertAfter(str,"Hello"," there");
```



Insert Text After Character Position

Insert text to form the string `"Hello there, world!"`.

```
str = "Hello, world!";
newStr = insertAfter(str,5," there");
```

Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`substr` — Substring
string scalar

Substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`pos` — Character position
positive integer

Character position, specified as a positive integer.

`new` — New substring
string scalar

New substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`insertBefore` | `extractAfter`

Topics

“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

insertBefore

Insert string before substring

Syntax

```
newStr = insertBefore(str,subStr,new)
newStr = insertBefore(str,pos,new)
```

Description

`newStr = insertBefore(str,subStr,new)` inserts the string `new` into the string `str` before the substring `subStr`. `insertAfter` inserts `new` before every occurrence of `subStr`.

`newStr = insertBefore(str,pos,new)` inserts `new` into `str` before the character position `pos`.

Note The `insertBefore` operator is not supported in Stateflow charts that use C as the action language.

Examples

Insert Text Before Substring

Insert text to form the string `"Hello there, world!"`.

```
str = "Hello, world!";
newStr = insertBefore(str,",", " there");
```

Insert Text Before Position

Insert text to form the string `"Hello there, world!"`.

```
str = "Hello, world!";
newStr = insertBefore(str,6," there");
```

Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`substr` — Substring
string scalar

Substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`pos` — Character position
positive integer

Character position, specified as a positive integer.

`new` — New substring
string scalar

New substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History
Introduced in R2021b

See Also
`insertAfter` | `extractBefore`

Topics
“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

isletter

Determine which characters are letters

Syntax

```
tf = isletter(str)
```

Description

`tf = isletter(str)` returns a Boolean array based on whether each character of `str` is a letter or not.

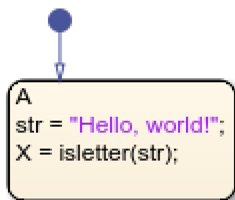
Note The `isletter` operator is not supported in Stateflow charts that use C as the action language.

Examples

Determine Which Characters of a String Are Letters

Create a Boolean vector of the form `[1 1 1 1 1 0 0 1 1 1 1 1 0]`.

```
str = "Hello, world!";
X = isletter(str);
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`isspace` | `isstring`

Topics

"Manage Textual Information by Using Strings"
"Share String Data with Custom C Code"

isspace

Determine which characters are spaces

Syntax

`tf = isspace(str)`

Description

`tf = isspace(str)` returns a Boolean array based on whether each character of `str` is a space or not.

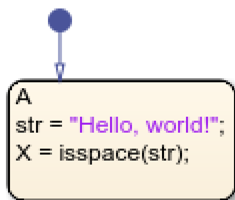
Note The `isspace` operator is not supported in Stateflow charts that use C as the action language.

Examples

Determine Which Characters of a String Are Spaces

Create a Boolean vector of the form `[0 0 0 0 0 0 1 0 0 0 0 0 0]`.

```
str = "Hello, world!";
X = isspace(str);
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`isletter` | `isstring`

Topics

"Manage Textual Information by Using Strings"
"Share String Data with Custom C Code"

isstring

Determine if input is string

Syntax

```
tf = isstring(X)
```

Description

`tf = isstring(X)` returns `1` (`true`) if `X` is a string. Otherwise, it returns `0` (`false`).

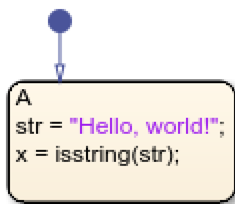
Note The `isstring` operator is not supported in Stateflow charts that use C as the action language.

Examples

Check Whether an Input Argument is a String Array

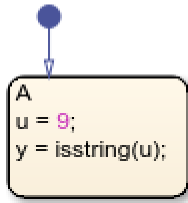
Return a value of `1` (`true`) because the input argument `"Hello, world!"` is a string.

```
str = "Hello, world";
x = isstring(str);
```



Return a value of `0` (`false`) because the input argument `9` is a not string.

```
u = 9;
y = isstring(u);
```



Input Arguments

X — Input value
scalar | vector | matrix | multidimensional array

Input value, specified as a scalar, vector, matrix, or multidimensional array. X can be any data type. If X is a string, it must be a string scalar.

Version History

Introduced in R2021b

See Also

isletter | isspace

Topics
"Manage Textual Information by Using Strings"
"Share String Data with Custom C Code"

isvalid

Determine if message is valid

Syntax

```
isvalid(message_name)
```

Description

`isvalid(message_name)` checks if an input or local message is valid. A message is valid if the chart has removed it from the queue and has not forwarded or discarded it.

Examples

Check Message in State Action

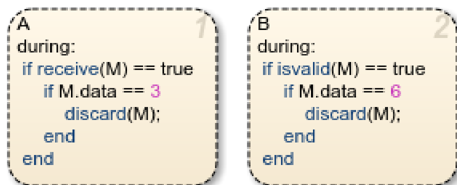
When state `A` is active, receive message `M`. If the message has a data value equal to 3, discard the message. Then, when state `B` is active, check that the message `M` is still valid. If the message is valid and has a data value equal to 6, discard the message.

In state `A`:

```
during:
   if receive(M) == true
      if M.data == 3
         discard(M);
      end
   end
```

In state `B`:

```
during:
   if isvalid(M) == true
      if M.data == 6
         discard(M);
      end
   end
```



Version History

Introduced in R2015b

See Also
`discard` | `forward` | `receive`

Topics
"Control Message Activity in Stateflow Charts"

length

Determine length of message queue

Syntax

`length(message_name)`

Description

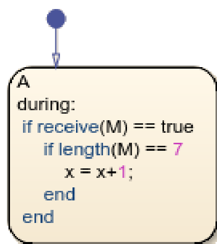
`length(message_name)` checks the number of messages in the internal receiving queue of an input or local message.

Examples

Check Queue Length in State Action

Check the queue for message `M`. If a message is present, remove it from the queue. If exactly seven messages remain in the queue, increment the value of `x`.

```
during:
   if receive(M) == true
      if length(M) == 7
         x = x+1;
      end
   end
```



Tips

- The `length` operator is not supported for input messages that use external receiving queues. To use the `length` operator, enable the **Use Internal Queue** property for this message.

Version History

Introduced in R2015b

See Also

`receive`

Topics

"Control Message Activity in Stateflow Charts"

lower

Convert string to lowercase

Syntax

```
newStr = lower(str)
```

Description

`newStr = lower(str)` converts the uppercase characters in the string `str` to the corresponding lowercase characters.

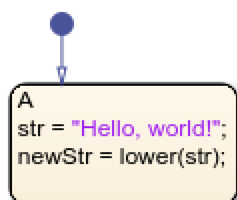
Note The `lower` operator is not supported in Stateflow charts that use C as the action language.

Examples

Convert String to Lowercase

Convert the uppercase characters and return the string `"hello, world!"`

```
str = "Hello, world!";
newStr = lower(str);
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`upper` | `reverse`

Topics

"Manage Textual Information by Using Strings"
"Share String Data with Custom C Code"

matches

Determine if two strings match

Syntax

```
tf = matches(str1,str2)
tf = matches(str1,str2,IgnoreCase=true)
```

Description

`tf = matches(str1,str2)` compares the strings `str1` and `str2`. The operator returns `1` (`true`) if the strings are identical, and returns `0` (`false`) otherwise.

`tf = matches(str1,str2,IgnoreCase=true)` compares strings `str1` and `str2`, ignoring any differences in letter case.

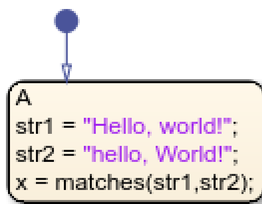
Note The `matches` operator is not supported in Stateflow charts that use C as the action language. For similar functionality, use `strcmp`.

Examples

Compare Strings

Return a value of `0` (`false`) because the strings do not match.

```
str1 = "Hello, world!";
str2 = "hello, World!";
x = matches(str1,str2);
```



Compare Strings While Ignoring Case

Return a value of `1` (`true`) because the strings match when you ignore case.

```
str1 = "Hello, world!";
str2 = "hello, World!";
x = matches(str1,str2,IgnoreCase=true);
```

Input Arguments

`str1`, `str2` — Input strings

string scalar

Input strings, specified as string scalars. Enclose literal string with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`strcmp` | `strcmpi` | `strncmp` | `strncmpi`

Topics

“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

plus, +

Concatenate strings

Syntax

```
newStr = str1 + str2
newStr = plus(str1,str2)
```

Description

`newStr = str1 + str2` concatenates the strings `str1` and `str2`.

`newStr = plus(str1,str2)` is an alternative way to execute `newStr = str1 + str2`.

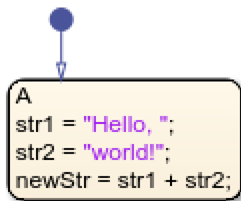
Note To concatenate strings in Stateflow charts that use C as the action language, use `strcat`.

Examples

Concatenate Strings

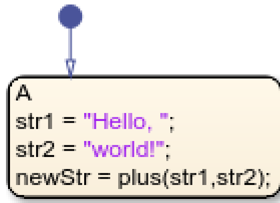
Concatenate strings to form `"Hello, world!"`.

```
str1 = "Hello, ";
str2 = "world!";
newStr = str1 + str2;
```



Alternatively, you can use `plus` to concatenate strings.

```
str1 = "Hello, ";
str2 = "world!";
newStr = plus(str1,str2);
```



Input Arguments

`str1`, `str2` — Input strings
string scalar

Input strings, specified as string scalars. Enclose literal string with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`extractAfter` | `extractBefore` | `strcat`

Topics
“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

receive

Extract message from queue

Syntax

```
receive(message_name)
```

Description

`tf = receive(message_name)` extracts an input or local message from its receiving queue. If a valid message exists, `receive` returns `true`. If a valid message does not exist but there is a message in the queue, the chart removes the message from the queue and `receive` returns `true`. If a valid message does not exist and there are no messages in the queue, `receive` returns `false`.

Examples

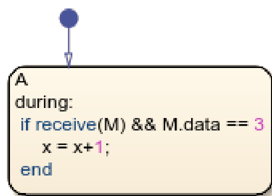
Extract Message in State Action

Check the queue for message `M` and increment the value of `x` if both of these conditions are true:

- A message is present in the queue.
- The data value of the message is equal to 3.

If a message is not present or if the data value is not equal to 3, then the value of `x` does not change. If a message is present, remove it from the queue regardless of the data value.

```
during:
   if receive(M) && M.data == 3
      x = x+1;
   end
```



Version History

Introduced in R2015b

See Also

`send`

Topics

"Control Message Activity in Stateflow Charts"

replace

Find and replace substrings

Syntax

```
newStr = replace(str,old,new)
```

Description

`newStr = replace(str,old,new)` replaces instances of the substring `old` that occur in the string `str` with the string `new`.

Note The `replace` operator is not supported in Stateflow charts that use C as the action language.

Examples

Replace Substring

Replace a substring to form the string `"Hello, Mars!"`.

```
str = "Hello, world!";
newStr = replace(str,"world","Mars");
```

Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`old` — Substring to replace
string scalar

Substring to replace, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`new` — New substring
string scalar

New substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Algorithms

The `replace` operator replaces sequential substrings. For example, `replace("abc 2 def 22 ghi 222 jkl 2222","22","*")` returns `"abc 2 def * ghi *2 jkl **"`. To replace overlapping substrings, use `strrep`. For more information, see “Replace Repeated Pattern”.

Version History

Introduced in R2021b

See Also

`replaceBetween` | `strrep`

Topics

“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

replaceBetween

Replace substrings between start and end points

Syntax

```
newStr = replaceBetween(str,startStr,endStr,new)
newStr = replaceBetween(str,startPos,endPos,new)
newStr = replaceBetween( ___ ,Boundaries=bounds)
```

Description

`newStr = replaceBetween(str,startStr,endStr,new)` replaces the substring in `str` between the substrings `startStr` and `endStr` with the string `new`. `replaceBetween` does not replace `startStr` and `endStr` themselves. `new` can have a different number of characters than the substring it replaces.

`newStr = replaceBetween(str,startPos,endPos,new)` replaces the substring in `str` between the character positions `startPos` and `endPos`, including the characters at those positions.

`newStr = replaceBetween( ___ ,Boundaries=bounds)` includes or excludes the boundaries specified in the previous syntaxes from the substring that the operator replaces. Specify `bounds` as `"inclusive"` or `"exclusive"`.

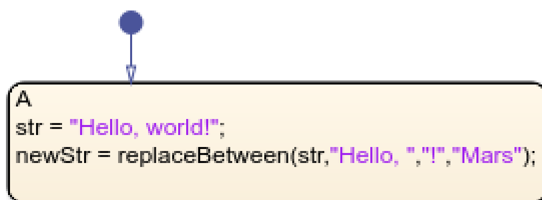
Note The `replaceBetween` operator is not supported in Stateflow charts that use C as the action language.

Examples

Replace Text Between Substrings

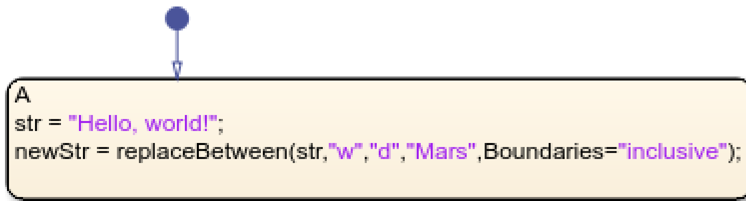
Replace a substring to form the string `"Hello, Mars!"`. By default, `replaceBetween` does not replace the boundary substrings.

```
str = "Hello, world!";
newStr = replaceBetween(str,"Hello, ","!","Mars");
```



Alternatively, use the option `Boundaries="inclusive"` to replace the boundary substrings.

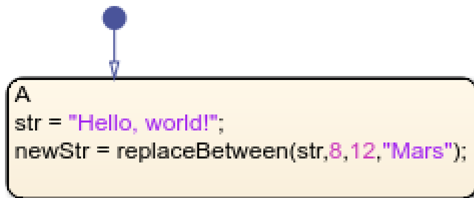
```
str = "Hello, world!";
newStr = replaceBetween(str,"w","d","Mars",Boundaries="inclusive");
```



Replace Text Between Start and End Positions

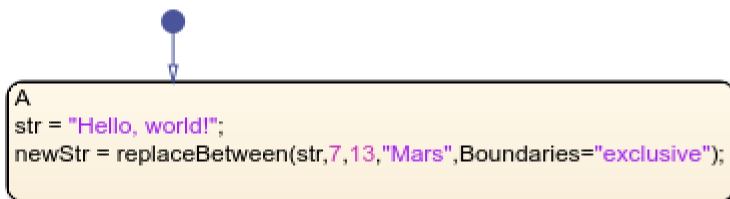
Replace a substring to form the string `"Hello, Mars!"`. By default, `replaceBetween` replaces the boundary characters.

```
str = "Hello, world!";
newStr = replaceBetween(str,8,12,"Mars");
```



Alternatively, use the `Boundaries="exclusive"` option to prevent replacement of the boundary characters.

```
str = "Hello, world!";
newStr = replaceBetween(str,7,13,"Mars",Boundaries="exclusive");
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`startStr` — Starting substring
string scalar

Staring substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: "Hello"

endStr — Ending substring
string scalar

Ending substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: "Hello"

startPos — Starting character position
positive integer

Starting character position, specified as a positive integer.

endPos — Ending character position
positive integer

Ending character position, specified as a positive integer.

new — New substring
string scalar

New substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: "Hello"

bounds — Boundary type
"inclusive" | "exclusive"

Boundary type, specified as "inclusive" or "exclusive". When you set bounds to "exclusive", replaceBetween replaces only the text between the boundaries. When you set bounds to "inclusive", replaceBetween also replaces the boundaries themselves.

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

replace | strrep | eraseBetween

Topics

“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

reverse

Reverse order of characters in strings

Syntax

```
newStr = reverse(str)
```

Description

`newStr = reverse(str)` reverses the order of the characters in the string `str`.

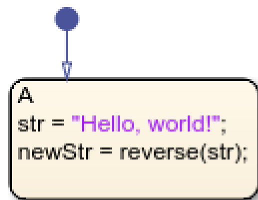
Note The `reverse` operator is not supported in Stateflow charts that use C as the action language.

Examples

Reverse String

Reverse the order of characters and return the string `"!dlrow ,olleH"`

```
str = "Hello, world!";
newStr = reverse(str);
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`lower` | `upper`

Topics

“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

rising

Detect rising edge in data since last time step

Syntax

```
tf = rising(expression)
```

Description

`tf = rising(expression)` returns 1 (`true`) if:

- The previous value of `expression` was negative and its current value is zero or positive.
- The previous value of `expression` was zero and its current value is positive.

Otherwise, the operator returns 0 (`false`). If `expression` changes value from negative to zero to positive at three consecutive time steps, the operator detects a single edge when the value of `expression` becomes zero.

The argument `expression`:

- Must be a scalar-valued expression
- Can combine chart input data, constants, nontunable parameters, continuous-time local data, and state data from Simulink based states
- Can include addition, subtraction, and multiplication of scalar variables, elements of a matrix, fields in a structure, or any valid combination of structure fields and matrix elements

Index elements of a matrix by using numbers or expressions that evaluate to a constant integer.

Note Edge detection is supported only in Stateflow charts in Simulink models.

Examples

Detect Rising Edge

Transition out of state if the value of the input data `signal` rises above a threshold of 2.5.

```
[rising(signal-2.5)]
```

The rising edge is detected when the value of the expression `signal-2.5` becomes zero or positive.

Tips

- The operator `rising` imitates the behavior of a Trigger block with **Trigger Type** set to `rising`.
- Edge detection for continuous-time local data and state data from Simulink based states is supported only in transition conditions.
- In atomic subcharts, map all input data that you use in edge detection expressions to input data or nontunable parameters in the main chart. Mapping these input data to output data, local data, or tunable parameters can result in undefined behavior.

Version History

Introduced in R2021b

See Also

`crossing` | `falling` | Trigger

Topics

"Detect Changes in Data and Expression Values"
"Operations for Vectors and Matrices in Stateflow"
"Index and Assign Values to Stateflow Structures"

send

Broadcast message or event

Syntax

```
send(message_name)
send(event_name)
send(local_event_name,state_name)
send(state_name.local_event_name)
```

Description

`send(message_name)` sends a local or output message.

`send(event_name)` sends a local or output event.

`send(local_event_name,state_name)` broadcasts a local event to `state_name` and any offspring of that state in the hierarchy.

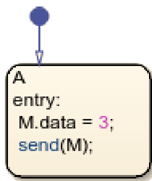
`send(state_name.local_event_name)` broadcasts a local event to its parent state `state_name` and any offspring of that state in the hierarchy.

Examples

Broadcast Message

Send a local or output message `M` with a data value of `3`.

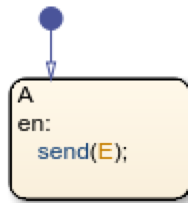
```
M.data = 3;
send(M);
```



Broadcast Output Event

Send an output event `E`.

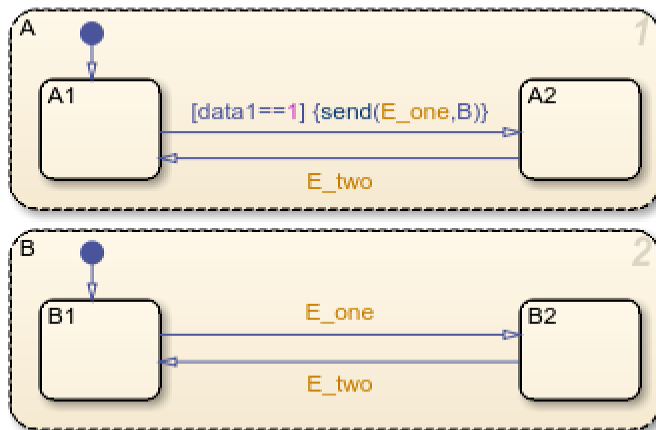
```
send(E);
```



Broadcast Directed Local Event

Send a local event E_one to state B and any of its substates.

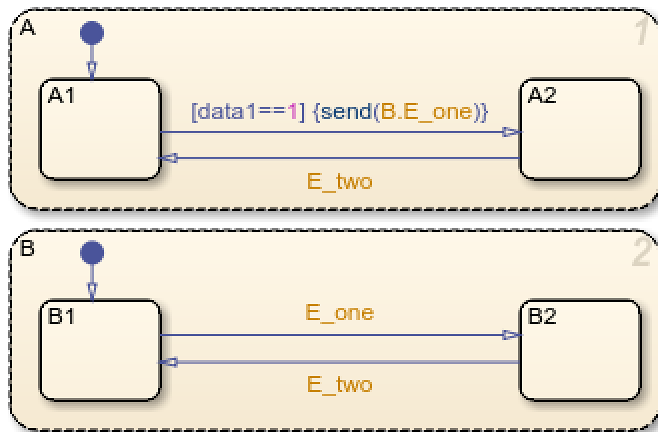
```
send(E_one,B);
```



Broadcast by Using Qualified Event Name

Send a local event E_one to its parent state B and any of its substates.

```
send(B.E_one);
```



Tips

- If a chart sends a message that exceeds the capacity of the receiving queue, a queue overflow occurs. The result of the queue overflow depends on the type of receiving queue.
  - When an overflow occurs in an internal queue, the Stateflow chart drops the new message. You can control the level of diagnostic action by setting the **Queue Overflow Diagnostic** property for the message. See “Queue Overflow Diagnostic”.
  - When an overflow occurs in an external queue, the Queue block either drops the new message or overwrites the oldest message in the queue, depending on the configuration of the block. See “Overwrite the oldest element if queue is full” (Simulink). An overflow in an external queue always results in a warning.
- Avoid using undirected local event broadcasts. Undirected local event broadcasts can cause unwanted recursive behavior in your chart. Instead, send local events by using directed broadcasts. For more information, see “Broadcast Local Events to Synchronize Parallel States”.
- Use the `send` operator to send events to the Schedule Editor. The Schedule Editor enables you to schedule the execution of aperiodic partitions. For more information on using the `send` operator with the Schedule Editor, see “Events in Schedule Editor” (Simulink).

Version History

Introduced before R2006a

See Also

`receive`

Topics

“Control Message Activity in Stateflow Charts”
“Activate a Simulink Block by Sending Output Events”
“Broadcast Local Events to Synchronize Parallel States”

startsWith

Determine if string starts with substring

Syntax

```
tf = startsWith(str,substr)
tf = startsWith(str,substr,IgnoreCase=true)
```

Description

`tf = startsWith(str,substr)` returns `1` (`true`) if the string `str` starts with the substring `substr`, and returns `0` (`false`) otherwise.

`tf = startsWith(str,substr,IgnoreCase=true)`checks if `str` starts with `substr`, ignoring any differences in letter case.

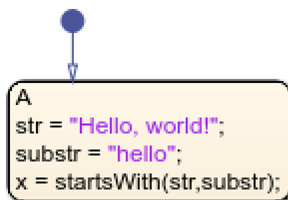
Note The `startsWith` operator is not supported in Stateflow charts that use C as the action language.

Examples

Determine If String Starts with Substring

Return a value of `0` (`false`) because the string `"Hello, world!"` does not start with the substring `"hello"`.

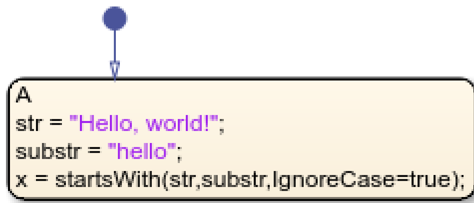
```
str = "Hello, world!";
substr = "hello";
x = startsWith(str,substr);
```



Determine If String Starts with Substring While Ignoring Case

Return a value of `1` (`true`) because the string `"Hello, world!"` starts with the substring `"hello"` when you ignore case.

```
str = "Hello, world!";
substr = "hello";
x = startsWith(str,substr,IgnoreCase=true);
```



Input Arguments

`str` — Input string

string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`substr` — Substring

string scalar

Substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`contains` | `endsWith` | `strfind`

Topics

“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

str2ascii

Convert string to array of type `uint8`

Syntax

`A = str2ascii(str,n)`

Description

`A = str2ascii(str,n)` returns array of type `uint8` containing ASCII values for the first `n` characters in `str`, where `n` is a positive integer. If `str` has fewer than `n` characters, the remaining elements of `A` are set to `0`.

Use of variables or expressions for `n` is not supported.

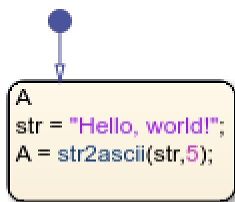
Note The operator `str2ascii` is supported only in Stateflow charts that use C as the action language.

Examples

String to ASCII Values

Return `uint8` array `{72,101,108,108,111}`.

```
str = "Hello, world!";
A = str2ascii(str,5);
```



Tips

- Enclose literal strings with single or double quotes.

Version History

Introduced in R2018b

See Also

`ascii2str`

Topics

“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

str2double, double

Convert string to double-precision value

Syntax

```
X = str2double(str)
X = double(str)
```

Description

`X = str2double(str)` converts the text in string `str` to a double-precision value.

- In a chart that uses MATLAB as the action language, `str2double` returns a complex value.
- In a chart that uses C as the action language, `str2double` returns a real value.

If `str2double` cannot convert the text to a number, it returns a `NaN` value.

`X = double(str)` is an alternative way to execute `str2double(str)` in charts that use MATLAB as the action language.

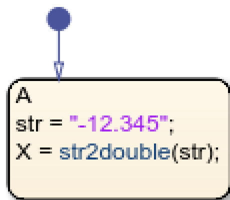
Note Stateflow charts that use C as the action language support calling `double` only with numeric arguments.

Examples

Convert String That Contains Decimal Notation

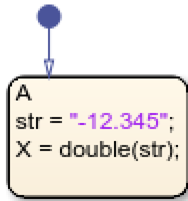
Convert the string `"-12.345"` to a double-precision numeric value.

```
str = "-12.345";
X = str2double(str);
```



Alternatively, in charts that use MATLAB as the action language, you can use the operator `double`:

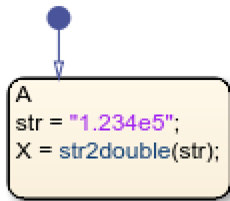
```
str = "-12.345";
X = double(str);
```



Convert String That Contains Exponential Notation

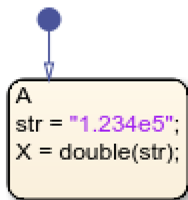
Return a value of `123400`.

```
str = "1.234e5";
X = str2double(str);
```



Alternatively, in charts that use MATLAB as the action language, you can use the operator `double`:

```
str = "1.234e5";
X = str2double(str);
```



Input Arguments

`str` — Input value

string scalar

Input value, specified as a string scalar.

`str` must contain text that represents a number, including:

- Digits
- A decimal point
- A leading `+` or `-` sign
- An `e` preceding a power of 10 scale factor

- An imaginary part followed by an `i` or a `j` (not supported in charts that use C as the action language)

In charts that use MATLAB as the action language, enclose literal strings with double quotes.

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2018b

See Also

`string` | `tostring`

Topics

“Manage Textual Information by Using Strings”

strcat

Concatenate strings

Syntax

```
newStr = strcat(str1,...,strN)
```

Description

`newStr = strcat(str1,...,strN)` concatenates strings `str1,...,strN`.

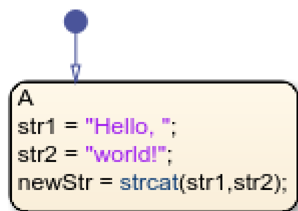
Note The operator `strcat` is supported only in Stateflow charts that use C as the action language. In charts that use MATLAB as the action language, use `plus`.

Examples

Concatenate Strings

Concatenate strings to form `"Hello, world!"`.

```
str1 = "Hello, ";
str2 = "world!";
newStr = strcat(str1,str2);
```



Tips

Enclose literal strings with single or double quotes.

Version History

Introduced in R2018b

See Also

`substr` | `plus`

Topics

“Manage Textual Information by Using Strings”

strcmp

Compare strings (case sensitive)

Syntax

```
tf = strcmp(str1,str2)
tf = strcmp(str1,str2,n)
```

Description

`tf = strcmp(str1,str2)` compares strings `str1` and `str2`.

- In charts that use MATLAB as the action language, the operator returns `1` (`true`) if the strings are identical and `0` (`false`) otherwise.
- In charts that use C as the action language, the operator returns `0` if the strings are identical. Otherwise, it returns a nonzero integer that depends on the compiler that you use. This value can differ in simulation and generated code.

`tf = strcmp(str1,str2,n)` compares the first `n` characters in `str1` and `str2`.

Note This syntax is supported only in Stateflow charts that use C as the action language. In charts that use MATLAB as the action language, use `strncmp`.

Examples

Compare Strings in Charts That Use MATLAB as the Action Language

Return a value of `1` (`true`) because the strings are equal.

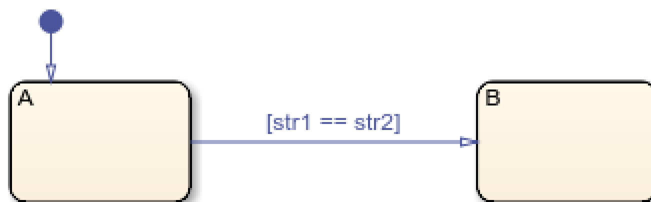
```
x = strcmp("Hello","Hello");
```

Return a value of `0` (`false`) because the strings are not equal.

```
y = strcmp("Hello","Hello!");
```

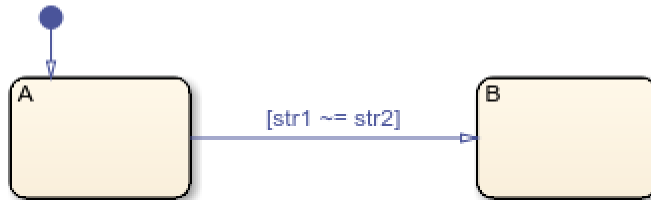
You can compare strings by using relational operators. Use == to determine if two strings are equal.

```
[str1 == str2]
```



Use ~= to determine if two strings are not equal.

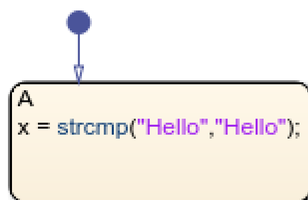
```
[str1 ~= str2]
```



Compare Strings in Charts That Use C as the Action Language

Return a value of `0` because the strings are equal.

```
x = strcmp("Hello","Hello");
```

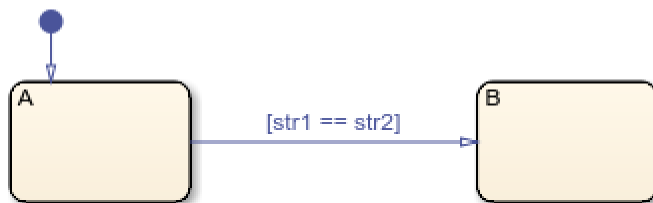


Return a nonzero value because the strings are not equal.

```
y = strcmp("Hello","Hello!");
```

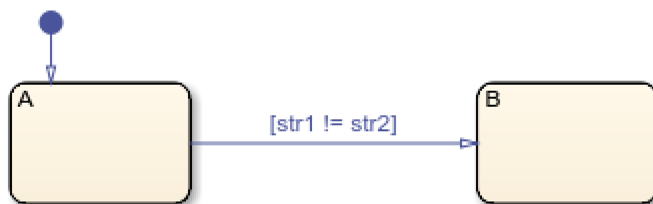
You can compare strings by using relational operators. Use == to determine if two strings are equal.

```
[str1 == str2]
```



Use != or ~= to determine if two strings are not equal.

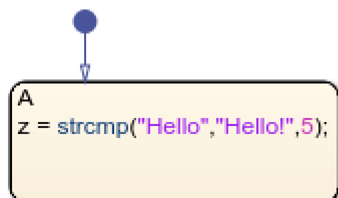
```
[str1 != str2]
```



Compare First N Characters of Strings

Return a value of 0 because the strings start with the same five characters.

```
z = strcmp("Hello","Hello!",5);
```



This syntax is supported only in Stateflow charts that use C as the action language. In charts that use MATLAB as the action language, use `strncmp`.

Input Arguments

`str1`, `str2` — Input strings

string scalar

Input strings, specified as string scalars. In charts that use MATLAB as the action language, enclose literal strings with double quotes.

Example: `"Hello"`

`n` — Number of characters checked

positive integer

Number of characters checked, starting at the beginning of each string, specified as a positive integer.

Version History

Introduced in R2018b

See Also

`matches` | `strcmpi` | `strncmp` | `strncmpi`

Topics

"Manage Textual Information by Using Strings"

strcmpi

Compare strings (case insensitive)

Syntax

`tf = strcmpi(str1,str2)`

Description

`tf = strcmpi(str1,str2)` compares strings `str1` and `str2`, ignoring differences in letter case. The operator returns `1` (`true`) if the strings are identical and `0` (`false`) otherwise.

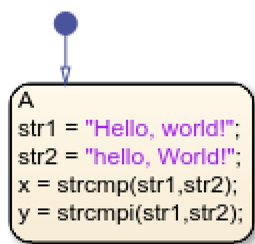
Note The operator `strcmpi` is not supported in Stateflow charts that use C as the action language.

Examples

Compare Strings While Ignoring Case

Set `x` to `0` (`false`) because the strings do not match. Set `y` to `1` (`true`) because the strings match when you ignore case.

```
str1 = "Hello, World!";
str2 = "hello, world!";
x = strcmp(str1,str2);
y = strcmpi(str1,str2);
```



Input Arguments

`str1`, `str2` — Input strings
string scalar

Input strings, specified as string scalars. Enclose literal string with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see "Access Bus Signals Through Stateflow Structures".

Version History

Introduced in R2021b

See Also

`matches` | `strcmp` | `strncmp` | `strncmpi`

Topics

"Manage Textual Information by Using Strings"

strcpy

Assign string value

Syntax

```
str1 = str2
strcpy(str1,str2)
```

Description

`str1 = str2` assigns string `str1` to string `str2`.

`strcpy(str1,str2)` is an alternative way to execute `str1 = str2`.

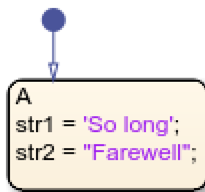
Note The operator `strcpy` is supported only in Stateflow charts that use C as the action language.

Examples

Assign String Data

Assign string data to `str1` and `str2`.

```
str1 = 'So Long';
str2 = "Farewell";
```



Alternatively, in charts that use C as the action language, you can use `strcpy` to assign string data.

```
strcpy(str3,'Auf Wiedersehen');
strcpy(str4,"Adieu");
```

Tips

- Source and destination arguments must refer to different symbols.
- Enclose literal strings with single or double quotes.

Version History

Introduced in R2018b

See Also

Topics

"Manage Textual Information by Using Strings"

strfind

Find substring within a string

Syntax

`k = strfind(str,substr)`

Description

`k = strfind(str,substr)` searches the string `str` for occurrences of the substring `substr`. The operator returns a vector that contains the starting index of each occurrence of `substr` in `str`. The search is case-sensitive.

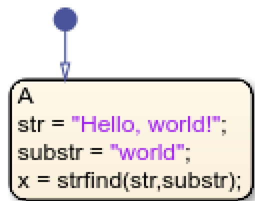
Note The `strfind` operator is not supported in Stateflow charts that use C as the action language.

Examples

Find Start of Substring

Return a value of 8, the starting character position of the substring `"world"` in the string `"Hello, world!"`.

```
str = "Hello, world!";
substr = "world";
x = strfind(str,substr);
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`substr` — Substring
string scalar

Substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Output Arguments

`k` — Starting character position of substring

vector of doubles

Starting character position of each occurrence of `subStr` in `str`, returned as a vector of doubles that contains the starting index of each occurrence of `substr` in `str`. If `strfind` does not find `subStr`, then `k` is an empty array.

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`contains` | `startsWith` | `endsWith`

Topics

“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

string

Convert value to string

Syntax

```
str = string(X)
```

Description

`str = string(X)` converts the input `X` to a string.

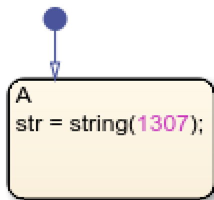
Note The operator `string` is not supported in Stateflow charts that use C as the action language. For similar functionality, use `tostring`.

Examples

Convert Integer Scalar to String

Convert integer scalar to string `"1307"`.

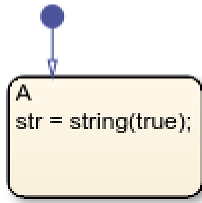
```
str = string(1307);
```



Convert Boolean Value to String

Convert Boolean value to string `"true"`.

```
str = string(true);
```



Input Arguments

X — Input value

integer scalar | Boolean scalar | character array

Input value, specified as a integer scalar, Boolean scalar, or character array. To create a character array of spaces, use the operator `blanks`.

Example: `str = string(1307)`

Example: `str = string(true)`

Example: `str = string(blanks(n))`

Version History

Introduced in R2021b

See Also

`str2double` | `blanks` | `tostring`

Topics

"Manage Textual Information by Using Strings"

strip

Remove leading and trailing characters from string

Syntax

```
newStr = strip(str)
newStr = strip(str,side)
newStr = strip( ___ ,stripCharacter)
```

Description

`newStr = strip(str)` removes consecutive whitespace characters from the beginning and end of the string `str`.

`newStr = strip(str,side)` removes consecutive white space characters from the side specified by `side`.

`newStr = strip( ___ ,stripCharacter)` strips the character specified by `stripCharacter`. You can use any of the input arguments in the previous syntaxes.

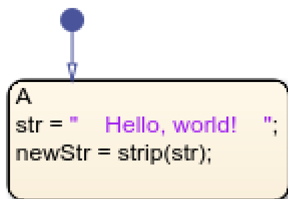
Note The `strip` operator is not supported in Stateflow charts that use C as the action language.

Examples

Strip Leading and Trailing Spaces from String

Remove the leading and trailing spaces and return the string `"Hello, world!"` .

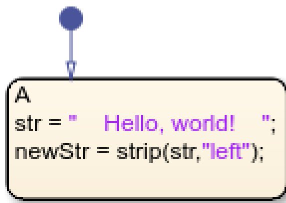
```
str = "    Hello, world!      ";
newStr = strip(str);
```



Strip Spaces from One Side of String

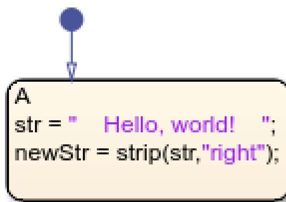
Remove the leading spaces and return the string `"Hello, world!     "`.

```
str = "    Hello, world!    ";
str1 = strip(h,"left");
```



Remove the trailing spaces and return the string `"     Hello, world!"`.

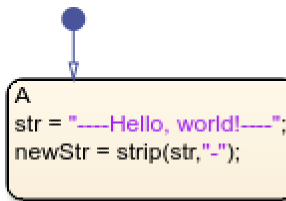
```
str = "     Hello, world!     ";
str1 = strip(h,"right");
```



Strip Leading and Trailing Characters from String

Remove the leading and trailing hyphens and return the string `"Hello, world!"` .

```
str = "----Hello, world!----";
newStr = strip(str);
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`side` — Side of string to strip
`"both"` (default) | `"left"` | `"right"`

Side of string to strip, specified as `"left"`, `"right"`, or `"both"`.

`stripCharacter` — Character to remove
" " (default) | string scalar

Character to remove, specified as a string scalar.

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`strtrim`

Topics
“Manage Textual Information by Using Strings”
“Share String Data with Custom C Code”

strlen

Determine length of string

Syntax

```
L = strlen(str)
```

Description

`L = strlen(str)` returns the number of characters in the string `str`.

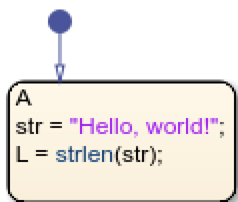
Note The operator `strlen` is supported only in Stateflow charts that use C as the action language. In charts that use MATLAB as the action language, use `strlength`.

Examples

Determine Length of String

Return a value of `13`, the number of characters in the string.

```
str = "Hello, world!";
L = strlen(str);
```



Tips

- Enclose literal strings with single or double quotes.

Version History

Introduced in R2018b

See Also

`strlength`

Topics

"Manage Textual Information by Using Strings"

strlength

Determine length of string

Syntax

`L = strlength(str)`

Description

`L = strlength(str)` returns the number of characters in the string `str`.

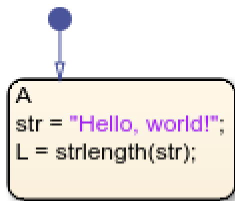
Note The operator `strlength` is not supported in Stateflow charts that use C as the action language. For similar functionality, use `strlen`.

Examples

Determine Length of String

Return a value of 13, the number of characters in the string.

```
str = "Hello, world!";
L = strlength(str);
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Version History

Introduced in R2021b

See Also

`string` | `contains` | `strlen`

Topics
"Manage Textual Information by Using Strings"

strncmp

Compare first N characters of strings (case sensitive)

Syntax

```
tf = strncmp(str1,str2,n)
```

Description

`tf = strncmp(str1,str2,n)` compares the first `n` characters of `str1` and `str2`. The operator returns `1` (`true`) if the strings are identical and `0` (`false`) otherwise.

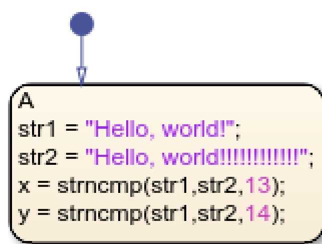
Note The operator `strncmp` is not supported in Stateflow charts that use C as the action language. For similar functionality, use `strcmp`.

Examples

Compare First N Characters of Strings

Set `x` to `1` (`true`) because the first 13 characters in the strings match. Set `y` to `0` (`false`) because the first 14 characters in the strings do not match.

```
str1 = "Hello, world!";
str2 = "Hello, world!!!!!!!!!!!!";
x = strncmp(str1,str2,13);
y = strncmp(str1,str2,14);
```



Input Arguments

`str1`, `str2` — Input strings
string scalar

Input strings, specified as string scalars. Enclose literal string with double quotes.

Example: `"Hello"`

`n` — Number of characters checked
positive integer

Number of characters checked, starting at the beginning of each string, specified as a positive integer.

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see "Access Bus Signals Through Stateflow Structures".

Version History

Introduced in R2021b

See Also

`matches` | `strcmp` | `strcmpi` | `strncmpi`

Topics

"Manage Textual Information by Using Strings"
"Share String Data with Custom C Code"

strncmpi

Compare first N characters of strings (case insensitive)

Syntax

```
tf = strncmpi(str1,str2,n)
```

Description

`tf = strncmpi(str1,str2,n)` compares the first `n` characters of `str1` and `str2`, ignoring any differences in letter case. The operator returns `1` (`true`) if the strings are identical and `0` (`false`) otherwise.

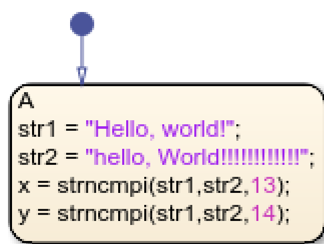
Note The operator `strncmpi` is not supported in Stateflow charts that use C as the action language.

Examples

Compare First N Characters While Ignoring Case

Set `x` to `1` (`true`) because the first 13 characters in the strings match when you ignore case. Set `y` to `0` (`false`) because the first 14 characters in the strings do not match.

```
str1 = "Hello, world!";
str2 = "hello, World!!!!!!!!!!!!";
x = strncmpi(str1,str2,13);
y = strncmpi(str1,str2,14);
```



Input Arguments

`str1`, `str2` — Input strings
string scalar

Input strings, specified as string scalars. Enclose literal string with double quotes.

Example: `"Hello"`

`n` — Number of characters checked
positive integer

Number of characters checked, starting at the beginning of each string, specified as a positive integer.

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`matches` | `strcmp` | `strcmpi` | `strncmp`

Topics

“Manage Textual Information by Using Strings”

strrep

Find and replace substrings

Syntax

`newStr = strrep(str,old,new)`

Description

`newStr = strrep(str,old,new)` replaces instances of the substring `old` that occur in the string `str` with the string `new`.

Note The `strrep` operator is not supported in Stateflow charts that use C as the action language.

Examples

Replace Substring

Replace a substring to form the string `"Hello, Mars!"`.

```
str = "Hello, world!";
newStr = strrep(str,"world","Mars");
```

Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`old` — Substring to replace
string scalar

Substring to replace, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

`new` — New substring
string scalar

New substring, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see "Access Bus Signals Through Stateflow Structures".

Algorithms

The `strrep` operator replaces overlapping substrings. For example, `strrep("abc 2 def 22 ghi 222 jkl 2222","22","*")` returns `"abc 2 def * ghi ** jkl ***"`. To replace only sequential substrings, use `replace`. For more information, see "Replace Repeated Pattern".

Version History

Introduced in R2021b

See Also

`replace` | `replaceBetween`

Topics

"Manage Textual Information by Using Strings"
"Share String Data with Custom C Code"

strtrim

Remove leading and trailing white space from string

Syntax

```
newStr = strtrim(str)
```

Description

`newStr = strtrim(str)` removes the leading and trailing whitespace characters from the string `str`.

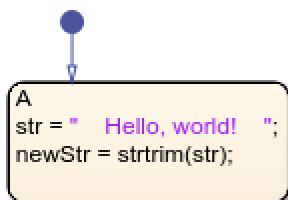
Note The `strtrim` operator is not supported in Stateflow charts that use C as the action language.

Examples

Remove Leading and Trailing Spaces from String

Remove the leading and trailing spaces and return the string `"Hello, world!"`

```
str = "    Hello, world!    ";
newStr = strtrim(str);
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see "Access Bus Signals Through Stateflow Structures".

Version History

Introduced in R2021b

See Also

`strip`

Topics

"Manage Textual Information by Using Strings"
"Share String Data with Custom C Code"

substr

Extract substring from string

Syntax

```
newStr = substr(str,pos,length)
```

Description

`newStr = substr(str,pos,length)` returns the substring of `str` that starts at the character position `pos` and is `length` characters long. Use zero-based indexing.

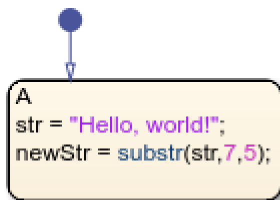
Note The operator `substr` is supported only in Stateflow charts that use C as the action language. In charts that use MATLAB as the action language, use `extractAfter` or `extractBefore`.

Examples

Extract Substring

Extract substring `"world"` from a longer string.

```
str = "Hello, world!";
newStr = substr(str,7,5);
```



Tips

- Use zero-based indexing.
- Enclose literal strings with single or double quotes.

Version History

Introduced in R2018b

See Also

`extractAfter` | `extractBefore`

Topics

"Manage Textual Information by Using Strings"

temporalCount

Number of events, chart executions, or time since state became active

Syntax

```
temporalCount(E)
temporalCount(tick)
temporalCount(time_unit)
```

Description

`temporalCount(E)` returns the number of occurrences of the event `E` since the associated state became active.

`temporalCount(tick)` returns the number of times that the chart has woken up since the associated state became active.

The implicit event `tick` is not supported when a Stateflow chart in a Simulink model has input events.

`temporalCount(time_unit)` returns the length of time that has elapsed since the associated state became active. Specify `time_unit` as seconds (`sec`), milliseconds (`msec`), or microseconds (`usec`).

Note Standalone Stateflow charts in MATLAB support using `temporalCount` only as an absolute-time temporal logic operator.

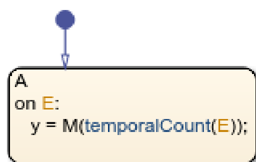
Examples

Perform Action on Event Broadcast

Access successive elements of the array `M` each time that the chart processes a broadcast of the event `E`.

In charts in a Simulink model, enter:

```
on E:
   y = M(temporalCount(E));
```



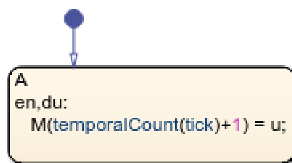
Using `temporalCount` as an event-based temporal logic operator is not supported in standalone charts in MATLAB.

Perform Action on Chart Execution

Store the value of the input data `u` in successive elements of the array `M`.

In charts in a Simulink model, enter:

```
en,du:
   M(temporalCount(tick)+1) = u;
```

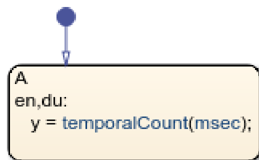


Using `temporalCount` as an event-based temporal logic operator is not supported in standalone charts in MATLAB.

Determine Time of State Activity

Store the number of milliseconds since the state became active.

```
en,du:
   y = temporalCount(msec);
```

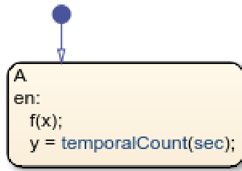


Tips

- You can use quotation marks to enclose the keywords `'tick'`, `'sec'`, `'msec'`, and `'usec'`. For example, `temporalCount('tick')` is equivalent to `temporalCount(tick)`.
- The Stateflow chart resets the counter used by the `temporalCount` operator each time the associated state reactivates.
- The timing for absolute-time temporal logic operators depends on the type of Stateflow chart:

  - Charts in a Simulink model define temporal logic in terms of simulation time.
  - Standalone charts in MATLAB define temporal logic in terms of wall-clock time.

  The difference in timing can affect the behavior of a chart. For example, suppose that this chart is executing the `entry` action of state `A`.



- In a Simulink model, the function call to `f` executes in a single time step and does not contribute to the simulation time. After calling the function `f`, the chart assigns a value of zero to `y`.
- In a standalone chart, the function call to `f` can take several seconds of wall-clock time to complete. After calling the function `f`, the chart assigns the nonzero time that has elapsed since state `A` became active to `y`.

Version History

Introduced in R2008a

See Also

`count` | `duration` | `elapsed`

Topics

"Control Chart Execution by Using Temporal Logic"
"Count Events by Using the temporalCount Operator"

this

Access chart data during simulation

Syntax

```
this
```

Description

`this` provides external MATLAB code, such as functions and apps, access to chart data during simulation.

- For charts in Simulink models, external MATLAB code can access inputs, outputs, and local data.
- For standalone charts in MATLAB, external MATLAB code can access local data and call `step`, input event functions, and graphical and MATLAB functions in the chart. For more information, see “Execute a Standalone Chart”.

Note In charts in Simulink models, the keyword `this` is supported only as an argument to external MATLAB code. Any other use of the keyword in the chart results in a compile-time error.

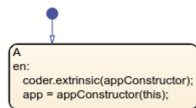
Examples

Connect Chart to MATLAB App

Create a bidirectional connection between a Stateflow chart and a MATLAB app created in App Designer. Call the app as an extrinsic function using `this` as an argument to the constructor. In the app, create a custom property to interface with the chart during simulation. In the chart, store the value returned by the function call to the app as a local data object.

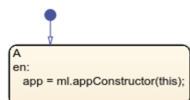
In a chart that uses MATLAB as the action language, enter:

```
coder.extrinsic(appConstructor);
app = appConstructor(this);
```



In a chart that uses C as the action language, enter:

```
app = ml.appConstructor(this);
```



For additional examples that illustrate this workflow, see “Model a Power Window Controller” and “Simulate a Media Player”.

Change Data Value While Debugging Standalone Chart

Modify the value of the local data `x` while debugging a standalone Stateflow chart in MATLAB.

At the debugging prompt, enter:

```
this.x = 7
```

For more information, see “Examine and Change Values of Chart Data”.

Note When debugging a chart in a Simulink model, you can access all Stateflow data directly at the debugging prompt. For more information, see “View and Modify Data in the MATLAB Command Window”.

Tips

- Do not use the keyword `this` to access chart data after simulation has stopped.
- Calling an external function named `this` from a chart disables the keyword `this` throughout the chart. To use the keyword, rename the extrinsic function.

Version History

Introduced in R2020b

See Also

`coder.extrinsic`

Topics

“Model a Power Window Controller”
“Simulate a Media Player”
“Model a Fitness Tracker”
“Call Extrinsic MATLAB Functions in Stateflow Charts”
“Access MATLAB Functions and Workspace Data in C Charts”
“Debug a Standalone Stateflow Chart”

tostring

Convert value to string

Syntax

```
str = tostring(X)
```

Description

`str = tostring(X)` converts numeric, Boolean, or enumerated data `X` to a string.

Note The operator `tostring` is supported only in Stateflow charts that use C as the action language. In charts that use MATLAB as the action language, use `string`.

Examples

Convert Numeric Value to String

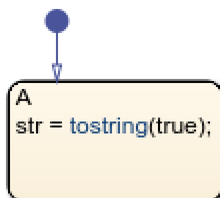
Convert numeric value to string `"1.2345"`.

```
str = tostring(1.2345);
```

Convert Boolean Value to String

Convert Boolean value to string `"true"`.

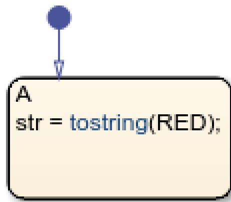
```
str = tostring(true);
```



Convert Enumerated Value to String

Convert enumerated value to string "RED".

```
str = tostring(RED);
```



Version History

Introduced in R2018b

See Also

`str2double` | `string`

Topics

“Manage Textual Information by Using Strings”

type

Type of Stateflow data object

Syntax

```
type(data_name)
```

Description

`type(data_name)` returns the type of a Stateflow data object. Use the `type` operator to derive the type of a Stateflow data object from other data objects.

In charts that use C as the action language, you can also use the return value in place of an explicit type in a `cast` operation to convert the value of an expression to the same type as another data object.

Tip In charts that use MATLAB as the action language, convert the value of an expression to the same type as another data object by calling the `cast` function with the keyword `"like"`. For more information, see "Cast Type Based on Other Data".

Examples

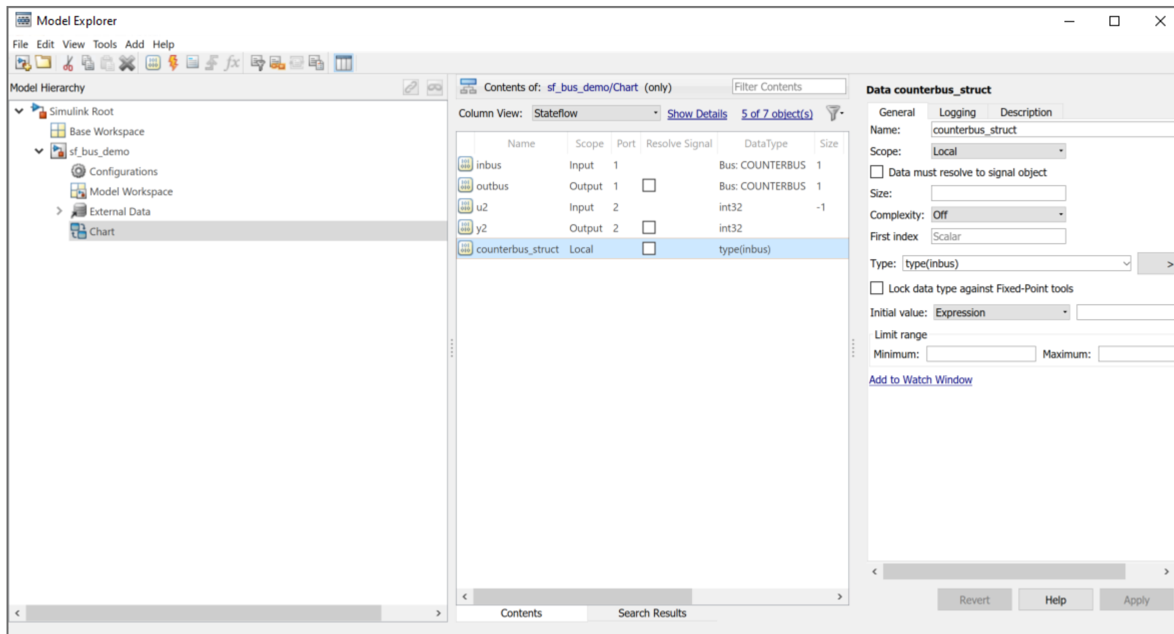
Derive Data Type from Other Data Objects

Open the example `sf_bus_demo`.

```
openExample("stateflow/InterfaceSimulinkBusSignalsIntegrateCustomCCodeExample")
```

In the Property Inspector or Model Explorer, use the data type of the input structure `inbus` to define the data type of the local structure `counterbus_struct`.

```
type(inbus)
```



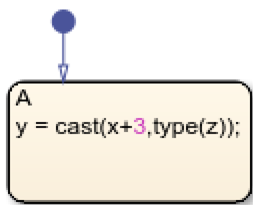
Because `inbus` derives its type from the `Simulink.Bus` object `COUNTERBUS`, `counterbus_struct` also derives its data type from `COUNTERBUS`.

For more information about this example, see "Integrate Custom Structures in Stateflow Charts".

Cast Type Based on Other Data

In a chart that uses C as the action language, cast the expression `x+3` to the same type as data `z` and assign its value to `y`.

```
y = cast(x+3,type(z));
```



Input Arguments

`data_name` — Data name
name of Stateflow data

Data name, specified as the name of a Stateflow data object.

Version History

Introduced before R2006a

See Also

`cast`

Topics

"Type Cast Operations"
"Specify Type of Stateflow Data"

upper

Convert a string to uppercase

Syntax

```
newStr = upper(str)
```

Description

`newStr = upper(str)` converts the lowercase characters in the string `str` to the corresponding uppercase characters.

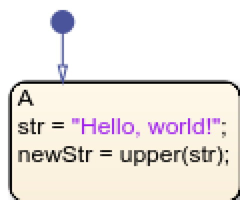
Note The `upper` operator is not supported in Stateflow charts that use C as the action language.

Examples

Convert String to Uppercase

Convert the lowercase characters and return the string `"HELLO, WORLD!"`

```
str = "Hello, world!";
newStr = upper(str);
```



Input Arguments

`str` — Input string
string scalar

Input string, specified as a string scalar. Enclose literal strings with double quotes.

Example: `"Hello"`

Limitations

- This operator does not support the use of Stateflow structure fields. For more information about structures in Stateflow, see “Access Bus Signals Through Stateflow Structures”.

Version History

Introduced in R2021b

See Also

`lower` | `reverse`

Topics

"Manage Textual Information by Using Strings"
"Share String Data with Custom C Code"

4

Objects

Stateflow.op.BlockOperatingPoint

Operating point information for Stateflow chart

Description

A `Stateflow.op.BlockOperatingPoint` object contains a snapshot of a Stateflow chart during simulation. The operating point includes information about:

- Active states
- Chart output data
- Chart, state, and function local data
- Persistent variables in MATLAB functions and truth tables

Creation

When you save the final operating point for a Simulink model, as described in “Save Operating Points”, you create a `Simulink.op.ModelOperatingPoint` object that contains a `Stateflow.op.BlockOperatingPoint` object for each Stateflow chart in the model.

Access the `Stateflow.op.BlockOperatingPoint` object for a chart by calling the `get` function and using the block path to the chart. For example, if the final operating point for the model is `xFinal` and the block path to your chart is `"myModel/Chart"`, enter:

```
op = get(xFinal,"myModel/Chart");
```

Properties

The `Stateflow.op.BlockOperatingPoint` object contains a property for each state, box, function, local data, and output data in the chart. The name of the property matches the name of the state, function, box, or data. For example:

- If a chart has a state named `state`, the `Stateflow.op.BlockOperatingPoint` object for the chart has a property named `state` that is a `Stateflow.op.OperatingPointContainer` object.
- If a chart has a chart output named `output`, the `Stateflow.op.BlockOperatingPoint` object for the chart has a property named `output` that is a `Stateflow.op.OperatingPointData` object.

Object Functions

highlightActiveStates	Highlight active states
removeHighlighting	Remove highlighting of active states
clone	Copy operating point for Stateflow chart
open	Display object in editing environment

Examples

Modify Operating Point Information for State Activity

1. Open the `sf_aircraft` model.

   ```
   openExample("stateflow/FaultDetectionControlLogicInAnAircraftControlSystemExample")
   ```

   For more information about this model, see “Detect Faults in Aircraft Elevator Control System”.
2. Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   1. Select **Final states** and enter a name for the operating point. For this example, use `xSteadyState`.
   2. Select **Save final operating point**.
   3. Click **OK**.
3. Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to `3`.
4. Run the simulation.
5. Access the `Stateflow.op.BlockOperatingPoint` object that contains the operating point information for the `Mode Logic` chart.

   ```
   blockpath = "sf_aircraft/Mode Logic";
   op = get(xSteadyState,blockpath)
   ```

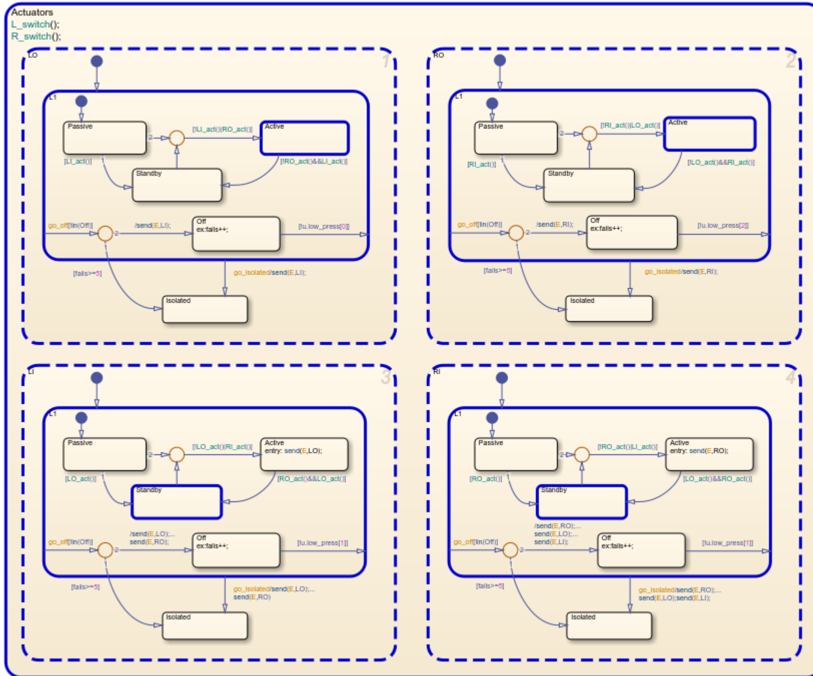
   ```
   op =

     Block:    "Mode Logic"    (handle)    (active)
     Path:     sf_aircraft/Mode Logic

     Contains:

       + Actuators          "State (OR)"                        (active)
       + LI_act             "Function"
       + LO_act             "Function"
       + L_switch           "Function"
       + RI_act             "Function"
       + RO_act             "Function"
       + R_switch           "Function"
         LI_mode            "State output data"        sf_aircraft_ModeType [1,1]
         LO_mode            "State output data"        sf_aircraft_ModeType [1,1]
         RI_mode            "State output data"        sf_aircraft_ModeType [1,1]
         RO_mode            "State output data"        sf_aircraft_ModeType [1,1]
   ```
6. Access the `Stateflow.op.OperatingPointContainer` object that contains the operating point information for the `Actuators` state.

   ```
   op.Actuators
   ```

   ```
   ans =

     State:    "Actuators"    (handle)    (active)
     Path:     sf_aircraft/Mode Logic/Actuators

     Contains:

       + LI            "State (AND)"         (active)
       + LO            "State (AND)"         (active)
       + RI            "State (AND)"         (active)
       + RO            "State (AND)"         (active)
   ```

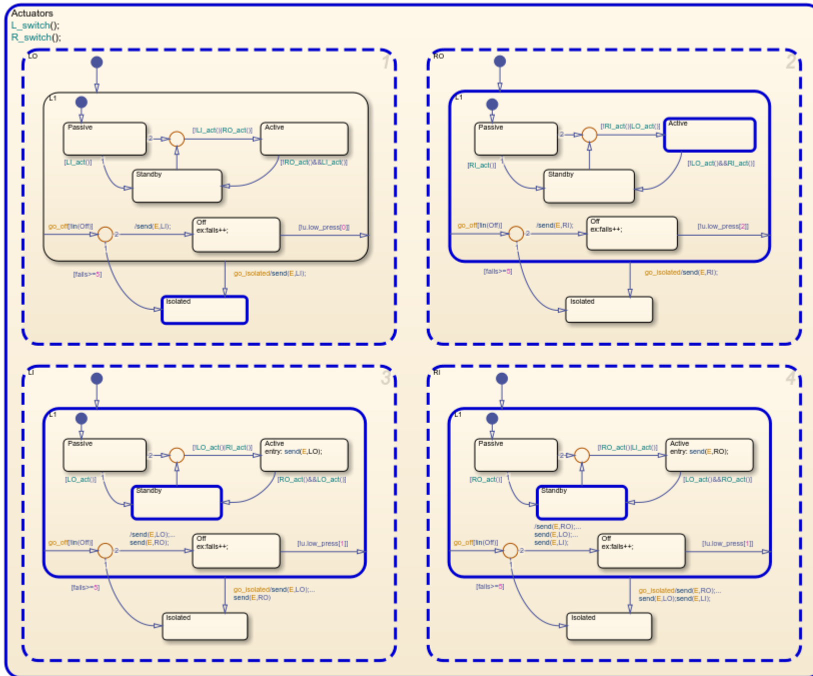
7 Highlight the states that are active in the chart at $t = 3$.

```
highlightActiveStates(op)
```



8 Change the substate activity in the state `LO` to reflect a failure of the left-outer actuator.

```
setActive(op.Actuators.LO.Isolated)
```



9 Verify that the substate `Isolated` in the state `LO` is active in the modified operating point.

```
isActive(op.Actuators.LO.Isolated)

ans =

  logical

   1
```

10 Remove the highlighting of active states in the Stateflow Editor.

```
removeHighlighting(op)
```

Modify Operating Point Information for Output Data

1 Open the model `old_sf_car`.

```
openExample("stateflow/AutomaticTransmissionLegacyExample")
```

2 Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   1 Select **Final states** and enter a name for the operating point. For this example, use `xSteadyState`.

   2 Select **Save final operating point**.

   3 Click **OK**.

3 Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to `10`.

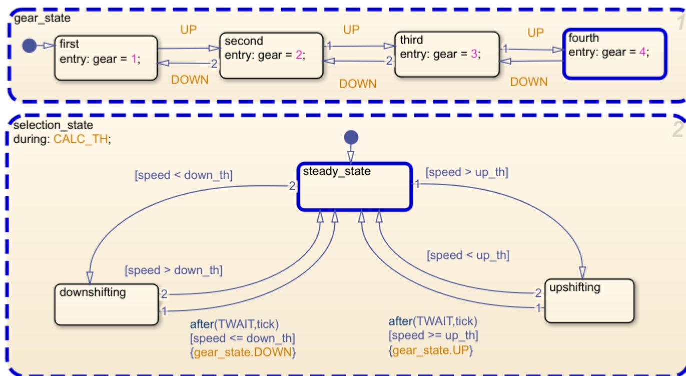
4 Run the simulation.

5 Access the `Stateflow.op.BlockOperatingPoint` object that contains the operating point information for the `shift_logic` chart.

```
blockpath = "old_sf_car/shift_logic";
op = get(xSteadyState,blockpath)

op =

Block:    "shift_logic"    (handle)    (active)
  Path:     old_sf_car/shift_logic

  Contains:

    + gear_state         "State (AND)"          (active)
    + selection_state    "State (AND)"          (active)
      gear               "Block output data"    double [1, 1]
```

6 Highlight the states that are active in the chart at $t = 10$.

```
highlightActiveStates(op)
```



7. Access the `Stateflow.op.OperatingPointData` object that contains the operating point information for the chart output `gear`.

   ```
   op.gear
   ```

   ```
   ans = 

         Description: 'Block output data'
            DataType: 'double'
                Size: '[1, 1]'
               Range: [1x1 struct]
        InitialValue: [1x0 double]
               Value: 4
   ```

8. Change the value of `gear` to `1`.

   ```
   op.gear.Value = 1;
   ```

9. Inspect the modified operating point information for the chart output `gear`.

   ```
   op.gear
   ```

   ```
   ans = 

         Description: 'Block output data'
            DataType: 'double'
                Size: '[1, 1]'
               Range: [1x1 struct]
        InitialValue: [1x0 double]
               Value: 1
   ```

Version History

Introduced in R2009b

R2019a: Renamed from `Stateflow.SimState.BlockSimState`

Behavior changed in R2019a

`Stateflow.SimState.BlockSimState` is now called `Stateflow.op.BlockOperationPoint`. The behavior remains the same.

See Also

Objects
`Stateflow.op.OperatingPointContainer` | `Stateflow.op.OperatingPointData` | `Simulink.op.ModelOperatingPoint`

Functions
`get`

Topics
"Save and Restore Operating Points for Stateflow Charts"
"Use Operating Points to Specify Initial State of Simulation"
"Test Difficult-to-Reproduce Chart Configurations"
"Test Chart with Fault Detection and Redundant Logic"

Stateflow.op.OperatingPointContainer

Operating point information for state, box, or function

Description

A `Stateflow.op.OperatingPointContainer` object contains a snapshot of a state, box, or function in a Stateflow chart during simulation. The operating point includes information about:

- Active substates
- State and function local data
- Persistent variables in MATLAB functions and truth tables

Creation

When you save the final operating point for a Simulink model, as described in "Save Operating Points", you create a `Stateflow.op.BlockOperatingPoint` object for each Stateflow chart in the model. This object contains a `Stateflow.op.OperatingPointContainer` object for each state, box, or function in the chart.

Access a `Stateflow.op.OperatingPointContainer` object by using the property that matches the name of the state, box, or function in the parent `Stateflow.op.BlockOperatingPoint` or `Stateflow.op.OperatingPointContainer`. For example, suppose that `op` is the `Stateflow.op.BlockOperatingPoint` object for a chart. To access the `Stateflow.op.OperatingPointContainer` object for a top-level state called `state`, enter:

```
op.state
```

Similarly, to access the `Stateflow.op.OperatingPointContainer` object for a substate called `substate` in the top-level state `state`, enter:

```
op.state.substate
```

Properties

The `Stateflow.op.OperatingPointContainer` object contains a property for each substate, box, function, local data, and persistent variables in the state, box, or function. The name of the property matches the name of the state, function, box, or data. For example:

- If a state has a substate named `substate`, the `Stateflow.op.OperatingPointContainer` object for the state has a property named `substate` that is specified as a `Stateflow.op.OperatingPointContainer` object.
- If a MATLAB function has a persistent variable named `persistentVar`, the `Stateflow.op.OperatingPointContainer` object for the function has a property named `persistentVar` that is specified as a `Stateflow.op.OperatingPointData` object.

Object Functions

setActive	Set state as active
isActive	Determine if state is active
getPrevActiveChild	Get previously active substate
setPrevActiveChild	Set previously active substate
open	Display object in editing environment

Examples

Modify Operating Point Information for State Activity

1. Open the `sf_aircraft` model.

   ```
   openExample("stateflow/FaultDetectionControlLogicInAnAircraftControlSystemExample")
   ```

   For more information about this model, see “Detect Faults in Aircraft Elevator Control System”.
2. Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   1. Select **Final states** and enter a name for the operating point. For this example, use `xSteadyState`.
   2. Select **Save final operating point**.
   3. Click **OK**.
3. Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to `3`.
4. Run the simulation.
5. Access the `Stateflow.op.BlockOperatingPoint` object that contains the operating point information for the `Mode Logic` chart.

   ```
   blockpath = "sf_aircraft/Mode Logic";
   op = get(xSteadyState,blockpath)
   ```

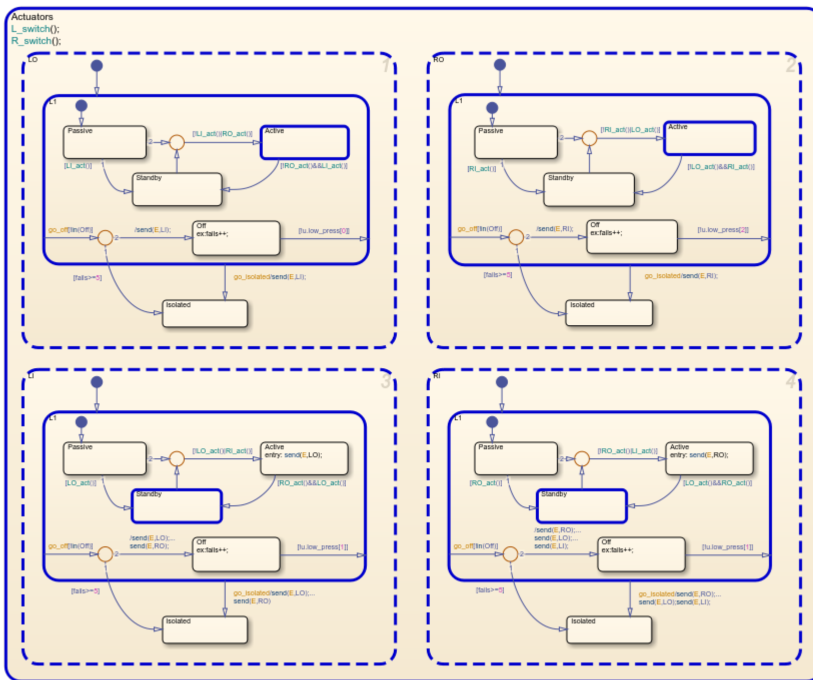
   ```
   op =

     Block:    "Mode Logic"    (handle)    (active)
     Path:     sf_aircraft/Mode Logic

     Contains:

       + Actuators          "State (OR)"                        (active)
       + LI_act             "Function"
       + LO_act             "Function"
       + L_switch           "Function"
       + RI_act             "Function"
       + RO_act             "Function"
       + R_switch           "Function"
         LI_mode            "State output data"       sf_aircraft_ModeType [1,1]
         LO_mode            "State output data"       sf_aircraft_ModeType [1,1]
         RI_mode            "State output data"       sf_aircraft_ModeType [1,1]
         RO_mode            "State output data"       sf_aircraft_ModeType [1,1]
   ```
6. Access the `Stateflow.op.OperatingPointContainer` object that contains the operating point information for the `Actuators` state.

   ```
   op.Actuators
   ```

```
ans = 

  State:     "Actuators"     (handle)     (active)
  Path:      sf_aircraft/Mode Logic/Actuators

  Contains:

    + LI             "State (AND)"         (active)
    + LO             "State (AND)"         (active)
    + RI             "State (AND)"         (active)
    + RO             "State (AND)"         (active)
```

7 Highlight the states that are active in the chart at $t = 3$.

```
highlightActiveStates(op)
```



8 Change the substate activity in the state `LO` to reflect a failure of the left-outer actuator.

```
setActive(op.Actuators.LO.Isolated)
```

9. Verify that the substate `Isolated` in the state `LO` is active in the modified operating point.

```
isActive(op.Actuators.LO.Isolated)
```

```
ans =

  logical

   1
```

10. Remove the highlighting of active states in the Stateflow Editor.

```
removeHighlighting(op)
```

Version History

Introduced in R2009b

R2019a: Renamed from `Stateflow.SimState.SimStateContainer`

Behavior changed in R2019a

`Stateflow.SimState.SimStateContainer` is now called `Stateflow.op.OperationPointContainer`. The behavior remains the same.

See Also

Objects

`Stateflow.op.BlockOperatingPoint` | `Stateflow.op.OperatingPointData` | `Simulink.op.ModelOperatingPoint`

Topics

"Save and Restore Operating Points for Stateflow Charts"

“Use Operating Points to Specify Initial State of Simulation”
“Test Difficult-to-Reproduce Chart Configurations”
“Test Chart with Fault Detection and Redundant Logic”

Stateflow.op.OperatingPointData

Operating point information for chart data

Description

A `Stateflow.op.OperatingPointData` object contains a snapshot of a data object in a Stateflow chart during simulation.

Creation

When you save the final operating point for a Simulink model, as described in "Save Operating Points", you create a `Stateflow.op.BlockOperatingPoint` object for each Stateflow chart in the model. This object contains a `Stateflow.op.OperatingPointData` object for each:

- Chart output data
- Chart, state, and function local data
- Persistent variable in a MATLAB function or truth table

Access a `Stateflow.op.OperatingPointData` object by using the property that matches the name of the data in the parent `Stateflow.op.BlockOperatingPoint` or `Stateflow.op.OperatingPointContainer`. For example, suppose that `op` is the `Stateflow.op.BlockOperatingPoint` object for a chart. To access the `Stateflow.op.OperatingPointData` object for a chart output called `output`, enter:

```
op.output
```

Similarly, to access the `Stateflow.op.OperatingPointData` object for a persistent variable called `persistentVar` in the MATLAB function `function`, enter:

```
op.function.persistentVar
```

Properties

`Description` — Description of saved operating point
character vector

This property is read-only.

Description of the saved operating point, specified as a character vector.

Data Types: `char`

`DataType` — Type of data
character vector

This property is read-only.

Type of data, specified as a character vector. For more information, see "Type".

Data Types: `char`

`Size` — Size of data
character vector

This property is read-only.

Size of the data, specified as a character vector. For more information, see “Size”.

Data Types: `char`

`Range` — Range of acceptable values for data
structure

This property is read-only.

Range of acceptable values for the data, specified as a structure with fields `Minimum` and `Maximum`. For more information, see “Limit range”.

Data Types: `struct`

`InitialValue` — Initial value of data
any data type

This property is read-only.

Initial value of data, specified as a value of the type determined by `DataType`. For more information, see “Initial value”.

`Value` — Value of data
any data type

Value of data, specified as a value of the type determined by `DataType`.

Object Functions

open   Display object in editing environment

Examples

Modify Operating Point Information for Output Data

1. Open the model `old_sf_car`.

   ```
   openExample("stateflow/AutomaticTransmissionLegacyExample")
   ```
2. Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   1. Select **Final states** and enter a name for the operating point. For this example, use `xSteadyState`.
   2. Select **Save final operating point**.
   3. Click **OK**.
3. Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to `10`.
4. Run the simulation.

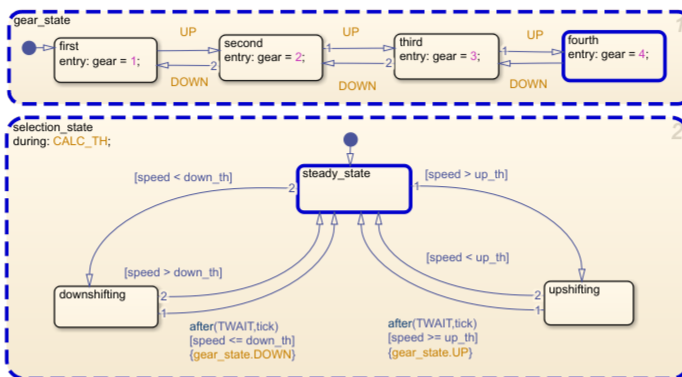
5 Access the `Stateflow.op.BlockOperatingPoint` object that contains the operating point information for the `shift_logic` chart.

```
blockpath = "old_sf_car/shift_logic";
op = get(xSteadyState,blockpath)
```

```
op = 

Block:    "shift_logic"    (handle)    (active)
  Path:     old_sf_car/shift_logic

  Contains:

    + gear_state         "State (AND)"          (active)
    + selection_state    "State (AND)"          (active)
      gear               "Block output data"    double [1, 1]
```

6 Highlight the states that are active in the chart at $t = 10$.

```
highlightActiveStates(op)
```



7 Access the `Stateflow.op.OperatingPointData` object that contains the operating point information for the chart output `gear`.

```
op.gear
```

```
ans = 

      Description: 'Block output data'
         DataType: 'double'
             Size: '[1, 1]'
            Range: [1x1 struct]
     InitialValue: [1x0 double]
            Value: 4
```

8 Change the value of `gear` to `1`.

```
op.gear.Value = 1;
```

9 Inspect the modified operating point information for the chart output `gear`.

```
op.gear
```

```
ans = 

      Description: 'Block output data'
```

```
     DataType: 'double'
         Size: '[1, 1]'
        Range: [1x1 struct]
 InitialValue: [1x0 double]
        Value: 1
```

Version History

Introduced in R2009b

R2019a: Renamed from `Stateflow.SimState.SimStateData`

Behavior changed in R2019a

`Stateflow.SimState.SimStateData` is now called `Stateflow.op.OperationPointData`. The behavior remains the same.

See Also

Objects

`Stateflow.op.BlockOperatingPoint` | `Stateflow.op.OperatingPointContainer` | `Simulink.op.ModelOperatingPoint`

Topics

"Save and Restore Operating Points for Stateflow Charts"
"Use Operating Points to Specify Initial State of Simulation"
"Test Difficult-to-Reproduce Chart Configurations"
"Test Chart with Fault Detection and Redundant Logic"

Stateflow.SimulationData.Data

Data values during simulation

Description

Use `Stateflow.SimulationData.Data` to log the values of local and output data during simulation.

Creation

1. In the **Symbols** pane, select a local or output data object.
2. In the **Property Inspector**, under **Logging**, select the **Log signal data** check box.

Properties

Name — Logging name of data object
character array

Logging name of the data object, specified as a character array. By default, the logging name for a data object is the name of the data object. To assign another name to the data object, in the **Property Inspector**, under **Logging Name**, select `Custom` and enter a custom logging name.

Data Types: `char`

`BlockPath` — Block path for source block
`Simulink.SimulationData.BlockPath`

Block path for the source block, specified as a `Simulink.SimulationData.BlockPath` object.

Data Types: `Simulink.SimulationData.BlockPath`

`Values` — Logged data and time
`timeseries`

Logged data and time, specified as a `timeseries` object.

Data Types: `timeseries`

Object Functions

plot   Plot simulation results in Simulation Data Inspector

Examples

Access Logged Data

1. Open the `sf_semantics_hotel_checkin` model.

```
openExample("stateflow/SemanticsHotelCheckinExample")
```

For more information about this example, see “How Stateflow Objects Interact During Execution”.

2. Open the `Hotel` chart.
3. Open the **Symbols** pane. In the **Simulation** tab, in **Prepare**, click **Symbols Pane**.
4. Open the **Property Inspector**. In the **Simulation** tab, in **Prepare**, click **Property Inspector**.
5. Configure the `service` local data for logging.

   - In the **Symbols** pane, select `service`.
   - In the **Property Inspector**, on the **Logging** tab, select the **Log signal data** check box.
6. Return to the Simulink model.
7. Simulate the model. After starting the simulation, check into the hotel by toggling the first switch twice and order room service by toggling the second switch multiple times. During simulation, Stateflow saves logged data in a `Simulink.SimulationData.Dataset` signal logging object. The default name of the signal logging object is `logsout`. For more information, see “Save Signal Data Using Signal Logging” (Simulink).
8. Stop the simulation.
9. To access the signal logging object, at the MATLAB command prompt, enter:

```
logsout
```

```
logsout =

Simulink.SimulationData.Dataset 'logsout' with 1 element

                        Name         BlockPath
                        ___________  ________________________________
    1  [1x1 Data ]      service      sf_semantics_hotel_checkin/Hotel
```

10. To access logged element, use the `get` method.

```
serviceLog = logsout.get("service")
```

```
serviceLog =

  Stateflow.SimulationData.Data
  Package: Stateflow.SimulationData

  Properties:
         Name: 'service'
    BlockPath: [1×1 Simulink.SimulationData.BlockPath]
       Values: [1×1 timeseries]
```

11. To access the logged data and time of each logged element, use the `Values.Data` and `Values.Time` properties. For example, arrange logged data in tabular form by using the `table` function.

```
T = table(serviceLog.Values.Time,serviceLog.Values.Data);
T.Properties.VariableNames = ["Time" "Data"]
```

```
T =

  6×2 table

       Time       Data
    __________    ____

             0     0
    1.7076e+06     0
    1.8607e+06     1
```

```
    1.9653e+06     2
    1.9653e+06     3
    1.9653e+06     4
    2.2912e+06     5
```

In this example, the data points with values of `0` correspond to when the chart initializes the local data `service` to `0` at time `0` and when a default transition sets `service` to `0` at time `1.7076e+06`.

Tips

- `Stateflow.SimulationData.Data` objects record a data point every time that the Stateflow chart writes to the data you are logging, even if the data does not change value. For example, in “Access Logged Data” on page 4-17, the data points with values of `0` correspond to when the chart initializes the local data `service` to `0` at time `0` and when a default transition sets `service` to `0` at time `1.7076e+06`.

Version History

Introduced in R2017b

See Also

`Stateflow.SimulationData.State` | `Simulink.SimulationData.BlockPath` | `timeseries` | `plot`

Topics

“Log Simulation Output for States and Data”
“Save Signal Data Using Signal Logging” (Simulink)

Stateflow.SimulationData.State

State activity during simulation

Description

Use `Stateflow.SimulationData.State` to log the activity of a state during simulation.

Creation

1. In the Stateflow Editor, select a state.
2. In the **Simulation** tab, in **Prepare**, select **Log Self Activity**. Alternatively, in the **Property Inspector**,under **Logging**, select the **Log self activity** check box.

Properties

Name — Logging name of state
character array

Logging name of the state, specified as a character array. By default, the logging name for a state is the hierarchical name using a period (`.`) to separate each level in the hierarchy of states. To assign a shorter name to the state, in the **Property Inspector**, set **Logging Name** to `Custom` and enter a custom logging name.

Data Types: `char`

`BlockPath` — Block path for source block
`Simulink.SimulationData.BlockPath`

Block path for the source block, specified as a `Simulink.SimulationData.BlockPath` object.

Data Types: `Simulink.SimulationData.BlockPath`

`Values` — State activity
`timeseries`

State activity, specified as a `timeseries` object. Data values represent whether the state is active (`1`) or not active (`0`). Time values correspond to simulation time.

Data Types: `timeseries`

Object Functions

plot Plot simulation results in Simulation Data Inspector

Examples

Access Logged State Activity

1. Open the `sf_semantics_hotel_checkin` model.

   ```
   openExample("stateflow/SemanticsHotelCheckinExample")
   ```

   For more information about this example, see "How Stateflow Objects Interact During Execution".
2. Open the `Hotel` chart.
3. Open the **Symbols** pane. In the **Simulation** tab, in **Prepare**, click **Symbols Pane**.
4. Configure the `Dining_area` state for logging.

   - In the Stateflow Editor, select the `Dining_area` state.
   - In the **Simulation** tab, under **Prepare**, select **Log Self Activity**.

     In the **Property Inspector**, under **Logging**, select the **Log self activity** check box.
   - By default, the logging name for this state is the hierarchical signal name `Check_in.Checked_in.Executive_suite.Dining_area`. To assign a shorter name to the state, set **Logging Name** to `Custom` and enter `Dining Room`.
5. Return to the Simulink model.
6. Simulate the model. After starting the simulation, check into the hotel by toggling the first switch twice and order room service by toggling the second switch multiple times. During simulation, Stateflow saves logged data in a `Simulink.SimulationData.Dataset` signal logging object. The default name of the signal logging object is `logsout`. For more information, see "Save Signal Data Using Signal Logging" (Simulink).
7. Stop the simulation.
8. To access the signal logging object, at the MATLAB command prompt, enter:

   ```
   logsout
   ```

   ```
   logsout = 

   Simulink.SimulationData.Dataset 'logsout' with 1 element

                           Name         BlockPath                        
                           ___________  ________________________________ 
       1  [1x1 State]      Dining Room  sf_semantics_hotel_checkin/Hotel
   ```
9. To access logged elements, use the `get` method.

   ```
   diningLog = logsout.get("Dining Room")
   ```

   ```
   diningLog = 

     Stateflow.SimulationData.State
     Package: Stateflow.SimulationData

     Properties:
            Name: 'Dining Room'
       BlockPath: [1×1 Simulink.SimulationData.BlockPath]
          Values: [1×1 timeseries]
   ```
10. To access the logged data and time of each logged element, use the `Values.Data` and `Values.Time` properties. For example, arrange logged data in tabular form by using the `table` function.

    ```
    T = table(diningLog.Values.Time,diningLog.Values.Data);
    T.Properties.VariableNames = ["Time" "Data"]
    ```

```
T =

  6×2 table

        Time       Data
    ___________    ____

              0     0
     1.8607e+06     1
     1.9653e+06     0
     1.9653e+06     1
     1.9653e+06     0
     2.2912e+06     1
```

Version History

Introduced in R2017b

See Also

`Stateflow.SimulationData.Data` | `Simulink.SimulationData.BlockPath` | `timeseries` | `plot`

Topics

"Log Simulation Output for States and Data"
"Save Signal Data Using Signal Logging" (Simulink)

Object Functions

clone

Package: `Stateflow.op`

Copy operating point for Stateflow chart

Syntax

`newOp = clone(op)`

Description

`newOp = clone(op)` creates a copy of the operating point `op` for a Stateflow chart.

Examples

Copy Operating Point

1. Open the `sf_car` model.

   ```
   openExample("stateflow/AutomaticTransmissionWithActiveStateDataExample")
   ```

   For more information about this model, see “Simulate Chart as a Simulink Block With Local Events”.
2. Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   1. Select **Final states** and enter a name for the operating point. For this example, use `xFinal`.
   2. Select **Save final operating point**.
   3. Click **OK**.
3. Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to `10`.
4. Run the simulation.
5. Access the `Stateflow.op.BlockOperatingPoint` object that contains the operating point information for the `shift_logic` chart.

   ```
   blockpath = "sf_car/shift_logic";
   op = get(xFinal,blockpath);
   ```
6. Access the `Stateflow.op.OperatingPointContainer` object that contains the operating point information for the `gear_state` state.

   ```
   op.gear_state
   ```

   ```
   ans =

   State: "gear_state"    (handle)    (active)
     Path:     sf_car/shift_logic/gear_state

     Contains:

       + first          "State (OR)"
       + fourth         "State (OR)"
   ```

```
    + second         "State (OR)"         (active)
    + third          "State (OR)"
```

The operating point shows that the substate `second` is active.

7. Create a copy of the operating point.

   ```
   newOp = clone(op);
   ```
8. Modify the new operating point by changing the active substate of `gear_state`.

   ```
   setActive(newOp.gear_state.first)
   ```
9. Verify that the substate `first` is active in the modified operating point.

   ```
   newOp.gear_state

   ans =

   State: "gear_state"    (handle)    (active)
     Path:     sf_car/shift_logic/gear_state

     Contains:

       + first          "State (OR)"         (active)
       + fourth         "State (OR)"
       + second         "State (OR)"
       + third          "State (OR)"
   ```
10. Verify that the substate `second` is active in the original operating point.

   ```
   op.gear_state

   ans =

   State: "gear_state"    (handle)    (active)
     Path:     sf_car/shift_logic/gear_state

     Contains:

       + first          "State (OR)"
       + fourth         "State (OR)"
       + second         "State (OR)"         (active)
       + third          "State (OR)"
   ```

Input Arguments

op — Operating point for chart
`Stateflow.op.BlockOperatingPoint` object

Operating point for a Stateflow chart, specified as a `Stateflow.op.BlockOperatingPoint` object.

Output Arguments

newOp — Copy of operating point
`Stateflow.op.BlockOperatingPoint` object

Copy of operating point, returned as a `Stateflow.op.BlockOperatingPoint` object.

Version History

Introduced in R2009b

See Also

Objects

`Stateflow.op.BlockOperatingPoint`

Topics

“Save and Restore Operating Points for Stateflow Charts”

getPrevActiveChild

Package: `Stateflow.op`

Get previously active substate

Syntax

```
substateOp = getPrevActiveChild(stateOp)
```

Description

`substateOp = getPrevActiveChild(stateOp)` returns the operating point for the previously active substate in the operating point `stateOp`. `stateOp` must be an operating point for a state that contains a history junction.

Examples

Modify Previously Active Child

1. Open the `sf_boiler` model.

   ```
   openExample("stateflow/BangBangControlUsingTemporalLogicExample")
   ```

   For more information about this model, see "Model Bang-Bang Temperature Control System".
2. Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   1. Select **Final states** and enter a name for the operating point. For this example, use `xFinal`.
   2. Select **Save final operating point**.
   3. Click **OK**.
3. Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to `100`.
4. Run the simulation.
5. Access the `Stateflow.op.BlockOperatingPoint` object that contains the operating point information for the `Bang-Bang Controller` chart.

   ```
   blockpath = "sf_boiler/Bang-Bang Controller";
   op = get(xFinal,blockpath);
   ```
6. Verify that the state `On` in the box `Heater` is not active.

   ```
   isActive(op.Heater.On)
   ```

   ```
   ans =

     logical

      0
   ```
7. Find the previously active substate of state `On`.

   ```
   getPrevActiveChild(op.Heater.On)
   ```

```
ans = 

State: "HIGH"    (handle)
  Path:     sf_boiler/Bang-Bang Controller/Heater/On/HIGH

  Contains:

    []
```

8. Modify the previously active substate of state `On`. Specify the substate as a `Stateflow.op.OperatingPointContainer` object.

   ```
   setPrevActiveChild(op.Heater.On,op.Heater.On.NORM)
   ```

   Alternatively, specify the name of the substate by using a string scalar or a character vector.

   ```
   setPrevActiveChild(op.Heater.On,"NORM")
   ```

9. Verify that the substate `NORM` is the previously active substate in the modified operating point.

   ```
   getPrevActiveChild(op.Heater.On)
   ```

   ```
   ans = 

   State: "NORM"    (handle)
     Path:     sf_boiler/Bang-Bang Controller/Heater/On/NORM

     Contains:

       []
   ```

Input Arguments

`stateOp` — Operating point for state
`Stateflow.op.OperatingPointContainer` object

Operating point for a state that contains a history junction, specified as a `Stateflow.op.OperatingPointContainer` object.

Output Arguments

`substateOp` — Operating point for substate
`Stateflow.op.OperatingPointContainer` object

Operating point for the previously active substate, returned as a `Stateflow.op.OperatingPointContainer` object.

Version History

Introduced in R2009b

See Also

Objects
`Stateflow.op.OperatingPointContainer`

Functions
`setPrevActiveChild` | `isActive`

Topics
"Save and Restore Operating Points for Stateflow Charts"

highlightActiveStates

Package: Stateflow.op

Highlight active states

Syntax

highlightActiveStates(op)

Description

highlightActiveStates(op) highlights the active states in the Stateflow Editor for the operating point op.

Examples

Modify Operating Point Information for State Activity

1. Open the sf_aircraft model.

   ```
   openExample("stateflow/FaultDetectionControlLogicInAnAircraftControlSystemExample")
   ```

   For more information about this model, see "Detect Faults in Aircraft Elevator Control System".
2. Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   1. Select **Final states** and enter a name for the operating point. For this example, use xSteadyState.
   2. Select **Save final operating point**.
   3. Click **OK**.
3. Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to 3.
4. Run the simulation.
5. Access the Stateflow.op.BlockOperatingPoint object that contains the operating point information for the Mode Logic chart.

   ```
   blockpath = "sf_aircraft/Mode Logic";
   op = get(xSteadyState,blockpath)
   ```

   ```
   op = 

     Block:    "Mode Logic"    (handle)    (active)
     Path:     sf_aircraft/Mode Logic

     Contains:

       + Actuators          "State (OR)"                    (active)
       + LI_act             "Function"
       + LO_act             "Function"
       + L_switch           "Function"
       + RI_act             "Function"
   ```

```
    + RO_act            "Function"
    + R_switch          "Function"
      LI_mode           "State output data"       sf_aircraft_ModeType [1,1]
      LO_mode           "State output data"       sf_aircraft_ModeType [1,1]
      RI_mode           "State output data"       sf_aircraft_ModeType [1,1]
      RO_mode           "State output data"       sf_aircraft_ModeType [1,1]
```

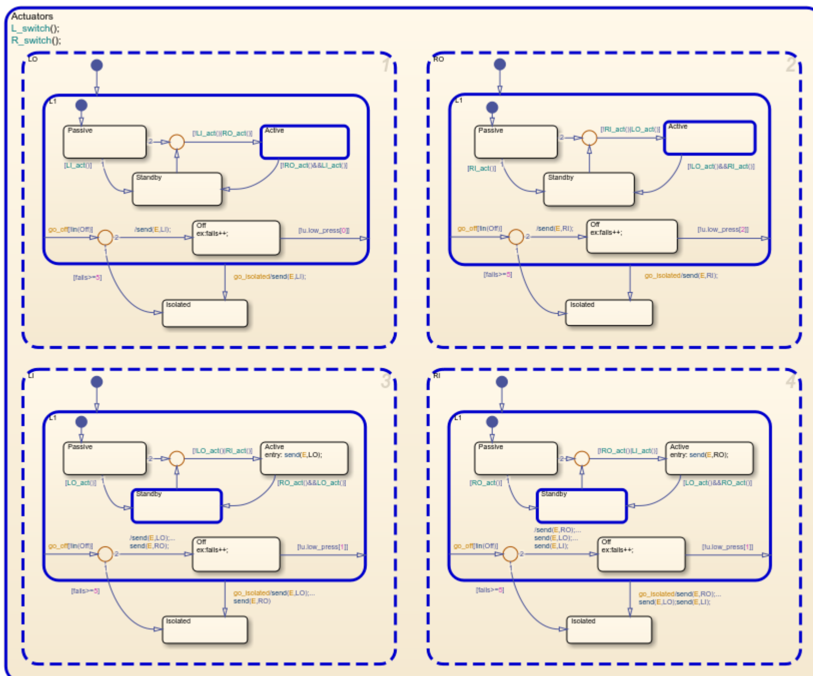
6 Access the `Stateflow.op.OperatingPointContainer` object that contains the operating point information for the `Actuators` state.

```
op.Actuators
```

```
ans = 

  State:    "Actuators"    (handle)    (active)
  Path:     sf_aircraft/Mode Logic/Actuators

  Contains:

    + LI            "State (AND)"        (active)
    + LO            "State (AND)"        (active)
    + RI            "State (AND)"        (active)
    + RO            "State (AND)"        (active)
```

7 Highlight the states that are active in the chart at $t = 3$.

```
highlightActiveStates(op)
```



8 Change the substate activity in the state `LO` to reflect a failure of the left-outer actuator.

```
setActive(op.Actuators.LO.Isolated)
```

9 Verify that the substate `Isolated` in the state `LO` is active in the modified operating point.

```
isActive(op.Actuators.LO.Isolated)
```

```
ans =

  logical

   1
```

10 Remove the highlighting of active states in the Stateflow Editor.

```
removeHighlighting(op)
```

Input Arguments

op — Operating point for chart
`Stateflow.op.BlockOperatingPoint` object

Operating point for a Stateflow chart, specified as a `Stateflow.op.BlockOperatingPoint` object.

Tips

- To clear the highlighting, use the `removeHighlighting` object function. Alternatively, in the **Debug** tab, under **Animation**, click the Remove animation highlighting button.

Version History
Introduced in R2009b

See Also

Objects
`Stateflow.op.BlockOperatingPoint`

Functions
`removeHighlighting` | `setActive` | `isActive`

Topics
"Save and Restore Operating Points for Stateflow Charts"

isActive

Package: Stateflow.op

Determine if state is active

Syntax

```
tf = isActive(stateOp)
```

Description

tf = isActive(stateOp) returns logical 1 (true) if stateOp is the operating point for a state that is active. Otherwise, the function returns logical 0 (false).

Examples

Modify Operating Point Information for State Activity

1. Open the sf_aircraft model.

   ```
   openExample("stateflow/FaultDetectionControlLogicInAnAircraftControlSystemExample")
   ```

   For more information about this model, see "Detect Faults in Aircraft Elevator Control System".
2. Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   1. Select **Final states** and enter a name for the operating point. For this example, use xSteadyState.
   2. Select **Save final operating point**.
   3. Click **OK**.
3. Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to 3.
4. Run the simulation.
5. Access the Stateflow.op.BlockOperatingPoint object that contains the operating point information for the Mode Logic chart.

   ```
   blockpath = "sf_aircraft/Mode Logic";
   op = get(xSteadyState,blockpath)
   ```

   ```
   op =

     Block:    "Mode Logic"    (handle)    (active)
     Path:     sf_aircraft/Mode Logic

     Contains:

       + Actuators          "State (OR)"                    (active)
       + LI_act             "Function"
       + LO_act             "Function"
       + L_switch           "Function"
       + RI_act             "Function"
   ```

```
    + RO_act            "Function"
    + R_switch          "Function"
      LI_mode           "State output data"       sf_aircraft_ModeType [1,1]
      LO_mode           "State output data"       sf_aircraft_ModeType [1,1]
      RI_mode           "State output data"       sf_aircraft_ModeType [1,1]
      RO_mode           "State output data"       sf_aircraft_ModeType [1,1]
```

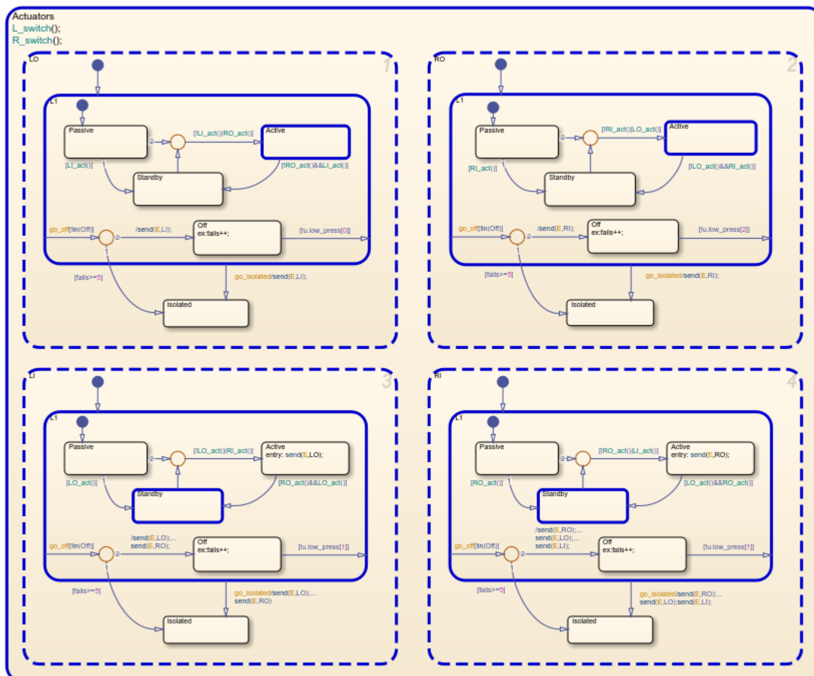
6 Access the `Stateflow.op.OperatingPointContainer` object that contains the operating point information for the `Actuators` state.

```
op.Actuators
```

```
ans = 

  State:     "Actuators"     (handle)     (active)
  Path:      sf_aircraft/Mode Logic/Actuators

  Contains:

    + LI            "State (AND)"         (active)
    + LO            "State (AND)"         (active)
    + RI            "State (AND)"         (active)
    + RO            "State (AND)"         (active)
```

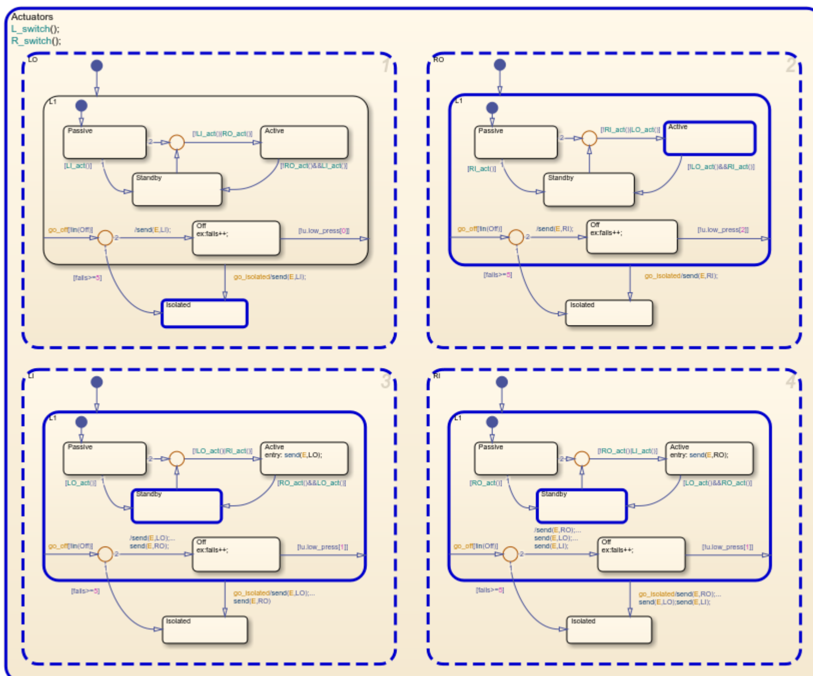
7 Highlight the states that are active in the chart at $t = 3$.

```
highlightActiveStates(op)
```



8 Change the substate activity in the state `LO` to reflect a failure of the left-outer actuator.

```
setActive(op.Actuators.LO.Isolated)
```



9 Verify that the substate `Isolated` in the state `LO` is active in the modified operating point.

```
isActive(op.Actuators.LO.Isolated)
```

```
ans =

  logical

   1
```

10 Remove the highlighting of active states in the Stateflow Editor.

```
removeHighlighting(op)
```

Input Arguments

`stateOp` — Operating point for state
`Stateflow.op.OperatingPointContainer` object

Operating point for a state, specified as a `Stateflow.op.OperatingPointContainer` object.

Version History

Introduced in R2009b

See Also

Objects
`Stateflow.op.OperatingPointContainer`

Functions
`setActive` | `highlightActiveStates` | `removeHighlighting`

Topics

“Save and Restore Operating Points for Stateflow Charts”
“Test Chart with Fault Detection and Redundant Logic”

open

Package: `Stateflow.op`

Display object in editing environment

Syntax

`open(op)`

Description

`open(op)` displays the object that corresponds to the operating point `op` in its editing environment. For example, charts, states, and boxes appear in the Stateflow Editor. Data appear in the Model Explorer. For more information, see `view`.

Examples

Display State in Chart

Suppose that `xFinal` is the operating point for a Stateflow chart that contains a top-level state `state` with a substate called `substate`.

Zoom in on and select the substate in the Stateflow Editor.

```
open(op.state.substate)
```

Input Arguments

op — Operating point
`Stateflow.op.BlockOperatingPoint` object | `Stateflow.op.OperatingPointContainer` object | `Stateflow.op.OperatingPointData` object

Operating point, specified as a `Stateflow.op.BlockOperatingPoint`, `Stateflow.op.OperatingPointContainer`, or `Stateflow.op.OperatingPointData` object.

Version History

Introduced in R2009b

See Also

Objects
`Stateflow.op.BlockOperatingPoint` | `Stateflow.op.OperatingPointContainer` | `Stateflow.op.OperatingPointData`

Functions
`view`

Topics

“Save and Restore Operating Points for Stateflow Charts”

removeHighlighting

Package: `Stateflow.op`

Remove highlighting of active states

Syntax

`removeHighlighting(op)`

Description

`removeHighlighting(op)` removes the highlighting of active states in the Stateflow Editor.

Examples

Modify Operating Point Information for State Activity

1. Open the `sf_aircraft` model.

   ```
   openExample("stateflow/FaultDetectionControlLogicInAnAircraftControlSystemExample")
   ```

   For more information about this model, see "Detect Faults in Aircraft Elevator Control System".
2. Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   1. Select **Final states** and enter a name for the operating point. For this example, use `xSteadyState`.
   2. Select **Save final operating point**.
   3. Click **OK**.
3. Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to `3`.
4. Run the simulation.
5. Access the `Stateflow.op.BlockOperatingPoint` object that contains the operating point information for the `Mode Logic` chart.

   ```
   blockpath = "sf_aircraft/Mode Logic";
   op = get(xSteadyState,blockpath)
   ```

   ```
   op = 

     Block:    "Mode Logic"    (handle)    (active)
     Path:     sf_aircraft/Mode Logic

     Contains:

       + Actuators          "State (OR)"                    (active)
       + LI_act             "Function"
       + LO_act             "Function"
       + L_switch           "Function"
       + RI_act             "Function"
       + RO_act             "Function"
   ```

```
    + R_switch          "Function"
      LI_mode           "State output data"        sf_aircraft_ModeType [1,1]
      LO_mode           "State output data"        sf_aircraft_ModeType [1,1]
      RI_mode           "State output data"        sf_aircraft_ModeType [1,1]
      RO_mode           "State output data"        sf_aircraft_ModeType [1,1]
```

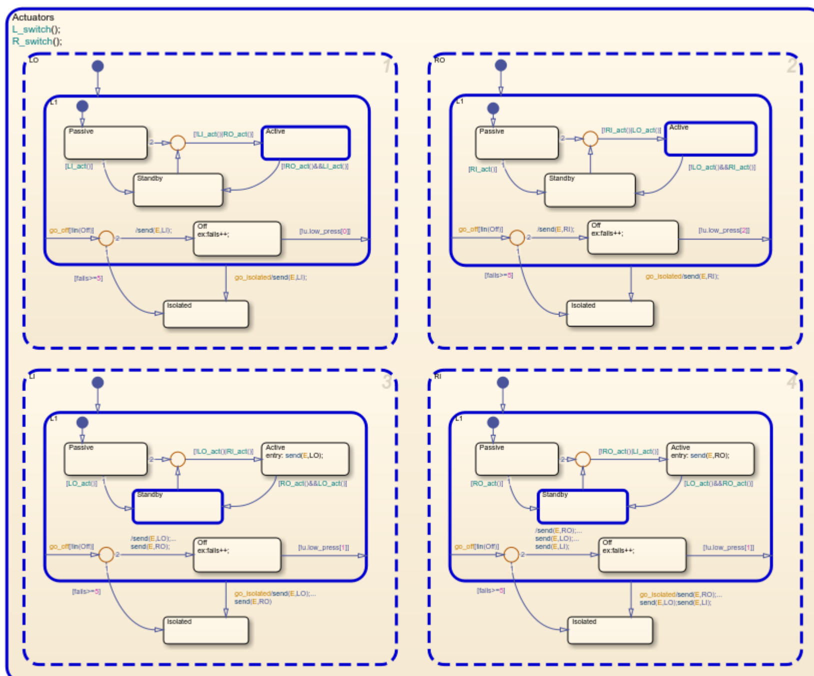
6 Access the `Stateflow.op.OperatingPointContainer` object that contains the operating point information for the `Actuators` state.

```
op.Actuators
```

```
ans =

  State:    "Actuators"    (handle)    (active)
  Path:     sf_aircraft/Mode Logic/Actuators

  Contains:

    + LI            "State (AND)"         (active)
    + LO            "State (AND)"         (active)
    + RI            "State (AND)"         (active)
    + RO            "State (AND)"         (active)
```

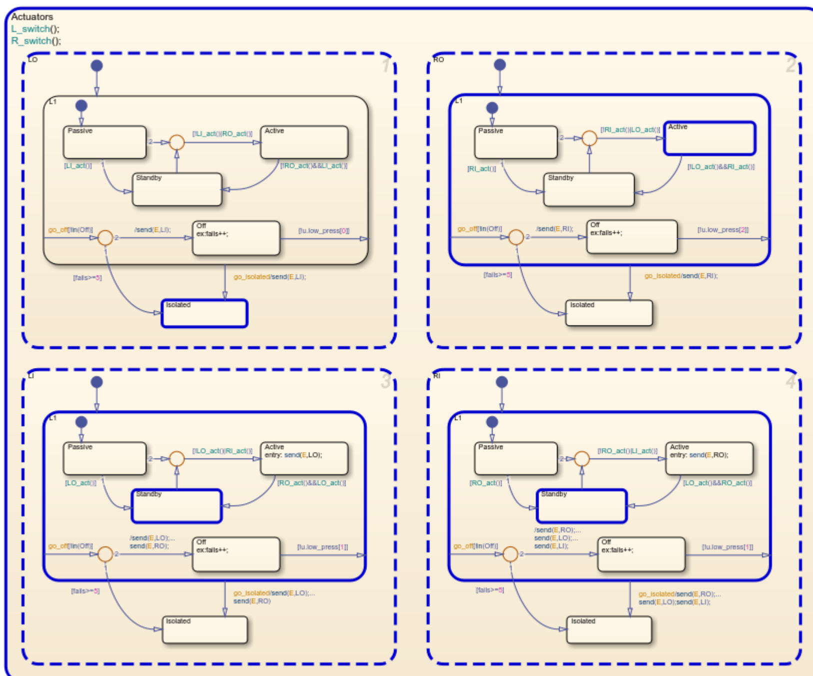
7 Highlight the states that are active in the chart at $t = 3$.

```
highlightActiveStates(op)
```



8 Change the substate activity in the state `LO` to reflect a failure of the left-outer actuator.

```
setActive(op.Actuators.LO.Isolated)
```



9 Verify that the substate `Isolated` in the state `LO` is active in the modified operating point.

```
isActive(op.Actuators.LO.Isolated)
```

```
ans =

  logical

   1
```

10 Remove the highlighting of active states in the Stateflow Editor.

```
removeHighlighting(op)
```

Input Arguments

op — Operating point for chart
`Stateflow.op.BlockOperatingPoint` object

Operating point for a Stateflow chart, specified as a `Stateflow.op.BlockOperatingPoint` object.

Version History

Introduced in R2009b

See Also

Objects
`Stateflow.op.BlockOperatingPoint`

Functions
`highlightActiveStates` | `setActive` | `isActive`

Topics
"Save and Restore Operating Points for Stateflow Charts"

setActive

Package: `Stateflow.op`

Set state as active

Syntax

`setActive(stateOp)`

Description

`setActive(stateOp)` sets the state that corresponds to the operating point `stateOp` as active. `stateOp` must be an operating point for a leaf state. When you call `setActive`, the chart maintains state consistency by:

- Exiting and entering the appropriate states
- Resetting temporal counters for newly active states
- Updating values of active state data
- Enabling or disabling function-call subsystems and Simulink functions that bind to states

However, the chart does not perform `exit` actions for the previously active states or `entry` actions for the newly active state. Additionally, the state does not reinitialize any state-parented local data.

Examples

Modify Operating Point Information for State Activity

1. Open the `sf_aircraft` model.

   ```
   openExample("stateflow/FaultDetectionControlLogicInAnAircraftControlSystemExample")
   ```

   For more information about this model, see "Detect Faults in Aircraft Elevator Control System".
2. Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   1. Select **Final states** and enter a name for the operating point. For this example, use `xSteadyState`.
   2. Select **Save final operating point**.
   3. Click **OK**.
3. Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to `3`.
4. Run the simulation.
5. Access the `Stateflow.op.BlockOperatingPoint` object that contains the operating point information for the `Mode Logic` chart.

   ```
   blockpath = "sf_aircraft/Mode Logic";
   op = get(xSteadyState,blockpath)
   ```

   ```
   op =
   ```

```
  Block:    "Mode Logic"    (handle)    (active)
  Path:     sf_aircraft/Mode Logic

  Contains:

    + Actuators          "State (OR)"                    (active)
    + LI_act             "Function"
    + LO_act             "Function"
    + L_switch           "Function"
    + RI_act             "Function"
    + RO_act             "Function"
    + R_switch           "Function"
      LI_mode            "State output data"        sf_aircraft_ModeType [1,1]
      LO_mode            "State output data"        sf_aircraft_ModeType [1,1]
      RI_mode            "State output data"        sf_aircraft_ModeType [1,1]
      RO_mode            "State output data"        sf_aircraft_ModeType [1,1]
```

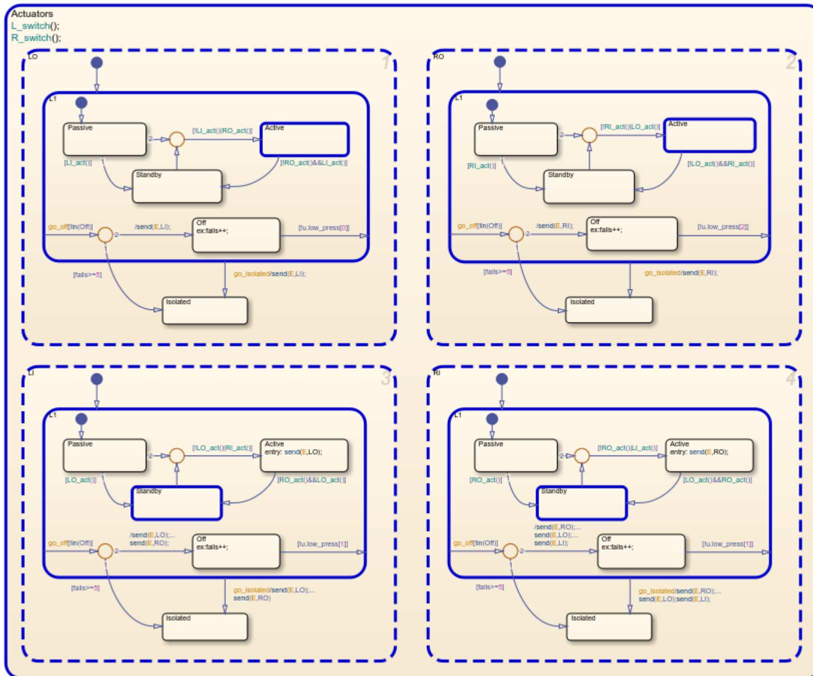
6 Access the `Stateflow.op.OperatingPointContainer` object that contains the operating point information for the `Actuators` state.

```
op.Actuators
```

```
ans =

  State:    "Actuators"    (handle)    (active)
  Path:     sf_aircraft/Mode Logic/Actuators

  Contains:

    + LI           "State (AND)"         (active)
    + LO           "State (AND)"         (active)
    + RI           "State (AND)"         (active)
    + RO           "State (AND)"         (active)
```

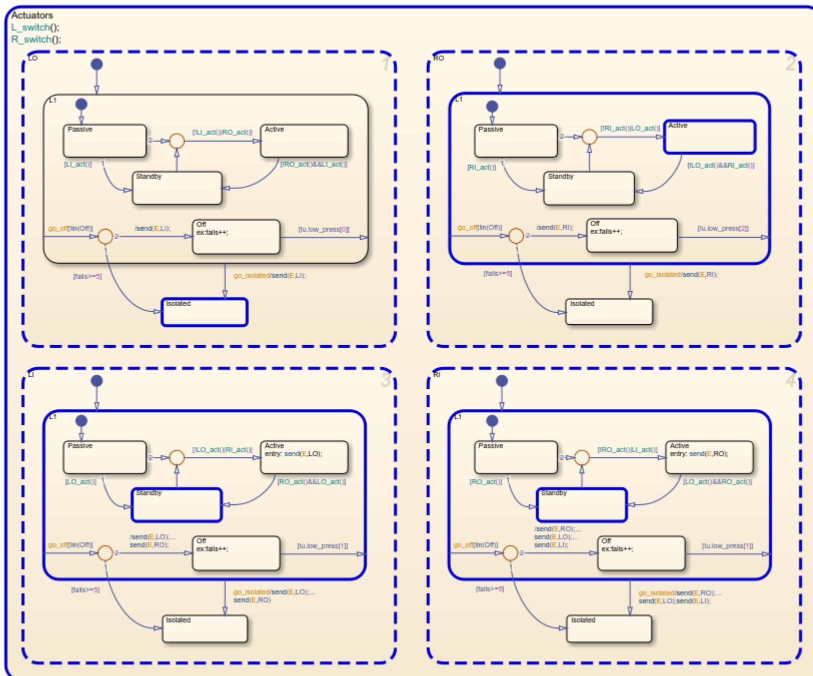
7 Highlight the states that are active in the chart at $t = 3$.

```
highlightActiveStates(op)
```



8 Change the substate activity in the state `LO` to reflect a failure of the left-outer actuator.

```
setActive(op.Actuators.LO.Isolated)
```



9 Verify that the substate `Isolated` in the state `LO` is active in the modified operating point.

```
isActive(op.Actuators.LO.Isolated)
```

```
ans =
```

```
logical

   1
```

10 Remove the highlighting of active states in the Stateflow Editor.

```
removeHighlighting(op)
```

Input Arguments

`stateOp` — Operating point for state
`Stateflow.op.OperatingPointContainer` object

Operating point for a leaf state, specified as a `Stateflow.op.OperatingPointContainer` object.

Version History

Introduced in R2009b

See Also

Objects
`Stateflow.op.OperatingPointContainer`

Functions
`isActive` | `highlightActiveStates` | `removeHighlighting`

Topics
"Save and Restore Operating Points for Stateflow Charts"
"Test Chart with Fault Detection and Redundant Logic"

setPrevActiveChild

Package: `Stateflow.op`

Set previously active substate

Syntax

```
setPrevActiveChild(stateOp,substateOp)
setPrevActiveChild(stateOp,substateName)
```

Description

`setPrevActiveChild(stateOp,substateOp)` sets the state that corresponds to the operating point `substateOp` as the previously active substate in the operating point `stateOp`. `stateOp` must be an operating point for an inactive state that contains a history junction.

`setPrevActiveChild(stateOp,substateName)` marks the state called `substateName` as the previously active substate in `stateOp`.

Examples

Modify Previously Active Child

1. Open the `sf_boiler` model.

   ```
   openExample("stateflow/BangBangControlUsingTemporalLogicExample")
   ```

   For more information about this model, see "Model Bang-Bang Temperature Control System".
2. Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   1. Select **Final states** and enter a name for the operating point. For this example, use `xFinal`.
   2. Select **Save final operating point**.
   3. Click **OK**.
3. Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to `100`.
4. Run the simulation.
5. Access the `Stateflow.op.BlockOperatingPoint` object that contains the operating point information for the `Bang-Bang Controller` chart.

   ```
   blockpath = "sf_boiler/Bang-Bang Controller";
   op = get(xFinal,blockpath);
   ```
6. Verify that the state `On` in the box `Heater` is not active.

   ```
   isActive(op.Heater.On)
   ```

   ```
   ans =

     logical

      0
   ```

7 Find the previously active substate of state On.

```
getPrevActiveChild(op.Heater.On)
```

```
ans = 

State: "HIGH"    (handle)
  Path:     sf_boiler/Bang-Bang Controller/Heater/On/HIGH

  Contains:

    []
```

8 Modify the previously active substate of state On. Specify the substate as a `Stateflow.op.OperatingPointContainer` object.

```
setPrevActiveChild(op.Heater.On,op.Heater.On.NORM)
```

Alternatively, specify the name of the substate by using a string scalar or a character vector.

```
setPrevActiveChild(op.Heater.On,"NORM")
```

9 Verify that the substate NORM is the previously active substate in the modified operating point.

```
getPrevActiveChild(op.Heater.On)
```

```
ans = 

State: "NORM"    (handle)
  Path:     sf_boiler/Bang-Bang Controller/Heater/On/NORM

  Contains:

    []
```

Input Arguments

stateOp — Operating point for state

`Stateflow.op.OperatingPointContainer` object

Operating point for an inactive state that contains a history junction, specified as a `Stateflow.op.OperatingPointContainer` object.

substateOp — Operating point for substate

`Stateflow.op.OperatingPointContainer` object

Operating point for the new previously active substate, specified as a `Stateflow.op.OperatingPointContainer` object.

substateName — Name of substate

string scalar | character vector

Name of the new previously active substate, specified as a string scalar or a character vector.

Version History

Introduced in R2009b

See Also

Objects
`Stateflow.op.OperatingPointContainer`

Functions
`getPrevActiveChild` | `isActive`

Topics
"Save and Restore Operating Points for Stateflow Charts"

6

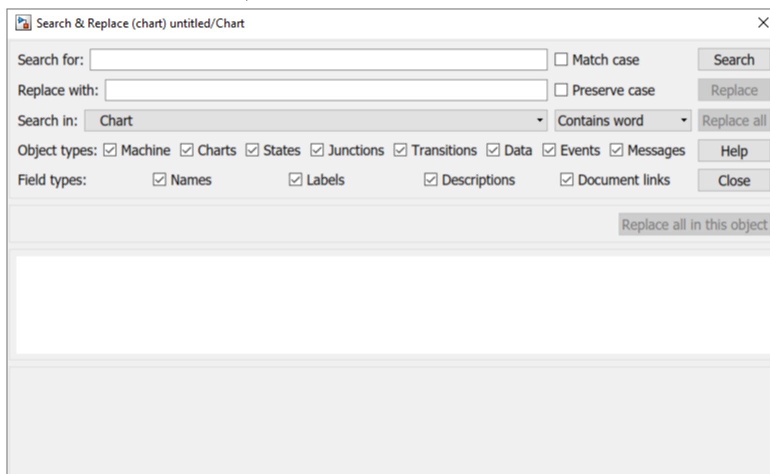
Tools

Search & Replace

Find and replace text in Stateflow charts

Description

Use the **Search & Replace** tool to find and modify text in your Stateflow charts. Search an individual chart or all of the charts in a Simulink model. Modify the scope of your search by enabling case-sensitive searching, matching only whole words, using regular expressions to define search patterns, or filtering by object and field types. Enable case-preserving replacements for lowercase, uppercase, title case, or sentence case text.



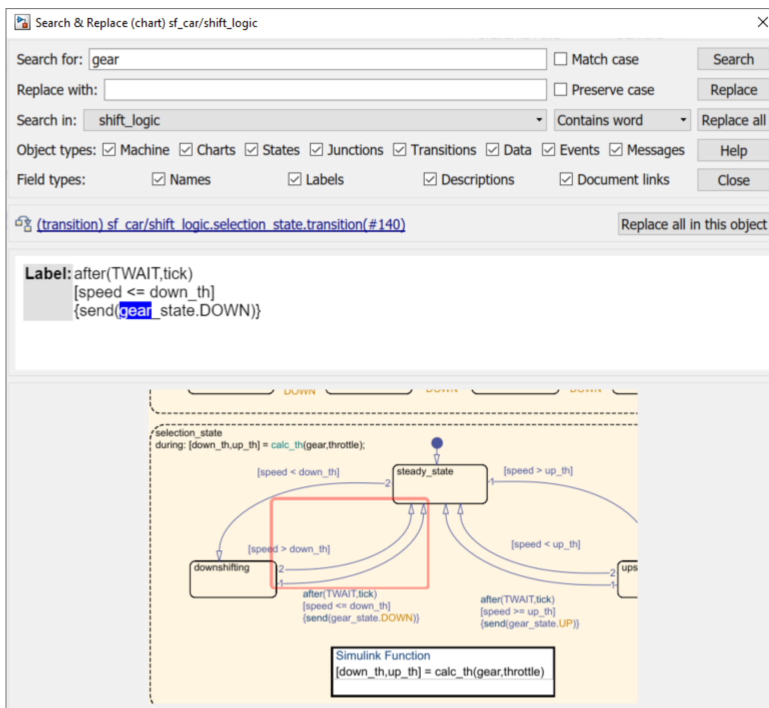
Open the Search & Replace

- Open a Stateflow chart. Then, in the **Modeling** tab, select **Find** > **Find & Replace in Chart**.

Examples

Search for Text

1. In the **Search for** field, enter the text for which to search.
2. Modify the scope of your search results.
   - To enable case-sensitive searching, select **Match case**.
   - To specify the chart or model in which to search, use the **Search in** drop-down list.
   - To match only whole words or to define a search pattern by using a regular expression, use the drop-down list to the right of **Search in** on page 6-0 .
   - To filter your search targets by object and field type, select one or more check boxes under **Object types** or **Field types**.
3. Click **Search**. The Search & Replace tool displays the matching text in the center pane of the tool. If the matching text belongs to a graphical object such as a state or transition, the graphical object appears highlighted in the bottom pane of the tool.



4 To highlight to the object in the Stateflow Editor, double-click the bottom pane.
5 To find the next match, click **Search** again.

Replace Text

1 Search for the text to replace, as described in “Search for Text” on page 6-2.
2 In the **Replace with** field, enter the text with which to replace the text found by your search.
3 To enable case-preserving replacements, select **Preserve case**.
4 Click one of these buttons:

- **Search** — Skip the current search result and search for the next match.
- **Replace** — Replace the current search result with the replacement text and search for the next match.
- **Replace all** — Replace all instances that match the search text. Replacement spans from the current search result to the end of the current Stateflow chart. The Search & Replace tool ignores any matches that you previously skipped by clicking the **Search** button.
- **Replace all in this object** — Replace all instances that match the search text in the current Stateflow object. The Search & Replace tool replaces any matches that you previously skipped by clicking the **Search** button.

Parameters

`Match case` — Case-sensitive searching
`off` (default) | `on`

Select this parameter to enable case-sensitive searching.

- When you select this parameter, the Search & Replace tool finds only text that exactly matches the text in the **Search for** field.
- When you clear this parameter, the Search & Replace tool matches the character sequence in the **Search for** field, regardless of case. For example, the search text `"gear"` matches the text `"gear"`, `"Gear"`, or `"GEAR"`.

`Preserve case` — Case-preserving replacement
`off` (default) | `on`

Select this parameter to enable case-preserving replacements. When you select this parameter, the Search & Replace tool replaces the matching text based on these conditions:

- If the matching text has only lowercase characters, the Search & Replace tool replaces the matching text entirely with the lowercase equivalent of all replacement characters. For example, if the replacement text is `"AnDreW"`, the matching text `"james"` is replaced by `"andrew"`.
- If the matching text has only uppercase characters, the Search & Replace tool replaces the matching text entirely with the uppercase equivalent of all replacement characters. For example, if the replacement text is `"AnDreW"`, the matching text `"JAMES"` is replaced by `"ANDREW"`.
- If the matching text uses title case, with uppercase characters in the first character position of each word, the Search & Replace tool replaces the matching text with the replacement text in title case. For example, if the replacement text is `"AnDreW jAcksOn"`, the matching text `"James Monroe"` is replaced by `"Andrew Jackson"`.
- If the matching text uses sentence case, with an uppercase character in the first character position of a sentence and all other sentence characters in lowercase, the Search & Replace tool replaces the matching text with the replacement text in sentence case. For example, if the replacement text is `"AnDreW is TALL"`, the matching text `"James is tall"` is replaced by `"Andrew is tall"`.

If the matching text does not follow any of these patterns, the Search & Replace tool replaces the matching text using the exact case specified by the replacement text.

`Search in` — Location to search
chart name (default) | model name

Specify the location to search. You can select an individual chart or all of the charts in a loaded Simulink model. By default, the Search & Replace tool searches only the chart in which you opened the tool.

Note The left drop-down list shows the charts in only one model at a time. To select a Stateflow chart in a different model, first select the model. Then open the drop-down list a second time and select the chart.

`Style` — Style of text for which to search
`Contains word` (default) | `Match whole word` | `Regular expression`

Use the drop-down list to the right of the **Search in** parameter to specify one of these options:

- `Contains word` — Search for text in any expression. For example, the search text `"gear"` matches the text `"gear_state"`.

- `Match whole word` — Search for whole word expressions delimited by a blank space or a character that is not alphanumeric or an underscore character. For example, the search text `"gear"` does not match the text `"gear_state"`.
- `Regular expression` — Treat the search text as a regular expression. For example,the search text `"g\w*_"` matches any text that begins with the letter `g` and ends with an underscore. For more information, see “Regular Expressions”.

`Object types` — Type of objects in which to search
`Machine` | `Charts` | `StatesJunctions` | `Transitions` | `Data` | `Events` | `Messages`

Specify the type of objects in which to search. You can limit your search to the Stateflow machine, charts, states, junctions, transitions, data, events, and messages. For more information, see “Overview of Stateflow Objects”.

`Field types` — Type of fields in which to search
`Names` | `Labels` | `Descriptions` | `Document links`

Specify the type of fields in which to search. You can limit your search to names, labels, descriptions, and document links.

Note The Search & Replace tool looks for matching text anywhere in a state label regardless of whether you limit your search to names or labels.

Tips

- The Search & Replace tool does not search the names of Simulink models and Stateflow charts. To change the names of models and charts, use the Simulink model window.

Version History

Introduced before R2006a

See Also

Topics
“Stateflow Editor Operations”
“Overview of Stateflow Objects”
“Regular Expressions”

Sequence Viewer

Visualize messages, events, states, transitions, and functions

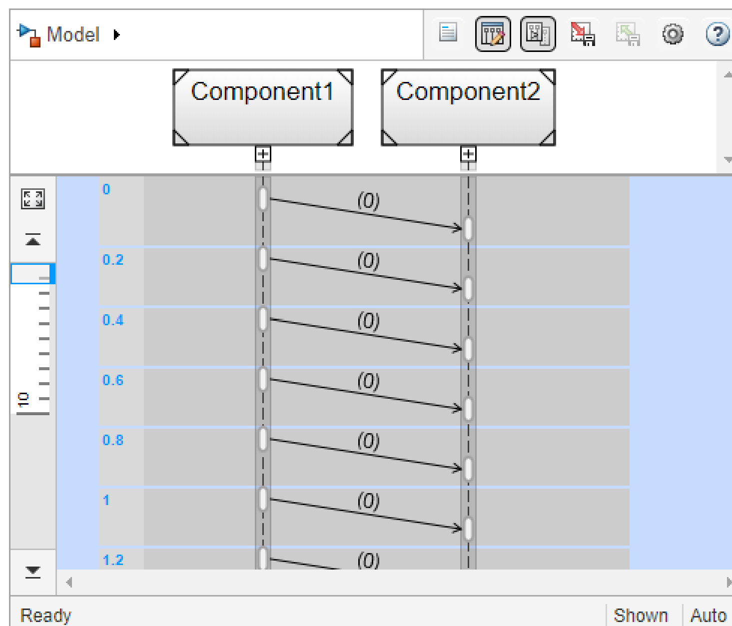
Description

The Sequence Viewer visualizes message flow, function calls, and state transitions.

Use the Sequence Viewer to see the interchange of messages, events, function calls in Simulink models, Simulink behavior models in System Composer™ and between Stateflow charts in Simulink models.

In the Sequence Viewer window, you can view event data related to Stateflow chart execution and the exchange of messages between Stateflow charts. The Sequence Viewer window shows messages as they are created, sent, forwarded, received, and destroyed at different times during model execution. The Sequence Viewer window also displays state activity, transitions, and function calls to Stateflow graphical functions, Simulink functions, and MATLAB functions. For more information, see "Use the Sequence Viewer to Visualize Messages, Events, and Entities".

Note The Sequence Viewer does not display function calls generated by MATLAB Function blocks and S-functions.



Open the Sequence Viewer

- Simulink Toolstrip: On the **Simulation** tab, in the **Review Results** section, click **Sequence Viewer**.

Examples

Using the Sequence Viewer Tool

1. To activate logging events, in the Simulink Toolstrip, under the **Simulation** tab, in the **Prepare** section, click **Log Events**.
2. Simulate your model.
3. To open the tool, in the Simulink Toolstrip, under the **Simulation** tab, in the **Review Results** section, click **Sequence Viewer**.

- "Use the Sequence Viewer to Visualize Messages, Events, and Entities"
- "Simulink Messages Overview" (Simulink)

Parameters

`Time Precision for Variable Step` — Digits for time increment precision
`3` (default) | scalar

Number of digits for time increment precision. When using a variable step solver, change this parameter to adjust the time precision for the sequence viewer. By default the block supports `3` digits of precision. Minimum and maximum precision are `1` and `16`, respectively.

Suppose the block displays two events that occur at times `0.1215` and `0.1219`. Displaying these two events precisely requires `4` digits of precision. If the precision is `3`, then the block displays two events at time `0.121`.

Programmatic Use
Block Parameter: `SequenceViewerTimePrecision`
Type: character vector
Values: `'3'` | scalar
Default: `'3'`

`History` — Maximum number of previous events to display
`1000` (default) | scalar

Total number of events before the last event to display. Minimum and maximum number of events are `0` and `25000`, respectively.

For example, if **History** is `5` and there are `10` events in your simulation, then the block displays `6` events, including the last event and the five events prior the last event. Earlier events are not displayed. The time ruler is greyed to indicate the time between the beginning of the simulation and the time of the first displayed event.

Each send, receive, drop, or function call event is counted as one event, even if they occur at the same simulation time.

Programmatic Use
Block Parameter: `SequenceViewerHistory`
Type: character vector

Values: `'1000'` | scalar
Default: `'1000'`

Version History

Introduced in R2020b

See Also

Blocks

Sequence Viewer

Topics

“Use the Sequence Viewer to Visualize Messages, Events, and Entities”
“Simulink Messages Overview” (Simulink)

Simulation Data Inspector

Inspect and compare data and simulation results to validate and iterate model designs

Description

The Simulation Data Inspector visualizes and compares multiple kinds of data.

Using the Simulation Data Inspector, you can inspect and compare time series data at multiple stages of your workflow. This example workflow shows how the Simulation Data Inspector supports all stages of the design cycle:

1. “View Data in the Simulation Data Inspector” (Simulink)

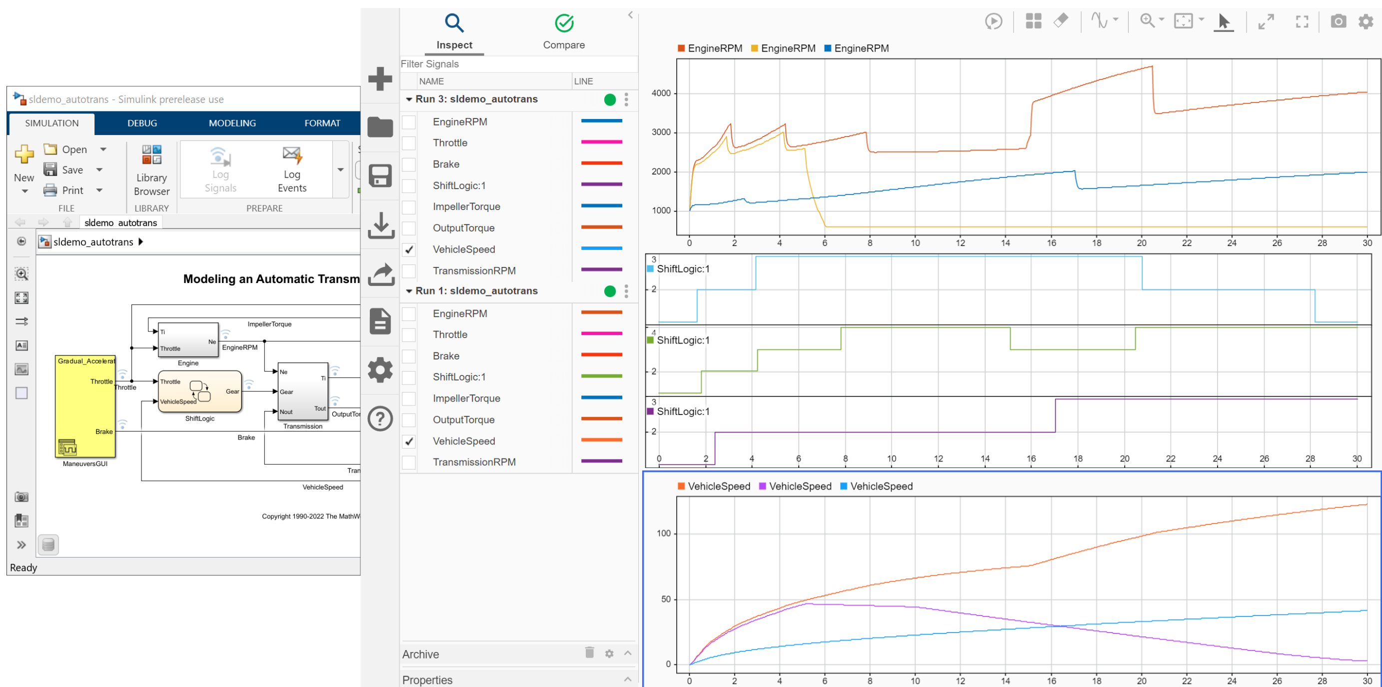
   Run a simulation in a model configured to log data to the Simulation Data Inspector, or import data from the workspace or a MAT-file. You can view and verify model input data or inspect logged simulation data while iteratively modifying your model diagram, parameter values, or model configuration.
2. “Inspect Simulation Data” (Simulink)

   Plot signals on multiple subplots, zoom in and out on specified plot axes, and use data cursors to understand and evaluate the data. “Create Plots Using the Simulation Data Inspector” (Simulink) to tell your story.
3. “Compare Simulation Data” (Simulink)

   Compare individual signals or simulation runs and analyze your comparison results with relative, absolute, and time tolerances. The compare tools in the Simulation Data Inspector facilitate iterative design and allow you to highlight signals that do not meet your tolerance requirements. For more information about the comparison operation, see “How the Simulation Data Inspector Compares Data” (Simulink).
4. “Save and Share Simulation Data Inspector Data and Views” (Simulink)

   Share your findings with others by saving Simulation Data Inspector data and views.

You can also harness the capabilities of the Simulation Data Inspector from the command line. For more information, see “Inspect and Compare Data Programmatically” (Simulink).



Open the Simulation Data Inspector

- Simulink Toolstrip: On the **Simulation** tab, under **Review Results**, click **Data Inspector**.
- Click the streaming badge on a signal to open the Simulation Data Inspector and plot the signal.
- MATLAB command prompt: Enter `Simulink.sdi.view`.

Examples

Apply a Tolerance to a Signal in Multiple Runs

You can use the Simulation Data Inspector programmatic interface to modify a parameter for the same signal in multiple runs. This example adds an absolute tolerance of `0.1` to a signal in all four runs of data.

First, clear the workspace and load the Simulation Data Inspector session with the data. The session includes logged data from four simulations of a Simulink® model of a longitudinal controller for an aircraft.

```
Simulink.sdi.clear
Simulink.sdi.load('AircraftExample.mldatx');
```

Use the `Simulink.sdi.getRunCount` function to get the number of runs in the Simulation Data Inspector. You can use this number as the index for a for loop that operates on each run.

```
count = Simulink.sdi.getRunCount;
```

Then, use a for loop to assign the absolute tolerance of `0.1` to the first signal in each run.

```
for a = 1:count
    runID = Simulink.sdi.getRunIDByIndex(a);
    aircraftRun = Simulink.sdi.getRun(runID);
    sig = getSignalByIndex(aircraftRun,1);
    sig.AbsTol = 0.1;
end
```

- “View Data in the Simulation Data Inspector” (Simulink)
- “Inspect Simulation Data” (Simulink)
- “Compare Simulation Data” (Simulink)
- “Iterate Model Design Using the Simulation Data Inspector” (Simulink)

Programmatic Use

`Simulink.sdi.view` opens the Simulation Data Inspector from the MATLAB command line.

Version History

Introduced in R2010b

See Also

Functions

`Simulink.sdi.clear` | `Simulink.sdi.clearPreferences` | `Simulink.sdi.snapshot`

Topics

“View Data in the Simulation Data Inspector” (Simulink)
“Inspect Simulation Data” (Simulink)
“Compare Simulation Data” (Simulink)
“Iterate Model Design Using the Simulation Data Inspector” (Simulink)