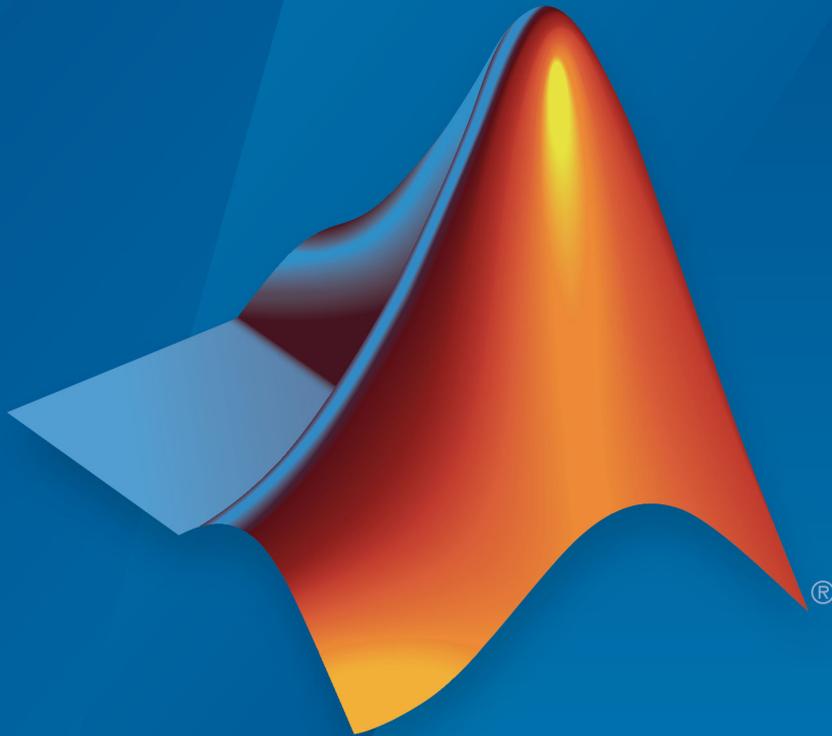


Simulink®
Reference



MATLAB® & SIMULINK®

R2017b



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.
3 Apple Hill Drive
Natick, MA 01760-2098

Simulink® Reference

© COPYRIGHT 2002–2017 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

July 2002	Online only	Revised for Simulink 5 (Release 13)
April 2003	Online only	Revised for Simulink 5.1 (Release 13SP1)
April 2004	Online only	Revised for Simulink 5.1.1 (Release 13SP1+)
June 2004	Online only	Revised for Simulink 6 (Release 14)
October 2004	Online only	Revised for Simulink 6.1 (Release 14SP1)
March 2005	Online only	Revised for Simulink 6.2 (Release 14SP2)
September 2005	Online only	Revised for Simulink 6.3 (Release 14SP3)
March 2006	Online only	Revised for Simulink 6.4 (Release 2006a)
September 2006	Online only	Revised for Simulink 6.5 (Release 2006b)
March 2007	Online only	Revised for Simulink 6.6 (Release 2007a)
September 2007	Online only	Revised for Simulink 7.0 (Release 2007b)
March 2008	Online only	Revised for Simulink 7.1 (Release 2008a)
October 2008	Online only	Revised for Simulink 7.2 (Release 2008b)
March 2009	Online only	Revised for Simulink 7.3 (Release 2009a)
September 2009	Online only	Revised for Simulink 7.4 (Release 2009b)
March 2010	Online only	Revised for Simulink 7.5 (Release 2010a)
September 2010	Online only	Revised for Simulink 7.6 (Release 2010b)
April 2011	Online only	Revised for Simulink 7.7 (Release 2011a)
September 2011	Online only	Revised for Simulink 7.8 (Release 2011b)
March 2012	Online only	Revised for Simulink 7.9 (Release 2012a)
September 2012	Online only	Revised for Simulink 8.0 (Release 2012b)
March 2013	Online only	Revised for Simulink 8.1 (Release 2013a)
September 2013	Online only	Revised for Simulink 8.2 (Release 2013b)
March 2014	Online only	Revised for Simulink 8.3 (Release 2014a)
October 2014	Online only	Revised for Simulink 8.4 (Release 2014b)
March 2015	Online only	Revised for Simulink 8.5 (Release 2015a)
September 2015	Online only	Revised for Simulink 8.6 (Release 2015b)
October 2015	Online only	Rereleased for Simulink 8.5.1 (Release 2015aSP1)
March 2016	Online only	Revised for Simulink 8.7 (Release 2016a)
September 2016	Online only	Revised for Simulink 8.8 (Release 2016b)
March 2017	Online only	Revised for Simulink 8.9 (Release 2017a)
September 2017	Online only	Revised for Simulink 9.0 (Release 2017b)

1	<u>Blocks — Alphabetical List</u>	
2	<u>Functions — Alphabetical List</u>	
3	<u>Mask Icon Drawing Commands</u>	
4	<u>Simulink Debugger Commands</u>	
5	<u>Simulink Classes</u>	
6	<u>Model and Block Parameters</u>	
	Model Parameters	6-2
	About Model Parameters	6-2

Examples of Setting Model Parameters	6-108
Common Block Properties	6-109
About Common Block Properties	6-109
Examples of Setting Block Properties	6-126
Block-Specific Parameters	6-128
Programmatic Parameters of Blocks and Models	6-128
Block-Specific Parameters and Programmatic Equivalents	6-129
Mask Parameters	6-278
About Mask Parameters	6-278

Fixed-Point Tool

7

Fixed-Point Tool Parameters and Dialog Box	7-2
Main Toolbar	7-2
Model Hierarchy Pane	7-5
Contents Pane	7-5
Customizing the Contents Pane View	7-8
Dialog Pane	7-10
Fixed-Point Advisor	7-12
Configure model settings	7-13
Run name	7-14
Simulate	7-15
Merge instrumentation results from multiple simulations	7-15
Derive ranges for selected system	7-16
Propose	7-16
Propose for	7-17
Default fraction length	7-17
Default word length	7-18
When proposing types use	7-18
Safety margin for simulation min/max (%)	7-18
Advanced Settings	7-20
Advanced Settings Overview	7-20
Fixed-point instrumentation mode	7-20
Data type override	7-21
Data type override applies to	7-23

Name of shortcut	7-25
Allow modification of fixed-point instrumentation settings . .	7-25
Allow modification of data type override settings	7-26
Allow modification of run name	7-27
Run name	7-27
Capture system settings	7-27
Fixed-point instrumentation mode	7-27
Data type override	7-28
Data type override applies to	7-29

Model Advisor Checks

8

Simulink Checks	8-2
Simulink Check Overview	8-4
Migrating to Simplified Initialization Mode Overview	8-5
Identify unconnected lines, input ports, and output ports	8-5
Check root model Inport block specifications	8-6
Check optimization settings	8-7
Check diagnostic settings ignored during accelerated model reference simulation	8-9
Check for parameter tunability information ignored for referenced models	8-10
Check for implicit signal resolution	8-11
Check for optimal bus virtuality	8-11
Check for Discrete-Time Integrator blocks with initial condition uncertainty	8-12
Identify disabled library links	8-13
Check for large number of function arguments from virtual bus across model reference boundary	8-14
Identify parameterized library links	8-15
Identify unresolved library links	8-16
Identify model reference variants and variant subsystems that override variant choice	8-17
Identify configurable subsystem blocks for converting to variant subsystem blocks	8-17
Check usage of function-call connections	8-18
Check model for upgradable Simulink Scope blocks	8-19
Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues	8-19

Check if read/write diagnostics are enabled for data store blocks	8-21
Check data store block sample times for modeling errors . . .	8-22
Check for potential ordering issues involving data store access	8-23
Check structure parameter usage with bus signals	8-24
Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition	8-25
Check for calls to <code>slDataTypeAndScale</code>	8-27
Check bus signals treated as vectors	8-29
Check for potentially delayed function-call subsystem return values	8-30
Identify block output signals with continuous sample time and non-floating point data type	8-31
Check usage of Merge blocks	8-32
Check usage of Outport blocks	8-35
Check usage of Discrete-Time Integrator blocks	8-47
Check model settings for migration to simplified initialization mode	8-48
Check S-functions in the model	8-50
Check for non-continuous signals driving derivative ports . .	8-51
Runtime diagnostics for S-functions	8-52
Check model for foreign characters	8-53
Identify unit mismatches in the model	8-54
Identify automatic unit conversions in the model	8-54
Identify disallowed unit systems in the model	8-55
Identify undefined units in the model	8-55
Check model for block upgrade issues	8-56
Check model for block upgrade issues requiring compile time information	8-57
Check that the model is saved in SLX format	8-58
Check model for SB2SL blocks	8-59
Check Model History properties	8-59
Identify Model Info blocks that can interact with external source control tools	8-60
Identify Model Info blocks that use the Configuration Manager	8-61
Check model for legacy 3DoF or 6DoF blocks	8-62
Check model and local libraries for legacy Aerospace Blockset blocks	8-62
Check and update masked blocks in library to use promoted parameters	8-63
Check and update mask image display commands with unnecessary <code>imread()</code> function calls	8-64

Check and update mask to affirm icon drawing commands dependency on mask workspace	8-65
Identify masked blocks that specify tabs in mask dialog using MaskTabNames parameter	8-66
Identify questionable operations for strict single-precision design	8-66
Check get_param calls for block CompiledSampleTime	8-68
Check model for parameter initialization and tuning issues .	8-70
Check for virtual bus across model reference boundaries . . .	8-71
Check model for custom library blocks that rely on frame status of the signal	8-73
Check Rapid accelerator signal logging	8-73
Check virtual bus inputs to blocks	8-74
Check for root outports with constant sample time	8-78
Analyze model hierarchy and continue upgrade sequence . . .	8-79
Check Access to Data Stores	8-81

Model Reference Conversion Advisor

9

Model Reference Conversion Advisor	9-2
Check Conversion Input Parameters	9-2

Performance Advisor Checks

10

Simulink Performance Advisor Checks	10-2
Simulink Performance Advisor Check Overview	10-2
Baseline	10-3
Checks that Require Update Diagram	10-3
Checks that Require Simulation to Run	10-3
Check Simulation Modes Settings	10-3
Check Compiler Optimization Settings	10-4
Create baseline	10-4
Identify resource-intensive diagnostic settings	10-4
Check optimization settings	10-4
Identify inefficient lookup table blocks	10-5

Check MATLAB System block simulation mode	10-5
Identify Interpreted MATLAB Function blocks	10-6
Identify simulation target settings	10-6
Check model reference rebuild setting	10-7
Identify Scope blocks	10-7
Identify active instrumentation settings on the model	10-7
Check model reference parallel build	10-8
Check Delay block circular buffer setting	10-10
Check continuous and discrete rate coupling	10-10
Check discrete signals driving derivative port	10-11
Check solver type selection	10-11
Select simulation mode	10-12
Select compiler optimizations on or off	10-13
Final Validation	10-14

Simulink Limits

11

Maximum Size Limits of Simulink Models	11-2
--------------------------------------------------	------

Block Reference Page Examples

12

Create Bus Ports in a Subsystem	12-2
Convert Bus Signal to a Vector	12-5
Assign Signal Values to a Bus	12-6
Initialize Your Model Using the Callback Button Block	12-7
Control a Parameter Value with Callback Button Blocks	12-9
Solve a Linear System of Algebraic Equations	12-12
Model a Planar Pendulum	12-13

Blocks — Alphabetical List

Abs

Output absolute value of input



Library

Math Operations

Description

The Abs block outputs the absolute value of the input.

For signed-integer data types, the absolute value of the most negative value is not representable by the data type. In this case, the **Saturate on integer overflow** check box controls the behavior of the block:

If you...	The block...	And...
Select this check box	Saturates to the most positive value of the integer data type	<ul style="list-style-type: none"> For 8-bit signed integers, -128 maps to 127. For 16-bit signed integers, -32768 maps to 32767. For 32-bit signed integers, -2147483648 maps to 2147483647.
Do not select this check box	Wraps to the most negative value of the integer data type	<ul style="list-style-type: none"> For 8-bit signed integers, -128 remains -128. For 16-bit signed integers, -32768 remains -32768. For 32-bit signed integers, -2147483648 remains -2147483648.

The Abs block supports zero-crossing detection. However, when you select **Enable zero-crossing detection** on the dialog box, the block does not report the simulation

minimum or maximum in the Fixed-Point Tool. If you want to use the Fixed-Point Tool to analyze a model, disable zero-crossing detection for all Abs blocks in the model first.

Data Type Support

The Abs block accepts real signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

The block also accepts complex floating-point inputs. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1 . To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Output minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” in *Simulink User's Guide* for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer). in the Fixed-Point Designer™ documentation.

Saturate on integer overflow

Action	Reason for Taking This Action	What Happens	Example
Select this check box.	Your model has possible overflow and you want explicit saturation protection in the generated code.	Overflows saturate to the maximum value that the data type can represent.	The number 130 does not fit in a signed 8-bit integer and saturates to 127.
Do not select this check box.	You want to optimize efficiency of your generated code.	Overflows wrap to the appropriate value that is representable by the data type.	The number 130 does not fit in a signed 8-bit integer and wraps to -126.

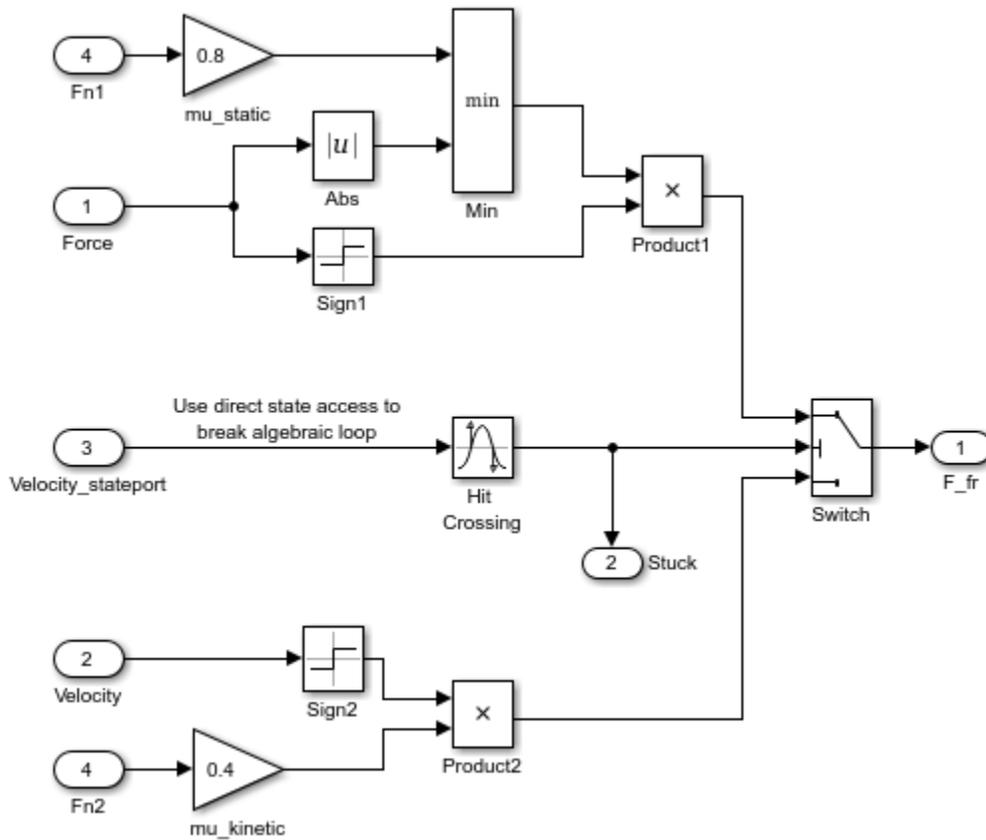
When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Examples

Usage as an Input to a MinMax Block

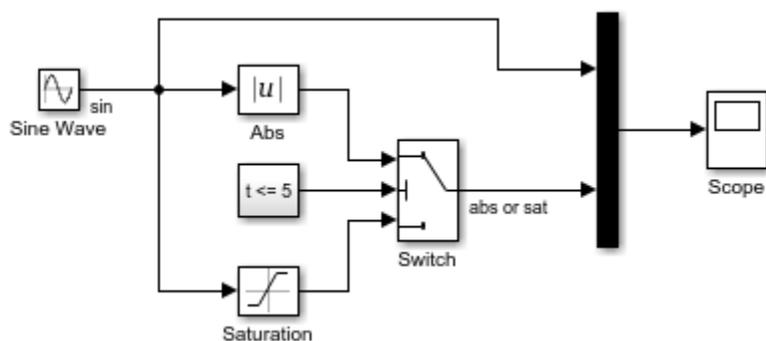
The `sldemo_hardstop` model shows how you can use the Abs block as an input to the MinMax block.

In the `sldemo_hardstop` model, the Abs block is in the Friction Model subsystem.



Usage as an Input to a Switch Block

The `sldemo_zeroring` model shows how you can use the `Abs` block as an input to the `Switch` block.



Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

Introduced before R2006a

Action Port

Implement Action subsystems used in `if` and `switch` control flow statements



Library

Ports & Subsystems

Description

Action Port blocks implement action subsystems used in `if` and `switch` control flow statements. The If Action Subsystem and the Switch Case Action Subsystem blocks each contain an Action Port block.

Data Type Support

Action Port blocks do not have data inputs or outputs.

Parameters

- “States when execution is resumed” on page 1-8
- “Propagate sizes of variable-size signals” on page 1-10

States when execution is resumed

Specify how to handle internal states when a subsystem with an Action Port block reenables.

Default: held

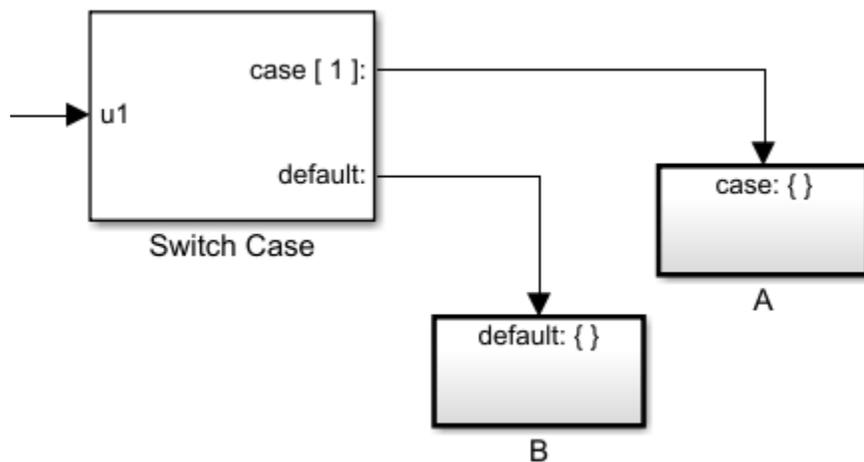
held

When the subsystem reenables, retains the states of the Action subsystem with their previous values. Retains the previous values of states between calls even if calling other member Action subsystems of an if-else or switch control flow statement.

reset

Reinitializes the states of the Action subsystem to initial values when the subsystem reenables.

Reenablement of a subsystem occurs when called and the condition of the call is true after having been previously false. In the following example, the Action Port blocks for both Action subsystems A and B have the **States when execution is resumed** parameter set to reset.



If case[1] is true, call Action subsystem A. This result implies that the default condition is false. When later calling B for the default condition, its states are reset. In the same way, Action subsystem A states are reset when calling A right after calling Action subsystem B.

Repeated calls to the Action subsystem of a case does not reset its states. If calling A again right after a previous call to A, this action does not reset the states of A. This behavior is because the condition of case[1] was not previously false. The same applies to B.

Parameter: InitializeStates

Type: character vector

Value: 'held' | 'reset' |

Default: 'held'

Propagate sizes of variable-size signals

Specify when to propagate a variable-size signal.

Default: Only when execution is resumed

Only when execution is resumed

Propagates variable-size signals only when reenabling the subsystem containing the Action Port block.

During execution

Propagates variable-size signals at each time step.

Parameter: PropagateVarSize

Type: character vector

Value: 'Only when execution is resumed' | 'During execution'

Default: 'Only when execution is resumed'

Characteristics

Sample Time	Inherited from driving If or Switch Case block
-------------	------------------------------------------------

See Also

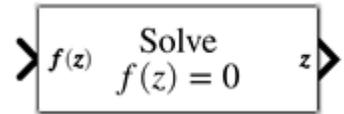
If, If Action Subsystem, Switch Case, Switch Case Action Subsystem

Introduced before R2006a

Algebraic Constraint

Constrain input signal

Library: Simulink / Math Operations



Description

The Algebraic Constraint block constrains the input signal $f(z)$ to z or 0 and outputs an algebraic state z . The block outputs a value that produces 0 or z at the input. The output must affect the input through a direct feedback path. In other words, the feedback path only contains blocks with direct feedthrough. For example, you can specify algebraic equations for index 1 differential-algebraic systems (DAEs).

Ports

Input

$f(z)$ — Input signal

real scalar or vector

Signal is subjected to the constraint $f(z) = 0$ or $f(z) = z$ to solve the algebraic loop.

Data Types: double

Output

z — Output state

real scalar or vector

Solution to the algebraic loop when the input signal $f(z)$ is subjected to the constraint $f(z) = 0$ or $f(z) = z$.

Data Types: double

Parameters

Constraint — Constraint on input signal

`f(z) = 0` (default) | `f(z) = z`

Type of constraint for which to solve. You can Choice to solve for $f(z) = 0$ or $f(z) = z$

Solver — Algebraic Loop Solver

`auto` (default) | `Trust Region` | `Line Search`

Choice between the Trust Region [1], [2] or Line Search [3] algorithms to solve the algebraic loop. By default this value is set to `auto`, which selects the solver based on the model configuration

Tolerance — Solver Tolerance

`auto` (default) | positive scalar

This option is visible when you explicitly specify a solver to be used (Trust region or Line Search) in the **Solver** dropdown menu. Specify a smaller value for higher accuracy or a larger value for faster execution. By default it is set to `auto`.

Initial Guess — Initial guess of solution value

`0` (default) | scalar

Initial guess for the algebraic state z that is close to the expected solution value to improve the efficiency of the algebraic loop solver. By default, this value is set to `0`

Model Examples

Block Characteristics

Data Types	double
Multidimensional Signals	No
Variable-Size Signals	No

References

- [1] Garbow, B. S., K. E. Hillstrom, and J. J. Moré. *User Guide for MINPACK-1*. Argonne, IL: Argonne National Laboratory, 1980.
- [2] Rabinowitz, P. H. *Numerical Methods for Nonlinear Algebraic Equations*. New York: Gordon and Breach, 1970.
- [3] Kelley, C. T. *Iterative Methods for Linear and Nonlinear Equations*. Society for Industrial and Applied Mathematics, Philadelphia, PA: 1995.

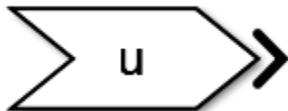
See Also

“Algebraic Loops”

Introduced before R2006a

Argument Inport

Argument input port for Simulink Function block



Description

This block is an input argument port for a function that you define in the Simulink Function block.

Data Type Support

The Argument Inport block accepts complex or real signals of any data type that Simulink supports, including fixed-point and enumerated data types. The Argument Inport block also accepts a bus object as a data type.

The numeric and data types of the block output are the same as those of its input. You can specify the signal type and data type of an input argument to an Argument Inport block using the **Signal type** and **Data type** parameters. For more information, see “Data Types Supported by Simulink”.

Parameters

- “Port number” on page 1-15
- “Argument Name” on page 1-15
- “Minimum” on page 1-16
- “Maximum” on page 1-16

- “Data type” on page 1-17
- “Show data type assistant” on page 1-18
- “Mode” on page 1-18
- “Data type override” on page 1-20
- “Signedness” on page 1-20
- “Word length” on page 1-21
- “Scaling” on page 1-21
- “Fraction length” on page 1-22
- “Slope” on page 1-22
- “Bias” on page 1-23
- “Output as nonvirtual bus” on page 1-23
- “Lock data type settings against changes by the fixed-point tools” on page 1-24
- “Port dimensions” on page 1-24
- “Signal type” on page 1-24

Port number

Specify the port number of the block.

Default: 1

This parameter controls the order in which the port that corresponds to the block appears in the parent subsystem or model block.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Argument Name

Default: u

This parameter provides the name of the input argument in the function prototype of the Simulink Function block.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Minimum

Specify the minimum value for the block to output.

Default: [] (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Maximum

Specify the maximum value for the block to output.

Default: [] (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Data type

Specify the output data type of the argument input.

Default: `double`

`double`

 Data type is `double`.

`single`

 Data type is `single`.

`int8`

 Data type is `int8`.

`uint8`

 Data type is `uint8`.

`int16`

 Data type is `int16`.

`uint16`

 Data type is `uint16`.

`int32`

 Data type is `int32`.

`uint32`

 Data type is `uint32`.

`boolean`

 Data type is `boolean`.

`fixdt(1,16,0)`

Data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`

Data type is fixed point `fixdt(1,16,2^0,0)`.

Enum: `<class name>`

Data type is enumerated, for example, Enum: `Basic Colors`.

Bus: `<object name>`

Data type is a bus object.

`<data type expression>`

The name of a data type object, for example `Simulink.NumericType`

Do not specify a bus object as the expression.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Mode

Select the category of data to specify.

Default: `Built in`

`Built in`

Built-in data types. Selecting `Built in` enables a second text box to the right. Select one of the following:

- `double` (default)

- single
- int8
- uint8
- int16
- uint16
- int32
- uint32
- boolean

Fixed point

Fixed-point data types.

Enumerated

Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.

Bus object

Bus object. Selecting `Bus` enables a **Bus object** parameter to the right, where you enter the name of a bus object that you want to use to define the structure of the bus. If you need to create or change a bus object, click **Edit** to the right of the **Bus object** field to open the Simulink Bus Editor. For details, see “Create Bus Objects with the Bus Editor”.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Do not specify a bus object as the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow® chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data as signed.

`Unsigned`

Specify the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope` and `bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > Slope and bias enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Output as nonvirtual bus

Output a nonvirtual bus.

Default: Off

On

Output a nonvirtual bus.

Off

Output a virtual bus.

- Select this option if you want code generated from this model to use a C structure to define the structure of the bus signal output by this block.
- All signals in a nonvirtual bus must have the same sample time, even if the elements of the associated bus object specify inherited sample times. Any bus operation that would result in a nonvirtual bus that violates this requirement generates an error. Therefore, if you select this option all signals in the bus must have the same sample time. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus, to allow the signal or bus to be included in a nonvirtual bus.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Lock data type settings against changes by the fixed-point tools

Select to lock data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks all data type settings for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change data type settings for this block.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Port dimensions

Specify the dimensions of the input signal to the block.

Default: 1

Valid values are:

n	Vector signal of width n accepted
[m n]	Matrix signal having m rows and n columns accepted

Signal type

Specify the numeric type of the argument input.

Default: real

real

Specify the numeric type as a real number.

complex

Specify the numeric type as a complex number.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Characteristics

Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

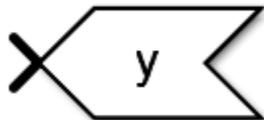
Argument Output

Simulink Function

Introduced in R2014b

Argument Output

Argument output port for Simulink Function block



Description

This block is an output argument port for a function that you define in the Simulink Function block.

Data Type Support

The Argument Output block accepts real or complex signals of any data type that Simulink supports. An Argument Output block can also accept fixed-point and enumerated data types when the block is not a root-level output port. The complexity and data type of the block output are the same as those of its input. The Argument Output block also accepts a bus object as a data type.

For more information, see “Data Types Supported by Simulink”.

Parameters

- “Port number” on page 1-27
- “Argument Name” on page 1-27
- “Minimum” on page 1-28
- “Maximum” on page 1-28
- “Data type” on page 1-29

- “Show data type assistant” on page 1-30
- “Mode” on page 1-30
- “Data type override” on page 1-32
- “Signedness” on page 1-32
- “Word length” on page 1-33
- “Scaling” on page 1-33
- “Fraction length” on page 1-34
- “Slope” on page 1-34
- “Bias” on page 1-35
- “Lock output data type setting against changes by the fixed-point tools” on page 1-35
- “Output as nonvirtual bus” on page 1-35
- “Port dimensions” on page 1-36
- “Signal type” on page 1-37

Port number

Specify the port number of the block.

Default: 1

This parameter controls the order in which the port that corresponds to the block appears on the parent subsystem or model block.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Argument Name

Specify the name of the output argument.

Default: y

This parameter provides the name of the output argument in the function prototype of the Simulink Function block.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Minimum

Specify the minimum value for the block to output.

Default: [] (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Maximum

Specify the maximum value for the block to output.

Default: [] (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Data type

Specify the output data type of the external input.

Default: double

double

Data type is double.

single

Data type is single.

int8

Data type is int8.

uint8

Data type is uint8.

int16

Data type is int16.

uint16

Data type is uint16.

int32

Data type is int32.

uint32

Data type is uint32.

boolean

Data type is boolean.

`fixdt(1,16,0)`

Data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`

Data type is fixed point `fixdt(1,16,2^0,0)`.

Enum: `<class name>`

Data type is enumerated, for example, Enum: `BasicColors`.

Bus: `<object name>`

Data type is a bus object.

`<data type expression>`

The name of a data type object, for example `Simulink.NumericType`

Do not specify a bus object as the expression.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Mode

Select the category of data to specify.

Default: `Built in`

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following:

- `double` (default)

- single
- int8
- uint8
- int16
- uint16
- int32
- uint32
- boolean

Fixed point

Fixed-point data types.

Enumerated

Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.

Bus object

Bus object. Selecting `Bus` enables a **Bus object** parameter to the right, where you enter the name of a bus object that you want to use to define the structure of the bus. If you need to create or change a bus object, click **Edit** to the right of the **Bus object** field to open the Simulink Bus Editor. For details, see “Create Bus Objects with the Bus Editor”.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Do not specify a bus object as the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data as signed.

`Unsigned`

Specify the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope` and `bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: `LockScale`

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Output as nonvirtual bus

Output a nonvirtual bus.

Default: Off



On

Output a nonvirtual bus.



Off

Output a virtual bus.

- Select this option if you want code generated from this model to use a C structure to define the structure of the bus signal output by this block.
- All signals in a nonvirtual bus must have the same sample time, even if the elements of the associated bus object specify inherited sample times. Any bus operation that would result in a nonvirtual bus that violates this requirement generates an error. Therefore, if you select this option all signals in the bus must have the same sample time. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus, to allow the signal or bus to be included in a nonvirtual bus.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Port dimensions

Specify the dimensions that a signal must have to connect to this Outport block.

Default: 1

Valid values are:

N	The signal connected to this port must be a vector of size N.
[R C]	The signal connected to this port must be a matrix having R rows and C columns.

Clearing **via bus object** enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Signal type

Specify the numeric type of the signal output by this block.

Default: `real`

`real`

Output a real-valued signal. The signal connected to this block must be real. If it is not, Simulink software displays an error if you try to update the diagram or simulate the model that contains this block.

`complex`

Output a complex signal. The signal connected to this block must be complex. If it is not, Simulink software displays an error if you try to update the diagram or simulate the model that contains this block.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Characteristics

Dimensionalized	Yes
Multidimensionalized	Yes
Zero-Crossing Detection	No

See Also

Argument Inport

Simulink Function

Introduced in R2014b

Assertion

Check whether signal is zero



Library

Model Verification

Description

The Assertion block checks whether any of the elements of the input signal is zero. If all elements are nonzero, the block does nothing. If any element is zero, the block halts the simulation, by default, and displays an error message. Use the block parameter dialog box to:

- Specify that the block should display an error message when the assertion fails but allow the simulation to continue.
- Specify a MATLAB® expression to evaluate when the assertion fails.
- Enable or disable the assertion.

You can also use the **Model Verification block enabling** setting in the Configuration Parameters dialog box to enable or disable all Assertion blocks in a model.

The Assertion block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to physically remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

The Assertion block works in normal, accelerator and rapid accelerator simulation modes.

Note For information about how Simulink Coder™ generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Data Type Support

The Assertion block accepts input signals of any dimensions and any numeric data type that Simulink supports, including fixed-point data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Enable assertion

Clearing this check box disables the Assertion block, that is, causes the model to behave as if the Assertion block did not exist. The **Model Verification block enabling** setting in the Configuration Parameters dialog box lets you enable or disable all Assertion blocks in a model regardless of the setting of this option.

Simulation callback when assertion fails

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Stop simulation when assertion fails

Selecting this check box causes the Simulink software to terminate the simulation when the block input is zero. After terminating the simulation, the software displays an error. Clearing this check box causes the Simulink software to continue the simulation and display a warning when the block input is zero.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

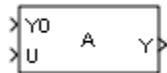
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Assignment

Assign values to specified elements of signal



Library

Math Operations

Description

The Assignment block assigns values to specified elements of the signal. You can specify the indices of the elements to be assigned values either by entering the indices in the block dialog box or by connecting an external indices source or sources to the block. The signal at the block data port, labeled U, specifies values to be assigned to Y. The block replaces the specified elements of Y with elements from the data signal.

Based on the value you enter for the **Number of output dimensions** parameter, a table of index options is displayed. Each row of the table corresponds to one of the output dimensions in **Number of output dimensions**. For each dimension, you can define the elements of the signal to work with. Specify a vector signal as a 1-D signal and a matrix signal as a 2-D signal. When you configure the Assignment block for multidimensional signal operations, the block icon changes.

For example, assume a 5-D signal with a one-based index mode. The table in the Assignment block dialog changes to include one row for each dimension. If you define each dimension with the following entries:

- 1

Index Option, select Assign all

- 2

Index Option, select Index vector (dialog)

Index, enter [1 3 5]

- 3

Index Option, select Starting index (dialog)

Index, enter 4

- 4

Index Option, select Starting index (port)

- 5

Index Option, select Index vector (port)

The assigned values will be $Y(1:end, [1\ 3\ 5], 4:3+size(U, 3), Idx4:Idx4+size(U, 4)-1, Idx5)=U$, where $Idx4$ and $Idx5$ are the input ports for dimensions 4 and 5.

The Assignment block's data port is labeled U. The rest of this section refers to the data port as U to simplify the explanation of the block's usage.

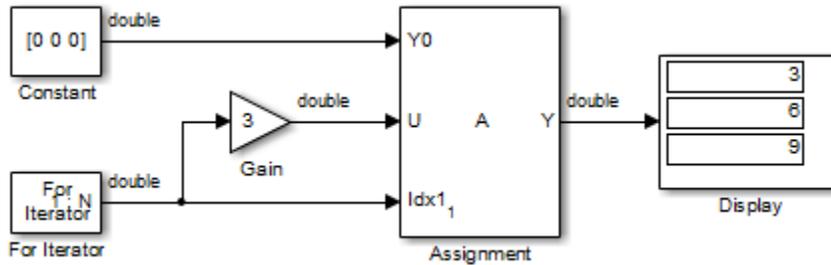
When using the Assignment block in Normal mode, Simulink initializes block outputs to zero even if the model does not explicitly initialize them. In Accelerator mode, Simulink converts the model into an S-Function. This involves code generation. The code generated may not do implicit initialization of block outputs. In such cases, you must explicitly initialize the model outputs.

You can use the block to assign values to vector, matrix, or multidimensional signals.

You can use an array of buses as an input signal to an Assignment block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Iterated Assignment

You can use the Assignment block to assign values computed in a For or While Iterator loop to successive elements of a vector, matrix, or multidimensional signal in a single time step. For example, the following model uses a For Iterator block to create a vector signal each of whose elements equals $3*i$ where i is the index of the element.

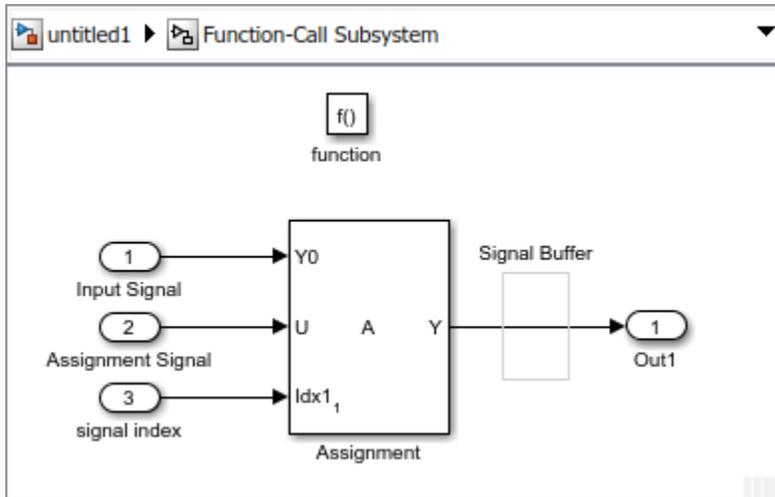


Iterated assignment uses an iterator (For or While) block to generate indices for the Assignment block. On the first iteration of an iterated assignment, the Assignment block copies the first input (Y0) to the output (Y) and assigns the second input (U) to the output Y(E₁). On successive iterations, the Assignment block assigns the current value of U to Y(E_i), that is, without first copying Y0 to Y. These actions occur in a single time step.

Assignment Block in Conditional Subsystem

If you place an Assignment block in a conditional subsystem block, a signal buffer can be inserted in many cases, and merging of signals from Assignment blocks with partial writes can cause an error.

However, if you select the **Ensure output is virtual** check box for the conditional subsystem Output block, such cases are supported and partial writes to arrays using Assignment blocks are possible.



Data Type Support

The data and initialization ports of the Assignment block accept signals of any data type that Simulink supports, including fixed-point, enumerated, and nonvirtual bus data types. The external indices port accepts any built-in data type, except Boolean data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Number of output dimensions

Enter the number of dimensions of the output signal.

Index mode

Select the indexing mode: *One-based* or *Zero-based*. If *One-based* is selected, an index of 1 specifies the first element of the input vector, 2, the second element, and so on. If *Zero-based* is selected, an index of 0 specifies the first element of the input vector, 1, the second element, and so on.

Index Option

Define, by dimension, how the elements of the signal are to be indexed. From the list, select:

- Assign all

This is the default. All elements are assigned.

- Index vector (dialog)

Enables the **Index** column. Enter the indices of elements.

- Index vector (port)

Disables the **Index** column. The index port defines the indices of elements.

- Starting index (dialog)

Enables the **Index** column. Enter the starting index of the range of elements to be assigned values.

- Starting index (port)

Disables the **Index** column. The index port defines the starting index of the range of elements to be assigned values.

If you choose `Index vector (port)` or `Starting index (port)` for any dimension in the table, you can specify the value for the **Initialize output (Y)** parameter to be one of the following:

- Initialize using input port <Y0>
- Specify size for each dimension in table

Otherwise, Y0 always initializes output port Y.

The **Index** and **Output Size** columns are displayed as relevant.

Index

If the **Index Option** is `Index vector (dialog)`, enter the index of each element you are interested in.

If the **Index Option** is `Starting index (dialog)`, enter the starting index of the range of elements to be selected. The number of elements from the starting point is determined by the size of this dimension at U.

Output Size

Enter the width of the block output signal. If you select `Specify size for each dimension in table` for the **Initialize output (Y)** parameter, this column is enabled.

Initialize output (Y)

Specify how to initialize the output signal. The **Initialize output** parameter appears when you set **Index Option** to `Index vector (port)` or `Starting index (port)`.

- Initialize using input port <Y0>

The signal at the input port Y0 initializes the output.

- Specify size for each dimension in table

The block requires you to specify the width of the block's output signal in the **Output Size** parameter. If the output has unassigned elements, the value of those elements is undefined.

Action if any output element is not assigned

Specify whether to produce a warning or error if you have not assigned all output elements. Options include:

- `Error` — Simulink software terminates the simulation and displays an error.
- `Warning` — Simulink software displays a warning and continues the simulation.
- `None` — Simulink software takes no action.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Characteristics

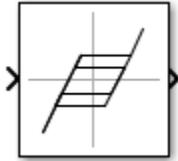
Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
------------	---------------------------------------------------------------------

Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Backlash

Model behavior of system with play

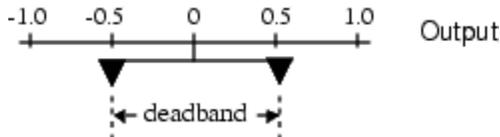


Library

Discontinuities

Description

The Backlash block implements a system in which a change in input causes an equal change in output. However, when the input changes direction, an initial change in input has no effect on the output. The amount of side-to-side play in the system is referred to as the *deadband*. The deadband is centered about the output. This figure shows the block's initial state, with the default deadband width of 1 and initial output of 0.



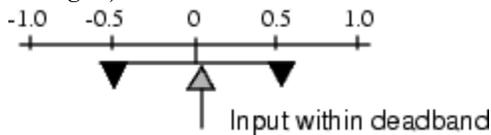
A system with play can be in one of three modes:

- Disengaged — In this mode, the input does not drive the output and the output remains constant.
- Engaged in a positive direction — In this mode, the input is increasing (has a positive slope) and the output is equal to the input *minus* half the deadband width.
- Engaged in a negative direction — In this mode, the input is decreasing (has a negative slope) and the output is equal to the input *plus* half the deadband width.

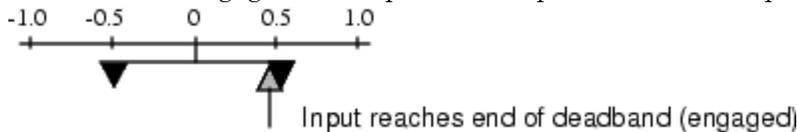
If the initial input is outside the deadband, the **Initial output** parameter value determines whether the block is engaged in a positive or negative direction, and the output at the start of the simulation is the input plus or minus half the deadband width.

For example, the Backlash block can be used to model the meshing of two gears. The input and output are both shafts with a gear on one end, and the output shaft is driven by the input shaft. Extra space between the gear teeth introduces *play*. The width of this spacing is the **Deadband width** parameter. If the system is disengaged initially, the output (the position of the driven gear) is defined by the **Initial output** parameter.

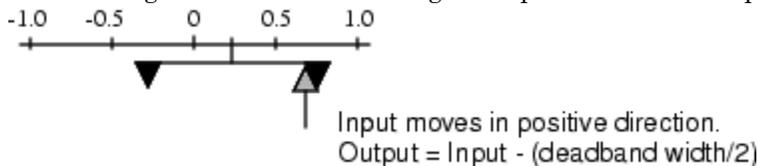
The following figures illustrate the block's operation when the initial input is within the deadband. The first figure shows the relationship between the input and the output while the system is in disengaged mode (and the default parameter values are not changed).



The next figure shows the state of the block when the input has reached the end of the deadband and engaged the output. The output remains at its previous value.



The final figure shows how a change in input affects the output while they are engaged.



If the input reverses its direction, it disengages from the output. The output remains constant until the input either reaches the opposite end of the deadband or reverses its direction again and engages at the same end of the deadband. Now, as before, movement in the input causes equal movement in the output.

For example, if the deadband width is 2 and the initial output is 5, the output, y , at the start of the simulation is as follows:

- 5 if the input, u , is between 4 and 6
- $u + 1$ if $u < 4$
- $u - 1$ if $u > 6$

Data Type Support

The Backlash block accepts and outputs real values of `single`, `double`, and built-in integer data types.

Parameters

Deadband width

Specify the width of the deadband. The default is 1.

Initial output

Specify the initial output value. The default value is 0. This parameter is tunable. Simulink does not allow the initial output of this block to be `inf` or `NaN`.

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- `Elements as channels (sample based)` — Treat each element of the input as a separate channel (sample-based processing).
- `Columns as channels (frame based)` — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox™ license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

- `Inherited` — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input `u`. All other input signals must be sample based.

Input Signal <code>u</code>	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Enable zero-crossing detection

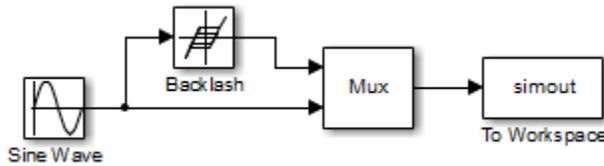
Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Sample time

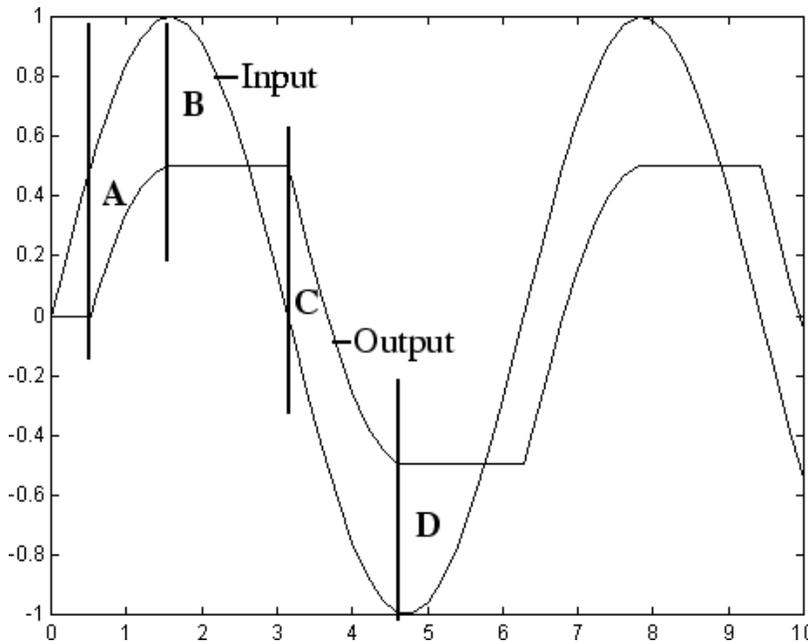
Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Examples

The following model shows the effect of a sine wave passing through a Backlash block.



The Backlash block uses default parameter values: the deadband width is 1 and the initial output is 0. The following plot shows that the Backlash block output is zero until the input reaches the end of the deadband (at 0.5). Now the input and output are engaged and the output moves as the input does until the input changes direction (at 1.0). When the input reaches 0, it again engages the output at the opposite end of the deadband.



A Input engages in positive direction. Change in input causes equal change in output.

B Input disengages. Change in input does not affect output.

C Input engages in negative direction. Change in input causes equal change in output.

D Input disengages. Change in input does not affect output.

Characteristics

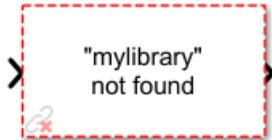
Data Types	Double Single Base Integer
Sample Time	Inherited from driving block

Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

Introduced before R2006a

Unresolved Link

Indicate unresolved reference to library block



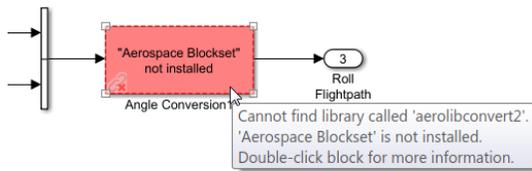
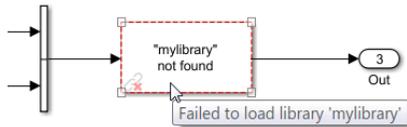
Description

This block indicates an unresolved reference to a library block (see “Linked Blocks”). You can use this block's parameter dialog box to fix the reference to point to the actual location of the library block.

Parameters

Details

The **Details** field contains a description of the cause of the unresolved link. Simulink tries to help you find and install missing products that a model needs to run. For missing products, the block description provides a link. Click the link to open Add-On Explorer and install the missing products.

You can customize the Unresolved Link block description for your library to include URLs as follows:

```
set_param(library1, 'libraryinfo', 'https://www.mathworks.com');
```

Here, `library1` is the name of the library for which you want to change the description, and `libraryinfo` is the property that provides the description of the unresolved link.

Source block

Path of the library block that this link represents. To fix a bad link, either click the link in the description to open Add-On Explorer and install a missing product, or edit the **Source block** field to the correct path of the library block. Then select Apply or OK to apply the fix and close the dialog box.

Alternatively, to fix an unresolved link, you can:

- Delete the unresolved block and copy the library block back into your model.
- Add the folder that contains the required library to the MATLAB path and select either **Simulation > Update Diagram** or **Diagram > Refresh Blocks**.

Source type

Type of library block that this link represents.

See Also

Topics

“Linked Blocks”

“Fix Unresolved Library Links”

Introduced in R2014a

Band-Limited White Noise

Introduce white noise into continuous system



Library

Sources

Description

Simulation of White Noise

The Band-Limited White Noise block generates normally distributed random numbers that are suitable for use in continuous or hybrid systems.

Theoretically, continuous white noise has a correlation time of 0, a flat power spectral density (PSD), and a total energy of infinity. In practice, physical systems are never disturbed by white noise, although white noise is a useful theoretical approximation when the noise disturbance has a correlation time that is very small relative to the natural bandwidth of the system.

In Simulink software, you can simulate the effect of white noise by using a random sequence with a correlation time much smaller than the shortest time constant of the system. The Band-Limited White Noise block produces such a sequence. The correlation time of the noise is the sample rate of the block. For accurate simulations, use a correlation time much smaller than the fastest dynamics of the system. You can get good results by specifying

$$tc \approx \frac{1}{100} \frac{2\pi}{f_{max}},$$

where f_{max} is the bandwidth of the system in rad/sec.

Comparison with the Random Number Block

The primary difference between this block and the Random Number block is that the Band-Limited White Noise block produces output at a specific sample rate. This rate is related to the correlation time of the noise.

Usage with the Averaging Power Spectral Density Block

The Band-Limited White Noise block specifies a two-sided spectrum, where the units are Hz. The Averaging Power Spectral Density block specifies a one-sided spectrum, where the units are the square of the magnitude per unit radial frequency: $\text{Mag}^2/(\text{rad}/\text{sec})$. When you feed the output of a Band-Limited White Noise block into an Averaging Power Spectral Density block, the average PSD value is π times smaller than the **Noise power** of the Band-Limited White Noise block. This difference is the result of converting the units of one block to the units of the other: $1/(1/2)(2\pi) = 1/\pi$, where:

- $1/2$ is the factor for converting from a two-sided to one-sided spectrum
- 2π is the factor for converting from Hz to rad/sec

Algorithm

To produce the correct intensity of this noise, the covariance of the noise is scaled to reflect the implicit conversion from a continuous PSD to a discrete noise covariance. The appropriate scale factor is $1/tc$, where tc is the correlation time of the noise. This scaling ensures that the response of a continuous system to the approximate white noise has the same covariance as the system would have to true white noise. Because of this scaling, the covariance of the signal from the Band-Limited White Noise block is not the same as the **Noise power** (intensity) parameter. This parameter is actually the height of the PSD of the white noise. This block approximates the covariance of white noise as the **Noise power** divided by tc .

Data Type Support

The Band-Limited White Noise block outputs real values of type `double`.

Parameters

Noise power

Specify the height of the PSD of the white noise. The default value is 0.1.

Sample time

Specify the correlation time of the noise. The default value is 0.1. For more information, see “Specify Sample Time” in the Simulink documentation.

Seed

Specify the starting seed for the random number generator. The default value is 23341.

Interpret vector parameters as 1-D

Select to output a 1-D array when the block parameters are vectors. Otherwise, output a 2-D array one of whose dimensions is 1. See “Determining the Output Dimensions of Source Blocks” in the Simulink documentation.

Examples

The following Simulink examples show how to use the Band-Limited White Noise block:

- `slexAircraftExample`
- `sldemo_radar_eml`

Characteristics

Data Types	Double
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

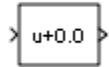
See Also

Random Number

Introduced before R2006a

Bias

Add bias to input



Library

Math Operations

Description

The Bias block adds a bias, or offset, to the input signal according to

$$Y = U + bias,$$

where U is the block input and Y is the output.

Data Type Support

The Bias block accepts and outputs real or complex values of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Bias

Specify the value of the offset to add to the input signal.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Do not select this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</p>	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code> , which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code> , is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Bit Clear

Set specified bit of stored integer to zero



Library

Logic and Bit Operations

Description

The Bit Clear block sets the specified bit, given by its index, of the stored integer to zero. Scaling is ignored.

You can specify the bit to be set to zero with the **Index of bit** parameter, where bit zero is the least significant bit.

Data Type Support

The Bit Clear block supports Simulink integer, fixed-point, and Boolean data types. The block does not support true floating-point data types or enumerated data types.

Parameters

Index of bit

Index of bit where bit 0 is the least significant bit.

Examples

If the Bit Clear block is turned on for bit 2, bit 2 is set to 0. A vector of constants $2.^{[0\ 1\ 2\ 3\ 4]}$ is represented in binary as [00001 00010 00100 01000 10000]. With bit 2 set to 0, the result is [00001 00010 00000 01000 10000], which is represented in decimal as [1 2 0 8 16].

Characteristics

Data Types	Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

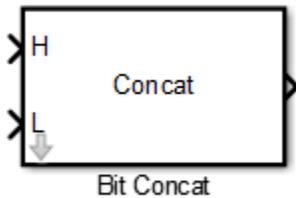
See Also

Bit Set

Introduced before R2006a

Bit Concat

Concatenates up to 128 input words into single output



Library

HDL Coder / HDL Operations

Description

The Bit Concat block concatenates up to 128 input words into a single output. The input port labeled **L** designates the lowest-order input word. The port labeled **H** designates the highest-order input word. The right-to-left ordering of words in the output follows the low-to-high ordering of input signals.

How the block operates depends on the number and dimensions of the inputs, as follows:

- Single input: The input is a scalar or a vector. When the input is a vector, the coder concatenates the individual vector elements.
- Two inputs: Inputs are any combination of scalar and vector. When one input is scalar and the other is a vector, the coder performs scalar expansion. Each vector element is concatenated with the scalar, and the output has the same dimension as the vector. When both inputs are vectors, they must have the same size.
- Three or more inputs (up to a maximum of 128 inputs): Inputs are uniformly scalar or vector. All vector inputs must have the same size.

Parameters

Number of Inputs: Enter an integer specifying the number of input signals. The number of block input ports updates when you change **Number of Inputs**.

- Default: 2
- Minimum: 1
- Maximum: 128

Caution Make sure that the **Number of Inputs** is equal to the number of signals you connect to the block. If the block has unconnected inputs, an error occurs at code generation time.

Ports

The block has up to 128 input ports, with H representing the highest-order input word, and L representing the lowest-order input word. The maximum concatenated output word size is 128 bits.

Supported Data Types

- Input: Fixed-point, integer (signed or unsigned), Boolean
- Output: Unsigned fixed-point or integer

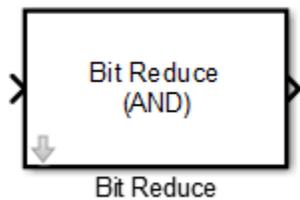
See Also

[Bit Shift](#) | [Bit Reduce](#) | [Bit Rotate](#) | [Bit Slice](#)

Introduced in R2014a

Bit Reduce

AND, OR, or XOR bit reduction on all input signal bits to single bit



Library

HDL Coder / HDL Operations

Description

The Bit Reduce block performs a selected bit-reduction operation (AND, OR, or XOR) on all the bits of the input signal, for a single-bit result.

Parameters

Reduction Mode

Specifies the reduction operation:

- AND (default): Perform a bitwise AND reduction of the input signal.
- OR: Perform a bitwise OR reduction of the input signal.
- XOR: Perform a bitwise XOR reduction of the input signal.

Ports

The block has the following ports:

Input

- Supported data types: Fixed-point, integer (signed or unsigned), Boolean
- Minimum bit width: 2
- Maximum bit width: 128

Output

Supported data type: `ufix1`

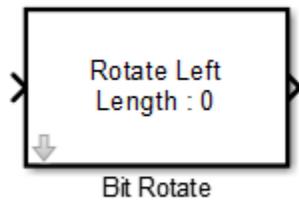
See Also

[Bit Shift](#) | [Bit Concat](#) | [Bit Rotate](#) | [Bit Slice](#)

Introduced in R2014a

Bit Rotate

Rotate input signal by bit positions



Library

HDL Coder / HDL Operations

Description

The Bit Rotate block rotates the input signal left or right by the specified number of bit positions.

Parameters

Rotate Mode: Specifies direction of rotation, left or right. The default is `Rotate Left`.

Rotate Length: Specifies the number of bits to rotate. Specify a value greater than or equal to zero. The default is 0.

Ports

The block has the following ports:

Input

- Supported data types: Fixed-point, integer (signed or unsigned), Boolean
- Minimum bit width: 2
- Maximum bit width: 128

Output

Has the same data type as the input signal.

See Also

[Bit Shift](#) | [Bit Concat](#) | [Bit Reduce](#) | [Bit Slice](#)

Introduced in R2014a

Bit Set

Set specified bit of stored integer to one



Library

Logic and Bit Operations

Description

The Bit Set block sets the specified bit of the stored integer to one. Scaling is ignored.

You can specify the bit to be set to one with the **Index of bit** parameter, where bit zero is the least significant bit.

Data Type Support

The Bit Set block supports Simulink integer, fixed-point, and Boolean data types. The block does not support true floating-point data types or enumerated data types.

Parameters

Index of bit

Index of bit where bit 0 is the least significant bit.

Examples

If the Bit Set block is turned on for bit 2, bit 2 is set to 1. A vector of constants $2.^{[0 \ 1 \ 2 \ 3 \ 4]}$ is represented in binary as [00001 00010 00100 01000 10000]. With bit 2 set to 1, the result is [00101 00110 00100 01100 10100], which is represented in decimal as [5 6 4 12 20].

Characteristics

Data Types	Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

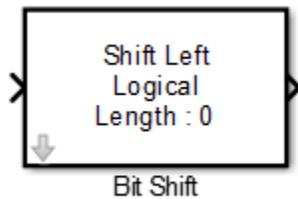
See Also

Bit Clear

Introduced before R2006a

Bit Shift

Logical or arithmetic shift of input signal



Library

HDL Coder / HDL Operations

Description

The Bit Shift block performs a logical or arithmetic shift on the input signal.

Parameters

Shift Mode

Default: Shift Left Logical

Specifies the type and direction of shift:

- Shift Left Logical (default)
- Shift Right Logical
- Shift Right Arithmetic

Shift Length

Specifies the number of bits to be shifted. Specify a value greater than or equal to zero. The default is 0.

Ports

The block has the following ports:

Input

- Supported data types: Fixed-point, integer (signed or unsigned), Boolean
- Minimum bit width: 2
- Maximum bit width: 128

Output

Has the same data type and bit width as the input signal.

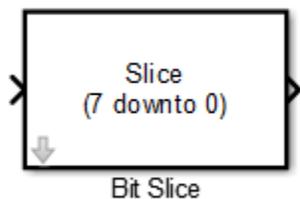
See Also

[Bit Rotate](#) | [Bit Concat](#) | [Bit Reduce](#) | [Bit Slice](#)

Introduced in R2014a

Bit Slice

Return field of consecutive bits from input signal



Library

HDL Coder / HDL Operations

Description

The Bit Slice block returns a field of consecutive bits from the input signal. Specify the lower and upper boundaries of the bit field by using zero-based indices in the **LSB Position** and **MSB Position** parameters.

Parameters

MSB Position

Specifies the bit position (zero-based) of the most significant bit (MSB) of the field to extract. The default is 7.

For an input word size *WS*, **LSB Position** and **MSB Position** must satisfy the following constraints:

```
WS > MSB Position >= LSB Position >= 0;
```

The word length of the output is computed as $(\text{MSB Position} - \text{LSB Position}) + 1$.

LSB Position

Specifies the bit position (zero-based) of the least significant bit (LSB) of the field to extract. The default is 0.

Ports

The block has the following ports:

Input

- Supported data types: Fixed-point, integer (signed or unsigned), Boolean
- Maximum bit width: 128

Output

Supported data types: unsigned fixed-point or unsigned integer.

See Also

[Bit Rotate](#) | [Bit Concat](#) | [Bit Reduce](#) | [Bit Shift](#)

Introduced in R2014a

Bitwise Operator

Specified bitwise operation on inputs



Library

Logic and Bit Operations

Description

Bitwise Operations

The Bitwise Operator block performs the bitwise operation that you specify on one or more operands. Unlike logic operations of the Logical Operator block, bitwise operations treat the operands as a vector of bits rather than a single value.

You can select one of the following bitwise operations:

Bitwise Operation	Description
AND	TRUE if the corresponding bits are all TRUE
OR	TRUE if at least one of the corresponding bits is TRUE
NAND	TRUE if at least one of the corresponding bits is FALSE
NOR	TRUE if no corresponding bits are TRUE
XOR	TRUE if an odd number of corresponding bits are TRUE
NOT	TRUE if the input is FALSE (available only for single input)

Restrictions on Block Operations

The Bitwise Operator block does not support shift operations. For shift operations, use the Shift Arithmetic block.

When configured as a multi-input XOR gate, this block performs modulo-2 addition according to the IEEE® Standard for Logic Elements.

Behavior of Inputs and Outputs

The output data type, which the block inherits from the driving block, must represent zero exactly. Data types that satisfy this condition include signed and unsigned integer data types.

The size of the block output depends on the number of inputs, the vector size, and the operator you select:

- The NOT operator accepts only one input, which can be a scalar or a vector. If the input is a vector, the output is a vector of the same size containing the bitwise logical complements of the input vector elements.
- For a single vector input, the block applies the operation (except the NOT operator) to all elements of the vector.
 - If you do not specify a bit mask, the output is a scalar.
 - If you do specify a bit mask, the output is a vector.
- For two or more inputs, the block performs the operation between all of the inputs. If the inputs are vectors, the block performs the operation between corresponding elements of the vectors to produce a vector output.

Bit Mask Behavior

Block behavior changes depending on whether you use a bit mask.

If the Use bit mask check box is...	The block accepts...	And you specify...	By using...
Selected	One input	Bit Mask	Any valid MATLAB expression, such as $2^5+2^2+2^0$ for the bit mask 00100101
Not selected	Multiple inputs, all having the same base data type	Number of input ports	Any positive integer greater than 1

Tip You can also use character vectors to specify a hexadecimal bit mask such as {'FE73', '12AC'}.

Bit Set and Bit Clear Operations

You can use the bit mask to set or clear a bit on the input.

To perform a...	Set the Operator parameter to...	And create a bit mask with...
Bit set	OR	A 1 for each corresponding input bit that you want to set to 1
Bit clear	AND	A 0 for each corresponding input bit that you want to set to 0

Suppose you want to set the fourth bit of an 8-bit input vector. The bit mask would be 00010000, which you can specify as 2^4 for the **Bit Mask** parameter. To clear the bit, the bit mask would be 11101111, which you can specify as $2^7+2^6+2^5+2^3+2^2+2^1+2^0$ for the **Bit Mask** parameter.

Data Type Support

The Bitwise Operator block supports the following data types:

- Built-in integer
- Fixed point
- Boolean

The block does not support floating-point data types or enumerated data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Operator

Specify the bitwise logical operator for the block operands.

Use bit mask

Select to use the bit mask. Clearing this check box enables **Number of input ports** and disables **Bit Mask** and **Treat mask as**.

Number of input ports

Specify the number of inputs. The default value is 1.

Bit Mask

Specify the bit mask to associate with a single input. This parameter is available only when you select **Use bit mask**.

Tip Do not use a mask greater than 53 bits. Otherwise, an error message appears during simulation.

Treat mask as

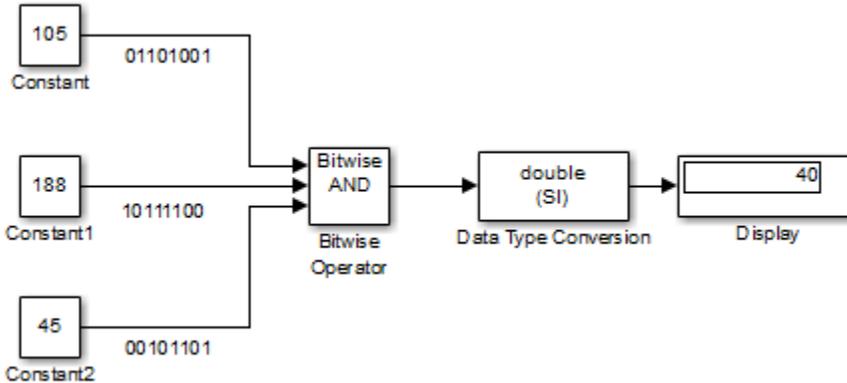
Specify whether to treat the mask as a real-world value or a stored integer. This parameter is available only when you select **Use bit mask**.

The encoding scheme is $V = SQ + B$, as described in “Scaling” (Fixed-Point Designer) in the Fixed-Point Designer documentation. **Real World Value** treats the mask as V . **Stored Integer** treats the mask as Q .

Examples

Unsigned Inputs for the Bitwise Operator Block

The following model shows how the Bitwise Operator block works for unsigned inputs.

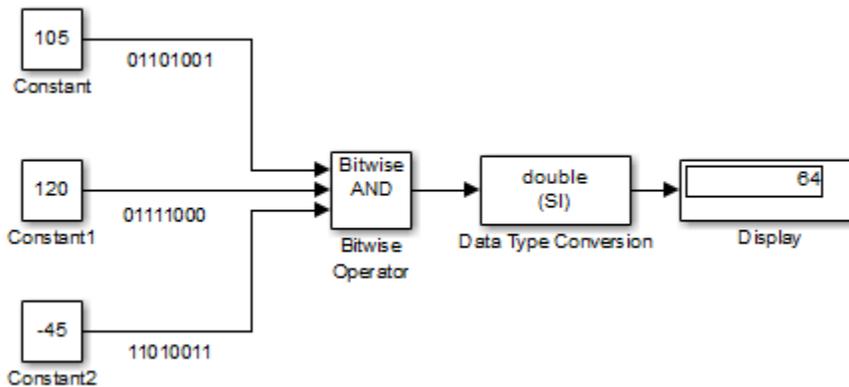


Each Constant block outputs an 8-bit unsigned integer (`uint8`). To determine the binary value of each Constant block output, use the `dec2bin` function. The results for all logic operations appear in the next table.

Operation	Binary Value	Decimal Value
AND	00101000	40
OR	11111101	253
NAND	11010111	215
NOR	00000010	2
XOR	11111000	248
NOT	N/A	N/A

Signed Inputs for the Bitwise Operator Block

The following model shows how the Bitwise Operator block works for signed inputs.



Each Constant block outputs an 8-bit signed integer (`int8`). To determine the binary value of each Constant block output, use the `dec2bin` function. The results for all logic operations appear in the next table.

Operation	Binary Value	Decimal Value
AND	01000000	64
OR	11111011	-5
NAND	10111111	-65
NOR	00000100	4
XOR	11000010	-62
NOT	N/A	N/A

Characteristics

Data Types	Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No

Code Generation	Yes
-----------------	-----

See Also

Logical Operator

Introduced before R2006a

Block Support Table

View data type support for Simulink blocks



Library

Model-Wide Utilities

Description

The Block Support Table block helps you access a table that lists the data types that Simulink blocks support. Double-click the block to view the table.

Data Type Support

Not applicable

Parameters

Not applicable

Characteristics

Data Types	Not applicable
Multidimensional Signals	No

Variable-Size Signals	No
Code Generation	Yes

Alternatives

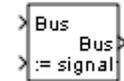
To access the information in the Block Support Table, you can enter `showblockdatatypetable` at the MATLAB command prompt.

Introduced in R2007b

Bus Assignment

Replace specified bus elements

Library: Simulink / Signal Routing



Description

The Bus Assignment block assigns the values of a signal to bus elements. Use a Bus Assignment block to change bus element values without adding Bus Selector and Bus Creator blocks that select bus elements and reassemble them into a bus.

Connect the bus signal to the first input port. To other input ports, connect one or more signals whose values you want to assign to a bus element. Use the Block Parameters dialog box to specify the bus elements to be replaced. The block displays an assignment input port for each such element. For an example of a model that uses a Bus Assignment block, see “Assign Signal Values to a Bus”.

By default, Simulink repairs broken selections in the Bus Assignment Block Parameters dialog boxes that are due to upstream bus hierarchy changes. Simulink generates a warning to highlight that it modified the model. To prevent Simulink from making these repairs automatically, in the **Model Configuration Parameters > Diagnostics > Connectivity** pane, set the “Repair bus selections” diagnostic to `Error without repair`.

Limitations

When using arrays of buses with a Bus Assignment block, these limitations apply:

- You can assign or replace a subbus that is an array of buses. However, the nested bus cannot be nested inside an array of buses.

- To replace a signal in an array of buses, use a Selector block to select the index for the bus element that you want to use with the Bus Assignment block. Then, use that selected bus element with the Bus Assignment block.

Ports

Input

Bus — Accept bus signal for bus element value assignment

real or complex values of any data type supported by Simulink

Input bus signals can have real or complex values of any data type supported by Simulink, including bus objects, arrays of buses, fixed-point, and enumerated data types. For details about data types, see Simulink, “Data Types Supported by Simulink”.

The signal connected to the assignment port must have the same structure, data type, and sample time as the bus element to which it corresponds. You can use a Rate Transition block to change the sample time of an individual signal or signals in a bus, to include the signal or bus in a nonvirtual bus. See “Virtual and Nonvirtual Buses” for more information.

:= — Accept signals whose value are assigned to bus elements

real or complex values of any data type supported by Simulink

Assignment input ports can accept signals can have real or complex values of any data type supported by Simulink, including bus objects, arrays of buses, fixed-point, and enumerated data types. You cannot use the Bus Assignment block to replace a bus that is nested within an array of buses. For details about data types, see Simulink, “Data Types Supported by Simulink”.

The Bus Assignment block assigns signals connected to its assignment input ports to specified elements of the bus connected to its bus input port. The block replaces the signals previously assigned to those elements. The change does not affect the composition of the bus; it affects only the values of the signals themselves. Signals not replaced are unaffected by the replacement of other signals.

Output

Bus — Output bus signal

virtual or nonvirtual bus

Bus that includes the assigned bus element values and the values of the bus elements of the input bus that you did not assign values to.

Parameters

Signals in the bus — Bus element signals of input bus

list of signal names

List of the bus element signals of the input bus signal. An arrow next to a signal name indicates that the input signal is a bus. To display the signals in an input bus, click the arrow.

Click any item in the list to select it. To find the source of the selected signal, click **Find**. Simulink opens and highlights the system containing the signal source. To move the currently selected signal into the adjacent list of signals to be assigned values (see **Signals that are being assigned** below), click **Select>>**. To refresh the display to reflect modifications to the bus connected to the block, click **Refresh**.

Filter by name — Filter set of displayed signals

text

Specify a search term to use for filtering a long list of input signals. Do not enclose the search term in quotation marks. The filter does a partial string search. To access filtering options, including using a regular expression for specifying the search term, click the  button on the right of the **Filter by name** edit box.

Enable regular expression — Filter set of displayed input signals

off (default) | on

Enable the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** edit box displays all signals whose names end with a lowercase `t` (and their immediate parents). For details, see “Regular Expressions” (MATLAB).

Dependencies

To access this parameter, click the  button on the right of the **Filter by name** edit box.

Show filtered results as a flat list — Filter set of displayed input signals
off (default) | on

By default, the list displays as a tree list of filtered signals, based on the search text in the **Filter by name** edit box. To use a flat list format that uses dot notation to reflect the hierarchy of bus signals, select this parameter.

Dependencies

To access this parameter, click the  button on the right of the **Filter by name** edit box

Signals that are being assigned — Bus element signals to be assigned
list of signal names

Names of bus elements to be assigned values. This block displays an assignment input port for each bus element in this list. The label of the corresponding input port contains the name of the element. You can order the signals by using the **Up**, **Down**, or **Remove**. Port connectivity is maintained when you change the signal order.

If an input bus no longer contains a bus element, three question marks (???) appear before the name of that bus element. The reason for this event is that the bus has changed since the last time you refreshed the Bus Assignment block input and bus element assignment lists. To address this issue, either modify the bus to include a signal of the specified name or remove the name from the list of bus elements designated to be assigned values.

Programmatic Use

Block Parameter:OutputSignals

Type:character vector

Values: 'signal1' | 'signal2'

Default:none

Model Examples

Block Characteristics

Data Types	double single Boolean base integer fixed point enumerated bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes

Extended Capabilities

C/C++ Code Generation

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

Actual data type or capability support depends on block implementation.

HDL Code Generation

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

Fixed-Point Conversion

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

Actual data type or capability support depends on block implementation.

See Also

Bus Creator | Bus Selector

Topics

“Getting Started with Buses”

“Combine Buses into an Array of Buses”

“Composite Signals”

“Buses and Libraries”

Introduced before R2006a

Bus Creator

Create bus signal from input signals

Library: Simulink / Commonly Used Blocks
Simulink / Signal Routing



Description

The Bus Creator block combines a set of signals into a bus. To bundle a group of signals with a Bus Creator block, set the block parameter **Number of inputs** to the number of signals in the group. The block displays the number of inport ports that you specify. Connect to the resulting input ports the signals that you want to group.

You can connect any type of signal to the inputs, including other bus signals. To access individual signals in a bus signal, connect the output port of the block to a Bus Selector block port.

The signals in the bus are ordered from the top input port to the bottom input port. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations. To rearrange the signals in the output bus signal, use buttons such as **Up** or **Down** buttons.

Simulink hides the name of a Bus Creator block when you copy it from the Simulink library to a model.

Tip For models that include bus signals composed of many bus elements that feed subsystems, consider using the In Bus Element and Out Bus Element blocks. You can use these bus element port blocks instead of Inport with Bus Selector blocks for inputs, and Outport with Bus Creator blocks for outputs. These bus element port blocks:

- Reduce signal line complexity and clutter in a block diagram.
- Make it easier to change the interface incrementally.

- Allow access to a bus element closer to the point of usage, avoiding the use of a Bus Selector and Goto block configuration.
-

Ports

Input

Port_1 — Accept signal to include in bus

real or complex values of any data type supported by Simulink

Input signals can have real or complex values of any data type supported by Simulink, including bus objects, arrays of buses, fixed-point, and enumerated data types. For details about data types, see Simulink, “Data Types Supported by Simulink”.

Output

Port_1 — Output bus signal

virtual or nonvirtual bus

Bus that combines the input signals.

Parameters

Number of inputs — Number of input ports

2 (default) | integer

Number of inputs, not fewer than two. Increasing the number of connected ports adds empty ports below the connected ports. Before you simulate the model, make sure an input signal is connected to each input port.

Tip As you draw a new signal line close to input side of a virtual Bus Creator block, if all input ports are already connected, the :

- Adds another input port to the Bus Creator block
- Updates the **Number of inputs** parameter

- Add to the list of bus signals a signal name for the new signal
-

Programmatic Use

Block Parameter:Inputs

Type:character vector

Values:integer greater than or equal to 2

Default:' 2 '

Signals in the bus — Input signals

list of signal names

List of input signals to combine into a bus signal. An arrow next to a signal name indicates that the input signal is a bus. To display the signals in an input bus, click the arrow. For information about working with the signals in the list, see “Bus Creation Using Bus Creator Blocks” .

Filter by name — Filter set of displayed input signals

text

Specify a search term to filter a long list of input signals. Do not enclose the search term in quotation marks. The filter does a partial string search. To access filtering options, including using a regular expression to specify the search term, click the  button on the right side of the **Filter by name** edit box.

Enable regular expression — Filter set of displayed input signals

off (default) | on

Enable the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** edit box displays all signals whose names end with a lowercase `t` (and their immediate parents). For details, see “Regular Expressions” (MATLAB).

Dependencies

To access this parameter, click the  button on the right side of the **Filter by name** edit box.

Show filtered results as a flat list — Filter set of displayed input signals

off (default) | on

By default, a tree list displays the filtered signals, based on the search text in the **Filter by name** edit box. To use a flat list format that uses dot notation to reflect the hierarchy of bus signals, select this parameter.

Dependencies

To access this parameter, click the  button on the right-hand side of the **Filter by name** edit box

Output data type — Data type of output signal

`{'Inherit: auto'} (default) | 'Bus: <object name>' | <datatype expression>`

Data type of the output bus signal.

Determine whether you want the Bus Creator block to output a virtual or nonvirtual bus.

- For a virtual bus, use the **Output data type** parameter default (`Inherit: auto`) or set the parameter to specify a bus object using `Bus: <object name>`.
- For a nonvirtual bus, set the **Output data type** parameter to specify a bus object using `Bus: <object name>` and click **Output as nonvirtual bus**.

If you select `'Bus: <object name>'`, specify a bus object in the edit box. The bus object must be in the base workspace when you perform an update diagram or simulate the model. To define a bus object using the Bus Editor, click **Show data type assistant** and then click **Edit**. For details, see “Create Bus Objects with the Bus Editor”.

If you select `'<data type expression>'`, specify an expression that evaluates to a data type. Do not specify a bus object as the expression.

Programmatic Use

Block Parameter:`OutDataTypeStr`

Type:character vector

Values:`{'Inherit: auto'} | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'boolean' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>' | 'Bus: <object name>'`

Default:`'inherit: auto'`

Require input signal names to match signals above — Require input signal names to match names listed in dialog box

`off (default) | on`

To check that the input signal names match the signal names in the Bus Creator Block Parameters dialog box, enable this parameter (the default setting).

Tip The **Require input signal names to match signals above** parameter might be removed in a future release. To enforce strong data typing, consider using the **Override bus signal names from inputs** parameter. If you select **Override bus signal names from inputs**, the **Require input signal names to match signals above** setting is ignored.

Programmatic Use

Block Parameter: OutDataTypeStr

Type: character vector

Values: 'on' | 'off'

Default: 'on'

Rename selected signal — Change name of currently selected signal

' ' (default) | character vector

Specify a name for the currently selected input signal. See “Signal Names and Labels” for guidelines for signal names.

Dependencies

The **Require input signal names to match signals above** parameter must be enabled.

Override bus signal names from inputs — Override bus signal names from input signals

off (default) | on

By default, the Bus Creator block overrides bus signal names from inputs. To inherit bus signal names from a bus object, clear this parameter. Clearing the parameter:

- Enforces strong data typing.
- Avoids having to enter a signal name multiple times: in the bus object and in the model. Entering the name multiple times can accidentally create signal name mismatches.
- Supports the array of buses requirement to have consistent signal names across array elements.

Alternatively, you can enforce strong data typing and also check that input signal names match the bus object element names.

- Select the **Override bus signal names from inputs** check box.
- Set the **Element name mismatch** parameter to `error`.

Dependencies

To display this parameter, the **Output data type** parameter must be set to a bus object.

Programmatic Use

Block Parameter:`InheritFromInputs`

Type:character vector

Values:'on' | 'off'

Default:'on'

Output as nonvirtual bus — Output nonvirtual bus

on (default) | off

Output a nonvirtual bus from the Bus Creator block. All signals in a nonvirtual bus must have the same sample time, even if the elements of the associated bus object specify inherited sample times. Any bus operation resulting in a nonvirtual bus that violates this requirement generates an error. To include in a nonvirtual bus a signal or bus that has a different sample time than the other input signals, use a Rate Transition block to change the sample time that signal. For details, see “Specify Bus Signal Sample Times”.

Enable this parameter to generate code that uses a C structure to define the structure of the bus signal output by this block.

Dependencies

The **Output data type** parameter must be set to a bus object.

Programmatic Use

Block Parameter:`NonVirtualBus`

Type:character vector

Values:'on' | 'off'

Default:'on'

Model Examples

Buses in Simulink

Block Characteristics

Data Types	double single Boolean base integer fixed point enumerated bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes

Extended Capabilities

C/C++ Code Generation

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

Actual data type or capability support depends on block implementation.

HDL Code Generation

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

Fixed-Point Conversion

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

Actual data type or capability support depends on block implementation.

See Also

Bus Assignment | Bus Selector | Bus to Vector | Out Bus Element

Topics

“Getting Started with Buses”

“Simplify Subsystem Bus Interfaces”
“Nest Buses”
“Bus-Capable Blocks”
“Assign Signal Values to a Bus”
“Composite Signals”
“Specify Bus Signal Sample Times”

Introduced before R2006a

Bus Selector

Select signals from incoming bus

Library: Simulink / Commonly Used Blocks
Simulink / Signal Routing



Description

The Bus Selector block outputs a specified subset of the elements of the bus at its input. The block can output the specified elements as separate signals or as a new bus.

By default, Simulink implicitly converts a nonbus signal to a bus signal to support connecting the signal to a Bus Selector block. To prevent Simulink from performing that conversion, set the “Non-bus signals treated as bus signals” diagnostic to warning or error.

When the block outputs multiple elements, it outputs each element from a separate port from top to bottom of the block. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.

In the Simulink Editor, as you draw a new signal line close to output side of a Bus Selector block and all output ports are already connected, Simulink Editor:

- Adds a port
- Prompts you to specify the signal to be selected

You cannot use this automatic port addition approach in either of these cases if:

- There is no bus input signal connected to the Bus Selector block.
- You do not specify a signal in response to the prompt that appears when you draw a signal line close to the Bus Selector block icon.
- You select the **Output as bus** parameter.

Tip For models that include bus signals composed of many bus elements that feed subsystems, consider using the In Bus Element and Out Bus Element blocks. You can use these bus element port blocks instead of Inport with Bus Selector blocks for inputs, and Outport with Bus Creator blocks for outputs. These bus element port blocks:

- Reduce signal line complexity and clutter in a block diagram.
 - Make it easier to change the interface incrementally.
 - Allow access to a bus element closer to the point of usage, avoiding the use of a Bus Selector and Goto block configuration.
-

Ports

Input

Port_1 — Accept bus signal to select signals from

real or complex values of any data type supported by Simulink except for arrays of buses

Input bus signals can have real or complex values of any data type supported by Simulink except for arrays of buses. To work with an array of buses signal, select with a Selector block the index for the bus element that you want to use with the Bus Selector block. Then use that selected bus element with the Bus Selector block.

Output

Port_1 — Output selected bus elements of input bus

nonbus signal (default) | virtual bus

Output selected bus element signals of an input bus signal.

When the block outputs separate elements, it outputs each element from a separate port from top to bottom of the block. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.

If you select the **Output as bus** parameter, the output bus is virtual. To produce nonvirtual bus output, insert a Signal Conversion block after the Bus Selector block. Set the Signal Conversion block **Output** parameter to `Nonvirtual bus` and use a `Simulink.Bus` bus object for the **Data type** parameter. For an example, see the Signal Conversion documentation.

Parameters

Signals in the bus — Element signals in input bus

list of signal names

List of bus element signals of the input bus, from which to select signals to output. To select a signal to output, click the signal in the list and then click **Select>>**.

To refresh the display to reflect modifications to the bus connected to the block, click **Refresh**.

To find the source of a signal entering the block, select the signal in the list and click **Find**. The Simulink software highlights the signal source in the block diagram.

Programmatic Use

Block Parameter:InputSignals

Type:matrix

Values:signal name

Default:{ ' [] ' }

Filter by name — Filter set of displayed input signals

text

Specify a search term to use for filtering a long list of input signals. Do not enclose the search term in quotation marks. The filter does a partial string search. To access filtering options, including using a regular expression to specify the search term, click the  button on the right side of the **Filter by name** edit box.

Enable regular expression — Filter set of displayed input signals

off (default) | on

Enable the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** edit box displays all signals whose names end with a lowercase `t` (and their immediate parents). For details, see “Regular Expressions” (MATLAB).

Dependencies

To access this parameter, click the  button on the right side of the **Filter by name** edit box.

Show filtered results as a flat list — Filter set of displayed input signals

off (default) | on

By default, a tree displays the filtered signals, based on the search text in the **Filter by name** edit box. Select this parameter to use a flat list format that uses dot notation to reflect the hierarchy of bus signals.

Dependencies

To access this parameter, click the  button on the right side of the **Filter by name** edit box

Selected signals — Selected bus elements signals

list of signal names

If an output signal listed in the **Selected signals** list box is not an input to the Bus Selector block, the signal name starts with three question marks (???).

You can change the list by using the **Up**, **Down**, and **Remove** buttons. To save your changes, click **Apply**. You can select multiple contiguous signals to move or remove. You cannot rearrange leaf signals within a bus. For example, you can move bus signal `Bus1` up or down in the list, but you cannot reorder any of the bus elements of `Bus1`. Port connectivity is maintained when you change the signal order.

Programmatic Use

Block Parameter:`OutputSignals`

Type:character vector

Values:character vector in the form of `'signal1', signal2'`

Default:none

Output as bus — Output selected elements as bus

on (default) | off

By default, the block outputs the selected elements as standalone signals, each from an output port that is labeled with the corresponding bus element name. To output the selected bus element signals as a bus, select this parameter.

The output bus is virtual. To produce nonvirtual bus output, insert a Signal Conversion block after the Bus Selector block. Set the Signal Conversion block **Output** parameter to `Nonvirtual bus` and use a `Simulink.Bus` bus object for the **Data type** parameter. For an example, see the Signal Conversion documentation.

When the **Selected signals** list includes only one signal and you enable **Output as bus**, then if the selected signal is:

- A nonbus signal, it is treated as a nonbus signal (it is not wrapped in a bus).
- A bus signal, then the output is that bus signal.

Programmatic Use

Block Parameter:OutputAsBus

Type:character vector

Values:'on' | 'off'

Default:'off'

Model Examples

Buses in Simulink

Block Characteristics

Data Types	double single Boolean base integer fixed point enumerated bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes

Extended Capabilities

C/C++ Code Generation

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

Actual data type or capability support depends on block implementation.

HDL Code Generation

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

Fixed-Point Conversion

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

Actual data type or capability support depends on block implementation.

See Also

[Bus Assignment](#) | [Bus Creator](#) | [Bus to Vector](#) | [Out Bus Element](#)

Topics

[“Getting Started with Buses”](#)

[“Simplify Subsystem Bus Interfaces”](#)

[“Nest Buses”](#)

[“Bus-Capable Blocks”](#)

[“Composite Signals”](#)

Introduced before R2006a

Bus to Vector

Convert virtual bus to vector

Library: Simulink / Signal Attributes



Description

The Bus to Vector block converts a virtual bus signal to a vector signal. The input bus signal must consist of scalar, 1-D, or either row or column vectors having the same data type, signal type, and sampling mode. If the input bus contains row or column vectors, this block outputs a row or column vector, respectively; otherwise, it outputs a 1-D array.

Use the block only to replace an implicit bus-to-vector conversion with an equivalent explicit conversion. To identify and correct buses used as vectors without manually inserting Bus to Vector blocks, you can use the Simulink Model Advisor “**Check bus signals treated as vectors**” on page 8-29 check. Alternatively, you can use the `Simulink.BlockDiagram.addBusToVector` function, which automatically inserts Bus to Vector blocks wherever needed.

Note If you use **Save As** for a model in a version of the Simulink product before R2007a, Simulink replaces each Bus to Vector block with a null subsystem that outputs nothing. Before you can use the model, reconnect or otherwise correct each signal that used to contain a Bus to Vector block but now is interrupted by a null subsystem.

Ports

Input

Port_1 — Accept signal to convert to vector
virtual bus | nonbus signal

An input virtual bus signal must consist of scalar, 1-D, or either row or column vectors having the same data type, signal type, and sampling mode. If the input is a nonbus signal, the block does no conversion.

Output

Port_1 — Output vector signal

vector | 1-D array

Output a vector signal, based on input bus signal. If the input bus contains row or column vectors, the block outputs a row or column vector, respectively. Otherwise, it outputs a 1-D array.

Parameters

This block has no user-accessible parameters.

Model Examples

Block Characteristics

Data Types	double single Boolean base integer fixed point enumerated bus
Multidimensional Signals	No
Variable-Size Signals	No

Extended Capabilities

C/C++ Code Generation

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

Actual data type or capability support depends on block implementation.

HDL Code Generation

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

Fixed-Point Conversion

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

Actual data type or capability support depends on block implementation.

See Also

Bus Creator | Bus Selector | Data Type Conversion |
`Simulink.BlockDiagram.addBusToVector`

Topics

“Correct Buses Used as Vectors”
“Composite Signals”

Introduced before R2006a

Callback Button

Execute MATLAB code based on user input

Library: Simulink / Dashboard

A rectangular button with rounded corners and a light gray border, containing the text "Callback Button" in a sans-serif font.

Description

The Callback Button block executes MATLAB code based on user input. The Callback Button block reacts to clicks and presses from the user. You can specify separate code to execute for each action. The Callback Button block repeats the code specified for the press action at a specified rate as long as you continue to press the button.

The Callback Button block registers a click when you release the left mouse button with the cursor on the Callback Button. The code for a press executes when you click the Callback Button and hold for the specified **Press Delay**.

Double-clicking the Callback Button block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Parameters

Button Text — Text on button

'Callback Button' (default) | character array

Text that appears on the button.

Mouse Action — Mouse action to run code

'ClickFcn' (default) | 'PressFcn'

Mouse action that causes the MATLAB code to execute.

- Select `ClickFcn` to view and edit the code that executes when the Callback Button block is clicked.

- Select `PressFcn` to view and edit the code that executes when the button is pressed.

MATLAB Code — Code to execute based on user input

empty (default) | MATLAB Code

MATLAB code that executes based on user input.

Dependency

The MATLAB code displayed depends on the Mouse Action parameter selection.

- Select `ClickFcn` to view and edit the code that executes when the Callback Button block is clicked.
- Select `PressFcn` to view and edit the code that executes when the button is pressed.

Press Delay (ms) — Time to hold button for press

500 (default) | scalar

Amount of time required to cause the `PressFcn` code to execute.

Dependency

Press Delay (ms) is only visible when `PressFcn` is selected as the Mouse Action.

Repeat Interval (ms) — Time interval to repeat PressFcn code

0 (default) | scalar

Time interval after which the `PressFcn` code executes again if the Callback Button block is still pressed.

Dependency

Repeat Interval (ms) is only visible when `PressFcn` is selected as the Mouse Action.

Model Examples

See Also

Push Button

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2017b

Check Box

Select parameter or variable value

Library: Simulink / Dashboard



Description

The Check Box block allows you to set the value of a parameter or variable during simulation by checking or clearing the box. Use the Check Box block with other Dashboard blocks to create an interactive dashboard for your model.

Double-clicking the Check Box block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Connection

Dashboard block controls connect to variables and block parameters. To connect a Dashboard block to a variable or parameter, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting blocks of interest in your model.
- 3 Mark the option button next to the parameter or variable you want to adjust during simulation.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined using the variable `engine(1)` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

Parameters

Connection — Select a block parameter to connect

signal connection options

Select the block parameter to control using the **Connection** table. To open the dialog box, double-click the block. Then, populate the **Connection** table by selecting a block in your model. Select the variable parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram with **Ctrl+D**.

States

Value (Unchecked) — Value when unchecked

0 (default) | scalar

Value assigned to the connected parameter when the Check Box block is not checked.

Value (Checked) — Value when checked

0 (default) | scalar

Value assigned to the connected parameter when the Check Box block is checked.

Label — **Block label position**

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

See Also

Rocker Switch | Rotary Switch | Slider Switch | Toggle Switch

Topics

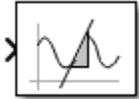
“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2017b

Check Discrete Gradient

Check that absolute value of difference between successive samples of discrete signal is less than upper bound



Library

Model Verification

Description

The Check Discrete Gradient block checks each signal element at its input to determine whether the absolute value of the difference between successive samples of the element is less than an upper bound. Use the block parameter dialog box to specify the value of the upper bound (1 by default). If the verification condition is true, the block does nothing. Otherwise, the block halts the simulation, by default, and displays an error in the Diagnostic Viewer.

The **Model Verification block enabling** setting in the Configuration Parameters dialog box lets you enable or disable all model verification blocks, including Check Discrete Gradient blocks, in a model.

The Check Discrete Gradient block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to physically remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Note For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Data Type Support

The Check Discrete Gradient block accepts `single`, `double`, `int8`, `int16`, and `int32` input signals of any dimensions. This block also supports fixed-point data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Maximum gradient

Specify the upper bound on the gradient of the discrete input signal.

Enable assertion

Clearing this check box disables the Check Discrete Gradient block, that is, causes the model to behave as if the block did not exist. The **Model Verification block enabling** setting in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Discrete Gradient blocks, regardless of the setting of this option.

Simulation callback when assertion fails

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Stop simulation when assertion fails

Selecting this check box causes the Check Discrete Gradient block to halt the simulation when the block's output is zero and the Simulink software displays an error. Otherwise, the Simulink software displays a warning and continues the simulation.

Output assertion signal

Selecting this check box causes the Check Discrete Gradient block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is `Boolean` if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

Specify the type of icon used to display this block in a block diagram: either `graphic` or `text`. The `graphic` option displays a graphical representation of the assertion condition on the icon. The `text` option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

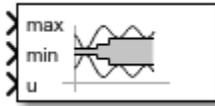
Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Check Dynamic Gap

Check that gap of possibly varying width occurs in range of signal's amplitudes



Library

Model Verification

Description

The Check Dynamic Gap block checks that a gap of possibly varying width occurs in the range of a signal's amplitudes. The test signal is the signal connected to the input labeled *sig*. The inputs labeled *min* and *max* specify the lower and upper bounds of the dynamic gap, respectively. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

The Check Dynamic Gap block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to physically remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Note For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Data Type Support

The Check Dynamic Gap block accepts input signals of any dimensions and of any numeric data type that Simulink supports. All three input signals must have the same

dimension and data type. If the inputs are nonscalar, the block checks each element of the input test signal to the corresponding elements of the reference signals.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Enable assertion

Clearing this check box disables the Check Dynamic Gap block, that is, causes the model to behave as if the block did not exist. The **Model Verification block enabling** setting in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Dynamic Gap blocks, regardless of the setting of this option.

Simulation callback when assertion fails

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Stop simulation when assertion fails

Selecting this check box causes the Check Dynamic Gap block to halt the simulation when the block's output is zero and the Simulink software displays an error. Otherwise, the Simulink software displays a warning and continues the simulation.

Output assertion signal

Selecting this check box causes the Check Dynamic Gap block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is `Boolean` if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

Specify the type of icon used to display this block in a block diagram: either `graphic` or `text`. The `graphic` option displays a graphical representation of the assertion condition on the icon. The `text` option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

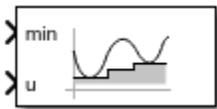
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Check Dynamic Lower Bound

Check that one signal is always less than another signal



Library

Model Verification

Description

The Check Dynamic Lower Bound block checks that the amplitude of a reference signal is less than the amplitude of a test signal at the current time step. The test signal is the signal connected to the input labeled *sig*. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

The Check Dynamic Lower Bound block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to physically remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Note For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Data Type Support

The Check Dynamic Lower Bound block accepts input signals of any numeric data type that Simulink supports. The test and the reference signals must have the same

dimensions and data type. If the inputs are nonscalar, the block checks each element of the input test signal to the corresponding elements of the reference signal.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Enable assertion

Clearing this check box disables the Check Dynamic Lower Bound block, that is, causes the model to behave as if the block did not exist. The **Model Verification block enabling** setting in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks, including Check Dynamic Lower Bound blocks, in a model regardless of the setting of this option.

Simulation callback when assertion fails

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Stop simulation when assertion fails

Selecting this check box causes the Check Dynamic Lower Bound block to halt the simulation when the block's output is zero and the Simulink software displays an error. Otherwise, the Simulink software displays a warning and continues the simulation.

Output assertion signal

Selecting this check box causes the Check Dynamic Lower Bound block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is `Boolean` if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

Specify the type of icon used to display this block in a block diagram: either `graphic` or `text`. The `graphic` option displays a graphical representation of the assertion condition on the icon. The `text` option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

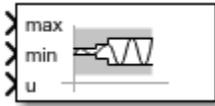
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Check Dynamic Range

Check that signal falls inside range of amplitudes that varies from time step to time step



Library

Model Verification

Description

The Check Dynamic Range block checks that a test signal falls inside a range of amplitudes at each time step. The width of the range can vary from time step to time step. The input labeled *sig* is the test signal. The inputs labeled *min* and *max* are the lower and upper bounds of the valid range at the current time step. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

The Check Dynamic Range block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to physically remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Note For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Data Type Support

The Check Dynamic Range block accepts input signals of any dimensions and of any numeric data type that Simulink supports. All three input signals must have the same dimension and data type. If the inputs are nonscalar, the block checks each element of the input test signal to the corresponding elements of the reference signals.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Enable assertion

Clearing this check box disables the Check Dynamic Range block, that is, causes the model to behave as if the block did not exist. The **Model Verification block enabling** setting in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Dynamic Range blocks, regardless of the setting of this option.

Simulation callback when assertion fails

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Stop simulation when assertion fails

Selecting this check box causes the Check Dynamic Range block to halt the simulation when the block's output is zero and the Simulink software display an error. Otherwise, the Simulink software displays a warning and continues the simulation.

Output assertion signal

Selecting this check box causes the Check Dynamic Range block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is `Boolean` if you selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

Specify the type of icon used to display this block in a block diagram: either `graphic` or `text`. The `graphic` option displays a graphical representation of the assertion

condition on the icon. The `text` option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

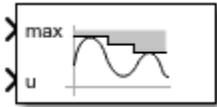
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Check Dynamic Upper Bound

Check that one signal is always greater than another signal



Library

Model Verification

Description

The Check Dynamic Upper Bound block checks that the amplitude of a reference signal is greater than the amplitude of a test signal at the current time step. The test signal is the signal connected to the input labeled *sig*. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

The Check Dynamic Upper Bound block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error-checking off by disabling the verification blocks. You do not have to physically remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Note For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Data Type Support

The Check Dynamic Upper Bound block accepts input signals of any dimensions and of any numeric data type that Simulink supports. The test and the reference signals must

have the same dimensions and data type. If the inputs are nonscalar, the block compares each element of the input test signal to the corresponding elements of the reference signal.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Enable assertion

Clearing this check box disables the Check Dynamic Upper Bound block, that is, causes the model to behave as if the block did not exist. The **Model Verification block enabling** setting in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks, including Check Dynamic Upper Bound blocks, in a model regardless of the setting of this option.

Simulation callback when assertion fails

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Stop simulation when assertion fails

Selecting this check box causes the Check Dynamic Upper Bound block to halt the simulation when the block's output is zero and the Simulink software displays an error. Otherwise, the Simulink software displays a warning and continues the simulation.

Output assertion signal

Selecting this check box causes the Check Dynamic Upper Bound block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is `Boolean` if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

Specify the type of icon used to display this block in a block diagram: either `graphic` or `text`. The `graphic` option displays a graphical representation of the assertion condition on the icon. The `text` option displays a mathematical expression that

represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

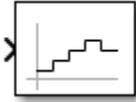
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Check Input Resolution

Check that input signal has specified resolution



Library

Model Verification

Description

The Check Input Resolution block checks whether the input signal has a specified scalar or vector resolution (see **Resolution** in “Parameters” on page 1-132). If the resolution is a scalar, the input signal must be a multiple of the resolution within a $10e-3$ tolerance. If the resolution is a vector, the input signal must equal an element of the resolution vector. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

The Check Input Resolution block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to physically remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Note For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Data Type Support

The Check Input Resolution block accepts input signals of data type `double` and of any dimension. If the input signal is nonscalar, the block checks the resolution of each element of the input test signal.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Resolution

Specify the resolution that the input signal must have.

Enable assertion

Clearing this check box disables the Check Input Resolution block, that is, causes the model to behave as if the block did not exist. The **Model Verification block enabling** setting in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Input Resolution blocks, regardless of the setting of this option.

Simulation callback when assertion fails

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Stop simulation when assertion fails

Selecting this check box causes the Check Input Resolution block to halt the simulation when the block's output is zero and the Simulink software displays an error. Otherwise, the Simulink software displays a warning and continues the simulation.

Output assertion signal

Selecting this check box causes the Check Input Resolution block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is `Boolean` if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

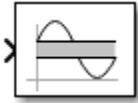
Characteristics

Data Types	Double
Sample Time	Inherited from driving block
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Check Static Gap

Check that gap exists in signal's range of amplitudes



Library

Model Verification

Description

The Check Static Gap block checks that each element of the input signal is less than (or optionally equal to) a static lower bound or greater than (or optionally equal to) a static upper bound at the current time step. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

The Check Static Gap block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to physically remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Note For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Data Type Support

The Check Static Gap block accepts input signals of any dimensions and of any numeric data type that Simulink supports.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Upper bound

Specify the upper bound of the gap in the input signal's range of amplitudes.

Inclusive upper bound

Selecting this check box specifies that the gap includes the upper bound.

Lower bound

Specify the lower bound of the gap in the input signal's range of amplitudes.

Inclusive lower bound

Selecting this check box specifies that the gap includes the lower bound.

Enable assertion

Clearing this check box disables the Check Static Gap block, that is, causes the model to behave as if the block did not exist. The **Model Verification block enabling** setting in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Static Gap blocks, regardless of the setting of this option.

Simulation callback when assertion fails

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Stop simulation when assertion fails

Selecting this check box causes the Check Static Gap block to halt the simulation when the block's output is zero and the Simulink software displays an error. Otherwise, the Simulink software displays a warning and continues the simulation.

Output assertion signal

Selecting this check box causes the Check Static Gap block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is `Boolean` if you have selected the **Implement logic signals as Boolean data** check box on the **Optimization** pane of the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

Specify the type of icon used to display this block in a block diagram: either `graphic` or `text`. The `graphic` option displays a graphical representation of the assertion condition on the icon. The `text` option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

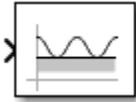
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Check Static Lower Bound

Check that signal is greater than (or optionally equal to) static lower bound



Library

Model Verification

Description

The Check Static Lower Bound block checks that each element of the input signal is greater than (or optionally equal to) a specified lower bound at the current time step. Use the block parameter dialog box to specify the value of the lower bound and whether the lower bound is inclusive. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

The Check Static Lower Bound block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to physically remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Note For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Data Type Support

The Check Static Lower Bound block accepts input signals of any dimensions and of any numeric data type that Simulink supports.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Lower bound

Specify the lower bound on the range of amplitudes that the input signal can have.

Inclusive boundary

Selecting this check box makes the range of valid input amplitudes include the lower bound.

Enable assertion

Clearing this check box disables the Check Static Lower Bound block, that is, causes the model to behave as if the block did not exist. The **Model Verification block enabling** setting in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Static Lower Bound blocks, regardless of the setting of this option.

Simulation callback when assertion fails

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Stop simulation when assertion fails

Selecting this check box causes the Check Static Lower Bound block to halt the simulation when the block's output is zero and the Simulink software displays an error. Otherwise, the Simulink software displays a warning and continues the simulation.

Output assertion signal

Selecting this check box causes the Check Static Lower Bound block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is `Boolean` if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

Specify the type of icon used to display this block in a block diagram: either `graphic` or `text`. The `graphic` option displays a graphical representation of the assertion

condition on the icon. The `text` option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

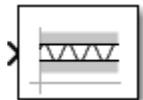
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Check Static Range

Check that signal falls inside fixed range of amplitudes



Library

Model Verification

Description

The Check Static Range block checks that each element of the input signal falls inside the same range of amplitudes at each time step. Use the block parameter dialog box to specify the upper and lower bounds of the valid amplitude range and whether the range includes the bounds. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

The Check Static Range block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to physically remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Note For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Data Type Support

The Check Static Range block accepts input signals of any dimensions and of any numeric data type that Simulink supports.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Upper bound

Specify the upper bound of the range of valid input signal amplitudes.

Inclusive upper bound

Selecting this check box specifies that the valid signal range includes the upper bound.

Lower bound

Specify the lower bound of the range of valid input signal amplitudes.

Inclusive lower bound

Selecting this check box specifies that the valid signal range includes the lower bound.

Enable assertion

Clearing this check box disables the Check Static Range block, that is, causes the model to behave as if the block did not exist. The **Model Verification block enabling** setting in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Static Range blocks, regardless of the setting of this option.

Simulation callback when assertion fails

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Stop simulation when assertion fails

Selecting this check box causes the Check Static Range block to halt the simulation when the block's output is zero and the Simulink software display an error. Otherwise, the Simulink software displays a warning and continues the simulation.

Output assertion signal

Selecting this check box causes the Check Static Range block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is `Boolean` if you have selected the

Implement logic signals as Boolean data check box in the Configuration Parameters dialog box. Otherwise the data type of the output signal is double.

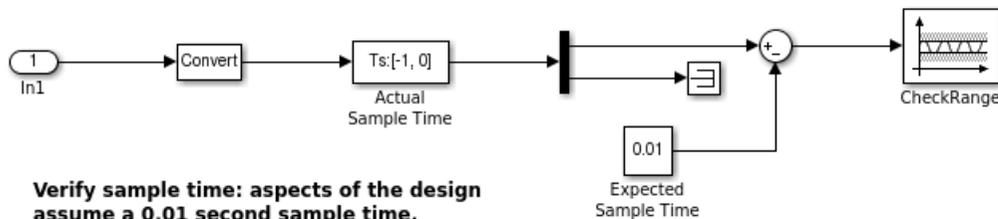
Select icon type

Specify the type of icon used to display this block in a block diagram: either `graphic` or `text`. The `graphic` option displays a graphical representation of the assertion condition on the icon. The `text` option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

Examples

The `sldemo_fuelsys` model shows how you can use the Check Static Range block to verify that the sample time is consistent throughout the model.

The Check Static Range block appears in the `sldemo_fuelsys/fuel_rate_control/validate_sample_time` subsystem.



Verify sample time: aspects of the design assume a 0.01 second sample time.

Characteristics

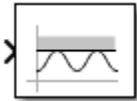
Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No

Code Generation	Yes
-----------------	-----

Introduced before R2006a

Check Static Upper Bound

Check that signal is less than (or optionally equal to) static upper bound



Library

Model Verification

Description

The Check Static Upper Bound block checks that each element of the input signal is less than (or optionally equal to) a specified upper bound at the current time step. Use the block parameter dialog box to specify the value of the upper bound and whether the bound is inclusive. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

The Check Static Upper Bound block and its companion blocks in the Model Verification library are intended to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to physically remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Note For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Data Type Support

The Check Static Upper Bound block accepts input signals of any dimensions and of any numeric data type that Simulink supports.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Upper bound

Specify the upper bound on the range of amplitudes that the input signal can have.

Inclusive boundary

Selecting this check box makes the range of valid input amplitudes include the upper bound.

Enable assertion

Clearing this check box disables the Check Static Upper Bound block, that is, causes the model to behave as if the block did not exist. The **Model Verification block enabling** setting in the Configuration Parameters dialog box allows you to enable or disable all model verification blocks in a model, including Check Static Upper Bound blocks, regardless of the setting of this option.

Simulation callback when assertion fails

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Stop simulation when assertion fails

Selecting this check box causes the Check Static Upper Bound block to halt the simulation when the block's output is zero and the Simulink software displays an error. Otherwise, the Simulink software displays a warning and continues the simulation.

Output assertion signal

Selecting this check box causes the Check Static Upper Bound block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is `Boolean` if you have selected the **Implement logic signals as Boolean data** check box in the Configuration Parameters dialog box. Otherwise the data type of the output signal is `double`.

Select icon type

Specify the type of icon used to display this block in a block diagram: either `graphic` or `text`. The `graphic` option displays a graphical representation of the assertion

condition on the icon. The `text` option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Chirp Signal

Generate sine wave with increasing frequency



Library

Sources

Description

The Chirp Signal block generates a sine wave whose frequency increases at a linear rate with time. You can use this block for spectral analysis of nonlinear systems. The block generates a scalar or vector output.

The parameters, **Initial frequency**, **Target time**, and **Frequency at target time**, determine the block's output. You can specify any or all of these variables as scalars or arrays. All the parameters specified as arrays must have the same dimensions. The block expands scalar parameters to have the same dimensions as the array parameters. The block output has the same dimensions as the parameters unless you select the **Interpret vector parameters as 1-D** check box. If you select this check box and the parameters are row or column vectors, the block outputs a vector (1-D array) signal.

The following limitations apply to the Chirp Signal block:

- The start time of the simulation must be 0. To confirm this value, go to the **Solver** pane in the Configuration Parameters dialog box and view the **Start time** field.
- Suppose that you use a Chirp Signal block in an enabled subsystem. Whenever the subsystem is enabled, the block output matches what would appear if the subsystem were enabled throughout the simulation.

Data Type Support

The Chirp Signal block outputs a real-valued signal of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Initial frequency

The initial frequency of the signal, specified as a scalar or matrix value. The default is 0.1 Hz.

Target time

The time at which the frequency reaches the **Frequency at target time** parameter value, a scalar or matrix value. The frequency continues to change at the same rate after this time. The default is 100 seconds.

Frequency at target time

The frequency of the signal at the target time, a scalar or matrix value. The default is 1 Hz.

Interpret vector parameters as 1-D

If selected, column or row matrix values for the **Initial frequency**, **Target time**, and **Frequency at target time** parameters result in a vector output whose elements are the elements of the row or column. For more information, see “Determining the Output Dimensions of Source Blocks” in the Simulink documentation.

Characteristics

Data Types	Double
Sample Time	Continuous
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No

Code Generation	Yes
-----------------	-----

Introduced before R2006a

Clock

Display and provide simulation time



Library

Sources

Description

The Clock block outputs the current simulation time at each simulation step. This block is useful for other blocks that need the simulation time.

When you need the current time within a discrete system, use the Digital Clock block.

Data Type Support

The Clock block outputs a real-valued signal of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Display time

Select this check box to display the current simulation time inside the Clock block icon.

Decimation

Specify a positive integer for the interval at which Simulink updates the Clock icon when you select **Display time**.

Suppose that the decimation is 1000. For a fixed integration step of 1 millisecond, the Clock icon updates at 1 second, 2 seconds, and so on.

Examples

The following Simulink examples show how to use the Clock block:

- `sldemo_tonegen_fixpt`
- `penddemo`

Characteristics

Data Types	Double
Sample Time	Continuous
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Digital Clock

Introduced before R2006a

Combinatorial Logic

Implement truth table



Library

Logic and Bit Operations

Description

The Combinatorial Logic block implements a standard truth table for modeling programmable logic arrays (PLAs), logic circuits, decision tables, and other Boolean expressions. You can use this block in conjunction with Memory blocks to implement finite-state machines or flip-flops.

You specify a matrix that defines all possible block outputs as the **Truth table** parameter. Each row of the matrix contains the output for a different combination of input elements. You must specify outputs for every combination of inputs. The number of columns is the number of block outputs.

The relationship between the number of inputs and the number of rows is:

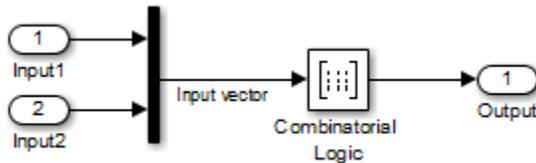
$$\text{number of rows} = 2^{\text{(number of inputs)}}$$

Simulink returns a row of the matrix by computing the row's index from the input vector elements. Simulink computes the index by building a binary number where input vector elements having zero values are 0 and elements having nonzero values are 1, then adding 1 to the result. For an input vector, u , of m elements:

$$\text{row index} = 1 + u(m) * 2^0 + u(m-1) * 2^1 + \dots + u(1) * 2^{m-1}$$

Two-Input AND Logic

This example builds a two-input AND function, which returns 1 when both input elements are 1, and 0 otherwise. To implement this function, specify the **Truth table** parameter value as [0; 0; 0; 1]. The portion of the model that provides the inputs to and the output from the Combinatorial Logic block might look like this:



The following table indicates the combination of inputs that generate each output. The input signal labeled “Input 1” corresponds to the column in the table labeled Input 1. Similarly, the input signal “Input 2” corresponds to the column with the same name. The combination of these values determines the row of the Output column of the table that is passed as block output.

For example, if the input vector is [1 0], the input references the third row:

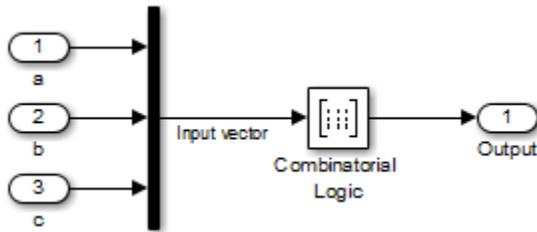
$$(2^{1*1} + 1)$$

The output value is 0.

Row	Input 1	Input 2	Output
1	0	0	0
2	0	1	0
3	1	0	0
4	1	1	1

Circuit Logic

This sample circuit has three inputs: the two bits (**a** and **b**) to be summed and a carry-in bit (**c**). It has two outputs: the carry-out bit (**c'**) and the sum bit (**s**).



The truth table and corresponding outputs for each combination of input values for this circuit appear in the following table.

Inputs			Outputs	
a	b	c	c'	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

To implement this adder with the Combinatorial Logic block, you enter the 8-by-2 matrix formed by columns **c'** and **s** as the **Truth table** parameter.

You can also implement sequential circuits (that is, circuits with states) with the Combinatorial Logic block by including an additional input for the state of the block and feeding the output of the block back into this state input.

Data Type Support

The type of signals accepted by a Combinatorial Logic block depends on whether you selected the Boolean logic signals option (see “Implement logic signals as Boolean data (vs. double)”). If this option is enabled, the block accepts real signals of type `Boolean` or `double`. The **Truth table** parameter can have Boolean values (0 or 1) of any data type,

including fixed-point data types. If the table contains non-Boolean values, the data type of the table must be `double`.

The type of the output is the same as that of the input except that the block outputs `double` if the input is `Boolean` and the truth table contains non-Boolean values.

If Boolean compatibility mode is disabled, the Combinatorial Logic block accepts only signals of type `Boolean`. The block outputs `double` if the truth table contains non-Boolean values of type `double`. Otherwise, the output is `Boolean`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Truth table

Specify the matrix of outputs. Each column corresponds to an element of the output vector and each row corresponds to a row of the truth table.

Sample time

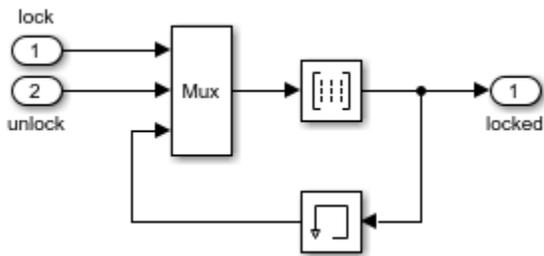
Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Examples

Usage with the Memory Block to Implement a Finite-State Machine

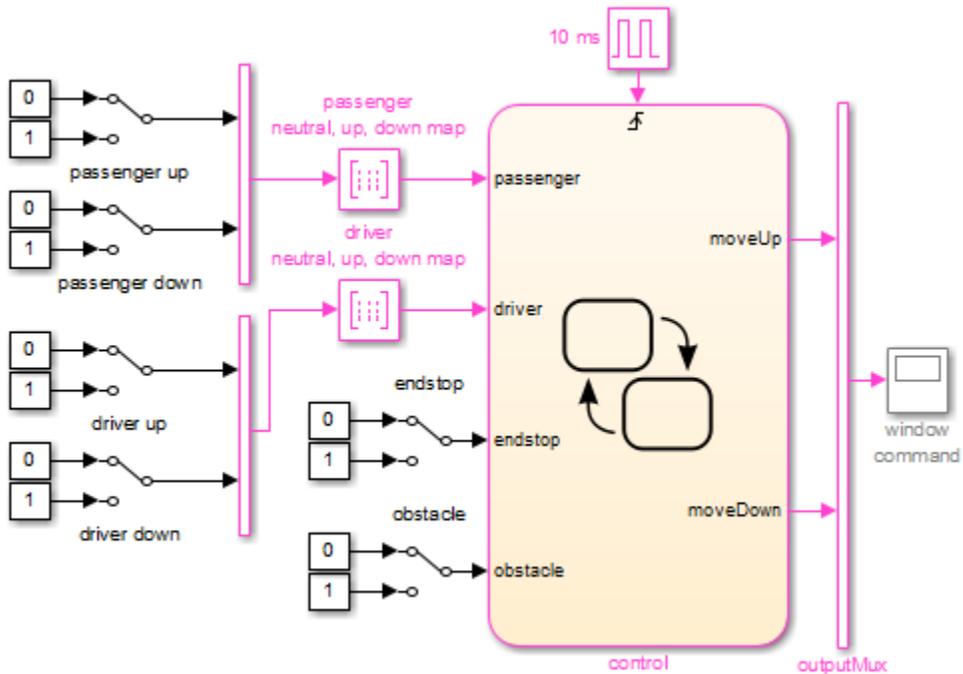
The `sldemo_clutch` model shows how you can use the Combinatorial Logic block with the Memory block to implement a finite-state machine.

The finite-state machine appears in the `Friction Mode Logic/Lockup FSM` subsystem.



Usage with a Stateflow Chart to Implement a Finite-State Machine

The powerwindow model shows how you can use two Combinatorial Logic blocks as inputs to a Stateflow chart to implement a finite-state machine.



Characteristics

Data Types	Double Boolean
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Combo Box

Select parameter value from drop-down menu

Library: Simulink / Dashboard



Description

The Combo Box block lets you set the value of a parameter to one of several values. You can define each selectable value and its label through the Combo Box block parameters. Use the Combo Box block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model.

Double-clicking the Combo Box block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Connection

Dashboard block controls connect to variables and block parameters. To connect a Dashboard block to a variable or parameter, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting blocks of interest in your model.
- 3 Mark the option button next to the parameter or variable you want to adjust during simulation.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined using the variable `engine(1)` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

Parameters

Connection — Select a block parameter to connect

signal connection options

Select the block parameter to control using the **Connection** table. To open the dialog box, double-click the block. Then, populate the **Connection** table by selecting a block in your model. Select the variable parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram with **Ctrl+D**.

States

value — Values for selected option

0/1/2 (default) | scalar

Values assigned to the connected parameter when you select the option with the corresponding **Label**. Click the **+** button to add options.

Label — Option labels

'Label1'/'Label2'/'Label3' (default) | character vector

Label for each option. You can use the **Label** to display the value the connected parameter takes when the switch is positioned at the bottom, or you can enter a text label.

Example: `Gain = 1`

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

See Also

Radio Button | Rotary Switch

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2017b

Compare To Constant

Determine how signal compares to specified constant



Library

Logic and Bit Operations

Description

The Compare To Constant block compares an input signal to a constant. Specify the constant in the **Constant value** parameter. Specify how the input is compared to the constant value with the **Operator** parameter. The **Operator** parameter can have the following values:

- == — Determine whether the input is equal to the specified constant.
- ~= — Determine whether the input is not equal to the specified constant.
- < — Determine whether the input is less than the specified constant.
- <= — Determine whether the input is less than or equal to the specified constant.
- > — Determine whether the input is greater than the specified constant.
- >= — Determine whether the input is greater than or equal to the specified constant.

The output is 0 if the comparison is false, and 1 if it is true.

Data Type Support

The Compare To Constant block accepts inputs of any data type that Simulink supports, including fixed-point and enumerated data types. The block first converts its **Constant**

value parameter to the input data type, and then performs the specified operation. The block output is `uint8` or `boolean` as specified by the **Output data type** parameter.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Operator

Specify how the input is compared to the constant value, as discussed in Description on page 1-161.

Constant value

Specify the constant value to which the input is compared.

Output data type

Specify the data type of the output, `boolean` or `uint8`.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

See Also

Compare To Zero

Introduced before R2006a

Compare To Zero

Determine how signal compares to zero



Library

Logic and Bit Operations

Description

The Compare To Zero block compares an input signal to zero. Specify how the input is compared to zero with the **Operator** parameter. The **Operator** parameter can have the following values:

- == — Determine whether the input is equal to zero.
- ~= — Determine whether the input is not equal to zero.
- < — Determine whether the input is less than zero.
- <= — Determine whether the input is less than or equal to zero.
- > — Determine whether the input is greater than zero.
- >= — Determine whether the input is greater than or equal to zero.

The output is 0 if the comparison is false, and 1 if it is true.

Data Type Support

The Compare To Zero block accepts inputs of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The block output is `uint8` or `boolean`, depending on your selection for the **Output data type** parameter. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Tip If the input data type cannot represent zero, parameter overflow occurs. To detect this overflow, go to the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box and set **Parameters > Detect overflow** to `warning` or `error`.

In this case, the block compares the input signal to the *ground value* of the input data type. For example, if you have an input signal of type `fixdt(0, 8, 2^0, 10)`, the input data type can represent unsigned 8-bit integers from 10 to 265 due to a bias of 10. The ground value is 10, instead of 0.

Parameters

Operator

Specify how the input is compared to zero, as discussed in Description on page 1-164.

Output data type

Specify the data type of the output, `boolean` or `uint8`.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	Yes

Variable-Size Signals	Yes
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

See Also

Compare To Constant

Introduced before R2006a

Complex to Magnitude-Angle

Compute magnitude and/or phase angle of complex signal



Library

Math Operations

Description

The Complex to Magnitude-Angle block accepts a complex-valued signal of type `double` or `single`. It outputs the magnitude and/or phase angle of the input signal, depending on the setting of the **Output** parameter. The outputs are real values of the same data type as the block input. The input can be an array of complex signals, in which case the output signals are also arrays. The magnitude signal array contains the magnitudes of the corresponding complex input elements. The angle output similarly contains the angles of the input elements.

Data Type Support

See the preceding description.

Parameters

Output

Determines the output of this block. Choose from the following values: `Magnitude` and `angle` (outputs the input signal's magnitude and phase angle in radians),

Magnitude (outputs the input's magnitude), Angle (outputs the input's phase angle in radians).

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Characteristics

Data Types	Double Single
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Complex to Real-Imag

Output real and imaginary parts of complex input signal



Library

Math Operations

Description

The Complex to Real-Imag block accepts a complex-valued signal of any data type that Simulink supports, including fixed-point data types. It outputs the real and/or imaginary part of the input signal, depending on the setting of the **Output** parameter. The real outputs are of the same data type as the complex input. The input can be an array (vector or matrix) of complex signals, in which case the output signals are arrays of the same dimensions. The real array contains the real parts of the corresponding complex input elements. The imaginary output similarly contains the imaginary parts of the input elements.

Data Type Support

See the preceding description. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Output

Determines the output of this block. Choose from the following values: `Real` and `imag` (outputs the input signal's real and imaginary parts), `Real` (outputs the input's real part), `Imag` (outputs the input's imaginary part).

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Configurable Subsystem

Represent any block selected from user-specified library of blocks



Library

Ports & Subsystems

Description

The Configurable Subsystem block represents one of a set of blocks contained in a specified library of blocks. The block's context menu lets you choose which block the configurable subsystem represents.

Configurable Subsystem blocks simplify creation of models that represent families of designs. For example, suppose that you want to model an automobile that offers a choice of engines. To model such a design, you would first create a library of models of the engine types available with the car. You would then use a Configurable Subsystem block in your car model to represent the choice of engines. To model a particular variant of the basic car design, a user need only choose the engine type, using the configurable engine block's dialog.

To create a configurable subsystem in a model, you must first create a library containing a master configurable subsystem and the blocks that it represents. You can then create configurable instances of the master subsystem by dragging copies of the master subsystem from the library and dropping them into models.

You can add any type of block to a master configurable subsystem library. Simulink derives the port names for the configurable subsystem by making a unique list from the port names of all the choices. However, Simulink uses default port names for non-subsystem block choices.

Note that you cannot break library links in a configurable subsystem because Simulink uses those links to reconfigure the subsystem when you choose a new configuration. Breaking links would be useful only if you do not intend to reconfigure the subsystem. In this case, you can replace the configurable subsystem with a nonconfigurable subsystem that implements the permanent configuration.

Creating a Master Configurable Subsystem

To create a master configurable subsystem:

- 1 Create a library of blocks representing the various configurations of the configurable subsystem.
- 2 Save the library.
- 3 Create an instance of the Configurable Subsystem block in the library.

To do this, drag a copy of the Configurable Subsystem block from the Simulink Ports & Subsystems library into the library you created in the previous step.

- 4 Display the Configurable Subsystem block dialog by double-clicking it. The dialog displays a list of the other blocks in the library.
- 5 Under **List of block choices** in the dialog box, select the blocks that represent the various configurations of the configurable subsystems you are creating.
- 6 Click the **OK** button to apply the changes and close the dialog box.
- 7 Select **Block Choice** from the Configurable Subsystem block's context menu.

The context menu displays a submenu listing the blocks that the subsystem can represent.

- 8 Select the block that you want the subsystem to represent by default.
- 9 Save the library.

Note If you add or remove blocks from a library, you must recreate any Configurable Subsystem blocks that use the library.

If you modify a library block that is the default block choice for a configurable subsystem, the change does not immediately propagate to the configurable subsystem. To propagate this change, do one of the following:

- Change the default block choice to another block in the subsystem, then change the default block choice back to the original block.

- Recreate the configurable subsystem block, including the selection of the updated block as the default block choice.

If a configurable subsystem in your model contains a broken link to a library block, editing the link and saving the model does not fix the broken link the next time you open the model. To fix a broken library link in your configurable subsystem, use one of the following approaches.

- Convert the configurable subsystem to a variant subsystem. Right-click the configurable subsystem, and select **Subsystem and Model Reference > Convert Subsystem to > Variant Subsystem**.
- Remove the library block from the master configurable subsystem library, add the library block back to the master configurable subsystem library, and then resave the master configurable subsystem library.

Creating an Instance of a Configurable Subsystem

To create an instance of a configurable subsystem in a model:

- 1 Open the library containing the master configurable subsystem.
- 2 Drag a copy of the master into the model.
- 3 Select **Block Choice** from the copy's context menu.
- 4 Select the block that you want the configurable subsystem to represent.

The instance of the configurable system displays the icon and parameter dialog box of the block that it represents.

Setting Instance Block Parameters

As with other blocks, you can use the parameter dialog box of a configurable subsystem instance to set the instance's parameters interactively and the `set_param` command to set the parameters from the MATLAB command line or in a MATLAB file. If you use `set_param`, you must specify the full path name of the configurable subsystem's current block choice as the first argument of `set_param`, for example:

```
curr_choice = get_param('mymod/myconfigsys', 'BlockChoice');  
curr_choice = ['mymod/myconfigsys/' curr_choice];  
set_param(curr_choice, 'MaskValues', ...);
```

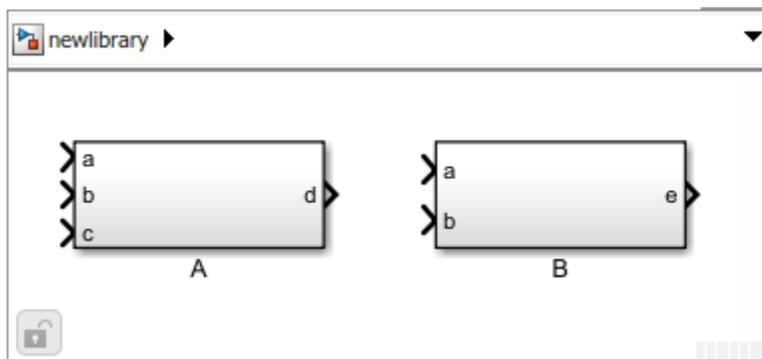
Mapping I/O Ports

A configurable subsystem displays a set of input and output ports corresponding to input and output ports in the selected library. Simulink uses the following rules to map library ports to Configurable Subsystem block ports:

- Map each uniquely named input/output port in the library to a separate input/output port of the same name on the Configurable Subsystem block.
- Map all identically named input/output ports in the library to the same input/output ports on the Configurable Subsystem block.
- Terminate any input/output port not used by the currently selected library block with a Terminator/Ground block.

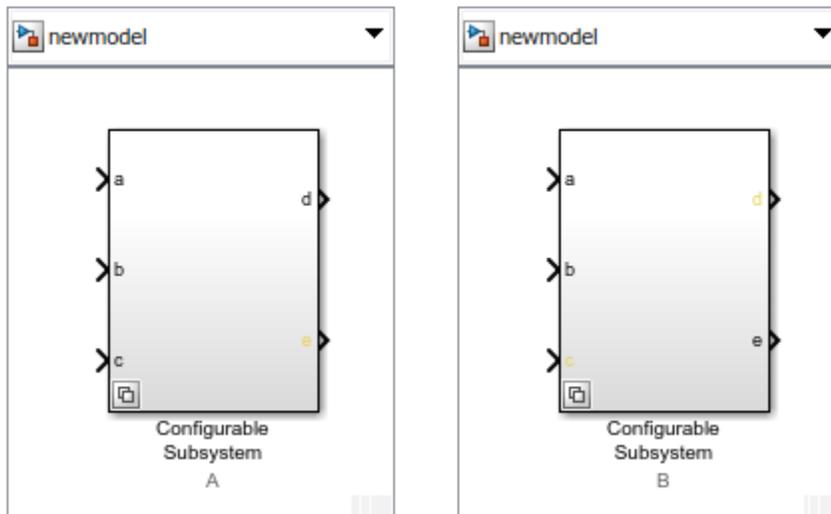
This mapping allows a user to change the library block represented by a Configurable Subsystem block without having to rewire connections to the Configurable Subsystem block.

For example, suppose that a library contains two blocks A and B and that block A has input ports labeled a, b, and c and an output port labeled d and that block B has input ports labeled a and b and an output port labeled e.



A Configurable Subsystem block based on this library would have three input ports labeled a, b, and c, respectively, and two output ports labeled d and e.

In this example, port a on the Configurable Subsystem block connects to port a of the selected library block no matter which block is selected. On the other hand, port c on the Configurable Subsystem block functions only if library block A is selected. Otherwise, it simply terminates.



Note A Configurable Subsystem block does not provide ports that correspond to non-I/O ports, such as the trigger and enable ports on triggered and enabled subsystems. Thus, you cannot use a Configurable Subsystem block directly to represent blocks that have such ports. You can do so indirectly, however, by wrapping such blocks in subsystem blocks that have input or output ports connected to the non-I/O ports.

Convert to Variant Subsystem

Right-click a configurable subsystem and select **Subsystems and Model Reference > Convert Subsystem To > Variant Subsystem**.

During conversion, Simulink performs the following operations:

- Replaces the Subsystem block with a Variant Subsystem block, preserving ports and connections.
- Adds the original subsystem as a variant choice in the Variant Subsystem block.
- Overrides the Variant Subsystem block to use the subsystem that was originally the active choice.
- Preserves links to libraries. For linked subsystems, Simulink adds the linked subsystem as a variant choice.

Simulink also preserves the subsystem block masks, and it copies the masks to the variant choice.

See Variant Subsystem for more information on variant choices.

Data Type Support

The Configurable Subsystem block accepts and outputs signals of the same types that are accepted or output by the block that it currently represents. The data types can be any that Simulink supports, including fixed-point data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

List of block choices

Select the blocks you want to include as members of the configurable subsystem. You can include user-defined subsystems as blocks.

Port names

Lists of input and output ports of member blocks. In the case of multiports, you can rearrange selected port positions by clicking the **Up** and **Down** buttons.

Characteristics

A Configurable Subsystem block has the characteristics of the block that it currently represents. Double-clicking the block opens the dialog box for the block that it currently represents.

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	Yes

Introduced before R2006a

Constant

Generate constant value



Library

Sources

Description

The Constant block generates a real or complex constant value.

The block generates scalar, vector, or matrix output, depending on:

- The dimensionality of the **Constant value** parameter
- The setting of the **Interpret vector parameters as 1-D** parameter

The output of the block has the same dimensions and elements as the **Constant value** parameter. If you specify for this parameter a vector that you want the block to interpret as a vector, select the **Interpret vector parameters as 1-D** parameter. Otherwise, if you specify a vector for the **Constant value** parameter, the block treats that vector as a matrix.

Data Type Support

By default, the Constant block outputs a signal whose data type and complexity are the same as those of the **Constant value** parameter. However, you can specify the output to be any data type that Simulink supports, including fixed-point and enumerated data types. The Enumerated Constant block can be more convenient than the Constant block

for outputting a constant enumerated value. You can also use a bus object as the output data type, which can help to simplify a model (see “Bus Support” on page 1-189 for details).

Note If you specify a bus object as the data type for this block, do not set the minimum and maximum values for bus data on the block. Simulink ignores these settings. Instead, set the minimum and maximum values for bus elements of the bus object specified as the data type. The values should be finite real double scalar.

For information on the Minimum and Maximum properties of a bus element, see `Simulink.BusElement`.

For more information about data type support, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Constant value

Specify the constant value output of the block.

Default: 1

Minimum: value from the **Output minimum** parameter

Maximum: value from the **Output maximum** parameter

- You can enter any expression that MATLAB evaluates as a matrix, including the Boolean keywords `true` and `false`.
- If you set the **Output data type** to be a bus object, you can specify either:
 - A full MATLAB structure corresponding to the bus object
 - 0 to indicate a structure corresponding to the ground value of the bus object

For details, see “Bus Support” on page 1-189.

- For non-bus data types, Simulink converts this parameter from its value data type to the specified output data type offline, using round toward nearest and saturation.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Interpret vector parameters as 1-D

Select this check box to output a vector of length N if the **Constant value** parameter evaluates to an N -element row or column vector.

Default: On

On

Outputs a vector of length N if the **Constant value** parameter evaluates to an N -element row or column vector. For example, the block outputs a matrix of dimension 1-by- N or N -by-1.

Off

Does not output a vector of length N if the **Constant value** parameter evaluates to an N -element row or column vector.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Sample time

Specify the interval between times that the Constant block output can change during simulation (for example, due to tuning the **Constant value** parameter).

Default: `inf`

This setting indicates that the block output can never change. This setting speeds simulation and generated code by avoiding the need to recompute the block output.

See “Specify Sample Time” in the online documentation for more information.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Output minimum

Lower value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Note If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum parameter for a bus element, see `Simulink.BusElement`.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMin

Type: character vector

Value: ' [] '

Default: ' [] '

Output maximum

Upper value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Note If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum parameter for a bus element, see `Simulink.BusElement`.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMax

Type: character vector

Value: ' [] '

Default: ' [] '

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting Slope and bias enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Output data type

Specify the output data type.

Default: Inherit: Inherit from 'Constant value'

Inherit: Inherit from 'Constant value'

Use data type of **Constant value**.

Inherit: Inherit via back propagation

Use data type of the driving block.

double

Output data type is double.

single

Output data type is single.

int8

Output data type is int8.

uint8

Output data type is uint8.

int16

Output data type is int16.

uint16

Output data type is uint16.

int32

Output data type is int32.

uint32

Output data type is uint32.

boolean

Output data type is boolean.

fixdt(1,16)

Output data type is fixed point fixdt(1,16).

fixdt(1,16,0)

Output data type is fixed point fixdt(1,16,0).

fixdt(1,16,2^0,0)

Output data type is fixed point fixdt(1,16,2^0,0).

Enum: <class name>

Use an enumerated data type, for example, Enum: BasicColors.

Bus: <object name>

Data type is a bus object.

<data type expression>

Data type is data type object, for example Simulink.NumericType.

Do not specify a bus object as the expression.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Control Signal Data Types” for more information.

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit from 'Constant value' (default)`
- `Inherit via back propagation`

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`

- uint16
- int32
- uint32
- boolean

Fixed point

Fixed-point data types.

Enumerated

Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.

Bus object

Bus object. Selecting `Bus` enables a **Bus object** parameter to the right, where you enter the name of a bus object that you want to use to define the structure of the bus. If you need to create or change a bus object, click **Edit** to the right of the **Bus object** field to open the Simulink Bus Editor. For details, see “Create Bus Objects with the Bus Editor”.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Do not specify a bus object as the expression.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data as signed.

`Unsigned`

Specify the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: `16`

Minimum: `0`

Maximum: `32`

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope and bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off



On

Locks the output data type setting for this block.



Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: `LockScale`

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Bus Support

Using Bus Objects as the Output Data Type

The Constant block supports nonvirtual buses as the output data type. If you use a bus object as the data type, set **Constant value** to 0 or a MATLAB structure that matches the bus object.

Using Structures for the Constant Value

The structure you specify must contain a value for every element of the bus represented by the bus object. The block output is a nonvirtual bus signal.

You can use the `Simulink.Bus.createMATLABStruct` to create a full structure that corresponds to a bus.

You can use `Simulink.Bus.createObject` to create a bus object from a MATLAB structure.

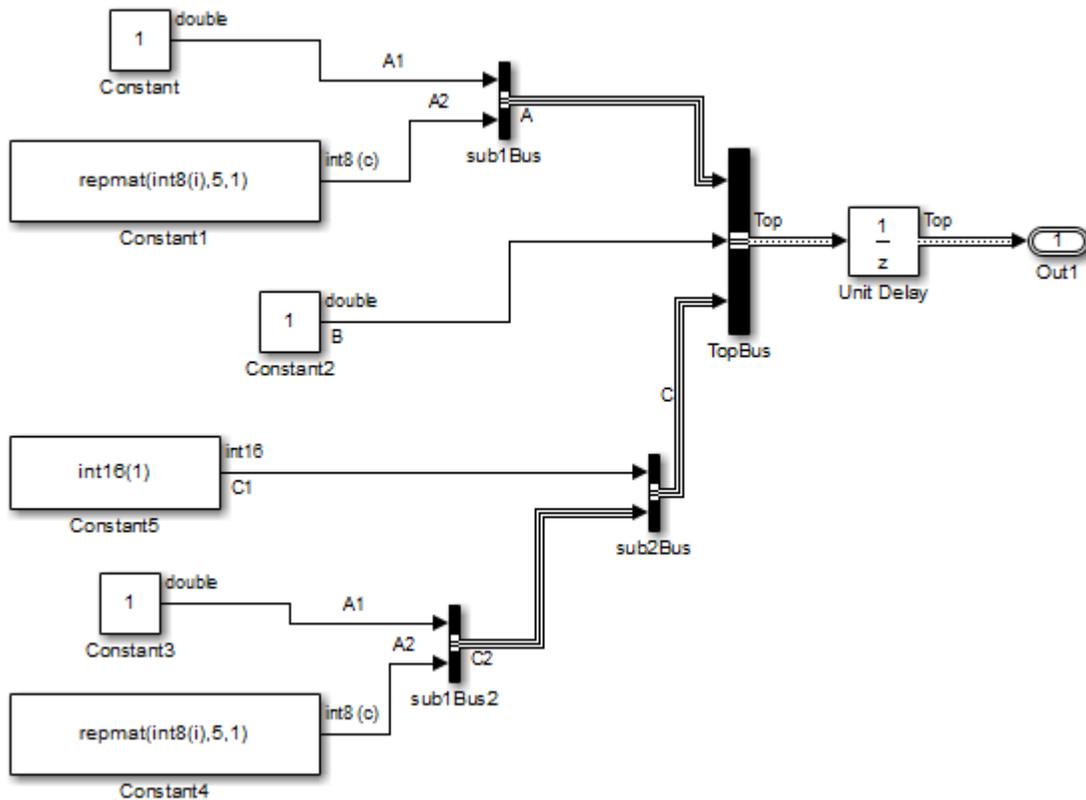
If the signal elements in the output bus use numeric data types other than `double`, you can specify the structure fields by using typed expressions such as `uint16(37)` or untyped expressions such as `37`. To control the field data types, you can use the bus object as the data type of a `Simulink.Parameter` object. To decide whether to use typed or untyped expressions, see “Control Data Types of Initial Condition Structure Fields”.

Example of Using a Bus Object for a Constant Block

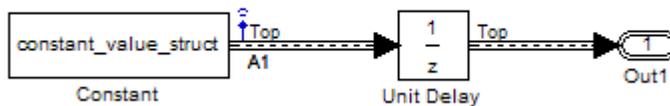
The following example illustrates how using a bus object as an output data type for a Constant block can help to simplify a model.

- 1 Open the `ex_busic` model and update it.

This model uses six Constant blocks. For details about the model, see “Create Partial Structures for Initialization”.



- Open the `ex_constantbus` model and update it. This model uses one Constant block that replaces the six Constant blocks in the `ex_basic` model.



- Simulate the `ex_constantbus` model. To verify that the output from the Constant block reflects the values from `constant_value_struct`, perform the next two steps.

- 4 At the MATLAB command line, examine the `constant_value_struct` structure that the Constant block uses for its **Constant value** parameter.

```
constant_value_struct

constant_value_struct =

    A: [1x1 struct]
    B: 5
    C: [1x1 struct]
```

- 5 Examine the logged data in the `logouts` variable, focusing on the B element of the A1 bus signal. The `constant_value_struct` structure sets the B element to 5.

```
logouts.get('A1').Values.B.Data
```

Group Constant Signals into an Array of Buses

You can use a Constant block to compactly represent multiple constant-valued signals as an array of buses. You can use this technique to reduce the number of signal lines in a model and the number of variables that the model uses, especially when the model repeats an algorithm with different parameter values.

To generate a constant-valued array of bus signals, use an array of MATLAB structures in a Constant block. The block output is an array of buses, and each field in the array of structures provides the simulation value for the corresponding signal element.

You can also use an array of structures to specify the `Value` property of a `Simulink.Parameter` object, and use the parameter object in a Constant block.

- 1 Open the example model `ex_constantbus_array`.

The variables `ParamBus` and `const_param_struct` appear in the base workspace. The variable `const_param_struct` contains a structure whose field names match the elements of the bus type that `ParamBus` defines.

- 2 Update the diagram to view the signal line widths.

The output of the Constant block is a single scalar bus of type `ParamBus`. The structure variable `const_param_struct` specifies the constant value in the block.

- 3 At the command prompt, create an array of two structures by copying the structure `const_param_struct`. Customize the field values by indexing into the individual structures in the array.

```

const_struct_array = ...
  [const_param_struct const_param_struct];

const_struct_array(2).Offset = 158;
const_struct_array(2).Gain = 3.83;
const_struct_array(2).Threshold = 1039.77

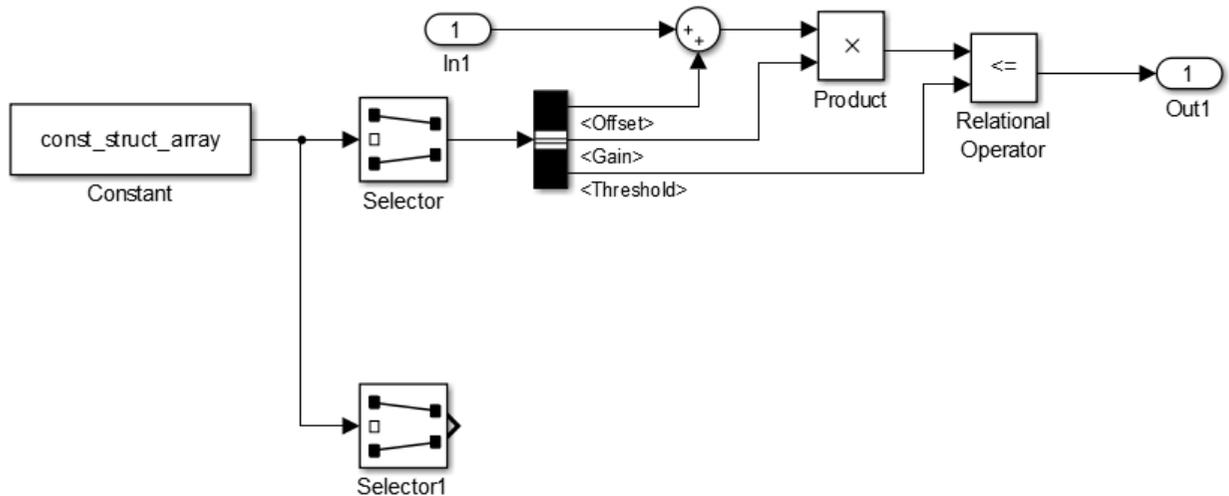
const_struct_array =

1x2 struct array with fields:

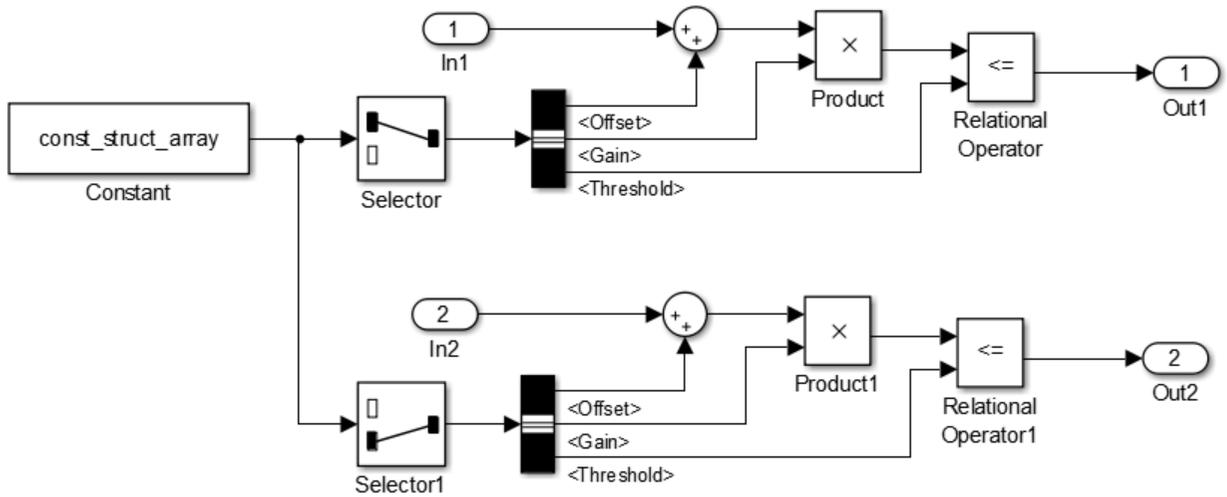
    Offset
    Gain
    Threshold

```

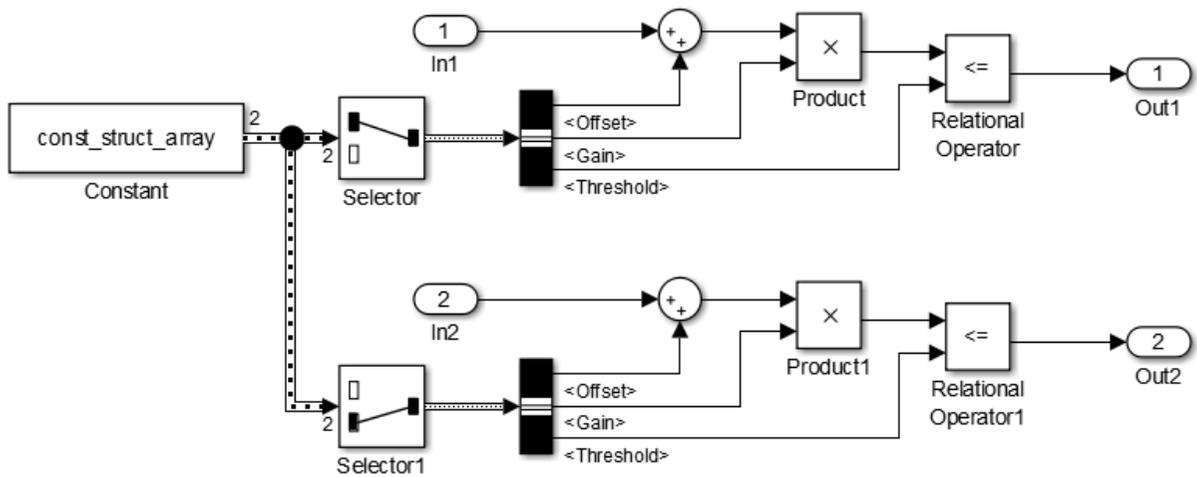
- 4 In the Constant block dialog box, set **Constant value** to `const_struct_array`.
- 5 Add two Selector blocks to the model, and connect the Constant block as shown.



- 6 In the Selector block dialog box, set **Index** to 1 and **Input port size** to 2.
- 7 In the Selector1 block dialog box, set **Index** to 2 and **Input port size** to 2.
- 8 Copy the block algorithm in the model, and connect the blocks as shown.



- 9 Update the diagram. The signal line width and style indicate that the output of the Constant block is an array of buses. The Selector blocks each extract one of the buses in the array.



Each copy of the algorithm uses the constant values provided by the corresponding structure in the variable `const_struct_array`.

To create an array of structures for a bus that uses a large hierarchy of signal elements, consider using the function `Simulink.Bus.createMATLABStruct`. You can also use this technique to create an array of structures if you do not have a scalar structure that you can copy.

Setting Configuration Parameters to Support Using a Bus Object Data Type

To enable the use of a bus object as an output data type, before you start a simulation, set **Configuration Parameters > Diagnostics > Data Validity > Advanced parameters > Underspecified initialization detection** to `Simplified`. For more information, see “Underspecified initialization detection”.

Examples

The following Simulink examples show how to use the Constant block:

- `sldemo_auto_climatecontrol`
- `sldemo_boiler`
- `sldemo_bounce`

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Specified in the Sample time parameter
Direct Feedthrough	N/A
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Enumerated Constant | `Simulink.Parameter`

Topics

“Set Block Parameter Values”

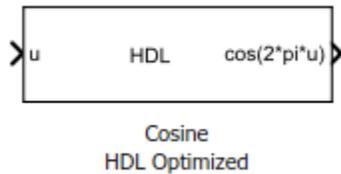
“Specify Initial Conditions for Bus Signals”

“When to Use Bus Objects”

Introduced before R2006a

Cosine HDL Optimized

Implement fixed-point cosine wave optimized for HDL code generation



Library

HDL Coder™ / Lookup Tables

Description

The Cosine HDL Optimized block implements a fixed-point cosine wave by using a lookup table method that exploits quarter-wave symmetry.

You define the number of lookup table points in the **Number of data points** parameter. The block implementation is most efficient for HDL code generation when you specify the lookup table data points to be (2^n) , where n is an integer. For information about the behavior of this block in HDL Coder, see Cosine HDL Optimized.

Depending on your selection of the **Output formula** parameter, the blocks can output these functions of the input signal:

- $\sin(2\pi u)$
- $\cos(2\pi u)$
- $\exp(i2\pi u)$
- $\sin(2\pi u)$ and $\cos(2\pi u)$

Use the **Table data type** parameter to specify the word length of the fixed-point output data type. The fraction length of the output is the output word length minus 2.

Data Type Support

The Cosine HDL Optimized block accepts signals of these data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The output of the block is a fixed-point data type.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Output formula

Select the signal(s) to output.

Number of data points

Specify the number of data points to retrieve from the lookup table. The implementation is most efficient when you specify the lookup table data points to be (2^n) , where n is an integer.

Table data type

Specify the table data type. You can specify an expression that evaluates to a data type, for example, `fixdt(1,16,0)`.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the table data type.

Show data type assistant

Select the mode of data type specification. If you select *Expression*, enter an expression that evaluates to a data type, for example, `fixdt(1,16,0)`.

If you select *Fixed point*, you can use the options in the **Data Type Assistant** to specify the fixed-point data type. In the *Fixed point* mode, you can choose binary

point scaling, and specify the signedness, word length, fraction length, and the data type override setting.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Sine HDL Optimized | Sine, Cosine | Trigonometric Function

Introduced in R2016b

Coulomb and Viscous Friction

Model discontinuity at zero, with linear gain elsewhere



Library

Discontinuities

Description

The Coulomb and Viscous Friction block models Coulomb (static) and viscous (dynamic) friction. The block models a discontinuity at zero and a linear gain otherwise.

The block output matches the MATLAB result for:

$$y = \text{sign}(x) .* (\text{Gain} .* \text{abs}(x) + \text{Offset})$$

where y is the output, x is the input, Gain is the signal gain for nonzero input values, and Offset is the Coulomb friction.

The block accepts one input and generates one output. The input can be a scalar, vector, or matrix with real and complex elements.

- For a scalar input, Gain and Offset can have dimensions that differ from the input. The output is a scalar, vector, or matrix depending on the dimensions of Gain and Offset .
- For a vector or matrix input, Gain and Offset must be scalar or have the same dimensions as the input. The output is a vector or matrix of the same dimensions as the input.

Data Type Support

The Coulomb and Viscous Friction block supports real inputs of the following data types:

- Floating point
- Built-in integer
- Fixed point

The block supports complex inputs only for floating-point data types, `double` and `single`. The output uses the same data type as the input.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Coulomb friction value

Specify the offset that applies to all input values.

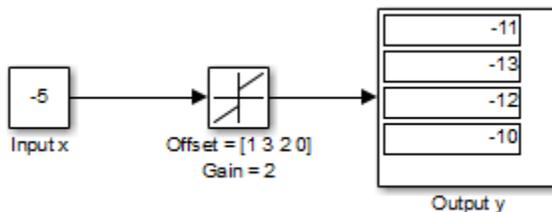
Coefficient of viscous friction

Specify the signal gain for nonzero input values.

Examples

Scalar Input

Suppose that you have the following model:



In this model, the block input x (-5) and Gain (2) are scalar values, but Offset is the vector [1 3 2 0]. Therefore, the block uses element-wise scalar expansion to compute the output.

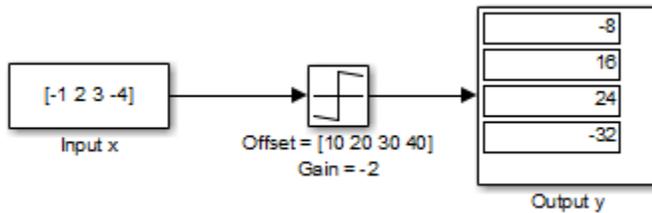
For example, for the first offset, 1, the output is calculated using the formula $y = \text{sign}(x) .* (\text{Gain} .* \text{abs}(x) + \text{Offset})$ as follows:

$$y = - * (2 * 5) + 1$$

$$y = -11$$

Vector Input

Suppose that you have the following model:



In this model, vector dimensions for block input x and Offset are the same.

Matrix Input

Suppose that you have the following model:



In this model, matrix dimensions for block input x and Offset are the same.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes
Code Generation	Yes

Introduced before R2006a

Counter Free-Running

Count up and overflow back to zero after reaching maximum value for specified number of bits



Library

Sources

Description

The Counter Free-Running block counts up until reaching the maximum value, $2^{Nbits} - 1$, where $Nbits$ is the number of bits. Then the counter overflows to zero and begins counting up again.

After overflow, the counter always initializes to zero. However, if you select the global doubles override, the Counter Free-Running block does not wrap back to zero.

This block does not report wrap on overflow warnings during simulation. To report these warnings, see the `Simulink.restoreDiagnostic` reference page. The block does report errors due to wrap on overflow.

Data Type Support

The Counter Free-Running block outputs an unsigned integer.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Number of Bits

Specify the number of bits.

When you use...	Such as...	The block counts up to...	Which is...
A positive integer	8	$2^8 - 1$	255
An unsigned integer expression	<code>uint8(8)</code>	<code>uint8(2^{uint8(8)} - 1)</code>	254

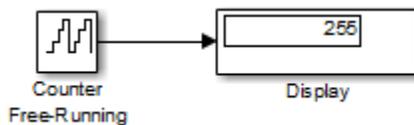
Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. See “Specify Sample Time” in the Simulink documentation.

Examples

Bit Specification Using a Positive Integer

Suppose that you have the following model:



The block parameters are:

Parameter	Setting
Number of Bits	8
Sample time	-1

The solver options for the model are:

Parameter	Setting
Stop time	255
Type	Fixed-step

Parameter	Setting
Solver	discrete (no continuous states)
Fixed-step size	1

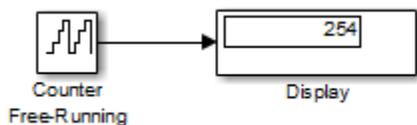
At $t = 255$, the counter reaches the maximum value:

$$2^8 - 1$$

If you change the stop time of the simulation to 256, the counter wraps to zero.

Bit Specification Using an Unsigned Integer Expression

Suppose that you have the following model:



The block parameters are:

Parameter	Setting
Number of Bits	uint8(8)
Sample time	-1

The solver options for the model are:

Parameter	Setting
Stop time	254
Type	Fixed-step
Solver	discrete (no continuous states)
Fixed-step size	1

At $t = 254$, the counter reaches the maximum value:

$$\text{uint8}(2^{\text{uint8}(8)} - 1)$$

If you change the stop time of the simulation to 255, the counter wraps to zero.

Characteristics

Data Types	Base Integer Fixed-Point
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Counter LimitedMATLAB Function

Introduced before R2006a

Counter Limited

Count up and wrap back to zero after outputting specified upper limit



Library

Sources

Description

The Counter Limited block counts up until the specified upper limit is reached. Then the counter wraps back to zero, and restarts counting up. The counter always initializes to zero.

You can specify the upper limit with the **Upper limit** parameter.

You can specify the sample time with the **Sample time** parameter. A **Sample time** of `-1` means that the sample time is inherited.

The output is an unsigned integer of 8, 16, or 32 bits, with the smallest number of bits needed to represent the upper limit.

This block does not report wrap on overflow warnings during simulation. To report these warnings, see the `Simulink.restoreDiagnostic` reference page. The block does report errors due to wrap on overflow.

Data Type Support

The Counter Limited block outputs an unsigned integer.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Upper limit

Specify the upper limit.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. See “Specify Sample Time” in the Simulink documentation.

Examples

The following Simulink examples show how to use the Counter Limited block:

- `sldemo_tonegen_fixpt`

Characteristics

Data Types	Boolean Base Integer Fixed-Point
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

Counter Free-Running

Introduced before R2006a

Dashboard Scope

Trace signals during simulation

Library: Simulink / Dashboard



Description

The Dashboard Scope block shows connected signals during simulation on a scope display. You can use the Dashboard Scope block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. The Dashboard Scope block provides a complete picture of a signal's behavior over the course of the simulation. Use the Dashboard Scope block to display real (not complex) signals of any data type that Simulink supports, including enumerated data types.

Connection

Dashboard blocks do not use ports to connect to signals. To connect a signal to a Dashboard block, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting signals of interest in your model.
- 3 Mark the option button next to the signal you want to display.
- 4 Click **Apply**.

Data Cursors

You can add data cursors to the Dashboard Scope to inspect the displayed signals. To add data cursors, right-click the Dashboard Scope block. Under the **Data Cursors** menu, select the number of cursors you want to add.

Zoom and Pan

You can also zoom and pan to inspect your signals. To change zoom and pan modes, right-click the Dashboard Scope block, and select the zoom or pan mode you want.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.
To save connections and properties, save the model file in the SLX format.
- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- If you turn off logging for a signal connected to a Dashboard block, the model stops sending data from that signal to the block. To view the signal again, reconnect the signal.
- External mode simulation does not support visualization Dashboard blocks.

Parameters

Connection — Select a signal to connect and display

signal connection options

Signal connection options. Populate the **Connection** table by selecting signals of interest in your model. Mark the check box next to the signal you want to display. Click **Apply** to connect the signal.

Time Span — Set horizontal axis span

auto (default) | scalar

A finite, real double, scalar value that sets the time span of the block's scope display.

When Time Span is specified as `auto`, the block sets its time span to the model's simulation stop time.

Min — Set y-axis minimum

-3 (default) | scalar

A finite, real, double, scalar value that sets the minimum of the vertical axis on the block's scope display.

The **Min** value must be less than the **Max** value.

Max — Set y-axis maximum

3 (default) | scalar

A finite, real, double, scalar value that sets the maximum of the vertical axis on the block's scope display.

The **Max** value must be greater than the **Min** value.

Legend — Set position of legend

'Top' (default) | 'Right' | 'Hide'

Options from the drop-down menu specify the position of the legend in the block's scope display. The legend shows the color chosen for each connected signal next to the signal's name.

Scale axes limits at stop — Autoscale axes limits when simulation finishes

on (default) | off

When on, performs a fit-to-view operation on the data displayed in the scope when the simulation stops.

Show "Double-click to connect" message — Show or hide message

on (default) | off

When on, shows instructional text if the block is not connected. When the block is not connected, you can turn this parameter off to hide the text.

Model Examples

See Also

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2015a

Data Store Memory

Define data store



Library

Signal Routing

Description

The Data Store Memory block defines and initializes a named shared data store, which is a memory region usable by Data Store Read and Data Store Write blocks that specify the same data store name.

The location of the Data Store Memory block that defines a data store determines which Data Store Read and Data Store Write blocks can access the data store:

- If the Data Store Memory block is in the *top-level system*, Data Store Read and Data Store Write blocks anywhere in the model can access the data store.
- If the Data Store Memory block is in a *subsystem*, Data Store Read and Data Store Write blocks in the same subsystem or in any subsystem below it in the model hierarchy can access the data store.

Data Store Read or Data Store Write blocks cannot access a Data Store Memory block that is either in a model that contains a Model block or in a referenced model.

Do not include a Data Store Memory block in a subsystem that a For Each Subsystem block represents.

Obtaining correct results from data stores requires ensuring that data store reads and writes occur in the expected order. For details, see:

- “Order Data Store Access”
- “Data Store Diagnostics”
- “Log Data Stores”

You can use `Simulink.Signal` objects in addition to, or instead of, Data Store Memory blocks to define data stores. A data store defined in the *base* workspace with a signal object is a *global* data store. Global data stores are accessible to every model, including all referenced models. See “Data Stores” for more information.

Data Type Support

The Data Store Memory block stores real or complex signals of these data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated
- Bus

The block does not support variable-size signals.

Note If you specify a bus object as the data type for this block, do not set the minimum and maximum values for bus data on the block. Simulink ignores these settings. Instead, set the minimum and maximum values for bus elements of the bus object specified as the data type. The values should be finite real double scalar.

For information on the Minimum and Maximum properties of a bus element, see `Simulink.BusElement`.

For more information, see “Data Types Supported by Simulink”.

You can use arrays of buses with a Data Store Memory block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Parameters

Data store name

Specify a name for the data store you are defining with this block. Data Store Read and Data Store Write blocks with the same name can read from, and write to, the data store initialized by this block. The name can represent a Data Store Memory block or a sign object defined to be a data store.

Rename All

Rename the data store everywhere the Data Store Read and Data Store Write blocks use it in a model.

You cannot use **Rename All** to rename a data store if you:

- Use a `Simulink.Signal` object in a workspace to control the code generated for the data store
- Use a `Simulink.Signal` object instead of a Data Store Memory block to define the data store

You must instead rename the corresponding `Simulink.Signal` object from Model Explorer. For an example, see “Rename Data Store Defined by Signal Object”.

Corresponding Data Store Read/Write blocks

List all the Data Store Read and Data Store Write blocks that have the same data store name as the current block, and that are in the current system or in any subsystem below it in the model hierarchy. Double-click a block in this list to highlight the block and bring it to the foreground.

Initial value

Specify the initial value or values of the data store. The **Minimum** parameter specifies the minimum value for this parameter, and the **Maximum** parameter specifies the maximum value.

If you specify a nonscalar value and set **Dimensions** to -1 (the default), the data store has the same dimensions as the array. Data that you write to the data store (by using Data Store Write blocks) must have these dimensions.

If you set the **Dimensions** parameter to a value other than -1 , the initial value dimensions must match the dimensions that you specify, unless the initial value is a scalar or a MATLAB structure. If you specify a scalar, each element of the data store

uses the scalar as the initial value. Use this technique to apply the same initial value (the scalar that you specify) to each element without manually matching the dimensions of the initial value with the dimensions of the data store.

To use this block to initialize a nonvirtual bus signal, specify the initial value as a MATLAB structure and set the model configuration parameter “Underspecified initialization detection” to `Simplified`. For more information about initializing nonvirtual bus signals using structures, see “Specify Initial Conditions for Bus Signals”.

Minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the `Minimum` property of a bus element, see `Simulink.BusElement`.

Simulink uses the minimum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For

information on the Maximum property of a bus element, see `Simulink.BusElement`.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Data type

Specify the output data type. You can set it to:

- A rule that inherits a data type (for example, `Inherit: auto`)
- The name of a built-in data type (for example, `single`)
- The name of a data type object (for example, a `Simulink.NumericType` object)
- An expression that evaluates to a data type (for example, `fixdt(1,16,0)`). Do not specify a bus object as the data type in an expression; use `Bus: <object name>` to specify a bus data type.
- `Bus: <object name>`; enter the name of a bus object that you want to use to define the structure of the bus. The bus must be a nonvirtual bus. If you need to create or change a bus object, click the **Show data type assistant** button and then click the **Edit** button to the right of the **Bus object** field to open the Simulink Bus Editor. For details, see “Create Bus Objects with the Bus Editor”.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Data type** parameter.

See “Control Signal Data Types”.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Signal type

Specify the numeric type, real or complex, of the values in the data store.

Dimensions (-1 to infer from Initial value)

Dimensions of the data store. The default value, -1, enables you to set the dimensions of the data store by using the **Initial value** parameter. However, in this case, you cannot use scalar expansion with the initial value. You must specify the initial value by using an array that has the dimensions that you want.

If you use a value other than -1, specify the same dimensions as the dimensions of the **Initial value** parameter, unless you specify the initial value as a scalar (for scalar expansion) or a MATLAB structure. If the data store represents an array of buses, and if you use a MATLAB structure for the initial value, you can specify dimensions to initialize the array of buses with this structure.

Interpret vector parameters as 1-D

Specify that the data store interpret vector initial values as one-dimensional.

By default, MATLAB represents vector data as matrices, which have two dimensions. For example, MATLAB represents the vector [1 2 3] as a 1-by-3 matrix.

When you select this parameter, the data store represents vector data by using only one dimension instead of two. For example, if you specify an initial value of [1 2 3], the data store stores a one-dimensional vector with three elements.

For more information, see “Determining the Output Dimensions of Source Blocks”.

Data store name must resolve to Simulink signal object

Specify that Simulink software, when compiling the model, searches the model and base workspace for a `Simulink.Signal` object having the same name, as described in “Symbol Resolution”. If Simulink does not find such an object, the compilation stops, with an error. Otherwise, Simulink compares the attributes of the signal object to the corresponding attributes of the Data Store Memory block. If the block and the object attributes are inconsistent, Simulink halts model compilation and displays an error.

Share across model instances

In a single model reference hierarchy, when you use multiple Model blocks to refer to a model that contains a Data Store Memory block, by default, each instance of the referenced model (each Model block) reads from and writes to a separate copy of the data store. When you select **Share across model instances**, instead of interacting with a separate copy, all of the instances read from and write to the same data store.

When you set the model configuration parameter **Code interface packaging** to `Reusable function` to generate reentrant code from a model (Simulink Coder), a data store with **Share across model instances** selected appears in the code as a global symbol that the generated entry-point functions access directly. For example, a global symbol is a global variable or a field of a global structure variable. Therefore, each call that your code makes to the entry-point functions (each instance of the model) shares the data.

For an example, see “Share Data Between Instances of a Reusable Algorithm”.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder® software, custom storage classes do not affect the generated code.

For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

To programmatically set this parameter, use `StateSignalObject`. See “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder).

Storage class

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

To programmatically set this parameter, use `StateStorageClass` or `StateSignalObject`. See “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder).

TypeQualifier

Specify a storage type qualifier such as `const` or `volatile`.

Setting **Storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

Note **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder software, custom storage classes and memory sections do not affect the generated code.

See “Discrete Block State Naming in Generated Code” (Simulink Coder) for more information.

Detect read before write

Select the diagnostic action to take if the model attempts to read data from a data store to which it has not written data in this time step. See also the “Detect read before write” diagnostic in the **Data Store Memory block** section of the **Model Configuration Parameters > Diagnostics > Data Validity** pane.

Default: `warning`

`none`

Simulink software takes no action.

`warning`

Simulink software displays a warning and continues the simulation.

`error`

Simulink software terminates the simulation and displays an error.

Detect write after read

Select the diagnostic action to take if the model attempts to write data to the data store after previously reading data from it in the current time step. See also the “Detect write after read” diagnostic in the **Data Store Memory block** section of the **Model Configuration Parameters > Diagnostics > Data Validity** pane.

Default: warning

none

Simulink software takes no action.

warning

Simulink software displays a warning and continues the simulation.

error

Simulink software terminates the simulation and displays an error.

Detect write after write

Select the diagnostic action to take if the model attempts to write data to the data store twice in succession in the current time step. See also the “Detect write after write” diagnostic in the **Data Store Memory block** section of the **Model Configuration Parameters > Diagnostics > Data Validity** pane.

Default: warning

none

Simulink software takes no action.

warning

Simulink software displays a warning and continues the simulation.

error

Simulink software terminates the simulation and displays an error.

Log signal data

Select this option to save the values of this signal to the MATLAB workspace during simulation. See “Signal Logging” for details.

Logging name

Use this pair of controls, consisting of a list box and an edit field, to specify the name associated with logged signal data.

Simulink uses the signal name as its logging name by default. To specify a custom logging name, select **Custom** from the list box and enter the custom name in the adjacent edit field.

Limit data points to last

Discard all but the last N data points, where N is the number that you enter in the adjacent edit field. For more information, see “Log Data Stores”.

Decimation

Log every N th data point, where N is the number that you enter in the adjacent edit field. For example, suppose that your model uses a fixed-step solver with a step size of 0.1 s. If you select this option and accept the default decimation value (2), Simulink records data points for this signal at times 0.0, 0.2, 0.4, and so on. For more information, see “Log Data Stores”.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

See Also

- “Data Stores”
- “Access Data Stores with Simulink Blocks”
- Data Store Read
- Data Store Write
- “Log Data Stores”

Introduced before R2006a

Data Store Read

Read data from data store



Library

Signal Routing

Description

The Data Store Read block copies data from the named data store to its output. More than one Data Store Read block can read from the same data store.

The data store from which the data is read is determined by the location of the Data Store Memory block or signal object that defines the data store. For more information, see “Data Stores” and Data Store Memory.

Obtaining correct results from data stores requires ensuring that data store reads and writes occur in the expected order. See “Order Data Store Access” and “Data Store Diagnostics” for details.

Data Type Support

The Data Store Read block can output a real or complex signal of these data types:

- Floating point
- Built-in integer
- Fixed point

- Boolean
- Enumerated
- Bus

The block does not support variable-size signals.

For more information, see “Data Types Supported by Simulink”.

You can use arrays of buses with a Data Store Read block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Parameters

Data store name

Specifies the name of the data store from which this block reads data. The adjacent pull-down list lists the names of Data Store Memory blocks that exist at the same level in the model as the Data Store Read block or at higher levels. The pulldown list also includes all `Simulink.Signal` objects in the base and model workspaces. To change the name, select a name from the pull-down list or enter the name directly in the edit field.

When Simulink software compiles the model containing this block, Simulink software searches the model upwards from this block's level for a Data Store Memory block having the specified data store name. If Simulink software does not find such a block, it searches the model workspace and the MATLAB workspace for a `Simulink.Signal` object having the same name. See “Symbol Resolution” for more information about the search path.

If Simulink software finds the signal object, it creates a hidden Data Store Memory block at the model's root level having the properties specified by the signal object and an initial value of 0. If Simulink software finds neither the Data Store Memory block nor the signal object, it halts the compilation and displays an error.

Data store memory block

This field lists the Data Store Memory block that initialized the store from which this block reads.

Data store write blocks

This parameter lists all the Data Store Write blocks with the same data store name as this block that are in the same (sub)system or in any subsystem below it in the

model hierarchy. Double-click any entry on this list to highlight the block and bring it to the foreground.

Sample time

The sample time, which controls when the block reads from the data store. A value of -1 indicates that the sample time is inherited. See “Specify Sample Time” for more information.

Use the **Element Selection** pane to select a subset of the bus or matrix elements defined for the associated data store. The Data Store Read block icon reflects the elements that you specify. For details, see “Accessing Specific Bus and Matrix Elements”.

Elements in the array or Signals in the bus (Prompt is specific to the type of data.)

For bus signals, lists the elements in the associated data store. The list displays the maximum dimensions for each element, in parentheses.

For data stores with a bus data type, you can expand the tree to view the bus elements. For data stores with arrays, you can read the whole data store, or you can specify one or more elements of the whole data store.

You can select an element and then use one of the following approaches:

- Click **Select>>** to display that element (and all its subelements) in the **Selected element(s)** list.
- Use the **Specify element(s) to select** edit box to specify the bus or matrix elements that you want to select for reading. Then click **Select>>**.

To refresh the display to reflect modifications to the bus or matrix used in the data store, click **Refresh**.

Specify element(s) to select

Enter a MATLAB expression to define the specific element that you want to read. For example, for a data store named `DSM` that has maximum dimensions of `[3, 5]`, you could enter expressions such as `DSM(2, 4)` or `DSM([1 3], 2)` in the edit box and then click **Select>>**.

To apply the element selection, click **OK**.

Selected Element(s)

Displays the elements that you select. The Data Store Read block icon displays a port for each element that you specify.

To change the order of bus or matrix elements in the list, select the element in the list and click **Up** or **Down**. Changing the order of the elements in the list changes the order of the ports. To remove an element, click **Remove**.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

See Also

- “Data Stores”
- “Rename Data Stores”
- “Access Data Stores with Simulink Blocks”
- Data Store Memory
- Data Store Write

Introduced before R2006a

Data Store Write

Write data to data store



Library

Signal Routing

Description

The Data Store Write block copies the value at its input to the named data store. Each write operation performed by a Data Store Write block writes over the data store, replacing the previous contents.

The data store to which this block writes is determined by the location of the Data Store Memory block or signal object that defines the data store. For more information, see “Data Stores” and Data Store Memory. The size of the data store is set by the signal object or the Data Store Memory block that defines and initializes the data store. Each Data Store Write block that writes to that data store must write the same amount of data.

More than one Data Store Write block can write to the same data store. However, if two Data Store Write blocks attempt to write to the same data store during the same simulation step, results are unpredictable.

Obtaining correct results from data stores requires ensuring that data store reads and writes occur in the expected order. For details, see “Order Data Store Access” and “Data Store Diagnostics”.

You can log the values of a local or global data store data variable for all the steps in a simulation. For details, see “Log Data Stores”.

Data Type Support

The Data Store Write block accepts a real or complex signal of these data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated
- Bus

The block does not support variable-size signals.

For more information, see “Data Types Supported by Simulink”.

You can use an array of buses with a Data Store Write block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Parameters

Data store name

Specifies the name of the data store to which this block writes data. The adjacent pull-down list lists the names of Data Store Memory blocks that exist at the same level in the model as the Data Store Write block or at higher levels. The pulldown list also includes all `Simulink.Signal` objects in the base and model workspaces. To change the name, select a name from the pull-down list or enter the name directly in the edit field.

When Simulink software compiles the model containing this block, Simulink software searches the model upwards from this block's level for a Data Store Memory block having the specified data store name. If Simulink does not find such a block, it searches the model workspace and the MATLAB workspace for a `Simulink.Signal` object having the same name. If Simulink software finds neither the Data Store Memory block nor the signal object, it halts the compilation and displays an error. See “Symbol Resolution” for more information about the search path.

If Simulink finds a signal object, it creates a hidden Data Store Memory block at the model's root level having the properties specified by the signal object and an initial

value set to a matrix of zeros. The dimensions of that matrix are inherited from the `Dimensions` property of the signal object.

Data store memory block

This field lists the Data Store Memory block that initialized the store to which this block writes.

Data store read blocks

This parameter lists all the Data Store Read blocks with the same data store name as this block that are in the same (sub)system or in any subsystem below it in the model hierarchy. Double-click any entry on this list to highlight the block and bring it to the foreground.

Sample time

Specify the sample time that controls when the block writes to the data store. A value of `-1` indicates that the sample time is inherited. See “Specify Sample Time” for more information.

Use the **Element Assignment** pane to assign a subset of the bus or matrix elements defined for writing to the associated data store. The Data Store Write block icon reflects the elements that you specify. For details, see “Accessing Specific Bus and Matrix Elements”.

Elements in the array or Signals in the bus (Prompt is specific to the type of data.)

For bus signals, lists the elements in the associated data store. The list displays the maximum dimensions for each element, in parentheses.

For data stores with a bus data type, you can expand the tree to view the bus elements. For data stores with arrays, you can write the whole data store, or you can assign one or more elements to the whole data store.

You can select an element and then use one of the following approaches:

- Click **Select>>** to display that element (and all its subelements) in the **Assigned element(s)** list.
- Use the **Specify element(s) to assign** edit box to specify the bus or matrix elements that you want to select for reading. Then click **Select>>**.

To refresh the display to reflect modifications to the bus or matrix used in the data store, click **Refresh**.

Specify element(s) to assign

Enter a MATLAB expression to define the specific element that you want to write. For example, for a data store named DSM that has maximum dimensions of [3, 5], you could enter expressions such as `DSM(2, 4)` or `DSM([1 3], 2)` in the edit box. Then click **Select>>**.

To apply the element selection, click **OK**.

Assigned Element(s)

Displays the elements that you selected for assignment. The Data Store Write block icon displays a port for each element that you specify.

To change the order of bus or matrix elements in the list, select the element in the list and click **Up** or **Down**. Changing the order of the elements in the list changes the order of the ports. To remove an element, click **Remove**.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

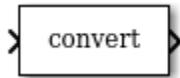
See Also

- “Data Stores”
- “Rename Data Stores”
- “Access Data Stores with Simulink Blocks”
- Data Store Memory
- Data Store Read
- “Log Data Stores”

Introduced before R2006a

Data Type Conversion

Convert input signal to specified data type



Library

Signal Attributes

Description

The Data Type Conversion block converts an input signal of any Simulink data type to the data type that you specify.

The input can be any real- or complex-valued signal. If the input is real, the output is real. If the input is complex, the output is complex.

Note To control the output data type by specifying block parameters, or to inherit a data type from a downstream block, use the Data Type Conversion block. To inherit a data type from a different signal in the model, use the Data Type Conversion Inherited block.

Convert Fixed-Point Signals

When you convert between fixed-point data types, the **Input and output to have equal** parameter controls block behavior. If neither input nor output use fixed-point scaling, because they are not of a fixed-point data type or have trivial fixed-point scaling, this parameter does not change the behavior of the block. For more information about fixed-point numbers, see “Fixed-Point Numbers in Simulink” (Fixed-Point Designer).

To convert a signal from one data type to another by attempting to preserve the real-world value of the input signal, select **Real World Value (RWV)**, the default setting. The block accounts for the scaling of the input and output and, within the limits of the specified data types, attempts to generate an output of equal real-world value.

To change the real-world value of the input signal by performing a scaling reinterpretation of the stored integer value, select `Stored Integer (SI)`. Within the limits of the specified data types, the block attempts to preserve the stored integer value of the signal during conversion. A best practice is to specify input and output data types using the same word length and signedness so that the block changes only the scaling of the signal. Specifying a different signedness or word length for the input and output could produce unexpected results such as range loss or unexpected sign extensions. For an example, see “Reinterpret Signal Using a Fixed-Point Data Type” on page 1-249.

If you select `Stored Integer (SI)`, the block does not perform a lower-level bit reinterpretation of a floating-point input signal. For example, if the input is of the data type `single` and has value 5, the bits that store the input in memory are given in hexadecimal by the following command.

```
num2hex(single(5))  
40a00000
```

However, the Data Type Conversion block does not treat the stored integer value as 40a00000, but instead as the real-world value, 5. After conversion, the stored integer value of the output is 5.

Cast Enumerated Signals

Use a Data Type Conversion block to cast enumerated signals as follows:

- 1 To cast a signal of enumerated type to a signal of any numeric type.

The underlying integers of all enumerated values input to the Data Type Conversion block must be within the range of the numeric type. Otherwise, an error occurs during simulation.

- 2 To cast a signal of any integer type to a signal of enumerated type.

The value input to the Data Type Conversion block must match the underlying value of an enumerated value. Otherwise, an error occurs during simulation.

You can enable the block’s **Saturate on integer overflow** parameter so that Simulink uses the default value of the enumerated type when the value input to the block does not match the underlying value of an enumerated value. See “Type Casting for Enumerations” (Simulink Coder).

You cannot use a Data Type Conversion block in the following cases:

- To cast a non-integer numeric signal to an enumerated signal.
- To cast a complex signal to an enumerated signal, regardless of the data types of the complex signal's real and imaginary parts.

See “Simulink Enumerations” for information on working with enumerated types.

Data Type Support

The Data Type Conversion block handles any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see “Data Types Supported by Simulink”.

Parameters

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: LockScale

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Input and output to have equal

Specify which type of input and output must be equal, in the context of fixed point data representation.

Default: Real World Value (RWV)

Real World Value (RWV)

Specifies the goal of making the Real World Value (RWV) of the input equal to the Real World Value (RWV) of the output.

Stored Integer (SI)

Specifies the goal of making the Stored Integer (SI) value of the input equal to the Stored Integer (SI) value of the output.

For the command-line information, see “Block-Specific Parameters” on page 6-128.

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: Floor

Ceiling

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

Floor

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

Nearest

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Parameter: `RndMeth`

Type: character vector

Value: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`

Default: `'Floor'`

For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

 Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Parameter: SaturateOnIntegerOverflow

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Output minimum

Lower value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMin

Type: character vector

Value: ' [] '

Default: ' [] '

Output maximum

Upper value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMax
Type: character vector
Value: ' [] '
Default: ' [] '

Output data type

Specify the output data type.

Default: Inherit: Inherit via back propagation

Inherit: Inherit via back propagation

Use data type of the driving block.

double

Output data type is double.

single

Output data type is single.

int8

Output data type is int8.

uint8

Output data type is uint8.

`int16`

Output data type is `int16`.

`uint16`

Output data type is `uint16`.

`int32`

Output data type is `int32`.

`uint32`

Output data type is `uint32`.

`boolean`

Output data type is `boolean`. The Data Type Conversion block converts real, nonzero numeric values (including NaN and Inf) to `boolean true` (1).

`fixdt(1,16,0)`

Output data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`

Output data type is fixed point `fixdt(1,16,2^0,0)`.

Enum: `<class name>`

Use an enumerated data type, for example, `Enum: BasicColors`.

`<data type expression>`

Use a data type object, for example, `Simulink.NumericType`.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Control Signal Data Types”.

Mode

Select the category of data to specify.

Default: `Inherit`

Inherit

Inheritance rules for data types. Selecting `Inherit` enables `Inherit` via back propagation.

Built in

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`

Fixed point

Fixed-point data types.

Enumerated

Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data as signed.

`Unsigned`

Specify the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope` and `bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Examples

Real-World Values and Stored Integers

The example model `ex_data_type_conversion_rwv_si` uses Data Type Conversion blocks to show the meaning of the real-world value and the stored integer of a signal. For basic information about fixed-point scaling, see “Scaling” (Fixed-Point Designer).

The Fixed-Point Constant block represents the real-world value 15 by using a fixed-point data type with binary-point scaling 2^{-5} . Due to the scaling, the output signal uses a stored integer value of 480.

The model uses Data Type Conversion blocks to convert the signal to a fixed-point data type with binary-point scaling 2^{-2} .

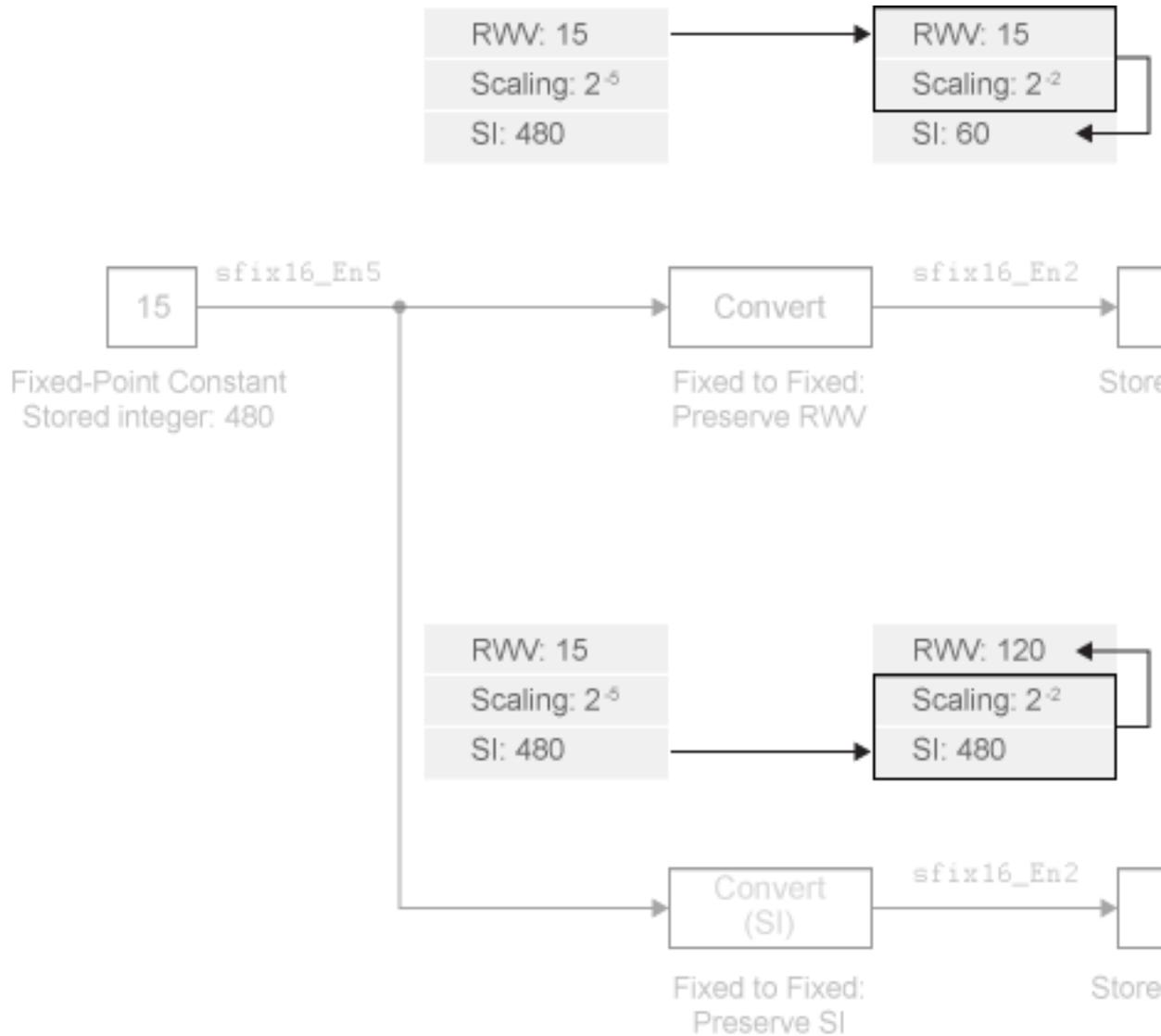
- The Fixed to Fixed: Preserve RWV block converts the input signal by preserving the real-world value, 15. The parameter **Input and output to have equal** is set to Real World Value (RWV).

The output signal has the same real-world value as the input, that is, 15. Due to the fixed-point scaling, the output uses a stored integer value of 60.

- The Fixed to Fixed: Preserve SI block converts the input signal by preserving the stored integer value, 480. The parameter **Input and output to have equal** is set to Stored Integer (SI).

The output signal uses the same stored integer value as the input, that is, 480. Due to the fixed-point scaling, the output has a real-world value of 120.

The figure shows the conversion mechanism for the two blocks.

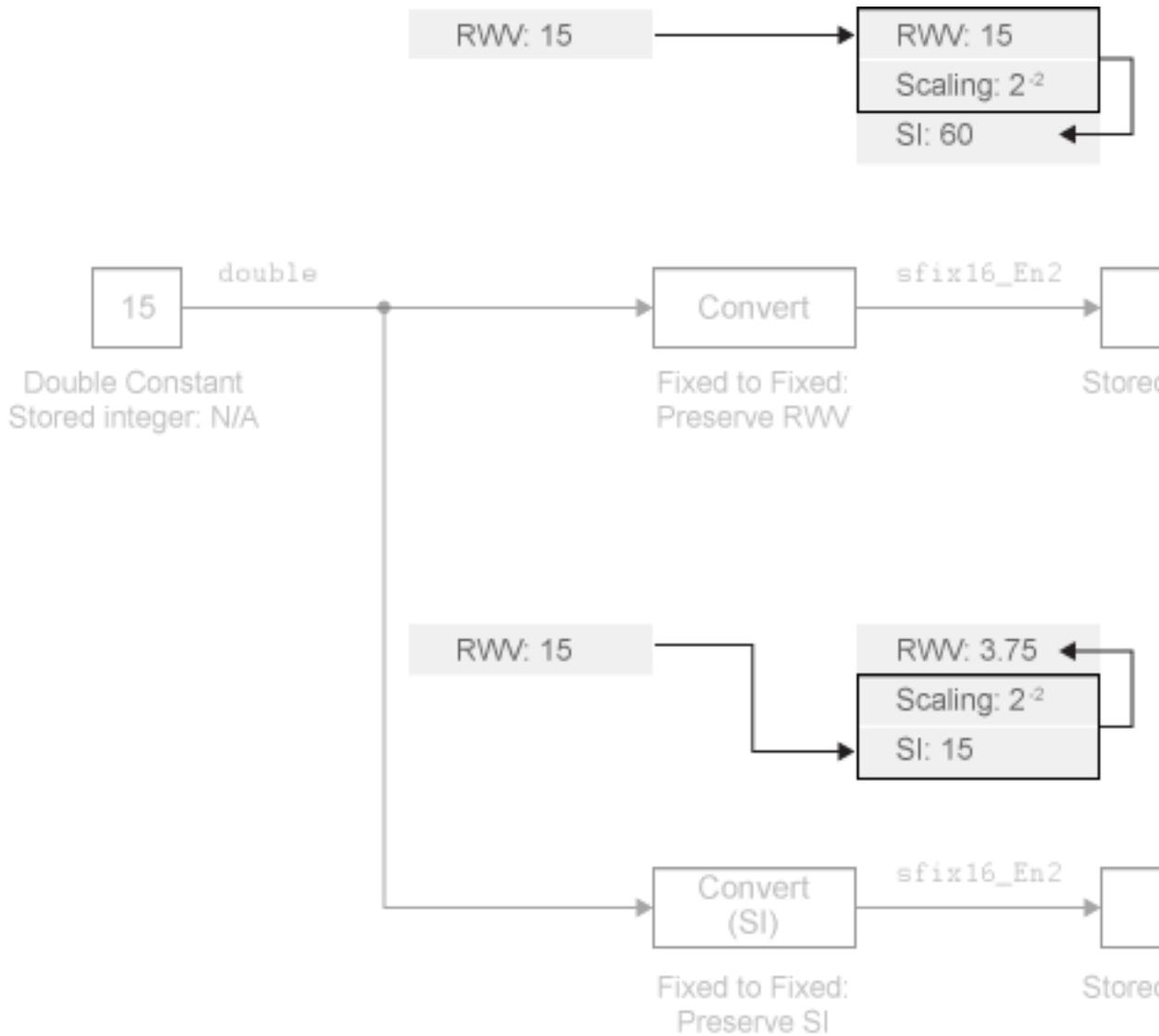


The Double Constant block represents the real-world value 15 by using the floating-point data type `double`. The output signal does not use fixed-point scaling.

The model uses Data Type Conversion blocks to convert the `double` signal to a fixed-point data type with binary-point scaling 2^{-2} .

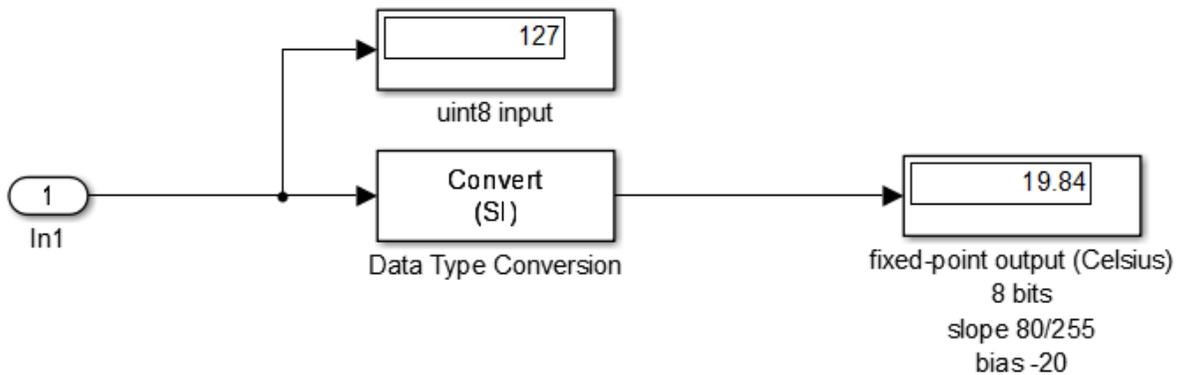
- The Float to Fixed: Preserve RWV block converts the input signal by preserving the real-world value, 15. The output signal has the same real-world value. Due to the fixed-point scaling, the output uses a stored integer value of 60.
- The Float to Fixed: Preserve SI block converts the input signal by attempting to preserve the stored integer value. However, the block does not use the underlying bits that store the floating-point signal in memory. Instead, the block uses the real-world value of the input, 15, as the stored integer of the output signal. Due to the fixed-point scaling, the real-world value of the output is 3.75.

The figure shows the conversion mechanism for the two blocks. The blocks also use these mechanisms if the input uses the floating-point data type `single`.



Reinterpret Signal Using a Fixed-Point Data Type

Suppose your hardware uses the data type `uint8` to store data from a temperature sensor. Also suppose that the minimum stored integer value 0 represents -20 degrees Celsius while the maximum 255 represents 60 degrees. The following model uses a Data Type Conversion block to convert the stored integer value of the sensor data to degrees Celsius.



The Data Type Conversion block parameter **Input and output to have equal** is set to `Stored Integer (SI)`. The block output signal is of a fixed-point data type with word length 8, slope $80/255$, and bias -20 .

The Data Type Conversion block reinterprets the integer input, 127, as a Celsius output, 19.84 degrees. The block output uses the specified slope and bias to scale the stored integer of the input.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	Yes

Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Data Type Conversion Inherited | Data Type Propagation

Topics

“Control Signal Data Types”
“About Data Types in Simulink”
“Fixed Point”

Introduced before R2006a

Data Type Conversion Inherited

Convert from one data type to another using inherited data type and scaling



Library

Signal Attributes

Description

The Data Type Conversion Inherited block forces dissimilar data types to be the same. The first input is used as the reference signal and the second input is converted to the reference type by inheriting the data type and scaling information. (See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.) Either input undergoes scalar expansion such that the output has the same width as the widest input.

Inheriting the data type and scaling provides these advantages:

- It makes reusing existing models easier.
- It allows you to create new fixed-point models with less effort since you can avoid the detail of specifying the associated parameters.

Data Type Support

The Data Type Conversion Inherited block accepts signals of the following data types:

- Floating point
- Built-in integer

- Fixed point
- Boolean
- Enumerated

For more information, see “Data Types Supported by Simulink”.

Parameters

Input and Output to have equal

Specify whether the `Real World Value (RWV)` or the `Stored Integer (SI)` of the input and output should be the same. Refer to `Description` on page 1-232 in the `Data Type Conversion` block reference page for more information about these choices.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer), in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	Yes

See Also

Data Type Conversion | Data Type Propagation

Topics

“Control Signal Data Types”

“About Data Types in Simulink”

“Fixed Point”

Introduced before R2006a

Data Type Duplicate

Force all inputs to same data type



Library

Signal Attributes

Description

The Data Type Duplicate block forces all inputs to have exactly the same data type. Other attributes of input signals, such as dimension, complexity, and sample time, are completely independent.

You can use the Data Type Duplicate block to check for consistency of data types among blocks. If all signals do not have the same data type, the block returns an error message.

The Data Type Duplicate block is typically used such that one signal to the block controls the data type for all other blocks. The other blocks are set to inherit their data types via back propagation.

The block is also used in a user created library. These library blocks can be placed in any model, and the data type for all library blocks are configured according to the usage in the model. To create a library block with more complex data type rules than duplication, use the Data Type Propagation block.

Data Type Support

The Data Type Duplicate block accepts signals of the following data types:

- Floating point

- Built-in integer
- Fixed point
- Boolean
- Enumerated

For more information, see “Data Types Supported by Simulink”.

Parameters

Number of input ports

Specify the number of inputs to this block.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from driving block
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Data Type Conversion | Data Type Propagation

Topics

“Control Signal Data Types”
 “About Data Types in Simulink”
 “Fixed Point”

Introduced before R2006a

Data Type Propagation

Set data type and scaling of propagated signal based on information from reference signals



Library

Signal Attributes

Description

The Data Type Propagation block allows you to control the data type and scaling of signals in your model. You can use this block in conjunction with fixed-point blocks that have their **Output data type** parameter configured to `Inherit: Inherit via back propagation`.

The block has three inputs: Ref1 and Ref2 are the reference inputs, while the Prop input back propagates the data type and scaling information gathered from the reference inputs. This information is then passed on to other fixed-point blocks.

The block provides you with many choices for propagating data type and scaling information. For example, you can:

- Use the number of bits from the Ref1 reference signal, or use the number of bits from widest reference signal.
- Use the range from the Ref2 reference signal, or use the range of the reference signal with the greatest range.
- Use a bias of zero, regardless of the biases used by the reference signals.
- Use the precision of the reference signal with the least precision.

You specify how data type information is propagated with the **Propagated data type** parameter list. If the parameter list is configured as `Specify` via dialog, then you manually specify the data type via the **Propagated data type** edit field. If the parameter list is configured as `Inherit` via propagation rule, then you must use the parameters described in “Parameters” on page 1-258.

You specify how scaling information is propagated with the **Propagated scaling** parameter list. If the parameter list is configured as `Specify` via dialog, then you manually specify the scaling via the **Propagated scaling** edit field. If the parameter list is configured as `Inherit` via propagation rule, then you must use the parameters described in “Parameters” on page 1-258.

After you use the information from the reference signals, you can apply a second level of adjustments to the data type and scaling by using individual multiplicative and additive adjustments. This flexibility has a variety of uses. For example, if you are targeting a DSP, then you can configure the block so that the number of bits associated with a MAC (multiply and accumulate) operation is twice as wide as the input signal, and has a certain number of guard bits added to it.

The Data Type Propagation block also provides a mechanism to force the computed number of bits to a useful value. For example, if you are targeting a 16-bit micro, then the target C compiler is likely to support sizes of only 8 bits, 16 bits, and 32 bits. The block will force these three choices to be used. For example, suppose the block computes a data type size of 24 bits. Since 24 bits is not directly usable by the target chip, the signal is forced up to 32 bits, which is natively supported.

There is also a method for dealing with floating-point reference signals. This makes it easier to create designs that are easily retargeted from fixed-point chips to floating-point chips or vice versa.

The Data Type Propagation block allows you to set up libraries of useful subsystems that will be properly configured based on the connected signals. Without this data type propagation process, a subsystem that you use from a library will almost certainly not work as desired with most integer or fixed-point signals, and manual intervention to configure the data type and scaling would be required. This block can eliminate the manual intervention in many situations.

Precedence Rules

The precedence of the dialog box parameters decreases from top to bottom. Additionally:

- Double-precision reference inputs have precedence over all other data types.
- Single-precision reference inputs have precedence over integer and fixed-point data types.
- Multiplicative adjustments are carried out before additive adjustments.
- The number of bits is determined before the precision or positive range is inherited from the reference inputs.

Data Type Support

The Data Type Propagation block accepts signals of the following data types:

- Floating-point
- Built-in integer
- Fixed-point
- Boolean

For more information, see “Data Types Supported by Simulink”.

Parameters

Propagated data type

Use the parameter list to propagate the data type via the dialog box, or inherit the data type from the reference signals. Use the edit field to specify the data type via the dialog box.

If any reference input is double, output is

Specify *single* or *double*. This parameter makes it easier to create designs that are easily retargeted from fixed-point chips to floating-point chips or vice versa.

This parameter is visible only when you set **Propagated data type** to *Inherit via propagation rule*.

If any reference input is single, output is

Specify *single* or *double*. This parameter makes it easier to create designs that are easily retargeted from fixed-point chips to floating-point chips or visa versa.

This parameter is visible only when you set **Propagated data type** to `Inherit` via propagation rule.

Is-Signed

Specify the sign of Prop as one of the following values:

Parameter Value	Description
<code>IsSigned1</code>	Prop is a signed data type if Ref1 is a signed data type.
<code>IsSigned2</code>	Prop is a signed data type if Ref2 is a signed data type.
<code>IsSigned1</code> or <code>IsSigned2</code>	Prop is a signed data type if either Ref1 or Ref2 are signed data types.
<code>TRUE</code>	Ref1 and Ref2 are ignored, and Prop is always a signed data type.
<code>FALSE</code>	Ref1 and Ref2 are ignored, and Prop is always an unsigned data type.

For example, if the Ref1 signal is `ufix(16)`, the Ref2 signal is `sfix(16)`, and the **Is-Signed** parameter is `IsSigned1` or `IsSigned2`, then Prop is forced to be a signed data type.

This parameter is visible only when you set **Propagated data type** to `Inherit` via propagation rule.

Number-of-bits: Base

Specify the number of bits used by Prop for the base data type as one of the following values:

Parameter Value	Description
<code>NumBits1</code>	The number of bits for Prop is given by the number of bits for Ref1.
<code>NumBits2</code>	The number of bits for Prop is given by the number of bits for Ref2.
<code>max([NumBits1 NumBits2])</code>	The number of bits for Prop is given by the reference signal with largest number of bits.
<code>min([NumBits1 NumBits2])</code>	The number of bits for Prop is given by the reference signal with smallest number of bits.
<code>NumBits1+NumBits2</code>	The number of bits for Prop is given by the sum of the reference signal bits.

For more information about the base data type, refer to Targeting an Embedded Processor (Fixed-Point Designer).

This parameter is visible only when you set **Propagated data type** to `Inherit` via propagation rule.

Number-of-bits: Multiplicative adjustment

Specify the number of bits used by Prop by including a multiplicative adjustment that uses a data type of `double`. For example, suppose you want to guarantee that the number of bits associated with a multiply and accumulate (MAC) operation is twice as wide as the input signal. To do this, you configure this parameter to the value 2.

This parameter is visible only when you set **Propagated data type** to `Inherit` via propagation rule.

Number-of-bits: Additive adjustment

Specify the number of bits used by Prop by including an additive adjustment that uses a data type of `double`. For example, if you are performing multiple additions during a MAC operation, the result might overflow. To prevent overflow, you can associate guard bits with the propagated data type. To associate four guard bits, you specify the value 4.

This parameter is visible only when you set **Propagated data type** to `Inherit` via propagation rule.

Number-of-bits: Allowable final values

Force the computed number of bits used by Prop to a useful value. For example, if you are targeting a processor that supports only 8, 16, and 32 bits, then you configure this parameter to `[8, 16, 32]`. The block always propagates the smallest specified value that fits. If you want to allow all fixed-point data types, you would specify the value `1:128`.

This parameter is visible only when you set **Propagated data type** to `Inherit` via propagation rule.

Propagated scaling

Use the parameter list to propagate the scaling via the dialog box, inherit the scaling from the reference signals, or calculate the scaling to obtain best precision.

Propagated scaling (Slope or [Slope Bias])

Specify the scaling as either a slope or a slope and bias.

This parameter is visible only when you set **Propagated scaling** to *Specify* via dialog.

Values used to determine best precision scaling

Specify any values to be used to constrain the precision, such as the upper and lower limits on the propagated input. Based on the data type, the scaling will automatically be selected such that these values can be represented with no overflow error and minimum quantization error.

This parameter is visible only when you set **Propagated scaling** to *Obtain best precision* via dialog.

Slope: Base

Specify the slope used by Prop for the base data type as one of the following values:

Parameter Value	Description
Slope1	The slope of Prop is given by the slope of Ref1.
Slope2	The slope of Prop is given by the slope of Ref2.
max([Slope1 Slope2])	The slope of Prop is given by the maximum slope of the reference signals.
min([Slope1 Slope2])	The slope of Prop is given by the minimum slope of the reference signals.
Slope1*Slope2	The slope of Prop is given by the product of the reference signal slopes.
Slope1/Slope2	The slope of Prop is given by the ratio of the Ref1 slope to the Ref2 slope.
PosRange1	The range of Prop is given by the range of Ref1.
PosRange2	The range of Prop is given by the range of Ref2.
max([PosRange1 PosRange2])	The range of Prop is given by the maximum range of the reference signals.
min([PosRange1 PosRange2])	The range of Prop is given by the minimum range of the reference signals.
PosRange1*PosRange2	The range of Prop is given by the product of the reference signal ranges.
PosRange1/PosRange2	The range of Prop is given by the ratio of the Ref1 range to the Ref2 range.

You control the precision of Prop with Slope1 and Slope2, and you control the range of Prop with PosRange1 and PosRange2. Additionally, PosRange1 and PosRange2 are one bit higher than the maximum positive range of the associated reference signal.

This parameter is visible only when you set **Propagated scaling** to `Inherit` via propagation rule.

Slope: Multiplicative adjustment

Specify the slope used by Prop by including a multiplicative adjustment that uses a data type of `double`. For example, if you want 3 bits of additional precision (with a corresponding decrease in range), the multiplicative adjustment is 2^{-3} .

This parameter is visible only when you set **Propagated scaling** to `Inherit` via propagation rule.

Slope: Additive adjustment

Specify the slope used by Prop by including an additive adjustment that uses a data type of `double`. An additive slope adjustment is often not needed. The most likely use is to set the multiplicative adjustment to 0, and set the additive adjustment to force the final slope to a specified value.

This parameter is visible only when you set **Propagated scaling** to `Inherit` via propagation rule.

Bias: Base

Specify the bias used by Prop for the base data type. The parameter values are described as follows:

Parameter Value	Description
Bias1	The bias of Prop is given by the bias of Ref1.
Bias2	The bias of Prop is given by the bias of Ref2.
max([Bias1 Bias2])	The bias of Prop is given by the maximum bias of the reference signals.
min([Bias1 Bias2])	The bias of Prop is given by the minimum bias of the reference signals.
Bias1*Bias2	The bias of Prop is given by the product of the reference signal biases.

Parameter Value	Description
Bias1/Bias2	The bias of Prop is given by the ratio of the Ref1 bias to the Ref2 bias.
Bias1+Bias2	The bias of Prop is given by the sum of the reference biases.
Bias1-Bias2	The bias of Prop is given by the difference of the reference biases.

This parameter is visible only when you set **Propagated scaling** to `Inherit` via propagation rule.

Bias: Multiplicative adjustment

Specify the bias used by Prop by including a multiplicative adjustment that uses a data type of `double`.

This parameter is visible only when you set **Propagated scaling** to `Inherit` via propagation rule.

Bias: Additive adjustment

Specify the bias used by Prop by including an additive adjustment that uses a data type of `double`.

If you want to guarantee that the bias associated with Prop is zero, you should configure both the multiplicative adjustment and the additive adjustment to 0.

This parameter is visible only when you set **Propagated scaling** to `Inherit` via propagation rule.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Ye
Zero-Crossing Detection	No

Code Generation	Yes
-----------------	-----

See Also

[Data Type Conversion](#) | [Data Type Conversion Inherited](#) | [Data Type Duplicate](#)

Topics

[“Control Signal Data Types”](#)
[“About Data Types in Simulink”](#)
[“Fixed Point”](#)

Introduced before R2006a

Data Type Scaling Strip

Remove scaling and map to built in integer



Library

Signal Attributes

Description

The Scaling Strip block strips the scaling off a fixed point signal. It maps the input data type to the smallest built in data type that has enough data bits to hold the input. The stored integer value of the input is the value of the output. The output always has nominal scaling (slope = 1.0 and bias = 0.0), so the output does not make a distinction between real world value and stored integer value.

Data Type Support

The Data Type Scaling Strip block accepts signals of any numeric data type that Simulink supports, including fixed-point data types.

For more information, see “Data Types Supported by Simulink”.

Characteristics

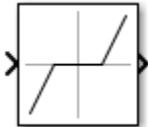
Data Types	Double Single Boolean Base Integer Fixed-Point
------------	--------------------------------------------------------

Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Dead Zone

Provide region of zero output



Library

Discontinuities

Description

The Dead Zone block generates zero output within a specified region, called its dead zone. You specify the lower limit (LL) and upper limit (UL) of the dead zone as the **Start of dead zone** and **End of dead zone** parameters, respectively. The block output depends on the input (U) and the values for the lower and upper limits:

Input	Output
$U \geq LL$ and $U \leq UL$	Zero
$U > UL$	$U - UL$
$U < LL$	$U - LL$

Data Type Support

The Dead Zone block accepts and outputs real signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Start of dead zone

Specify the lower limit of the dead zone. The default is -0.5 .

End of dead zone

Specify the upper limit of the dead zone. The default is 0.5 .

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Do not select this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</p>	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code> , which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code> , is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Treat as gain when linearizing

The linearization commands in Simulink software treat this block as a gain in state space. Select this check box to cause the commands to treat the gain as 1; otherwise, the commands treat the gain as 0.

Enable zero-crossing detection

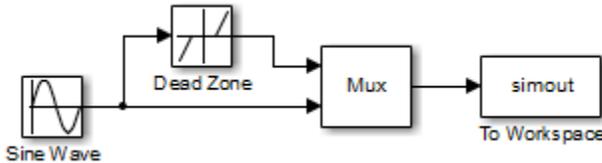
Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Sample time

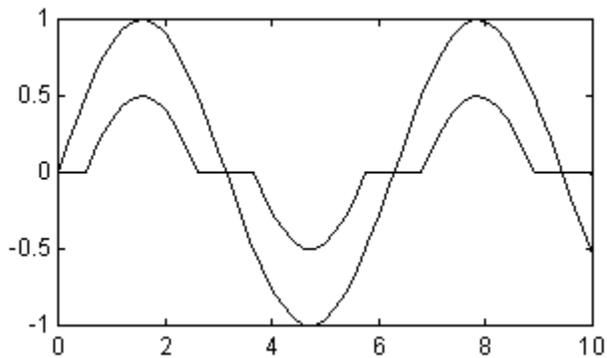
Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Examples

The following model uses lower and upper limits of -0.5 and 0.5, with a sine wave as input.



This plot shows the effect of the Dead Zone block on the sine wave. When the input sine wave is between -0.5 and 0.5 , the output is zero.



Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

See Also

Dead Zone Dynamic

Introduced before R2006a

Dead Zone Dynamic

Set inputs within bounds to zero



Library

Discontinuities

Description

The Dead Zone Dynamic block dynamically bounds the range of the input signal, providing a region of zero output. The bounds change according to the upper and lower limit input signals where

- The input within the bounds is set to zero.
- The input below the lower limit is shifted down by the lower limit.
- The input above the upper limit is shifted down by the upper limit.

The input for the upper limit is the `up` port, and the input for the lower limit is the `lo` port.

Data Type Support

The Dead Zone Dynamic block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

Dead Zone

Introduced before R2006a

Decrement Real World

Decrease real world value of signal by one



Library

Additional Math & Discrete / Additional Math: Increment - Decrement

Description

The Decrement Real World block decreases the real world value of the signal by one. Overflows always wrap.

Data Type Support

The Decrement Real World block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
------------	----------------------------------------------

Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Decrement Stored Integer, Decrement Time To Zero, Decrement To Zero, Increment Real World

Introduced before R2006a

Decrement Stored Integer

Decrease stored integer value of signal by one



Library

Additional Math & Discrete / Additional Math: Increment - Decrement

Description

The Decrement Stored Integer block decreases the stored integer value of a signal by one.

Floating-point signals also decrease by one, and overflows always wrap.

Data Type Support

The Decrement Stored Integer block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
------------	----------------------------------------------

Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

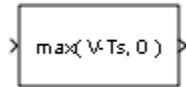
See Also

Decrement Real World, Decrement Time To Zero, Decrement To Zero, Increment Stored Integer

Introduced before R2006a

Decrement Time To Zero

Decrease real-world value of signal by sample time, but only to zero



Library

Additional Math & Discrete / Additional Math: Increment - Decrement

Description

The Decrement Time To Zero block decreases the real-world value of the signal by the sample time, T_s . The output never goes below zero. This block works only with fixed sample rates and does not work inside a triggered subsystem.

Data Type Support

The Decrement Time To Zero block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

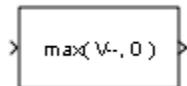
See Also

Decrement Real World, Decrement Stored Integer, Decrement To Zero

Introduced before R2006a

Decrement To Zero

Decrease real-world value of signal by one, but only to zero



Library

Additional Math & Discrete / Additional Math: Increment - Decrement

Description

The Decrement To Zero block decreases the real-world value of the signal by one. The output never goes below zero.

Data Type Support

The Decrement To Zero block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
------------	----------------------------------------------

Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Decrement Real World, Decrement Stored Integer, Decrement Time To Zero

Introduced before R2006a

Delay

Delay input signal by fixed or variable sample periods



Library

Discrete

Description

The Delay block outputs the input of the block after a delay. The block determines the delay time based on the value of the **Delay length** parameter. The block supports:

- Variable delay length
- Specification of the initial condition from an input port
- State storage
- Using a circular buffer instead of an array buffer for state storage
- Resetting the state to the initial condition with an external reset signal
- Controlling execution of the block at every time step with an external enable signal

The initial block output depends on a number of factors such as the **Initial condition** parameter and the simulation start time. For more information, see “Initial Block Output” on page 1-282. The **External reset** parameter determines if the block output resets to the initial condition on triggering. The **Show enable port** parameter determines if the block execution is controlled in every time step by an external enable signal.

Initial Block Output

The output of the Delay block in the first few time steps of the simulation depends on the block sample time, the delay length, and the simulation start time. The block supports

specifying or inheriting discrete sample times to determine the time interval between samples. For more information, see “Specify Sample Time”.

Suppose that the block inherits a discrete sample time as $[T_{\text{sampling}}, T_{\text{offset}}]$, where T_{sampling} is the sampling period and T_{offset} is the initial time offset. n is the value of the block’s **Delay length** parameter and T_{start} is the simulation start time for the model.

The table shows the Delay block output for the first few time steps.

Simulation Time Range	Block Output
(T_{start}) to $(T_{\text{start}} + T_{\text{offset}})$	Zero
$(T_{\text{start}} + T_{\text{offset}})$ to $(T_{\text{start}} + T_{\text{offset}} + n * T_{\text{sampling}})$	Initial condition parameter
After $(T_{\text{start}} + T_{\text{offset}} + n * T_{\text{sampling}})$	Input signal

Data Type Support

The block’s parameters have these dimensional requirements:

- **Delay length** and **External reset** must be scalar.
- **Initial condition** can be scalar or nonscalar.
- For frame-based processing, signal dimensions of the data input port u cannot be larger than two.

The block supports input signals with these data types.

Input Signal	Supported Data Types
Data input port u	<ul style="list-style-type: none"> • Floating point • Built-in integer • Fixed point • Boolean • Enumerated

Input Signal	Supported Data Types
Delay length d	<ul style="list-style-type: none"> Floating point Fixed-point integer Built-in integer
Enable port 	<ul style="list-style-type: none"> Floating point Built-in integer Fixed point Integer (only <code>ufix1</code>) Boolean
External reset port 	<ul style="list-style-type: none"> Floating point Built-in integer Fixed point Integer (only <code>ufix1</code>) Boolean
Initial condition $x0$	<ul style="list-style-type: none"> Floating point Built-in integer Fixed point Boolean Enumerated

When u is Boolean, $x0$ must be Boolean. When u uses an enumerated type, $x0$ must use the same enumerated type. Otherwise, $x0$ can use a floating-point, built-in integer, or fixed-point data type that fits in the data type of u . For example, when u uses `int32`, $x0$ can use `int8` but not `double`.

The data type of the output signal is the same as the input signal u .

For more information, see “Data Types Supported by Simulink”.

Variable-Size Support

The Delay block provides the following support for variable-size signals:

- The data input port u accepts variable-size signals. The other input ports do not accept variable-size signals.

- The output port has the same signal dimensions as the data input port u for variable-size inputs.

The rules that apply to variable-size signals depend on the input processing mode of the Delay block.

Input Processing Mode	Rules for Variable-Size Signal Support
Elements as channels (sample based)	<ul style="list-style-type: none"> • The signal dimensions change only during state reset when the block is enabled. • The initial condition must be scalar.
Columns as channels (frame based)	<ul style="list-style-type: none"> • No support
Inherited (where input is a sample-based signal)	<ul style="list-style-type: none"> • The signal dimensions change only during state reset when the block is enabled. • The initial condition must be scalar.
Inherited (where input is a frame-based signal)	<ul style="list-style-type: none"> • The channel size changes only during state reset when the block is enabled. • The initial condition must be scalar. • The frame size must be constant.

Bus Support

The Delay block provides the following support for bus signals:

- The data input port u accepts virtual and nonvirtual bus signals. The other input ports do not accept bus signals.
- The output port has the same bus type as the data input port u for bus inputs.
- Buses work with:
 - Sample-based and frame-based processing
 - Fixed and variable delay length
 - Array and circular buffers

To use a bus signal as the input to a Delay block, you should specify the initial condition on the dialog box. In other words, the initial condition cannot come from the input port

×0. Support for virtual and nonvirtual buses depends on the initial condition that you specify and whether the **State name** parameter is empty or not.

Initial Condition	State Name	
	Empty	Not Empty
Zero	Virtual and nonvirtual bus support	Nonvirtual bus support only
Nonzero scalar	Virtual and nonvirtual bus support	No bus support
Nonscalar	No bus support	No bus support
Structure	Virtual and nonvirtual bus support	Nonvirtual bus support only
Partial structure	Virtual and nonvirtual bus support	Nonvirtual bus support only

Parameters

Delay length

Specify whether to enter the delay length directly on the dialog box (fixed delay) or to inherit the delay from an input port (variable delay).

- If you set **Source** to `Dialog`, enter the delay length in the edit field under **Value**.
- If you set **Source** to `Input port`, verify that an upstream signal supplies a delay length for the d input port. You can also specify its maximum value by specifying the parameter **Upper limit**.

Specify the scalar delay length as a real, non-negative integer. An out-of-range or non-integer value in the dialog box (fixed delay) returns an error. An out-of-range value from an input port (variable delay) casts it into the range. A non-integer value from an input port (variable delay) truncates it to the integer.

This parameter is not tunable for simulation or code generation.

Initial condition

Specify whether to enter the initial condition directly on the dialog box or to inherit the initial condition from an input port.

- If you set **Source** to `Dialog`, enter the initial condition in the edit field under **Value**.
- If you set **Source** to `Input port`, verify that an upstream signal supplies an initial condition for the `x0` input port.

Simulink converts offline the data type of **Initial condition** to the data type of the input signal `u` using a round-to-nearest operation and saturation.

Note When **State name must resolve to Simulink signal object** is selected on the **State Attributes** pane, the block copies the initial value of the signal object to the **Initial condition** parameter. However, when the source for **Initial condition** is `Input port`, the block ignores the initial value of the signal object.

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- `Elements as channels (sample based)` — Treat each element of the input as a separate channel (sample-based processing).
- `Columns as channels (frame based)` — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

- `Inherited` — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input u . All other input signals must be sample based.

Input Signal u	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Use circular buffer for state

Select to use a circular buffer for storing the state in simulation and code generation. Otherwise, an array buffer stores the state.

Using a circular buffer can improve execution speed when the delay length is large. For an array buffer, the number of copy operations increases as the delay length goes up. For a circular buffer, the number of copy operations is constant for increasing delay length.

If one of the following conditions is true, an array buffer always stores the state because a circular buffer does not improve execution speed:

- For sample-based signals, the delay length is 1.
- For frame-based signals, the delay length is no larger than the frame size.

Prevent direct feedthrough

Select to increase the delay length from zero to the lower limit for the **Input processing** mode:

- For sample-based signals, increase the minimum delay length to 1.
- For frame-based signals, increase the minimum delay length to the frame length.

Selecting this check box prevents direct feedthrough from the input port, u , to the output port. However, this check box cannot prevent direct feedthrough from the initial condition port, $x0$, to the output port.

This check box is available when you set **Delay length: Source** to `Input port`.

Remove delay length check in generated code

Select to remove code that checks for out-of-range delay length.

Check Box	Result	When to Use
Selected	Generated code does not include conditional statements to check for out-of-range delay length.	For code efficiency
Cleared	Generated code includes conditional statements to check for out-of-range delay length.	For safety-critical applications

This check box is available when you set **Delay length: Source** to `Input port`.

Diagnostic for delay length

Specify whether to produce a warning or error when the input `d` is less than the lower limit or greater than the **Delay length: Upper limit**. The lower limit depends on the setting for **Prevent direct feedthrough**.

- If the check box is cleared, the lower limit is zero.
- If the check box is selected, the lower limit is 1 for sample-based signals and frame length for frame-based signals.

Options for the diagnostic include:

- `None` — Simulink software takes no action.
- `Warning` — Simulink software displays a warning and continues the simulation.
- `Error` — Simulink software terminates the simulation and displays an error.

This parameter is available when you set **Delay length: Source** to `Input port`.

Show enable port

Select to show an enable port for this block. This port can control execution of the block. The block is considered enabled when the input to this port is nonzero, and is disabled when the input is 0. The value of the input is checked at the same time step as the block execution.

External reset

Specify the trigger event to use to reset the states to the initial conditions.

Reset Mode	Behavior
None	No reset.
Rising	Reset on a rising edge.
Falling	Reset on a falling edge.
Either	Reset on either a rising or falling edge.
Level	Reset in either of these cases: <ul style="list-style-type: none"> • when the reset signal is nonzero at the current time step • when the reset signal value changes from nonzero at the previous time step to zero at the current time step
Level hold	Reset when the reset signal is nonzero at the current time step

The reset signal must be a scalar of type `single`, `double`, `boolean`, or `integer`. Fixed point data types, except for `ufix1`, are not supported.

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. This block supports discrete sample time, but not continuous sample time.

State name

Use this parameter to assign a unique name to the block state. The default is `' '`. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder).

State name must resolve to Simulink signal object

Select this check box to require that the state name resolve to a Simulink signal object. This check box is cleared by default.

State name enables this parameter. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class**.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see “Apply Custom Storage Classes Directly to Signal Lines, Block States, and Output Blocks” (Embedded Coder).

To programmatically set this parameter, use `StateSignalObject`.

For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

StorageClass

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

State name enables this parameter.

To programmatically set this parameter, use `StateStorageClass` or `StateSignalObject`. See “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder).

TypeQualifier

Note **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

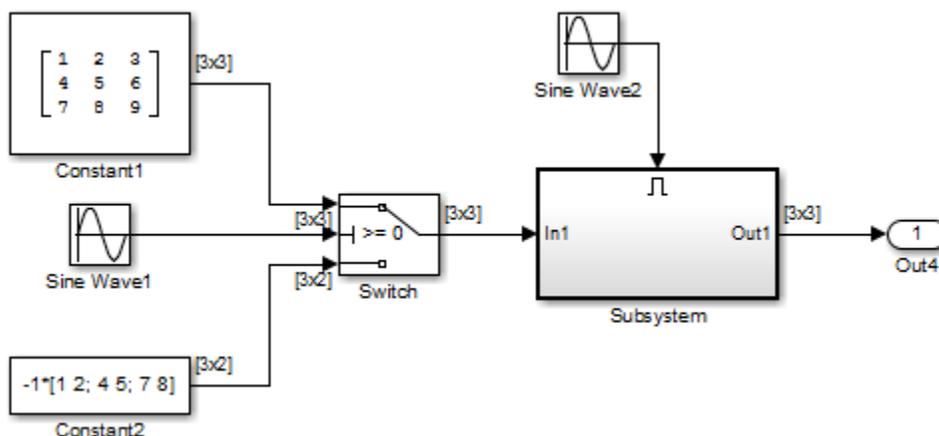
Specify a storage type qualifier such as `const` or `volatile`.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

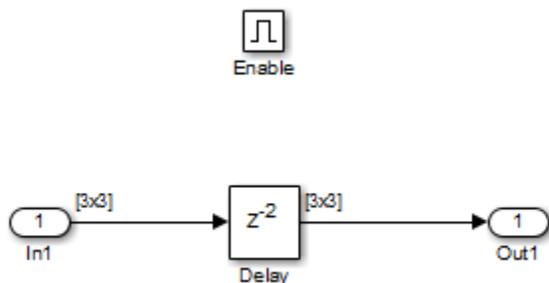
Examples

Variable-Size Signals for Sample-Based Processing

This model shows how the Delay block supports variable-size signals for sample-based processing.



The Switch block controls whether the input signal to the enabled subsystem is a 3-by-3 or 3-by-2 matrix. The Delay block appears inside the enabled subsystem.

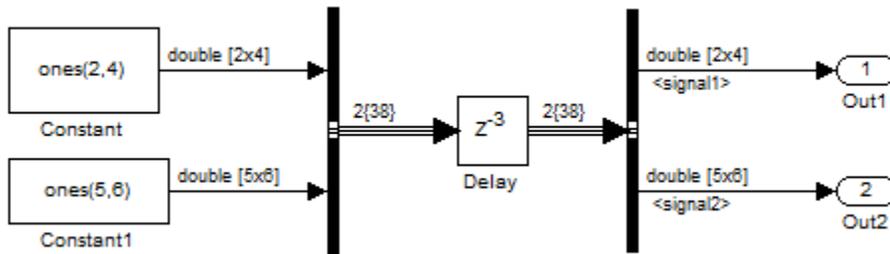


The model follows the rules for variable-size signals when the Delay block uses sample-based processing.

Rule	How the Model Follows the Rule
The signal dimensions change only during state reset when the block is enabled.	The Enable block sets Propagate sizes of variable-size signals to Only when enabling.
The initial condition must be scalar.	The Delay block sets Initial condition to 0.0, a scalar value.

Bus Signals for Frame-Based Processing

This model shows how the Delay block supports bus signals for frame-based processing.



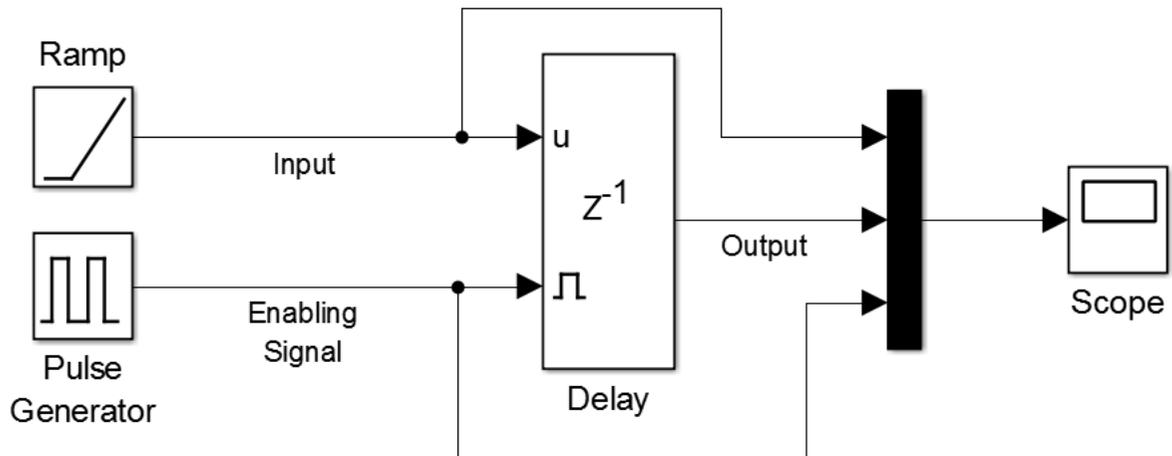
Each Constant block supplies an input signal to the Bus Creator block, which outputs a two-dimensional bus signal. After the Delay block delays the bus signal by three sample periods, the Bus Selector block separates the bus back into the two original signals.

The model follows the rules for bus signals when the Delay block uses frame-based processing.

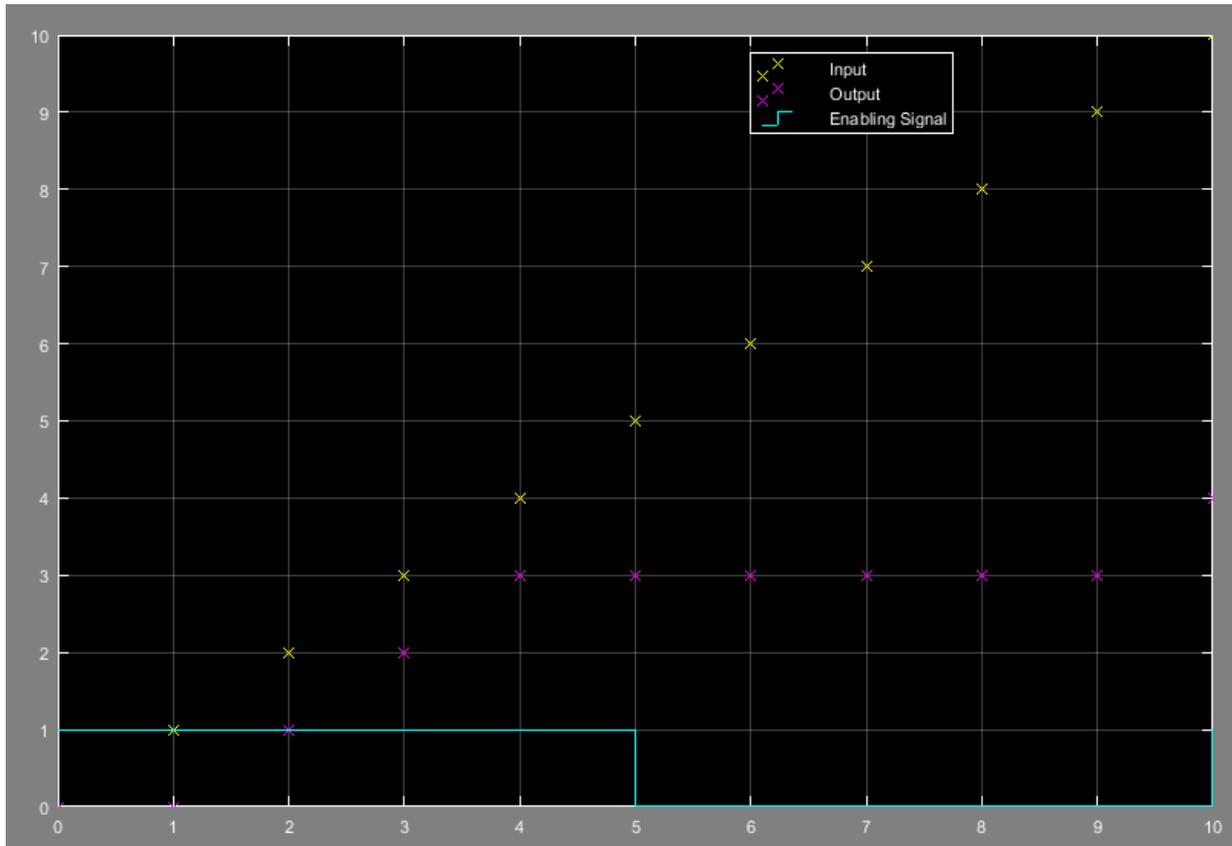
Rule	How the Model Follows the Rule
For the initial condition, set the value on the dialog box.	The Delay block sets Initial condition to 0, a scalar value.
For frame-based processing, signal dimensions of the data input port u cannot be larger than two. (This rule applies to all inputs for the port u , not just bus signals.)	The bus input to the Delay block has two dimensions.

Enable or Disable Execution of the Delay Block

This example shows how you can enable or disable the execution of the Delay block using the enable port of the block. Consider this model. A ramp input signal feeds into a Delay block whose execution is controlled by an enabling signal. A Pulse Generator block generates this enabling signal.



The Scope block displays the output of the Delay block along with the enabling signal and the ramp input. Simulating the model and viewing the scope output shows the following graph.



The magenta marks show that the Delay block outputs the input signal delayed by one time step only while the enabling signal is 1. At $t=5$ sec, the enabling signal becomes 0 and the Delay block does not execute. Hence, the output is held constant until the next time the enabling signal becomes 1.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Specified in the Sample time parameter
Direct Feedthrough	Yes, when you clear Prevent direct feedthrough

Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Resettable Delay | Tapped Delay | Unit Delay | Variable Integer Delay

Demux

Extract and output elements of virtual vector signal

Library: Simulink / Commonly Used Blocks
Simulink / Signal Routing



Description

The Demux block extracts the components of an input vector signal and outputs separate signals. The output signal ports are ordered from top to bottom. See “Mux Signals” for information about creating and decomposing vectors.

Ports

Input

Port_1 — Accept nonbus vector signal to extract and output signals from real or complex values of any nonbus data type supported by Simulink

Output

Port_1 — Output signals extracted from input vector signal nonbus signal with real or complex values of any data type supported by Simulink

Output signals extracted from the input vector. The output signal ports are ordered from top to bottom. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.

Parameters

Number of outputs — Number of outputs

2 (default) | scalar | vector

Specify the number and, optionally, the dimensionality of each output port. If you do not specify the dimensionality of the outputs, the block determines the dimensionality of the outputs.

The value can be a scalar specifying the number of outputs or a vector whose elements specify the widths of the block output ports. The block determines the size of its outputs from the size of the input signal and the value of the **Number of outputs** parameter.

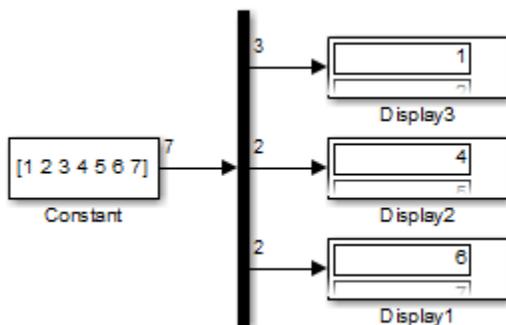
If you specify a scalar for the **Number of outputs** parameter and all of the output ports are connected, as you draw a new signal line close to output side of a Demux block, Simulink adds a port and updates the **Number of outputs** parameter.

For an input vector of width n , here is what the block outputs.

Parameter Value	Block outputs...	Examples and Comments
$p = n$	p scalar signals	If the input is a three-element vector and you specify three outputs, the block outputs three scalar signals.
$p > n$	Error	This value is not supported.
$p < n$ $n \bmod p = 0$	p vector signals each having n/p elements	If the input is a six-element vector and you specify three outputs, the block outputs three two-element vectors.
$p < n$ $n \bmod p = m$	m vector signals each having $(n/p) + 1$ elements and $p - m$ signals having n/p elements	If the input is a five-element vector and you specify three outputs, the block outputs two two-element vector signals and one scalar signal.

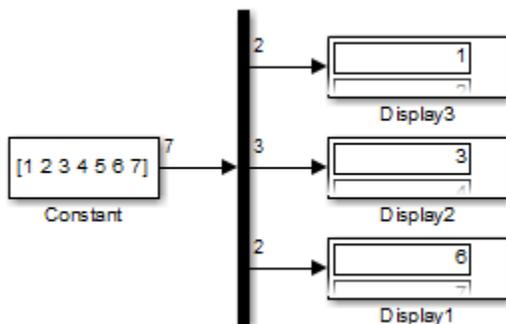
Parameter Value	Block outputs...	Examples and Comments
$[p_1 \ p_2 \ \dots \ p_m]$ $p_1+p_2+\dots+p_m=n$ $p_i > 0$	m vector signals having widths p_1, p_2, \dots, p_m	If the input is a five-element vector and you specify $[3, 2]$ as the output, the block outputs three of the input elements on one port and the other two elements on the other port.
An array that has one or more of m elements with a value of -1 , which specifies that Simulink infers the size for the element. For example, suppose that you have a four-element array with a total width of 14 and you specify the parameter to be $[p_1 \ p_2 \ -1 \ p_4]$. The value for the third element (the -1 element) is $14 - (p_1 + p_2 + p_4)$	m vector signals	If p_i is greater than zero, the corresponding output has width p_i . If p_i is -1 , the width of the corresponding output is computed dynamically.
$[p_1 \ p_2 \ \dots \ p_m]$ $p_1+p_2+\dots+p_m \neq n$ $p_i = > 0$	Error	This value is not supported

If you specify the number of outputs that is smaller than the number of input elements, the block distributes the elements as evenly as possible over the outputs. For example, this model, model distributes the seven input signals as evenly as possible.



You can use `-1` in a vector expression to indicate that the block dynamically sizes the corresponding port. For example, the expression `[-1, 3 -1]` causes the block to output three signals where the second signal always has three elements. The sizes of the first and third signals depend on the size of the input signal.

If a vector expression comprises positive values and `-1` values, the block assigns as many elements as needed to the ports with positive values. The block distributes the remaining elements as evenly as possible over the ports with `-1` values. For example, suppose that the block input is seven elements wide and you specify the output as `[-1, 3 -1]`. In this model, the block outputs two elements on the first port, three elements on the second, and two elements on the third.



Programmatic Use

Block Parameter:Outputs

Type:scalar or vector

Values:character vector

Default:{ '2' } or vector

Display option — Displayed block icon

bar (default) | none

By default, the block icon is a solid bar of the block foreground color. To display the icon as a box containing the block type name, select none.

Programmatic Use

Block Parameter:Display option

Type:character vector

Values:'bar' | 'none'

Default:'bar'

Block Characteristics

Data Types	double single Boolean base integer fixed point enumerated
Multidimensional Signals	No
Variable-Size Signals	No

Extended Capabilities

C/C++ Code Generation

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

Actual data type or capability support depends on block implementation.

HDL Code Generation

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

Fixed-Point Conversion

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

Actual data type or capability support depends on block implementation.

See Also

Bus Creator | Bus to Vector | Mux

Topics

“Virtual Signals”

“Simplify Subsystem Bus Interfaces”

“Composite Signals”

feedbacksystem

Introduced before R2006a

Derivative

Output time derivative of input



Library

Continuous

Description

The Derivative block approximates the derivative of the input signal u with respect to the simulation time t . You obtain the approximation of

$$\frac{du}{dt},$$

by computing a numerical difference $\Delta u/\Delta t$, where Δu is the change in input value and Δt is the change in time since the previous simulation (major) time step.

This block accepts one input and generates one output. The initial output for the block is zero.

The precise relationship between the input and output of this block is:

$$y(t) = \frac{\Delta u}{\Delta t} = \frac{u(t) - u(T_{previous})}{t - T_{previous}} \Big|_{t > T_{previous}},$$

where t is the current simulation time and $T_{previous}$ is the time of the last output time of the simulation. The latter is the same as the time of the last major time step.

The Derivative block output might be very sensitive to the dynamics of the entire model. The accuracy of the output signal depends on the size of the time steps taken in the

simulation. Smaller steps allow a smoother and more accurate output curve from this block. However, unlike with blocks that have continuous states, the solver does not take smaller steps when the input to this block changes rapidly. Depending on the dynamics of the driving signal and model, the output signal of this block might contain unexpected fluctuations. These fluctuations are primarily due to the driving signal output and solver step size.

Because of these sensitivities, structure your models to use integrators (such as Integrator blocks) instead of Derivative blocks. Integrator blocks have states that allow solvers to adjust step size and improve accuracy of the simulation. See “Circuit Model” for an example of choosing the best-form mathematical model to avoid using Derivative blocks in your models,

If you must use the Derivative block with a variable step solver, set the solver maximum step size settings to a value such that the Derivative block can generate answers with adequate accuracy. To determine this value, you might need to repeatedly run the simulation using different solver settings.

When the input to this block is a discrete signal, the continuous derivative of the input exhibits an impulse when the value of the input changes. Otherwise, it is 0. Alternatively, you can define the discrete derivative of a discrete signal using the difference of the last two values of the signal, as follows:

$$y(k) = \frac{1}{\Delta t}(u(k) - u(k - 1))$$

Taking the z -transform of this equation results in:

$$\frac{Y(z)}{u(z)} = \frac{1 - z^{-1}}{\Delta t} = \frac{z - 1}{\Delta t \cdot z}$$

The Discrete Derivative block models this behavior. Use this block instead of the Derivative block to approximate the discrete-time derivative of a discrete signal.

Improved Linearization with Transfer Fcn Blocks

The Laplace domain transfer function for the operation of differentiation is:

$$\frac{Y(s)}{X(s)} = s$$

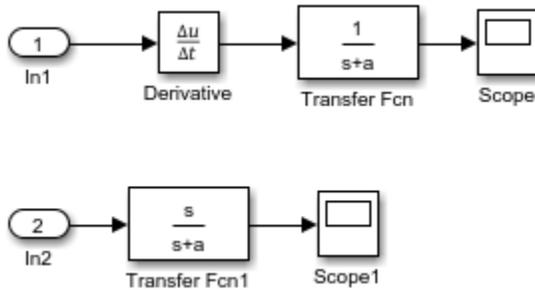
This equation is not a proper transfer function, nor does it have a state-space representation. As such, the Simulink software linearizes this block as an effective gain

of 0 unless you explicitly specify that a proper first-order transfer function should be used to approximate the linear behavior of this block (see “Coefficient c in the transfer function approximation $s/(c*s + 1)$ used for linearization” on page 1-306).

To improve linearization, you can also try to incorporate the derivative term in other blocks. For example, if you have a Derivative block in series with a Transfer Fcn block, try using a single Transfer Fcn block of the form

$$\frac{s}{s+a}$$

For example, you can replace the first set of blocks in this figure with the blocks below them.



Data Type Support

The Derivative block accepts and outputs a real signal of type `double`. For more information, see “Data Types Supported by Simulink”.

Parameters

Coefficient c in the transfer function approximation $s/(c*s + 1)$ used for linearization

Specify the time constant c to approximate the linearization of your system.

Default: `inf`

- The exact linearization of the Derivative block is difficult, because the dynamic equation for the block is $y = \dot{u}$, which you cannot represent as a state-space system. However, you can approximate the linearization by adding a pole to the Derivative to create a transfer function $s / (c * s + 1)$. The addition of a pole filters the signal before differentiating it, which removes the effect of noise.
- The default value `inf` corresponds to a linearization of 0.
- A best practice is to change the value of c to $\frac{1}{f_b}$, where f_b is the break frequency for the filter.
- **Coefficient c in the transfer function approximation $s/(c*s+1)$ used for linearization** must be a finite positive value. This value must be nonzero.

Parameter: `CoefficientInTFapproximation`

Type: character vector

Value: `'inf'`

Default: `'inf'`

Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

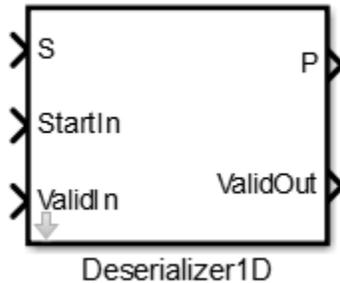
See Also

Discrete Derivative

Introduced before R2006a

Deserializer1D

Convert scalar stream or smaller vectors to vector signal



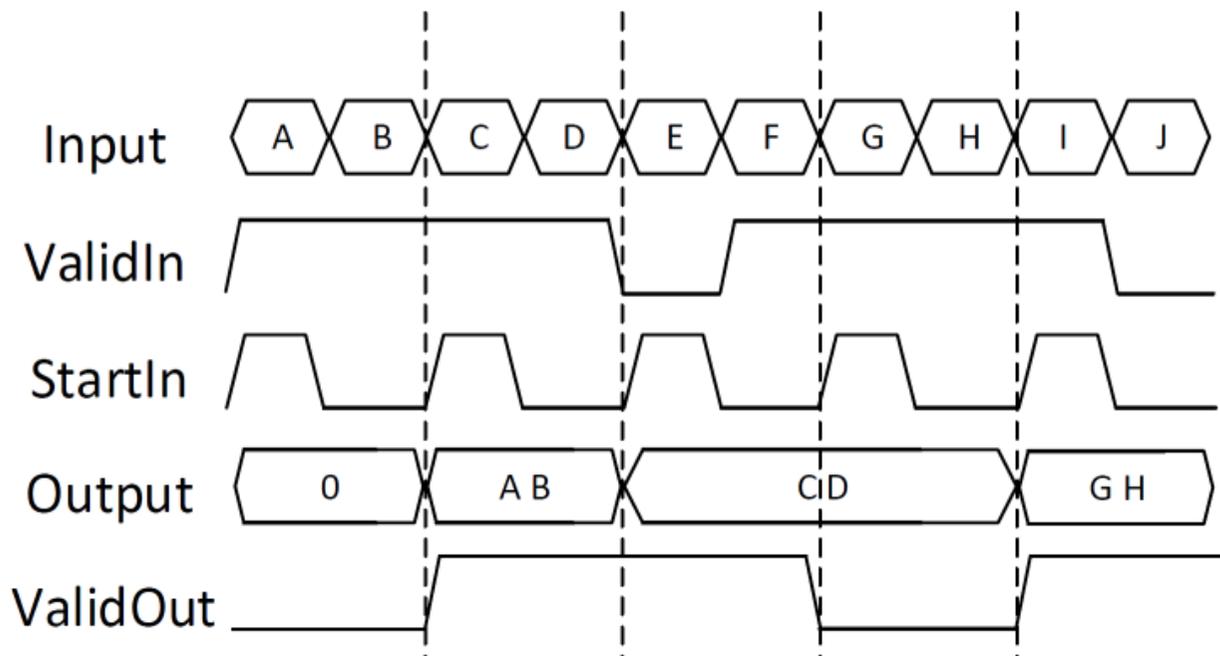
Library

HDL Coder / HDL Operations

Description

The Deserializer1D block buffers a faster, scalar stream or vector signals into a larger, slower vector signal. The faster input signal is converted to a slower signal based on the **Ratio** and **Idle Cycle** values, the conversion changes sample time. Also, the output signal is delayed one slow signal cycle because the serialized data needs to be collected before it can be output as a vector. See the examples below for more details.

You can configure the deserialization to depend on a valid input signal ValidIn and a start signal StartIn. If the **ValidIn** and **StartIn** block parameters are both selected, data collection starts only if both ValidIn and StartIn signals are true. Consider this example:

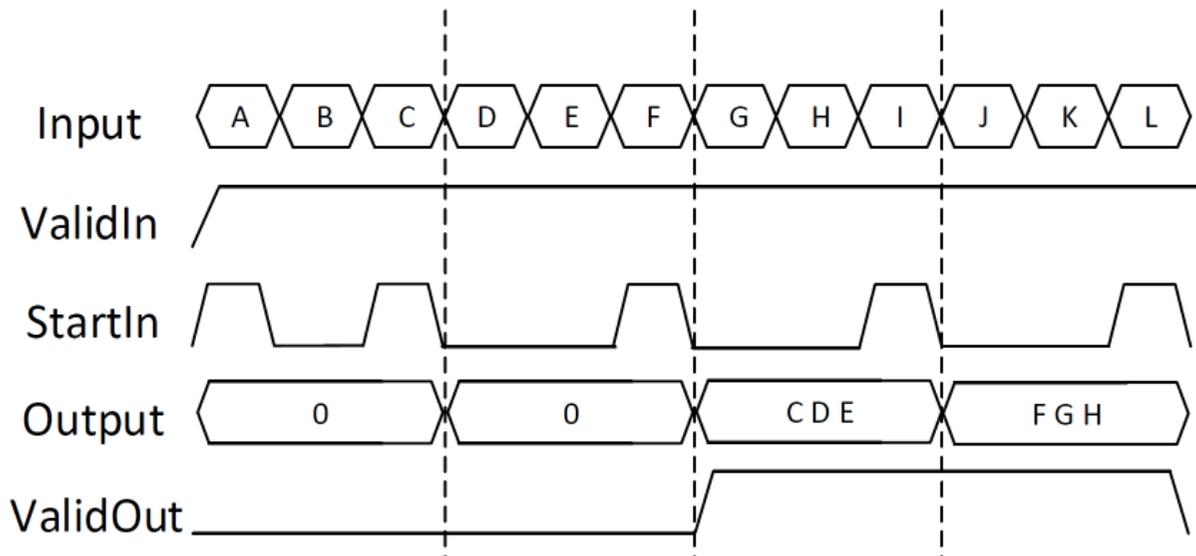


- **Ratio** is 2 and **Idle Cycles** is 0, so each output cycle is two input signals long with all data points considered.
- **ValidIn** and **StartIn** are selected, so data collection can begin only when both StartIn and ValidIn signals are true.
- **ValidOut** is selected.

In the first cycle, ValidIn and StartIn are true, so data collection begins for A and B. The block outputs the deserialized vector in the next valid cycle, so the AB vector is output in the next cycle. This is also true in the second cycle for C and D.

In the third cycle, starting at E, StartIn is true, but ValidIn is not. E is dropped. At F, ValidIn is true, but StartIn is not, so F is also dropped. Since it cannot collect data for E or F, Deserializer1D outputs the previous cycle vector, CD, but ValidOut changes to false.

Another scenario to consider is when the StartIn signal arrives too early. If the length between two StartIn signals is not long enough to collect a full ratio cycle, the insufficient signal data is dropped. Consider this example:



- **Ratio** is 3, so each cycle is two sections long.
- **Idle Cycles** is 0, so all data inputs are considered.
- **ValidIn** and **StartIn** are selected, so data collection can begin only when both StartIn and ValidIn signals are true.
- **ValidOut** is selected.

In the first cycle, ValidIn and StartIn are true, so data collection can begin for A and B. However, at C another StartIn signal arrives before three signals can be collected. Because the StartIn arrived early, A and B are dropped and no valid vector is collected during the first cycle. Therefore, the output of the second cycle is still zero. Deserialization begins at the StartIn at C, for C, D, and E. This vector is output at the next valid cycle, which is cycle 3. Similarly, deserialization starts again at the StartIn at F, and outputs the FGH vector in the fourth cycle.

You specify the block output for the first sampling period with the value of the **Initial condition** parameter.

Parameters

Ratio

Enter the deserialization ratio. Default is 1.

The ratio is the output vector size, divided by the input vector size. The ratio must be divisible by the input vector size.

Idle Cycles

Enter the number of idle cycles added to the end of each serialized input. Default is 0.

The value of **Idle Cycles** affects the deserialized output rate. For example, if **Ratio** is 2 and the input signal is A, B, B, C, D, D, . . . , without idle cycles the output would be AB, BC, DD. . . . However for the same input and ratio with **Idle Cycles** set to 1, the output is AB, CD. . . . The idle cycles, B and D, are dropped.

The Deserializer1D behavior changes if **Idle Cycles** is not zero, and **ValidIn** or **StartIn** are on. The idle cycles value affects only the output rate, while **ValidIn** and **StartIn** control what input data is deserialized.

Initial condition

Specify the initial output of the simulation. Default is 0.

StartIn

Select to activate the StartIn port. Default is off.

ValidIn

Select to activate the ValidIn port. Default is off.

ValidOut

Select to activate ValidOut port. Default is off.

Input data port dimensions (-1 for inherited)

Enter the size of the input data signal. The input size must be divisible by the ratio plus the number of idle cycles. By default, the block inherits size based on context within the model.

Input sample time (-1 for inherited)

Enter the time interval between sample time hits or specify another appropriate sample time such as continuous. By default, the block inherits its sample time based on context within the model. For more information, see “Sample Time”.

Input signal type

Specify the input signal type of the block as `auto`, `real`, or `complex`.

Ports

S

Input signal to deserialize. Bus data types are not supported.

ValidIn

Indicates valid input signal. Use with the Deserializer1D block. This port is available when you select the **ValidIn** check box.

Data type: Boolean

StartOut

Indicates where to start deserialization. Use with the Deserializer1D block. This port is available when you select the **StartOut** check box.

Data type: Boolean

P

Deserialized output signal. Bus data types are not supported.

ValidOut

Indicates valid output signal. This port is available when you select the **ValidOut** check box.

Data type: Boolean

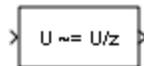
See Also

Serializer1D

Introduced in R2014b

Detect Change

Detect change in signal value



Library

Logic and Bit Operations

Description

The Detect Change block determines if an input does not equal its previous value.

- The output is true (equal to 1) when the input signal does not equal its previous value.
- The output is false (equal to 0) when the input signal equals its previous value.

Data Type Support

The Detect Change block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Initial condition

Set the initial condition for the previous input U/z .

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).
- **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

- **Inherited** — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the **Inherited** option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input u . All other input signals must be sample based.

Input Signal u	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes

Input Signal u	Input Processing Mode	Block Works?
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Code Generation	Yes

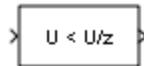
See Also

Detect Decrease, Detect Fall Negative, Detect Fall Nonpositive, Detect Increase, Detect Rise Nonnegative, Detect Rise Positive

Introduced before R2006a

Detect Decrease

Detect decrease in signal value



Library

Logic and Bit Operations

Description

The Detect Decrease block determines if an input is strictly less than its previous value.

- The output is true (equal to 1) when the input signal is less than its previous value.
- The output is false (equal to 0) when the input signal is greater than or equal to its previous value.

Data Type Support

The Detect Decrease block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Initial condition

Set the initial condition for the previous input U/z .

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- `Elements as channels (sample based)` — Treat each element of the input as a separate channel (sample-based processing).
- `Columns as channels (frame based)` — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

- `Inherited` — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input u . All other input signals must be sample based.

Input Signal u	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes

Input Signal u	Input Processing Mode	Block Works?
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Code Generation	Yes

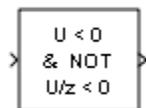
See Also

Detect Change, Detect Fall Negative, Detect Fall Nonpositive, Detect Increase, Detect Rise Nonnegative, Detect Rise Positive

Introduced before R2006a

Detect Fall Negative

Detect falling edge when signal value decreases to strictly negative value, and its previous value was nonnegative



Library

Logic and Bit Operations

Description

The Detect Fall Negative block determines if the input is less than zero, and its previous value was greater than or equal to zero.

- The output is true (equal to 1) when the input signal is less than zero, and its previous value was greater than or equal to zero.
- The output is false (equal to 0) when the input signal is greater than or equal to zero, or if the input signal is negative, its previous value was also negative.

Data Type Support

The Detect Fall Negative block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Initial condition

Set the initial condition of the Boolean expression $U/z < 0$.

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- `Elements as channels (sample based)` — Treat each element of the input as a separate channel (sample-based processing).
- `Columns as channels (frame based)` — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

- `Inherited` — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input `u`. All other input signals must be sample based.

Input Signal <code>u</code>	Input Processing Mode	Block Works?
Sample based	Sample based	Yes

Input Signal u	Input Processing Mode	Block Works?
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Code Generation	Yes

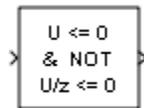
See Also

Detect Change, Detect Decrease, Detect Fall Nonpositive, Detect Increase, Detect Rise Nonnegative, Detect Rise Positive

Introduced before R2006a

Detect Fall Nonpositive

Detect falling edge when signal value decreases to nonpositive value, and its previous value was strictly positive



Library

Logic and Bit Operations

Description

The Detect Fall Nonpositive block determines if the input is less than or equal to zero, and its previous value was greater than zero.

- The output is true (equal to 1) when the input signal is less than or equal to zero, and its previous value was greater than zero.
- The output is false (equal to 0) when the input signal is greater than zero, or if it is nonpositive, its previous value was also nonpositive.

Data Type Support

The Detect Fall Nonpositive block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Initial condition

Set the initial condition of the Boolean expression $U/z \leq 0$.

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- `Elements as channels (sample based)` — Treat each element of the input as a separate channel (sample-based processing).
- `Columns as channels (frame based)` — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

- `Inherited` — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input `u`. All other input signals must be sample based.

Input Signal <code>u</code>	Input Processing Mode	Block Works?
Sample based	Sample based	Yes

Input Signal u	Input Processing Mode	Block Works?
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Code Generation	Yes

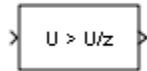
See Also

Detect Change, Detect Decrease, Detect Fall Negative, Detect Increase, Detect Rise Nonnegative, Detect Rise Positive

Introduced before R2006a

Detect Increase

Detect increase in signal value



Library

Logic and Bit Operations

Description

The Detect Increase block determines if an input is strictly greater than its previous value.

- The output is true (equal to 1) when the input signal is greater than its previous value.
- The output is false (equal to 0) when the input signal is less than or equal to its previous value.

Data Type Support

The Detect Increase block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Initial condition

Set the initial condition for the previous input U/z .

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- `Elements as channels (sample based)` — Treat each element of the input as a separate channel (sample-based processing).
- `Columns as channels (frame based)` — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

- `Inherited` — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input u . All other input signals must be sample based.

Input Signal u	Input Processing Mode	Block Works?
Sample based	Sample based	Yes

Input Signal u	Input Processing Mode	Block Works?
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Code Generation	Yes

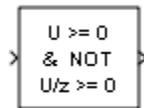
See Also

Detect Change, Detect Decrease, Detect Fall Negative, Detect Fall Nonpositive, Detect Rise Nonnegative, Detect Rise Positive

Introduced before R2006a

Detect Rise Nonnegative

Detect rising edge when signal value increases to nonnegative value, and its previous value was strictly negative



Library

Logic and Bit Operations

Description

The Detect Rise Nonnegative block determines if the input is greater than or equal to zero, and its previous value was less than zero.

- The output is true (equal to 1) when the input signal is greater than or equal to zero, and its previous value was less than zero.
- The output is false (equal to 0) when the input signal is less than zero, or if the input signal is nonnegative, its previous value was also nonnegative.

Data Type Support

The Detect Rise Nonnegative block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Initial condition

Set the initial condition of the Boolean expression $U/z \geq 0$.

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- `Elements as channels (sample based)` — Treat each element of the input as a separate channel (sample-based processing).
- `Columns as channels (frame based)` — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

- `Inherited` — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input u . All other input signals must be sample based.

Input Signal u	Input Processing Mode	Block Works?
Sample based	Sample based	Yes

Input Signal u	Input Processing Mode	Block Works?
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Code Generation	Yes

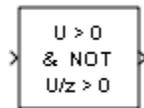
See Also

Detect Change, Detect Decrease, Detect Fall Negative, Detect Fall Nonpositive, Detect Increase, Detect Rise Positive

Introduced before R2006a

Detect Rise Positive

Detect rising edge when signal value increases to strictly positive value, and its previous value was nonpositive



Library

Logic and Bit Operations

Description

The Detect Rise Positive block determines if the input is strictly positive, and its previous value was nonpositive.

- The output is true (equal to 1) when the input signal is greater than zero, and the previous value was less than or equal to zero.
- The output is false (equal to 0) when the input is negative or zero, or if the input is positive, the previous value was also positive.

Data Type Support

The Detect Rise Positive block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Initial condition

Set the initial condition of the Boolean expression $U/z > 0$.

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- `Elements as channels (sample based)` — Treat each element of the input as a separate channel (sample-based processing).
- `Columns as channels (frame based)` — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

- `Inherited` — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input `u`. All other input signals must be sample based.

Input Signal <code>u</code>	Input Processing Mode	Block Works?
Sample based	Sample based	Yes

Input Signal u	Input Processing Mode	Block Works?
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Output data type

Set the output data type to `boolean` or `uint8`.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Code Generation	Yes

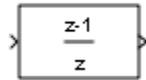
See Also

Detect Change, Detect Decrease, Detect Fall Negative, Detect Fall Nonpositive, Detect Increase, Detect Rise Nonnegative

Introduced before R2006a

Difference

Calculate change in signal over one time step



Library

Discrete

Description

The Difference block outputs the current input value minus the previous input value.

Data Type Support

The Difference block accepts signals of any numeric data type that Simulink supports, including fixed-point data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Initial condition for previous input

Set the initial condition for the previous input.

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).
- **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

- **Inherited** — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the **Inherited** option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input *u*. All other input signals must be sample based.

Input Signal <i>u</i>	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Output minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” in the *Simulink User's Guide* for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

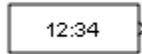
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Code Generation	Yes

Introduced before R2006a

Digital Clock

Output simulation time at specified sampling interval



Library

Sources

Description

The Digital Clock block outputs the simulation time only at the specified sampling interval. At other times, the block holds the output at the previous value. To control the precision of this block, set the **Sample time** parameter in the block dialog box.

Use this block rather than the Clock block (which outputs continuous time) when you need the current simulation time within a discrete system.

Data Type Support

The Digital Clock block outputs a real signal of type `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Sample time

Specify the sampling interval. The default value is 1 second. For more information, see [Specifying Sample Time](#) in the Simulink documentation.

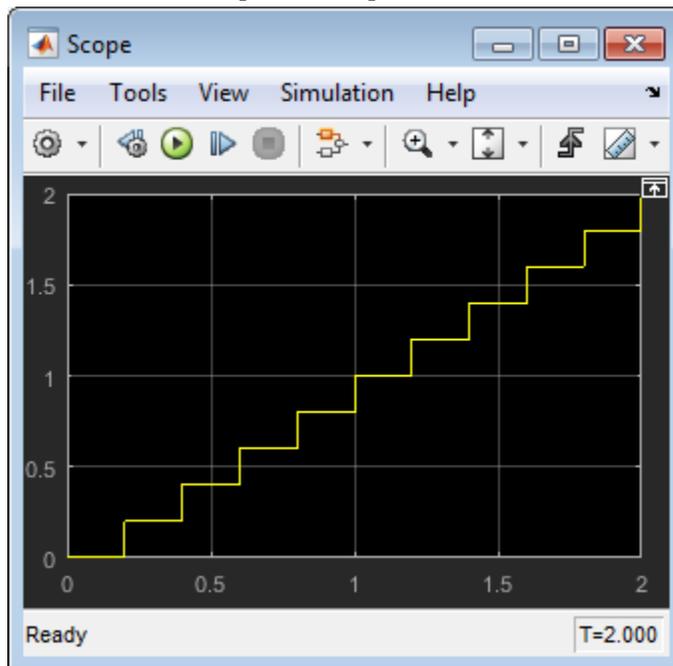
Do not specify a continuous sample time, either 0 or $[0, 0]$. Also, avoid specifying -1 (inheriting the sample time) because this block is a source.

Examples

In the following model, the Scope block shows the output of a Digital Clock block with a **Sample time** of 0.2.



The Digital Clock block outputs the simulation time every 0.2 seconds. Otherwise, the block holds the output at the previous value.



Characteristics

Data Types	Double
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

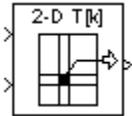
See Also

Clock

Introduced before R2006a

Direct Lookup Table (n-D)

Index into N-dimensional table to retrieve element, column, or 2-D matrix



Library

Lookup Tables

Description

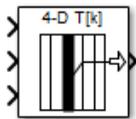
Block Inputs and Outputs

The Direct Lookup Table (n-D) block uses inputs as zero-based indices into an n -dimensional table. The number of inputs varies with the shape of the output: an element, column, or 2-D matrix.

You define a set of output values as the **Table data** parameter. The first input specifies the zero-based index to the table dimension that is *one higher* than the output dimensionality. The next input specifies the zero-based index to the next table dimension, and so on.

Output Shape	Output Dimensionality	Table Dimension That Maps to the First Input
Element	0	1
Column	1	2
Matrix	2	3

Suppose that you want to select a column of values from a 4-D table:



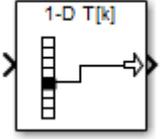
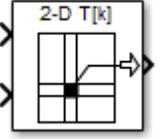
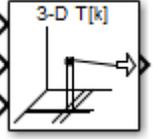
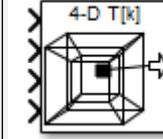
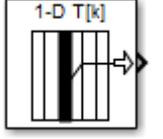
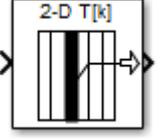
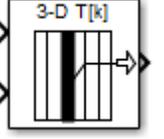
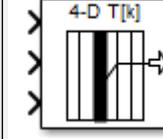
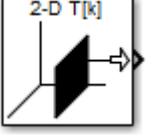
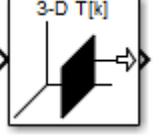
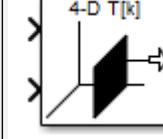
The following mapping of block input port to table dimension applies:

This input port...	Is the index for this table dimension...
1	2
2	3
3	4

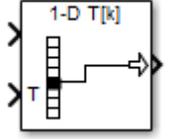
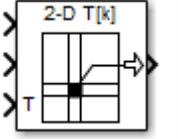
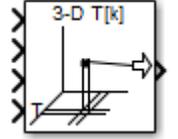
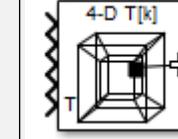
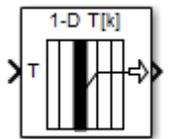
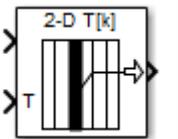
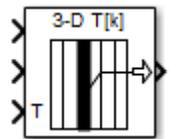
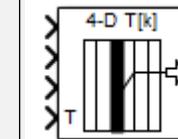
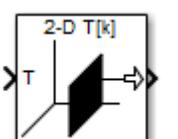
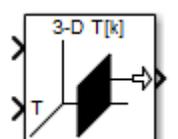
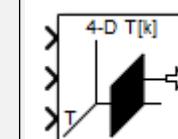
Changes in Block Icon Appearance

Depending on parameters you set, the block icon changes appearance. For table dimensions higher than 4, the icon matches the 4-D version but shows the exact number of dimensions in the top text.

When you use the **Table data** parameter, you see the following icons:

Object That Inputs Select from the Table	Number of Table Dimensions			
	1	2	3	4
Element				
Column				
2-D Matrix	Not applicable			

When you use the table input port, you see the following icons:

Object That Inputs Select from the Table	Number of Table Dimensions			
	1	2	3	4
Element				
Column				
2-D Matrix	Not applicable			

Data Type Support

The Direct Lookup Table (n-D) block accepts input signals of different data types.

Type of Input Port	Data Types Supported
Index port	<ul style="list-style-type: none"> • Floating point • Built-in integer • Boolean • Enumerated data types

Type of Input Port	Data Types Supported
Table port (with the label T)	<ul style="list-style-type: none"> • Floating point • Built-in integer • Fixed point • Boolean • Enumerated data types

The output data type is the same as the table data type. Inputs for indexing must be real, but table data can be complex.

When the table data is...	The block inherits the output type from...
Not an input	The Table data type parameter
An input	The table input port

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

- “Main tab” on page 1-345
- “Table Attributes tab” on page 1-347

Main tab

Number of table dimensions

Specify the number of dimensions that the **Table data** parameter must have. This value determines the number of independent variables for the table and the number of inputs to the block.

To specify...	Do this...
1, 2, 3, or 4	Select the value from the drop-down list.
A higher number of table dimensions	Enter a positive integer directly in the field. The maximum number of table dimensions that this block supports is 30.

Inputs select this object from table

Specify whether the output data is a single element, a column, or a 2-D matrix. The number of input ports for indexing depends on your selection.

Selection	Number of Input Ports for Indexing
Element	Number of table dimensions
Column	Number of table dimensions – 1
2-D Matrix	Number of table dimensions – 2

This numbering matches MATLAB indexing. For example, if you have a 4-D table of data, follow these guidelines:

To access...	Specify...	As in...
An element	Four indices	<code>array(1,2,3,4)</code>
A column	Three indices	<code>array(:,2,3,4)</code>
A 2-D matrix	Two indices	<code>array(:, :, 3, 4)</code>

Make table an input

Select this check box to force the Direct Lookup Table (n-D) block to ignore the **Table data** parameter. Instead, a new input port appears with **T** next to it. Use this port to input table data.

Table data

Specify the table of output values. The matrix size must match the dimensions of the **Number of table dimensions** parameter. The **Table data** field is available only if you clear the **Make table an input** check box.

Tip During block diagram editing, you can leave the **Table data** field empty. But for simulation, you must match the number of dimensions in **Table data** to the **Number of table dimensions**. For details on how to construct multidimensional MATLAB arrays, see “Multidimensional Arrays” (MATLAB) in the MATLAB documentation.

Click **Edit** to open the Lookup Table Editor. For more information, see “Edit Lookup Tables” in the Simulink documentation.

Diagnostic for out-of-range input

Specify whether to show a warning or error when an index is out of range with respect to the table dimension. Options include:

- `None` — Simulink software takes no action.
- `Warning` — Simulink software displays a warning and continues the simulation.
- `Error` — Simulink software terminates the simulation and displays an error.

When you select `None` or `Warning`, the block clamps out-of-range indices to fit table dimensions. For example, if the specified index is 5.3 and the maximum index for that table dimension is 4, the block clamps the index to 4.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Table Attributes tab

Note The parameters in the **Table Attributes** pane are not available if you select **Make table an input**. In this case, the block inherits all table attributes from the input port with the label `T`.

Table minimum

Specify the minimum value for table data. The default value is `[]` (unspecified).

Table maximum

Specify the maximum value for table data. The default value is `[]` (unspecified).

Table data type

Specify the table data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit from 'Table data'`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Table data type** parameter.

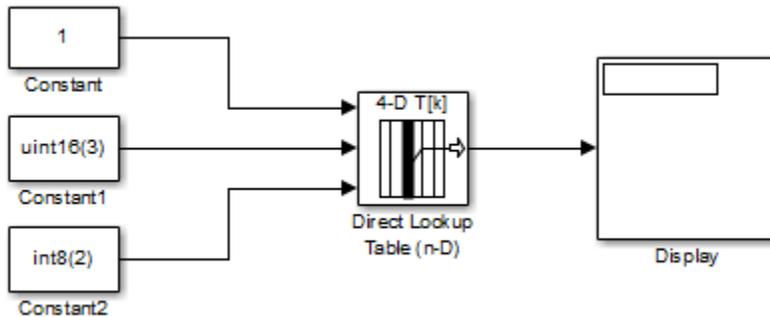
Lock data type settings against changes by the fixed-point tools

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Examples

When Table Data Is Not an Input

Suppose that you have the following model:



The Direct Lookup Table (n-D) block parameters are:

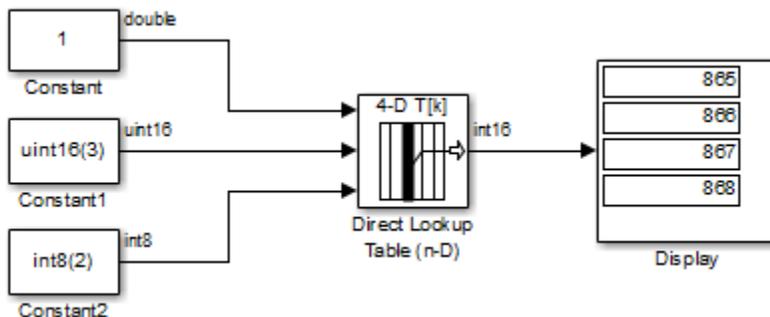
Block Parameter	Value
Number of table dimensions	4
Inputs select this object from table	Column
Make table an input	off
Table data	a
Diagnostic for out-of-range input	Warning
Sample time	-1
Table minimum	[]

Block Parameter	Value
Table maximum	[]
Table data type	int16
Lock data type settings against changes by the fixed-point tools	off

In this example, `a` is a 4-D array of linearly increasing values that you define with the following model preload function:

```
a = reshape(1:2800, [4 5 20 7]);
```

When you run the model, you get the following results:



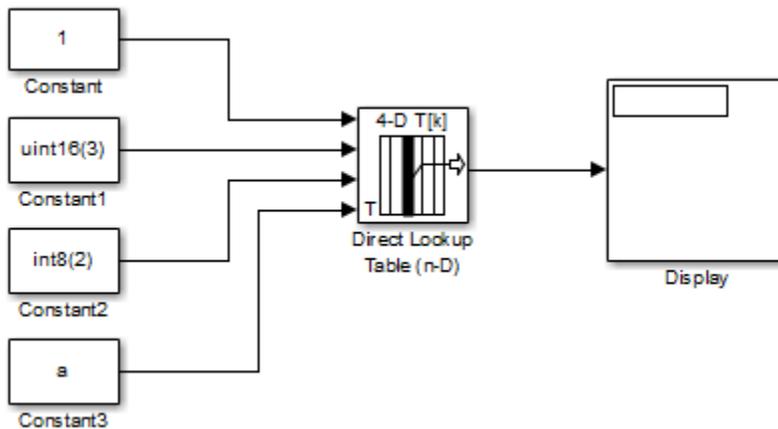
Because the Direct Lookup Table (n-D) block uses zero-based indexing, the output is:

```
a(:,2,4,3)
```

The output has the same data type as the table: `int16`.

When Table Data Is an Input

Suppose that you have the following model:



The Direct Lookup Table (n-D) block parameters are:

Block Parameter	Value
Number of table dimensions	4
Inputs select this object from table	Column
Make table an input	on
Diagnostic for out-of-range input	Warning
Sample time	-1

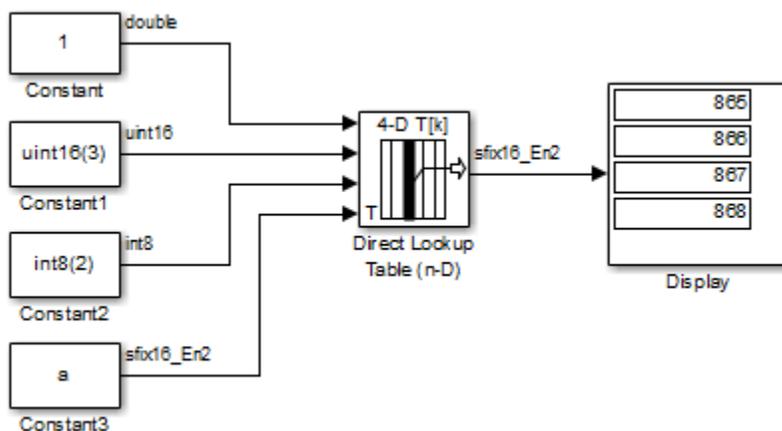
The key parameters of the Constant3 block are:

Block Parameter	Value
Constant value	a
Output data type	fixdt(1,16,2)

In this example, a is a 4-D array of linearly increasing values that you define with the following model preload function:

```
a = reshape(1:2800, [4 5 20 7]);
```

When you run the model, you get the following results:



The Constant3 block feeds the 4-D array to the Direct Lookup Table (n-D) block, using the fixed-point data type `fixdt(1, 16, 2)`. Because the Direct Lookup Table (n-D) block uses zero-based indexing, the output is:

```
a(:, 2, 4, 3)
```

The output has the same data type as the table: `fixdt(1, 16, 2)`.

Characteristics

Data Types	Double Single Boolean Base Integer Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

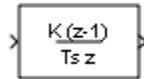
See Also

n-D Lookup Table

Introduced before R2006a

Discrete Derivative

Compute discrete-time derivative



Library

Discrete

Description

The Discrete Derivative block computes an optionally scaled discrete time derivative as follows

$$y(t_n) = \frac{Ku(t_n)}{T_s} - \frac{Ku(t_{n-1})}{T_s}$$

where

- $u(t_n)$ and $y(t_n)$ are the block's input and output at the current time step, respectively.
- $u(t_{n-1})$ is the block's input at the previous time step.
- K is a scaling factor.
- T_s is the simulation's discrete step size, which must be fixed.

Note Do not use this block in subsystems with a non-periodic trigger (for example, non-periodic function-call subsystems). This configuration will produce inaccurate results.

Data Type Support

The Discrete Derivative block supports all numeric Simulink data types, including fixed-point data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Gain value

Scaling factor used to weight the block's input at the current time step.

Initial condition for previous weighted input $K*u/Ts$

Set the initial condition for the previous scaled input.

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).
- **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

- **Inherited** — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the **Inherited** option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input u . All other input signals must be sample based.

Input Signal u	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Output minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” in *Simulink User's Guide* for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No

Variable-Size Signals	Yes
Code Generation	Yes

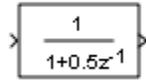
See Also

Derivative

Introduced before R2006a

Discrete Filter

Model Infinite Impulse Response (IIR) filters



Library

Discrete

Description

The Discrete Filter block independently filters each channel of the input signal with the specified digital IIR filter. You can specify the filter structure as one of | Direct form I | Direct form I transposed | Direct form II | Direct form II transposed. The block implements static filters with fixed coefficients. You can tune the coefficients of these static filters.

This block filters each channel of the input signal independently over time. The **Input processing** parameter allows you to specify how the block treats each element of the input. You can specify treating input elements as an independent channel (sample-based processing), or treating each column of the input as an independent channel (frame-based processing). To perform frame-based processing, you must have a DSP System Toolbox license.

The output dimensions equal those of the input, except when you specify a matrix of filter taps for the **Numerator coefficients** parameter. When you do so, the output dimensions depend on the number of different sets of filter taps you specify.

Use the **Numerator coefficients** parameter to specify the coefficients of the discrete filter numerator polynomial. Use the **Denominator coefficients** parameter to specify the coefficients of the denominator polynomial of the function. The **Denominator coefficients** parameter must be a vector of coefficients.

Specify the coefficients of the numerator and denominator polynomials in ascending powers of z^{-1} . The Discrete Filter block lets you use polynomials in z^{-1} (the delay operator) to represent a discrete system. This method is the one that signal processing engineers typically use. Conversely, the Discrete Transfer Fcn block lets you use polynomials in z to represent a discrete system. This method is the one that control engineers typically use. When the numerator and denominator polynomials have the same length, the two methods are identical.

Specifying Initial States

In **Dialog parameters** and **Input port(s)** modes, the block initializes the internal filter states to zero by default, which is equivalent to assuming past inputs and outputs are zero. You can optionally use the **Initial states** parameter to specify nonzero initial states for the filter delays.

To determine the number of initial state values you must specify, and how to specify them, see the following table on Valid Initial States and Number of Delay Elements (Filter States). The **Initial states** parameter can take one of four forms as described in the following table.

Valid Initial States

Initial state	Examples	Description
Scalar	5 Each delay element for each channel is set to 5.	The block initializes all delay elements in the filter to the scalar value.
Vector (for applying the same delay elements to each channel)	For a filter with two delay elements: $[d_1 \ d_2]$ The delay elements for all channels are d_1 and d_2 .	Each vector element specifies a unique initial condition for a corresponding delay element. The block applies the same vector of initial conditions to each channel of the input signal. The vector length must equal the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)).
Vector or matrix (for applying different delay elements to each channel)	For a 3-channel input signal and a filter with two delay elements: $[d_1 \ d_2 \ D_1 \ D_2 \ d_1 \ d_2]$ or $\begin{bmatrix} d_1 & D_1 & d_1 \\ d_2 & D_2 & d_2 \end{bmatrix}$ <ul style="list-style-type: none"> The delay elements for channel 1 are d_1 and d_2. The delay elements for channel 2 are D_1 and D_2. The delay elements for channel 3 are d_1 and d_2. 	Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel: <ul style="list-style-type: none"> The vector length must be equal to the product of the number of input channels and the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)). The matrix must have the same number of rows as the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)), and must have one column for each channel of the input signal.
Empty matrix	$[\]$ Each delay element for each channel is set to 0.	The empty matrix, $[\]$, is equivalent to setting the Initial conditions parameter to the scalar value 0.

The number of delay elements (filter states) per input channel depends on the filter structure, as indicated in the following table.

Number of Delay Elements (Filter States)

Filter Structure	Number of Delay Elements per Channel
Direct form I Direct form I transposed	<ul style="list-style-type: none"> • number of zeros - 1 • number of poles - 1
Direct form II Direct form II transposed	$\max(\text{number of zeros, number of poles}) - 1$

The following tables describe the valid initial states for different sizes of input and different number of channels. These tables provide this information according to whether you set the **Input processing** parameter to frame based or sample based.

Frame-Based Processing

Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
<ul style="list-style-type: none"> • Column vector (K-by-1) • Unoriented vector (K) 	1	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1) • Row vector (1-by-M) 	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1)
<ul style="list-style-type: none"> • Row vector (1-by-N) • Matrix (K-by-N) 	N	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1) • Row vector (1-by-M) • Matrix (M-by-N) 	<ul style="list-style-type: none"> • Scalar • Matrix (M-by-N)

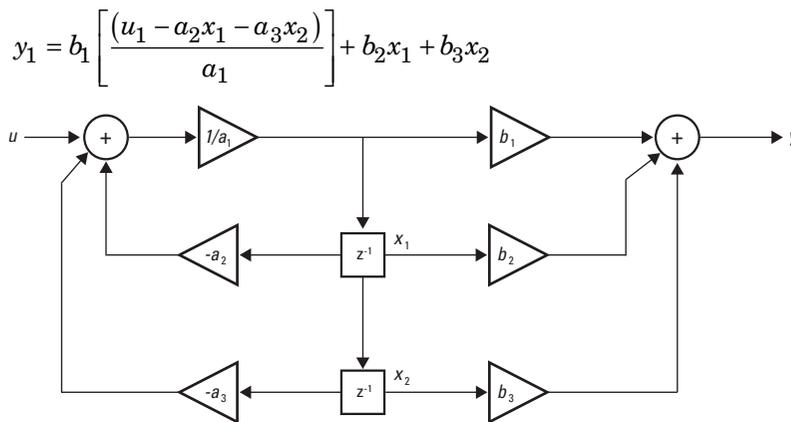
Sample-Based Processing

Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
<ul style="list-style-type: none"> • Scalar 	1	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1) • Row vector (1-by-M) 	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1) • Row vector (1-by-M)
<ul style="list-style-type: none"> • Row vector (1-by-N) • Column vector (N-by-1) • Unoriented vector (N) 	N	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1) • Row vector (1-by-M) • Matrix (M-by-N) 	<ul style="list-style-type: none"> • Scalar
<ul style="list-style-type: none"> • Matrix (K-by-N) 	$K \times N$	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1) • Row vector (1-by-M) • Matrix (M-by-$(K \times N)$) 	<ul style="list-style-type: none"> • Scalar

When the **Initial states** is a scalar, the block initializes all filter states to the same scalar value. Enter 0 to initialize all states to zero. When the **Initial states** is a vector or a matrix, each vector or matrix element specifies a unique initial state. This unique state corresponds to a delay element in a corresponding channel:

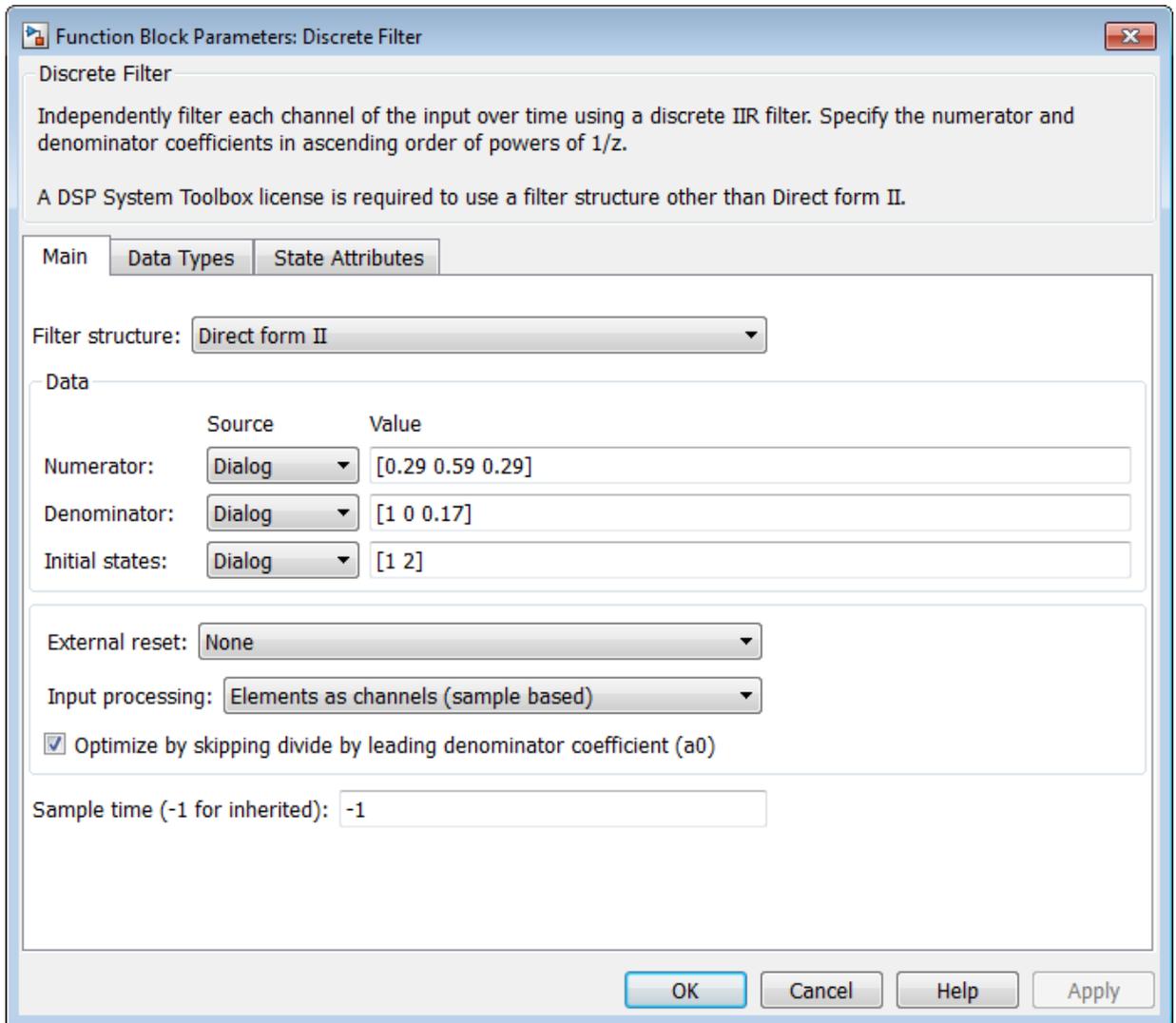
- The vector length must equal the number of delay elements in the filter, $M = \max(\text{number of zeros}, \text{number of poles})$.
- The matrix must have the same number of rows as the number of delay elements in the filter, $M = \max(\text{number of zeros}, \text{number of poles})$. The matrix must also have one column for each channel of the input signal.

The following example shows the relationship between the initial filter output and the initial input and state. Given an initial input u_1 , the first output y_1 is related to the initial state $[x_1, x_2]$ and initial input by:

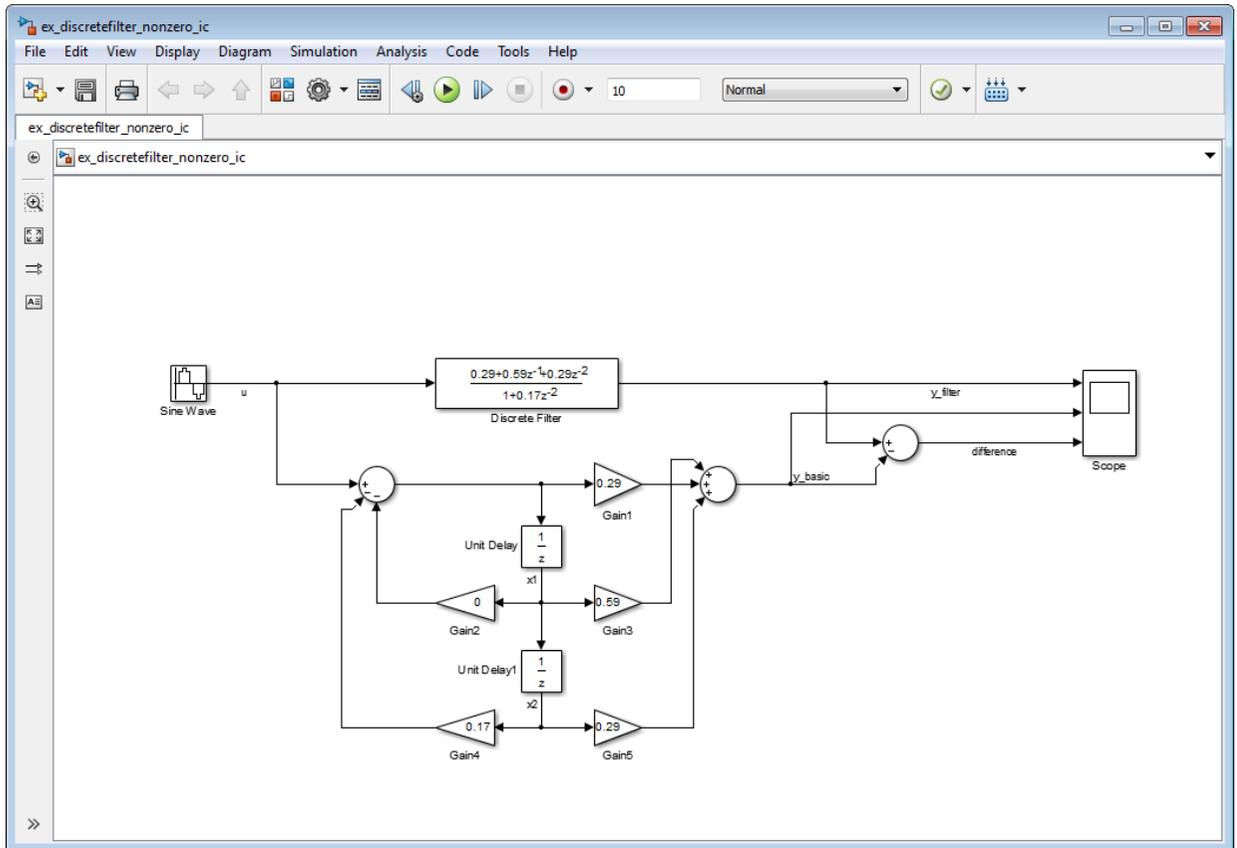


To see an example of how to set initial conditions as a vector:

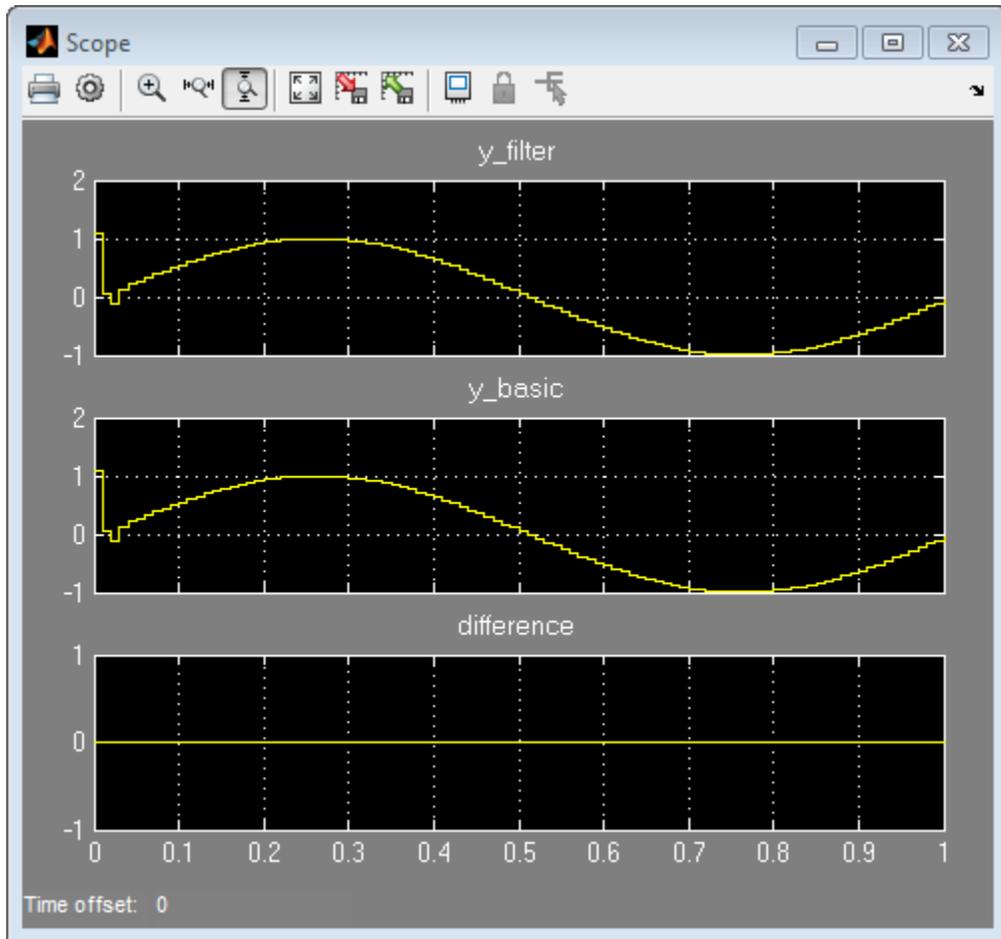
- Click on the model `ex_discretefilter_nonzero_ic`, or type it at the MATLAB command prompt.
- Double-click on the Discrete Filter block, and set the parameters. The following shows how to set the initial conditions of the Discrete Filter block to `[1 2]`.



- Simulate the model, by left-clicking the green simulation icon.



- Double-click the scope. You can see that the difference between the signal filtered by the Discrete Filter block, and the signal from the filter's building blocks, is zero.



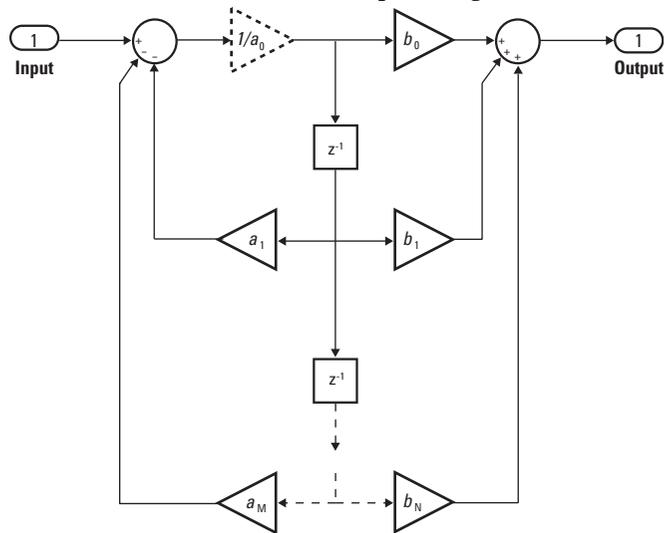
This demonstrates that you can enter the initial conditions of the Discrete Filter block as a vector of $[1 \ 2]$. You can also set the initial condition of the first Unit Delay to 1 and the second Unit Delay to 2. The resulting outputs are the same.

Data Type Support

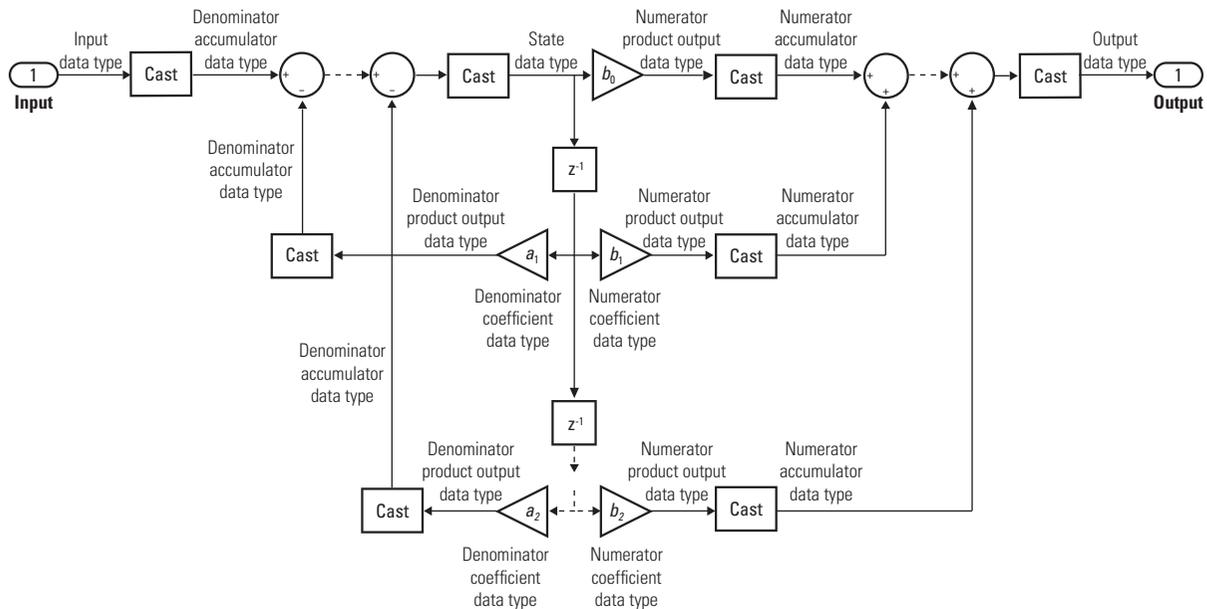
The Discrete Filter block accepts and outputs real and complex signals of any signed numeric data type that Simulink supports. The block supports the same types for the numerator and denominator coefficients.

Numerator and denominator coefficients must have the same complexity. They can have different word lengths and fraction lengths.

The following diagrams show the filter structure and the data types used within the Discrete Filter block for fixed-point signals.



The block omits the dashed divide when you select the **Optimize by skipping divide by leading denominator coefficient (a0)** parameter.



Parameters

Numerator

Numerator coefficients of the discrete filter. To specify the coefficients, set the **Source** to `Dialog`. Then, enter the coefficients in **Value** as descending powers of z . Use a row vector to specify the coefficients for a single numerator polynomial.

Denominator

Denominator coefficients of the discrete filter. To specify the coefficients, set the **Source** to `Dialog`. Then, enter the coefficients in **Value** as descending powers of z . Use a row vector to specify the coefficients for a single denominator polynomial.

Initial states

If the **Source** is `Dialog`, then, in **Value**, specify the initial states of the filter states. To learn how to specify initial states, see “Specifying Initial States” on page 1-359.

If the **Source** is `Input port`, then you do not need to specify **Value**.

External reset

Specify the trigger event to use to reset the states to the initial conditions.

Reset Mode	Behavior
None	No reset.
Rising	Reset on a rising edge.
Falling	Reset on a falling edge.
Either	Reset on either a rising or falling edge.
Level	Reset in either of these cases: <ul style="list-style-type: none"> • when the reset signal is nonzero at the current time step • when the reset signal value changes from nonzero at the previous time step to zero at the current time step
Level hold	Reset when the reset signal is nonzero at the current time step

The reset signal must be a scalar of type single, double, boolean, or integer. Fixed point data types, except for `ufix1`, are not supported.

Input processing

Specify whether the block performs sample- or frame-based processing.

- Elements as channels (sample based) — Process each element of the input as an independent channel.
- Columns as channels (frame based) — Process each column of the input as an independent channel.

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Optimize by skipping divide by leading denominator coefficient (a0)

Select when the leading denominator coefficient, a_0 , equals 1. This parameter optimizes your code.

When you select this check box, the block does not perform a divide-by- a_0 either in simulation or in the generated code. An error occurs if a_0 is not equal to one.

When you clear this check box, the block is fully tunable during simulation. It performs a divide-by- a_0 in both simulation and code generation.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in “How Simulink Works” in the *Simulink User's Guide*.

State

Specify the state data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **State** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

Numerator coefficients

Specify the numerator coefficient data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Numerator coefficients** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

Numerator coefficient minimum

Specify the minimum value that a numerator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

Numerator coefficient maximum

Specify the maximum value that a numerator coefficient can have. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

Numerator product output

Specify the product output data type for the numerator coefficients. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Numerator product output** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

Numerator accumulator

Specify the accumulator data type for the numerator coefficients. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Numerator accumulator** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

Denominator coefficients

Specify the denominator coefficient data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Denominator coefficients** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

Denominator coefficient minimum

Specify the minimum value that a denominator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

Denominator coefficient maximum

Specify the maximum value that a denominator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

Denominator product output

Specify the product output data type for the denominator coefficients. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Denominator product output** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

Denominator accumulator

Specify the accumulator data type for the denominator coefficients. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Denominator accumulator** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

Output

Specify the output data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See “Control Signal Data Types” for more information.

Output minimum

Specify the minimum value that the block can output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value that the block can output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer). in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</p>	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code> , which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code> , is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect

when overflow is not possible. In this case, the code generator does not produce saturation code.

State name

Use this parameter to assign a unique name to the block state. The default is ' '. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder).

State name must resolve to Simulink signal object

Select this check box to require that the state name resolve to a Simulink signal object. This check box is cleared by default.

State name enables this parameter. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class**.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see “Apply Custom Storage Classes Directly to Signal Lines, Block States, and Outport Blocks” (Embedded Coder).

To programmatically set this parameter, use `StateSignalObject`.

For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage

classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

StorageClass

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

State name enables this parameter.

To programmatically set this parameter, use `StateStorageClass` or `StateSignalObject`. See “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder).

TypeQualifier

Note **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name resolves
- Minimum and maximum values of the signal object

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder) in the Simulink Coder documentation.

Characteristics

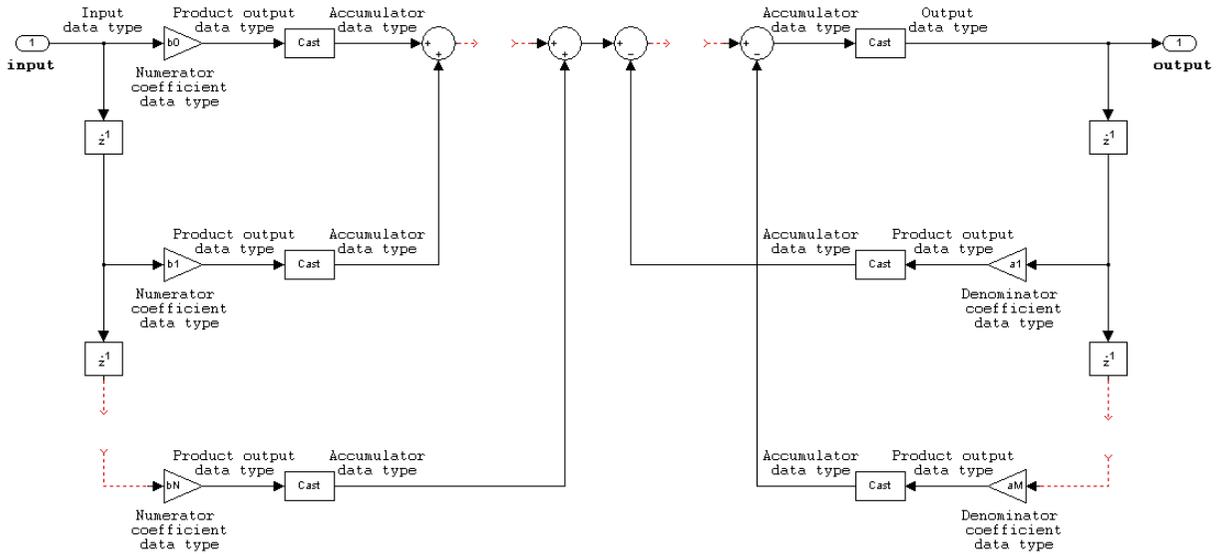
Data Types	Double Single Base Integer Fixed-Point
Sample Time	Specified in the Sample time parameter
Direct Feedthrough	Only when the leading numerator coefficient does not equal zero
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Filter Structure Diagrams

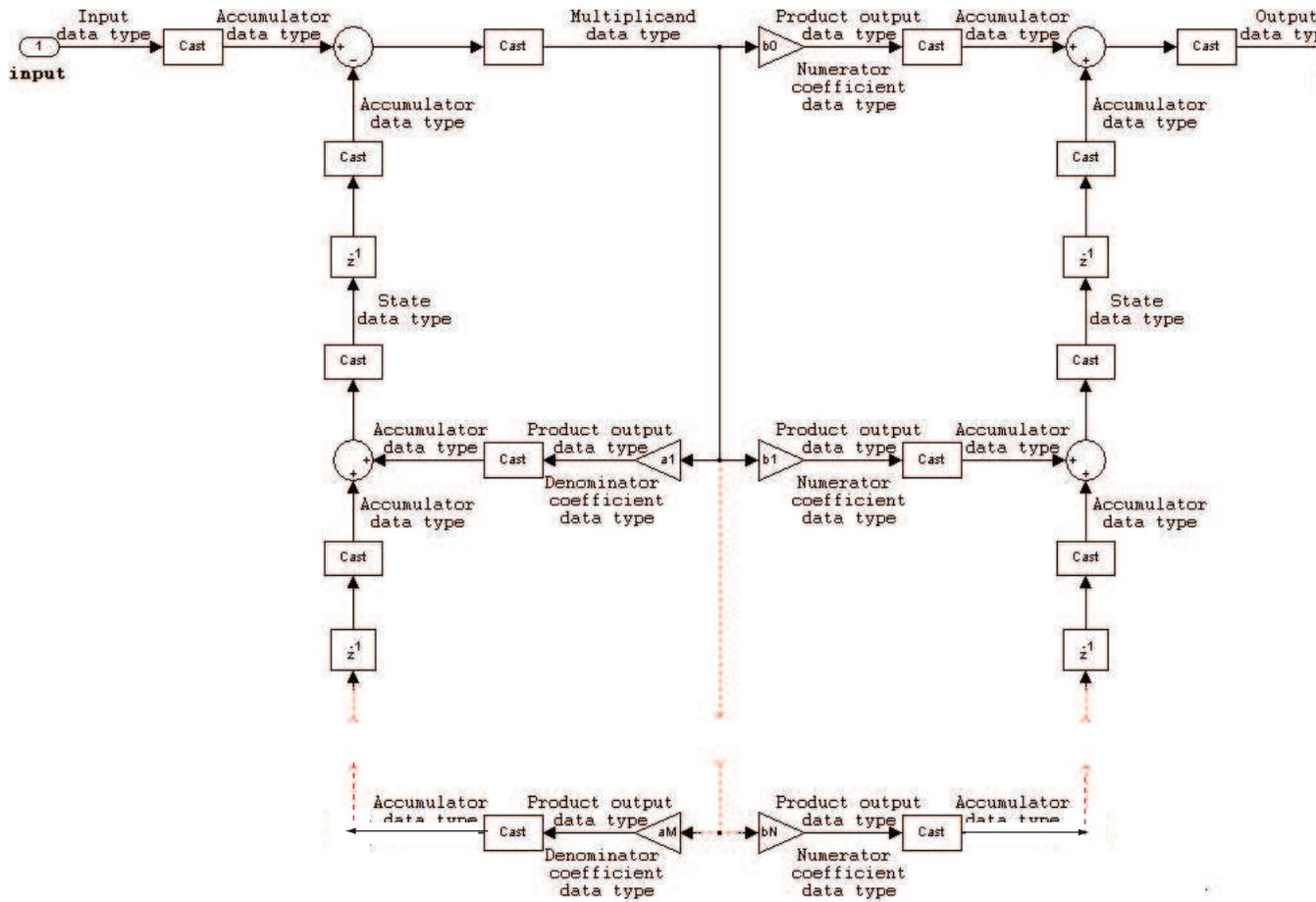
The diagrams in the following sections show the filter structures supported by the Digital Filter block. They also show the data types used in the filter structures for fixed-point signals. You can set the coefficient, output, accumulator, product output, and state data types shown in these diagrams in the block dialog.

- “IIR direct form I” on page 1-379
- “IIR direct form I transposed” on page 1-381
- “IIR direct form II” on page 1-384
- “IIR direct form II transposed” on page 1-386

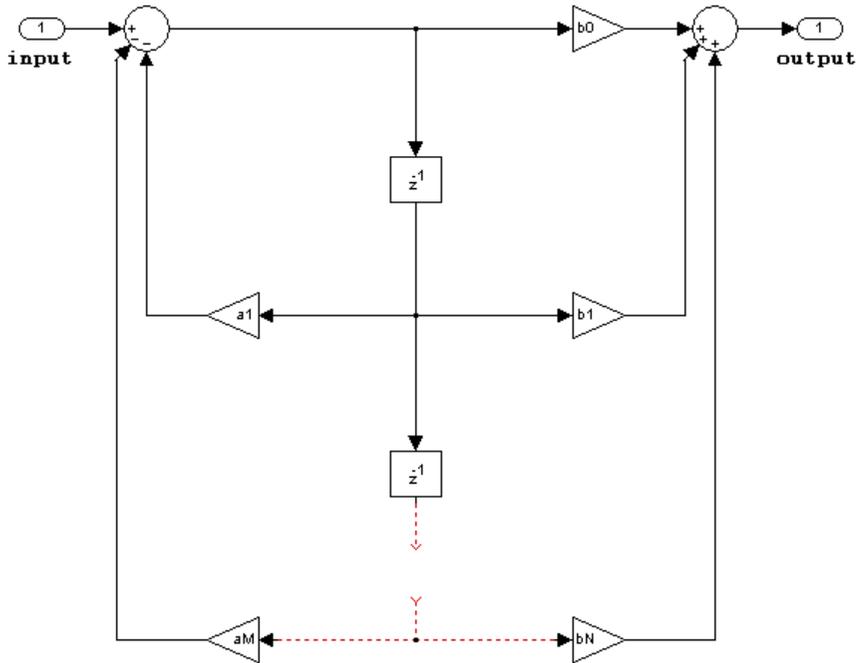
- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.
- The State data type cannot be specified on the block mask for this structure. Doing so is not possible because the input and output states have the same data types as the input and output buffers.



- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must have the same complexity characteristics.
 - When the numerator and denominator coefficients are specified using input ports and have different complexities from each other, you get an error.
 - When the numerator and denominator coefficients are specified in the dialog box and have different complexities from each other, the block does not error. Instead, it processes the filter as if two sets of complex coefficients are provided. The real-valued coefficient set is treated as if it is a complex vector with zero-valued imaginary parts.
- States are complex when either the input or the coefficients are complex.
- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.



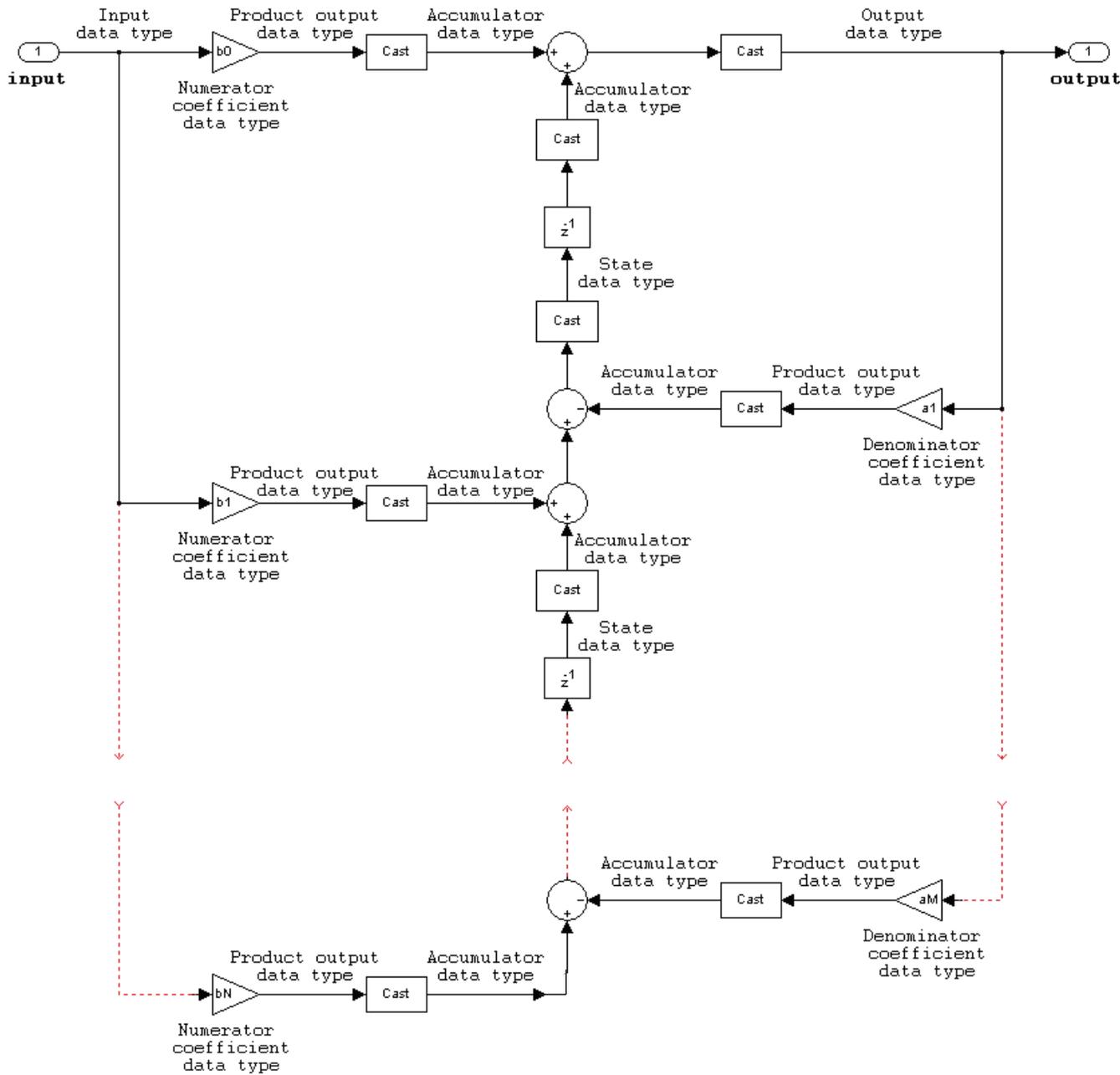
IIR direct form II



The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must have the same complexity characteristics.
 - When the numerator and denominator coefficients are specified using input ports and have different complexities from each other, you get an error.
 - When the numerator and denominator coefficients are specified in the dialog box and have different complexities from each other, the block does not error. Instead, it processes the filter as if two sets of complex coefficients are provided. The real-valued coefficient set is treated as if it is a complex vector with zero-valued imaginary parts.
- States are complex when either the inputs or the coefficients are complex.

- When the numerator and denominator coefficients are specified using input ports and have different complexities from each other, you get an error.
- When the numerator and denominator coefficients are specified in the dialog box and have different complexities from each other, the block does not error. Instead, it processes the filter as if two sets of complex coefficients are provided. The real-valued coefficient set is treated as if it is a complex vector with zero-valued imaginary parts.
- States are complex when either the inputs or the coefficients are complex.
- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.



Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

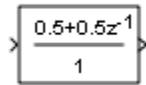
See Also

Allpole Filter	DSP System Toolbox
Digital Filter Design	DSP System Toolbox
Discrete FIR Filter	Simulink
Filter Realization Wizard	DSP System Toolbox
<code>dsp.IIRFilter</code>	DSP System Toolbox
<code>dsp.AllpoleFilter</code>	DSP System Toolbox
<code>filterDesigner</code>	DSP System Toolbox
<code>fvtool</code>	Signal Processing Toolbox

Introduced before R2006a

Discrete FIR Filter

Model FIR filters



Library

Discrete

Description

The Discrete FIR Filter block independently filters each channel of the input signal with the specified digital FIR filter. The block can implement static filters with fixed coefficients, as well as time-varying filters with coefficients that change over time. You can tune the coefficients of a static filter during simulation.

This block filters each channel of the input signal independently over time. The **Input processing** parameter allows you to specify whether the block treats each element of the input as an independent channel (sample-based processing), or each column of the input as an independent channel (frame-based processing). To perform frame-based processing, you must have a DSP System Toolbox license.

The output dimensions equal those of the input, except when you specify a matrix of filter taps for the **Coefficients** parameter. When you do so, the output dimensions depend on the number of different sets of filter taps you specify.

The outputs of this block numerically match the outputs of the DSP System Toolbox Digital Filter Design block.

This block supports the Simulink state logging feature. See “States”.

Filter Structure Support

You can change the filter structure implemented with the Discrete FIR Filter block by selecting one of the following from the **Filter structure** parameter:

- Direct form
- Direct form symmetric
- Direct form antisymmetric
- Direct form transposed
- Lattice MA

You must have an available DSP System Toolbox license to run a model with any of these filter structures other than `Direct form`.

Specifying Initial States

The Discrete FIR Filter block initializes the internal filter states to zero by default, which has the same effect as assuming that past inputs and outputs are zero. You can optionally use the **Initial states** parameter to specify nonzero initial conditions for the filter delays.

To determine the number of initial states you must specify and how to specify them, see the table on valid initial states. The **Initial states** parameter can take one of the forms described in the next table.

Valid Initial States

Initial Condition	Description
Scalar	The block initializes all delay elements in the filter to the scalar value.
Vector or matrix (for applying different delay elements to each channel)	<p>Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel:</p> <ul style="list-style-type: none"> • The vector length equal the product of the number of input channels and the number of delay elements in the filter, <code>#_of_filter_coeffs-1</code> (or <code>#_of_reflection_coeffs</code> for Lattice MA). • The matrix must have the same number of rows as the number of delay elements in the filter, <code>#_of_filter_coeffs-1</code> (<code>#_of_reflection_coeffs</code> for Lattice MA), and must have one column for each channel of the input signal.

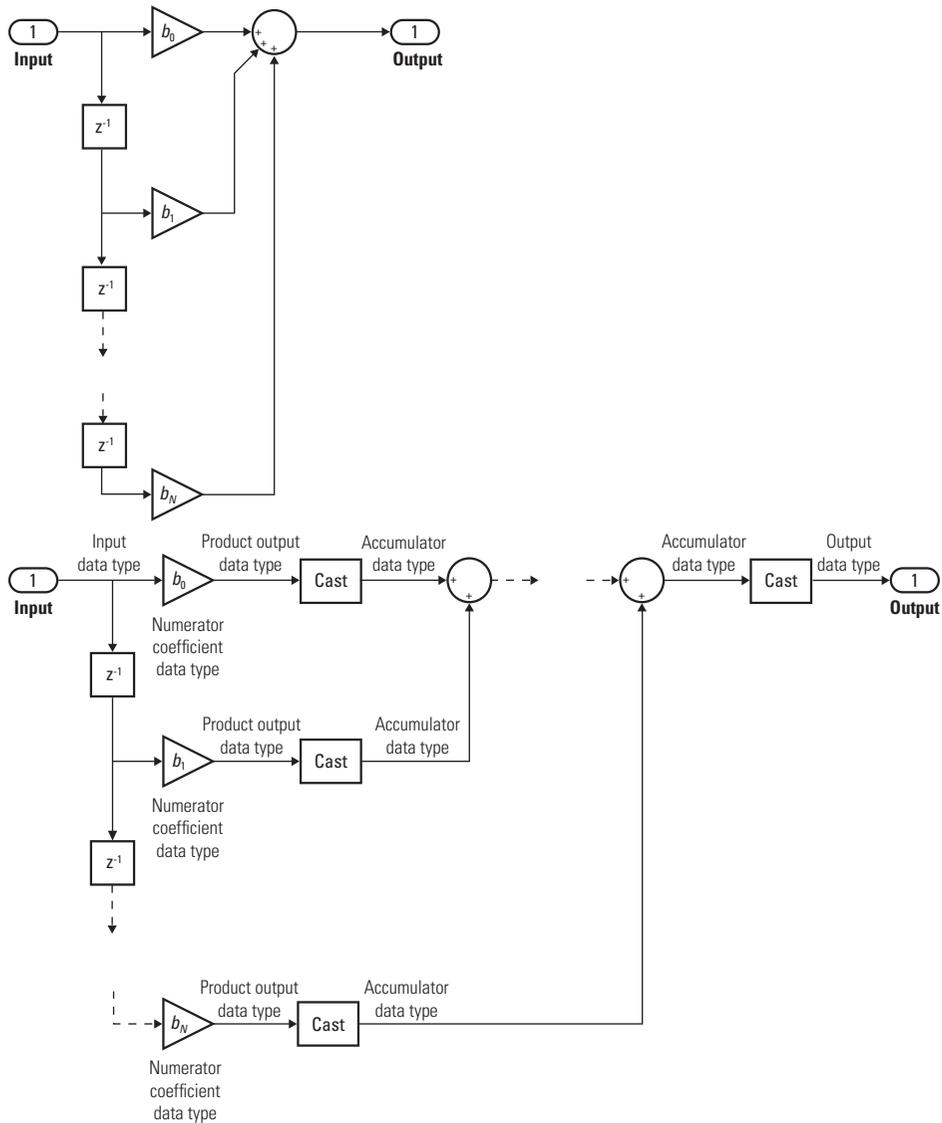
Data Type Support

The Discrete FIR Filter block accepts and outputs real and complex signals of any numeric data type supported by Simulink. The block supports the same types for the coefficients.

The following diagrams show the filter structure and the data types used within the Discrete FIR Filter block for fixed-point signals.

Direct Form

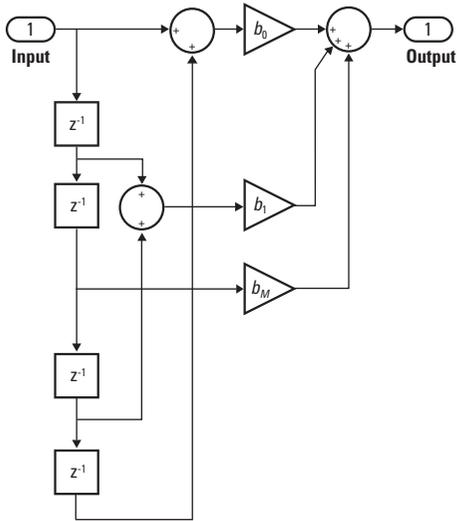
You cannot specify the state data type on the block mask for this structure because the input states have the same data types as the input.



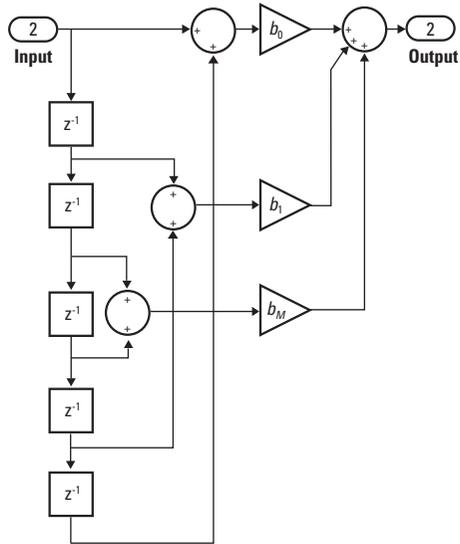
Direct Form Symmetric

You cannot specify the state data type on the block mask for this structure because the input states have the same data types as the input.

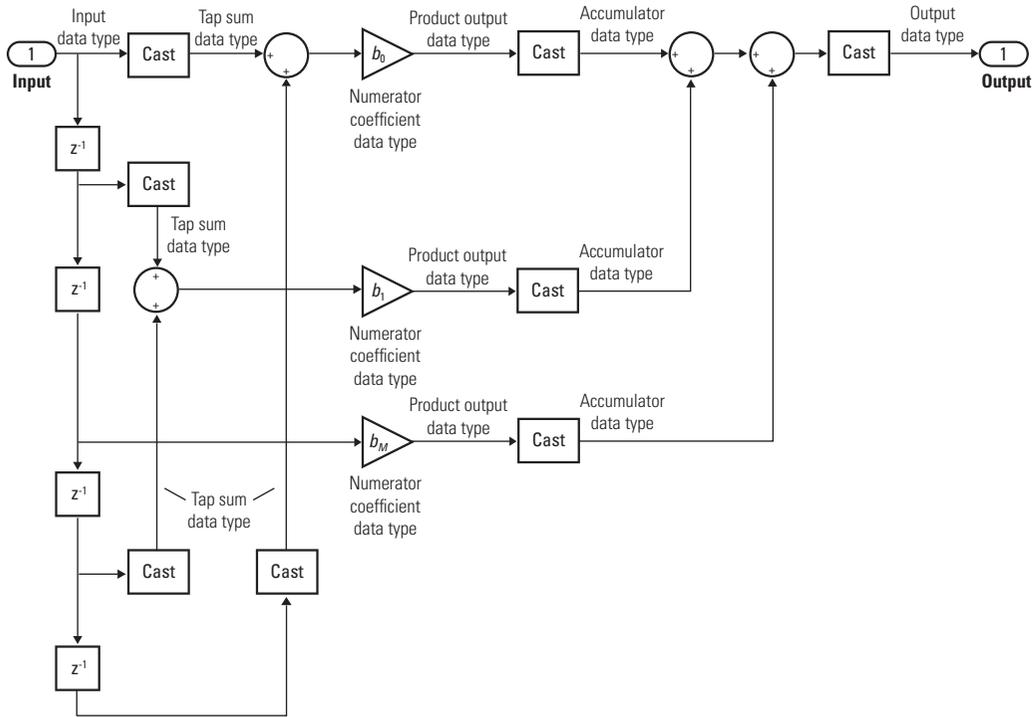
It is assumed that the filter coefficients are symmetric. The block only uses the first half of the coefficients for filtering.



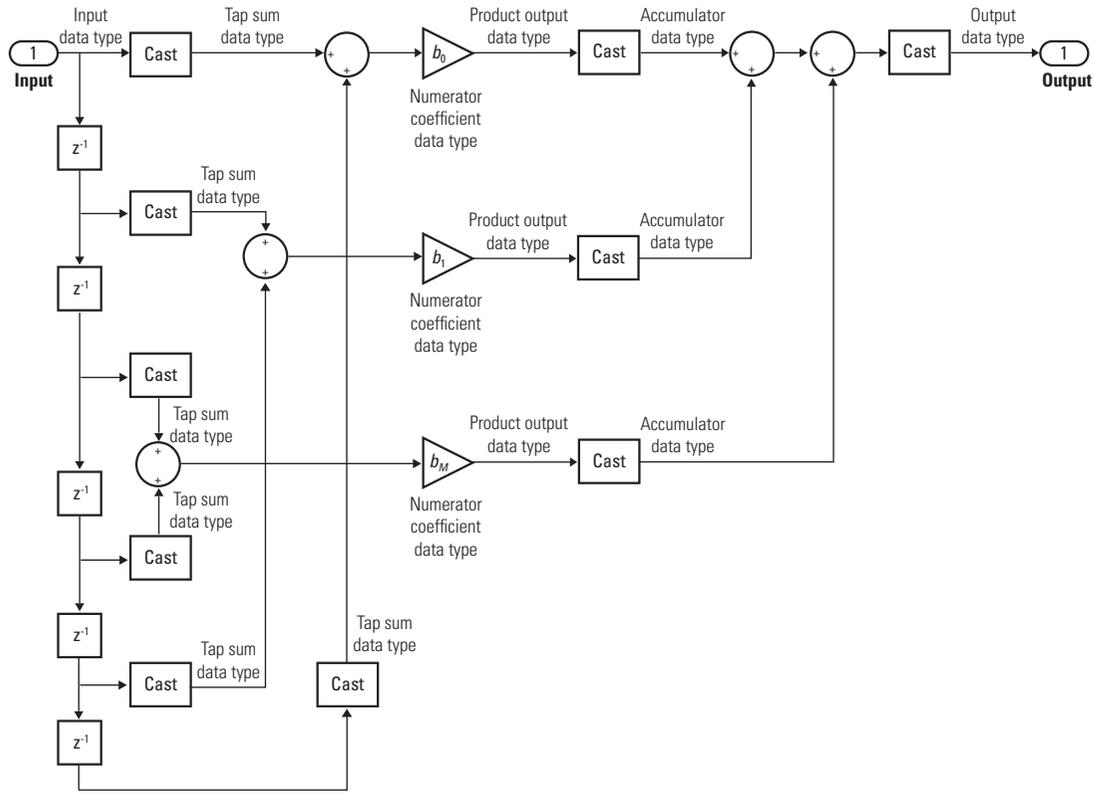
Even Order - Type I



Odd Order - Type II



Even Order - Type I

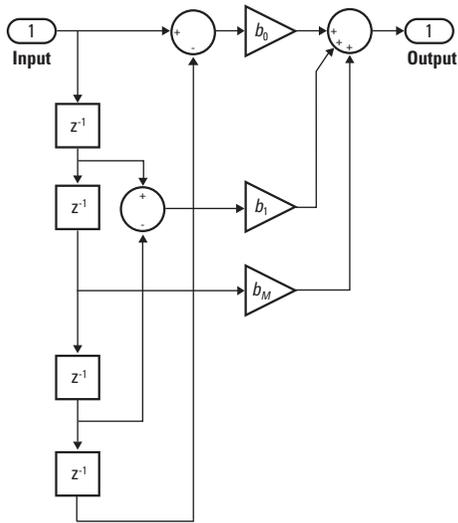


Odd Order - Type II

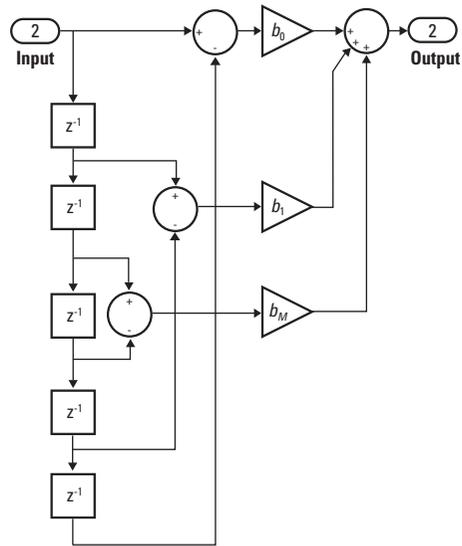
Direct Form Antisymmetric

You cannot specify the state data type on the block mask for this structure because the input states have the same data types as the input.

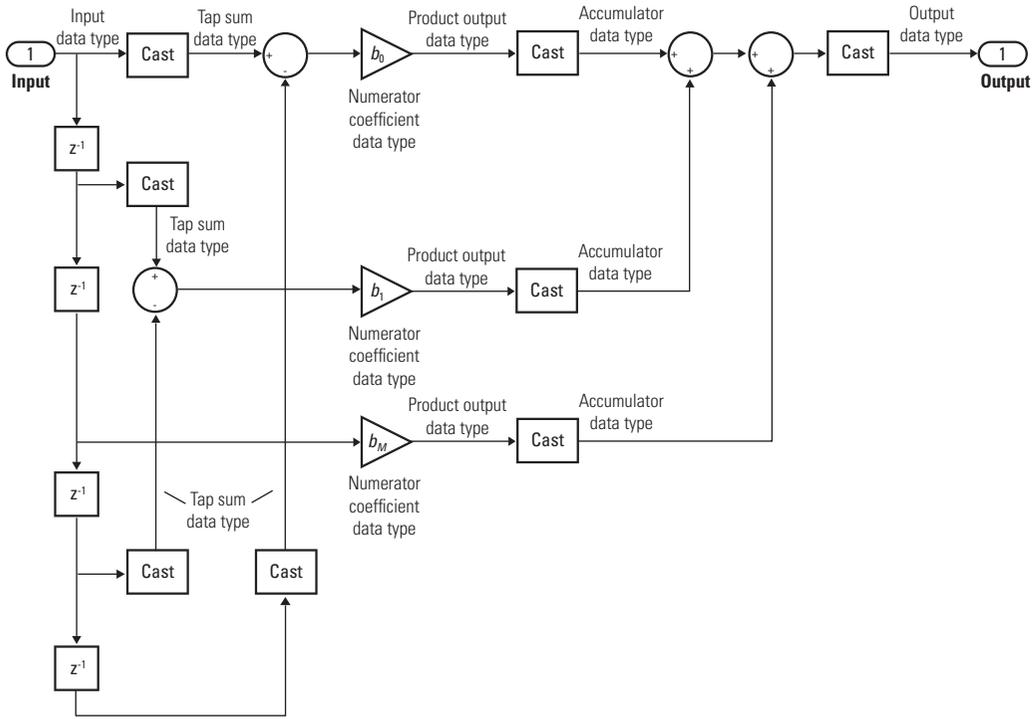
It is assumed that the filter coefficients are antisymmetric. The block only uses the first half of the coefficients for filtering.



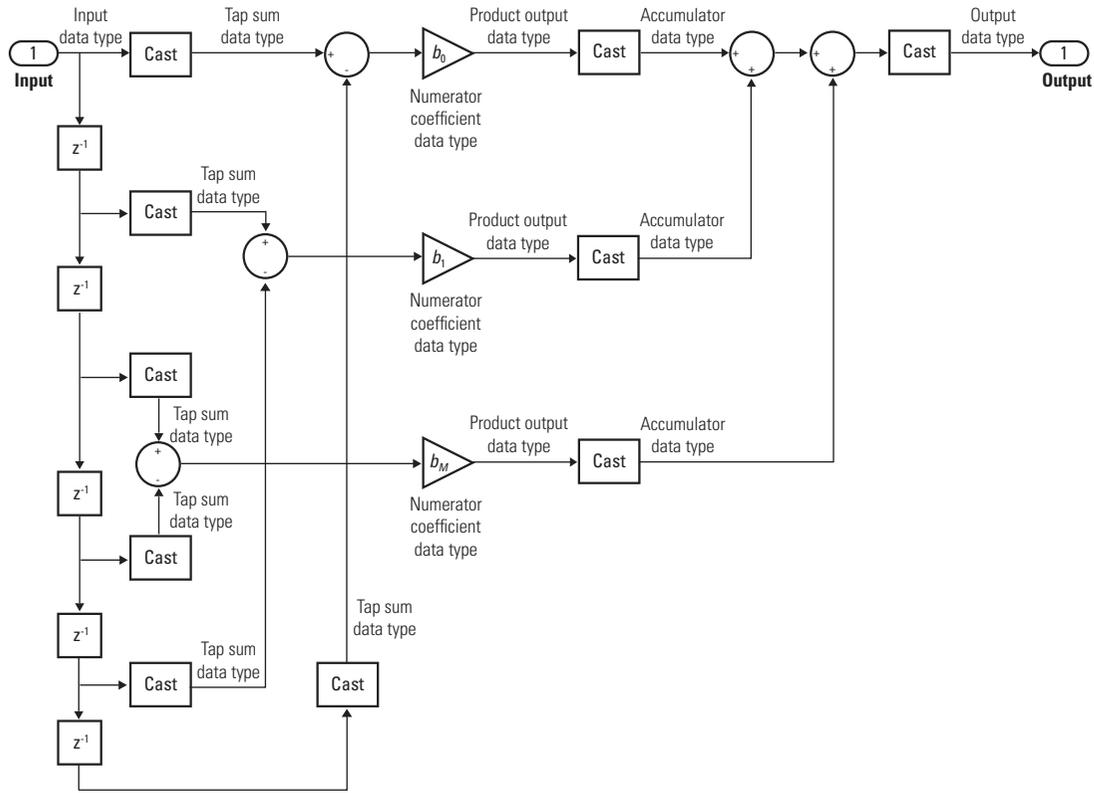
Even Order - Type III



Odd Order - Type IV



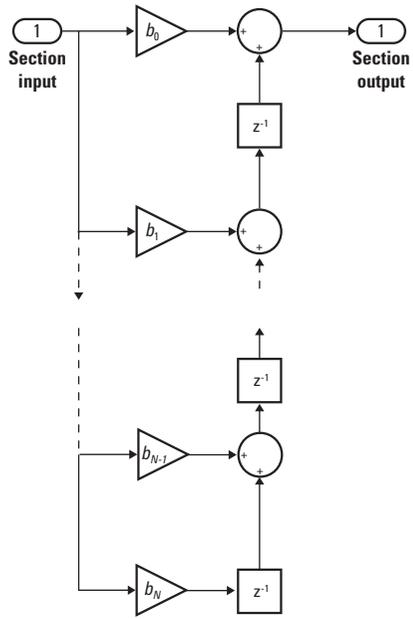
Even Order - Type III

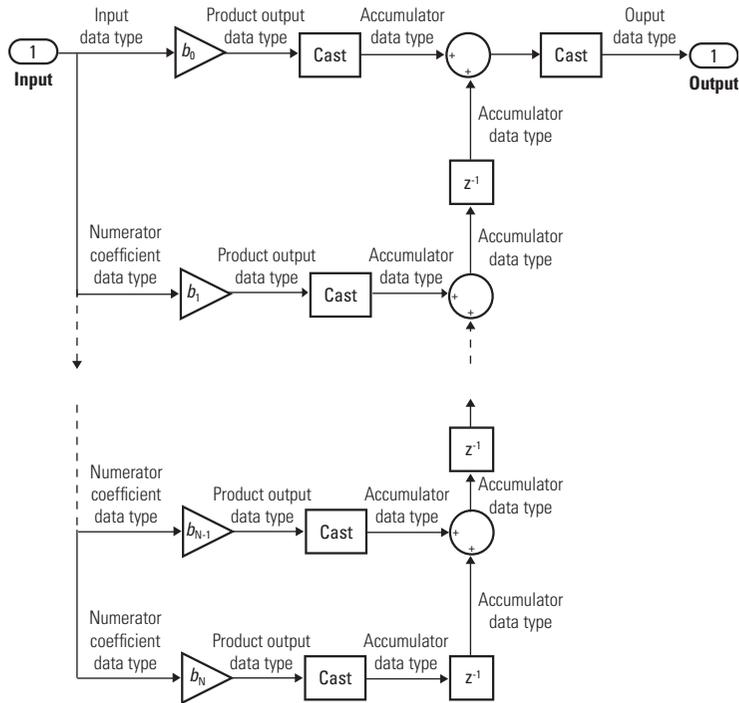


Odd Order - Type IV

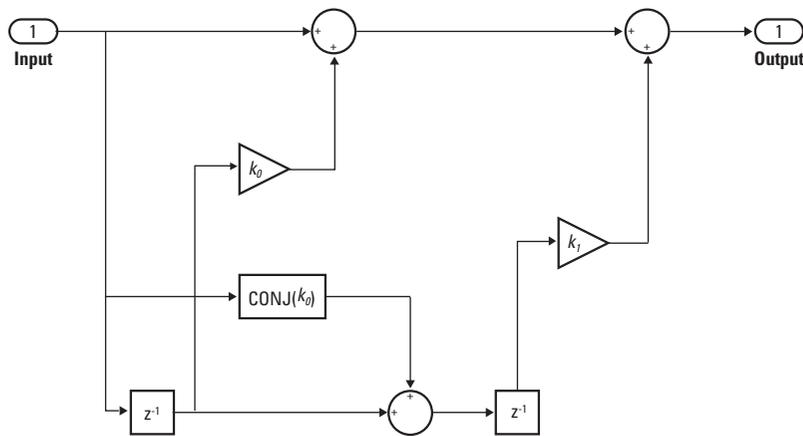
Direct Form Transposed

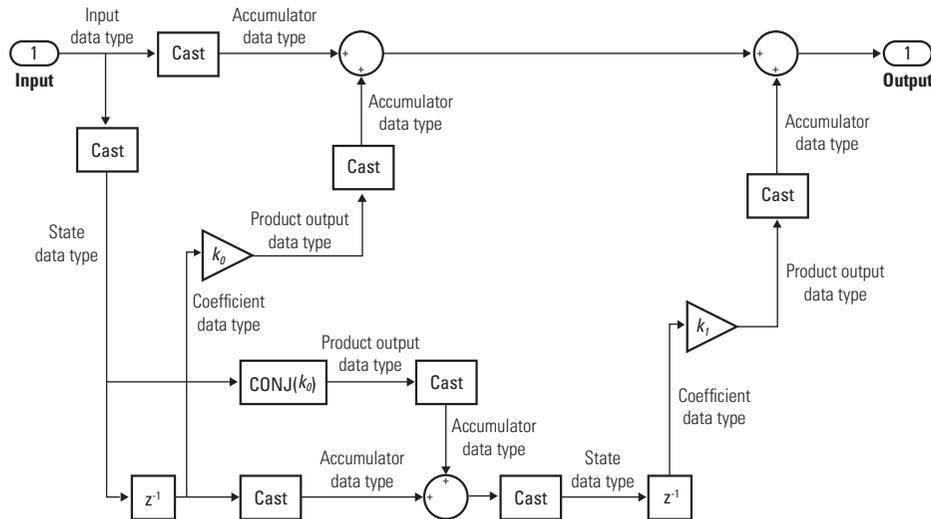
States are complex when either the inputs or the coefficients are complex.





Lattice MA





Parameters

Coefficient source

Select whether you want to specify the filter coefficients on the block mask or through an input port.

Filter structure

Select the filter structure you want the block to implement. You must have an available DSP System Toolbox license to run a model with a Discrete FIR Filter block that implements any filter structure other than direct form.

Coefficients

Specify the vector coefficients of the filter's transfer function. Filter coefficients must be specified as a row vector. When you specify a row vector of filter taps, the block applies a single filter to the input. To apply multiple filters to the same input, specify a matrix of coefficients, where each row represents a different set of filter taps. This parameter is visible only when **Coefficient source** is set to `Dialog parameters`. For multiple filter, **Filter structure** must be `Direct form`, and the input must be a scalar.

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- Elements as channels (sample based) — Treat each element of the input as an independent channel (sample-based processing).
- Columns as channels (frame based) — Treat each column of the input as an independent channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Initial states

Specify the initial conditions of the filter states. To learn how to specify initial states, see “Specifying Initial States” on page 1-391.

Show enable port

Select to show an enable port for this block. This port can control execution of the block. The block is considered enabled when the input to this port is nonzero, and is disabled when the input is 0. The value of the input is checked at the same time step as the block execution.

External reset

Specify the trigger event to use to reset the states to the initial conditions.

Reset Mode	Behavior
None	No reset.
Rising	Reset on a rising edge.
Falling	Reset on a falling edge.
Either	Reset on either a rising or falling edge.
Level	Reset in either of these cases: <ul style="list-style-type: none"> • when the reset signal is nonzero at the current time step • when the reset signal value changes from nonzero at the previous time step to zero at the current time step
Level hold	Reset when the reset signal is nonzero at the current time step

The reset signal must be a scalar of type `single`, `double`, `boolean`, or `integer`. Fixed point data types, except for `ufix1`, are not supported.

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in “How Simulink Works” in the *Simulink User's Guide*.

Tap sum

Specify the tap sum data type of a direct form symmetric or direct form antisymmetric filter, which is the data type the filter uses when it sums the inputs prior to multiplication by the coefficients. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

This parameter is only visible when the selected filter structure is either `Direct form symmetric` or `Direct form antisymmetric`.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Tap sum** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

Coefficients

Specify the coefficient data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Coefficients** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

Coefficients minimum

Specify the minimum value that a filter coefficient should have. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

Coefficients maximum

Specify the maximum value that a filter coefficient should have. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

Product output

Specify the product output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

Accumulator

Specify the accumulator data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`

- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

State

Specify the state data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

This parameter is only visible when the selected filter structure is `Lattice MA`.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **State** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

Output

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See “Control Signal Data Types” in the *Simulink User's Guide* for more information.

Output minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Lock data type settings against changes by the fixed-point tools

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer). in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</p>	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code> , which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code> , is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect

when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Specified in the Sample time parameter
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced in R2008a

Discrete State-Space

Implement discrete state-space system

$$\begin{cases} x_{n+1} = Ax_n + Bu_n \\ y_n = Cx_n + Du_n \end{cases}$$

Library

Discrete

Description

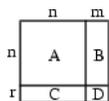
Block Behavior for Non-Empty Matrices

The Discrete State-Space block implements the system described by

$$\begin{aligned} x(n+1) &= Ax(n) + Bu(n) \\ y(n) &= Cx(n) + Du(n), \end{aligned}$$

where u is the input, x is the state, and y is the output. The matrix coefficients must have these characteristics, as illustrated in the following diagram:

- **A** must be an n-by-n matrix, where n is the number of states.
- **B** must be an n-by-m matrix, where m is the number of inputs.
- **C** must be an r-by-n matrix, where r is the number of outputs.
- **D** must be an r-by-m matrix.



The block accepts one input and generates one output. The width of the input vector is the number of columns in the **B** and **D** matrices. The width of the output vector is the

number of rows in the **C** and **D** matrices. To define the initial state vector, use the **Initial conditions** parameter.

To specify a vector or matrix of zeros for **A**, **B**, **C**, **D**, or **Initial conditions**, use the `zeros` function.

Block Behavior for Empty Matrices

When the matrices **A**, **B**, and **C** are empty (for example, `[]`), the functionality of the block becomes $y(n) = Du(n)$. If the **Initial conditions** vector is also empty, the block uses an initial state vector of zeros.

Data Type Support

The Discrete State Space block accepts and outputs a real signal of type `single` or `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

A, B, C, D

Specify the matrix coefficients, as defined in the Description section.

Initial conditions

Specify the initial state vector. The default value is 0. Simulink does not allow the initial states of this block to be `inf` or `NaN`.

Sample time (-1 for inherited)

Specify the time interval between samples. See “Specify Sample Time”.

State name

Use this parameter to assign a unique name to the block state. The default is `' '`. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.

- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder).

State name must resolve to Simulink signal object

Select this check box to require that the state name resolve to a Simulink signal object. This check box is cleared by default.

State name enables this parameter. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class**.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see “Apply Custom Storage Classes Directly to Signal Lines, Block States, and Output Blocks” (Embedded Coder).

To programmatically set this parameter, use `StateSignalObject`.

For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

StorageClass

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

State name enables this parameter.

To programmatically set this parameter, use `StateStorageClass` or `StateSignalObject`. See “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder).

TypeQualifier

Note **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name is resolved
- Min and Max values of the signal object

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder) in the Simulink Coder documentation.

Characteristics

Data Types	Double Single
Sample Time	Specified in the Sample time parameter
Direct Feedthrough	Only if $D \neq 0$
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Discrete-Time Integrator

Perform discrete-time integration or accumulation of signal



Library

Discrete

Description

Capabilities of the Discrete-Time Integrator Block

You can use the Discrete-Time Integrator block in place of the Integrator block to create a purely discrete system. With the Discrete-Time Integrator block, you can:

- Define initial conditions on the block dialog box or as input to the block.
- Define an input gain (K) value.
- Output the block state.
- Define upper and lower limits on the integral.
- Reset the state depending on an additional reset input.

Output Equations

The block starts from the first time step, $n = 0$, with either initial output $y(0) = IC$ or initial state $x(0) = IC$, depending on the **Initial condition setting** parameter value.

For a given step $n > 0$ with simulation time $t(n)$, Simulink updates output $y(n)$ as follows:

- Forward Euler method:

$$y(n) = y(n-1) + K * [t(n) - t(n-1)] * u(n-1)$$

- Backward Euler method:

$$y(n) = y(n-1) + K * [t(n) - t(n-1)] * u(n)$$

- Trapezoidal method:

$$y(n) = y(n-1) + K * [t(n) - t(n-1)] * [u(n) + u(n-1)] / 2$$

Simulink automatically selects a state-space realization of these output equations depending on the block sample time, which can be explicit or triggered. When using explicit sample time, $t(n) - t(n-1)$ reduces to the sample time T for all $n > 0$. For more information on these methods, see “Integration and Accumulation Methods” on page 1-416.

Integration and Accumulation Methods

The block can integrate or accumulate using the forward Euler, backward Euler, and trapezoidal methods. Assume that u is the input, y is the output, and x is the state. For a given step n , Simulink updates $y(n)$ and $x(n+1)$. In integration mode, T is the block sample time (delta T in the case of triggered sample time). In accumulation mode, $T = 1$. The block sample time determines when the output is computed but not the output value. K is the gain value. Values clip according to upper or lower limits.

- Forward Euler method (default), also known as forward rectangular, or left-hand approximation

For this method, the software approximates $1/s$ as $T / (z-1)$. The expressions for the output of the block at step n are:

$$\begin{aligned} x(n+1) &= x(n) + K * T * u(n) \\ y(n) &= x(n) \end{aligned}$$

The block uses the following steps to compute the output:

$$\begin{aligned} \text{Step 0:} \quad & y(0) = IC \text{ (clip if necessary)} \\ & x(1) = y(0) + K * T * u(0) \end{aligned}$$

$$\begin{aligned} \text{Step 1:} \quad & y(1) = x(1) \\ & x(2) = x(1) + K * T * u(1) \end{aligned}$$

$$\begin{aligned} \text{Step } n: \quad & y(n) = x(n) \\ & x(n+1) = x(n) + K \cdot T \cdot u(n) \quad (\text{clip if necessary}) \end{aligned}$$

Using this method, input port 1 does not have direct feedthrough.

- Backward Euler method, also known as backward rectangular or right-hand approximation

For this method, the software approximates $1/s$ as $T \cdot z / (z-1)$. The resulting expression for the output of the block at step n is

$$y(n) = y(n-1) + K \cdot T \cdot u(n).$$

Let $x(n) = y(n-1)$. The block uses these steps to compute the output.

If the parameter **Initial condition setting** is set to Output:

$$\begin{aligned} \text{Step } 0: \quad & y(0) = \text{IC (clipped if necessary)} \\ & x(1) = y(0) \end{aligned}$$

If the parameter **Initial condition setting** is set to State (most efficient):

$$\begin{aligned} \text{Step } 0: \quad & x(0) = \text{IC (clipped if necessary)} \\ & x(1) = y(0) = x(0) + K \cdot T \cdot u(0) \end{aligned}$$

$$\begin{aligned} \text{Step } 1: \quad & y(1) = x(1) + K \cdot T \cdot u(1) \\ & x(2) = y(1) \end{aligned}$$

$$\begin{aligned} \text{Step } n: \quad & y(n) = x(n) + K \cdot T \cdot u(n) \\ & x(n+1) = y(n) \end{aligned}$$

Using this method, input port 1 has direct feedthrough.

- Trapezoidal method

For this method, the software approximates $1/s$ as $T/2 \cdot (z+1) / (z-1)$.

When T is fixed (equal to the sampling period), the expressions to compute the output are:

$$\begin{aligned} x(n) &= y(n-1) + K \cdot T / 2 \cdot u(n-1) \\ y(n) &= x(n) + K \cdot T / 2 \cdot u(n) \end{aligned}$$

If the **Initial condition setting** parameter is set to Output:

Step 0: $y(0) = \text{IC (clipped if necessary)}$
 $x(1) = y(0) + K \cdot T / 2 \cdot u(0)$

If the **Initial condition setting** parameter is set to State (most efficient):

Step 0: $x(0) = \text{IC (clipped if necessary)}$
 $y(0) = x(0) + K \cdot T / 2 \cdot u(0)$
 $x(1) = y(0) + K \cdot T / 2 \cdot u(0)$

Step 1: $y(1) = x(1) + K \cdot T / 2 \cdot u(1)$
 $x(2) = y(1) + K \cdot T / 2 \cdot u(1)$

Step n: $y(n) = x(n) + K \cdot T / 2 \cdot u(n)$
 $x(n+1) = y(n) + K \cdot T / 2 \cdot u(n)$

Here, $x(n+1)$ is the best estimate of the next output. It is not the same as the state, in that $x(n)$ is not equal to $y(n)$.

If T is variable (for example, obtained from the triggering times), the block uses these steps to compute the output.

If the **Initial condition setting** parameter is set to Output:

Step 0: $y(0) = \text{IC (clipped if necessary)}$
 $x(1) = y(0)$

If the **Initial condition setting** parameter is set to State (most efficient):

Step 0: $x(0) = \text{IC (clipped if necessary)}$
 $x(1) = y(0) = x(0) + K \cdot T / 2 \cdot u(0)$

Step 1: $y(1) = x(1) + T / 2 \cdot (u(1) + u(0))$
 $x(2) = y(1)$

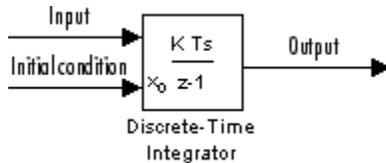
Step n: $y(n) = x(n) + T / 2 \cdot (u(n) + u(n-1))$
 $x(n+1) = y(n)$

Using this method, input port 1 has direct feedthrough.

Define Initial Conditions

You can define the initial conditions as a parameter on the block dialog box or input them from an external signal:

- To define the initial conditions as a block parameter, set the **Initial condition source** parameter to `internal` and enter the value in the **Initial condition** text box.
- To provide the initial conditions from an external source, set the **Initial condition source** parameter to `external`. An additional input port appears on the block.

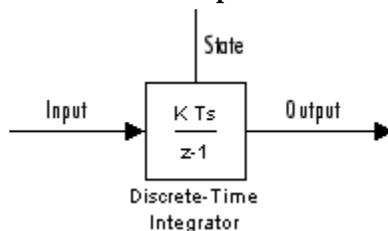


When to Use the State Port

Use the state port instead of the output port:

- When the output of the block is fed back into the block through the reset port or the initial condition port, causing an algebraic loop. For an example, see the `sldemo_bounce_two_integrators` model.
- When you want to pass the state from one conditionally executed subsystem to another, which can cause timing problems. For an example, see the `sldemo_clutch` model.

You can work around these problems by passing the state through the state port rather than the output port. Simulink generates the state at a slightly different time from the output, which protects your model from these problems. To output the block state, select the **Show state port** check box. The state port appears on the top of the block



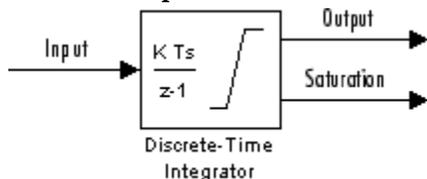
Limit the Integral

To keep the output within certain levels, select the **Limit output** check box and enter the limits in the corresponding text box. Doing so causes the block to function as a

limited integrator. When the output reaches the limits, the integral action turns off to prevent integral windup. During a simulation, you can change the limits but you cannot change whether the output is limited. The table shows how the block determines output.

Integral	Output
Less than or equal to the Lower saturation limit and the input is negative	Held at the Lower saturation limit
Between the Lower saturation limit and the Upper saturation limit	The integral
Greater than or equal to the Upper saturation limit and the input is positive	Held at the Upper saturation limit

To generate a signal that indicates when the state is being limited, select the **Show saturation port** check box. A new saturation port appears below the block output port:

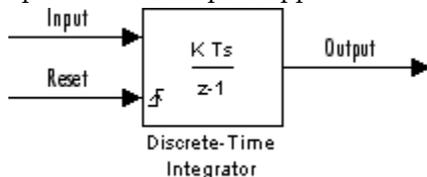


The signal has one of three values:

- 1 indicates that the upper limit is being applied.
- 0 indicates that the integral is not limited.
- -1 indicates that the lower limit is being applied.

Reset the State

The block can reset its state to the initial condition you specify, based on an external signal. To cause the block to reset its state, select one of the **External reset** parameter options. A reset port appears that indicates the reset trigger type:



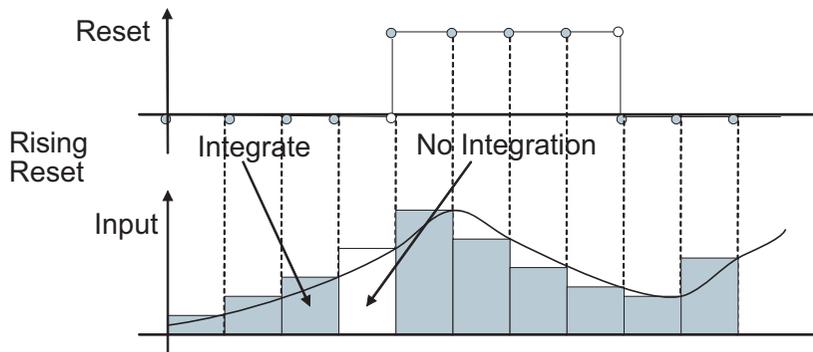
The reset port has direct feedthrough. If the block output feeds back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results.

To resolve this loop, feed the output of the block state port into the reset port instead. To access the block state, select the **Show state port** check box.

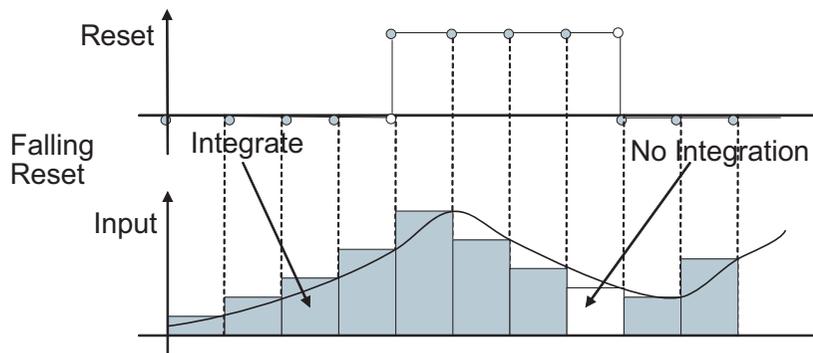
Reset Trigger Types

The **External reset** parameter lets you determine the attribute of the reset signal that triggers the reset. The trigger options include:

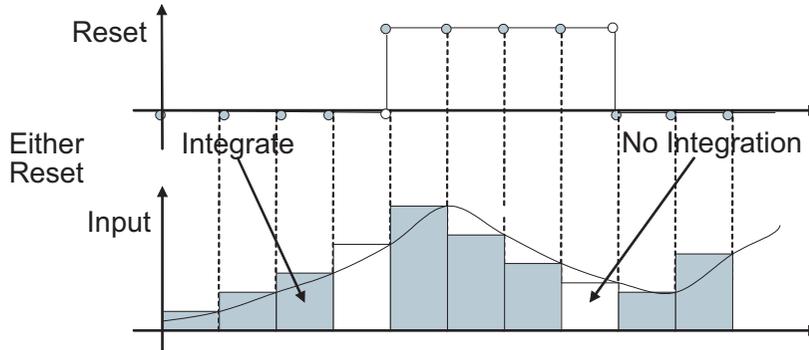
- `rising` – Resets the state when the reset signal has a rising edge. For example, this figure shows the effect that a rising reset trigger has on backward Euler integration.



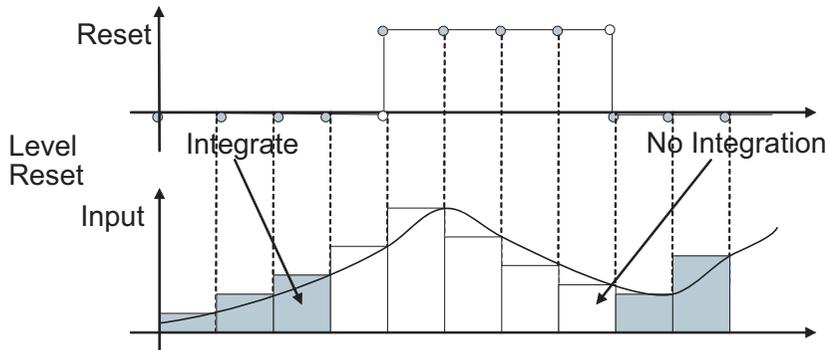
- `falling` – Resets the state when the reset signal has a falling edge. For example, this figure shows the effect that a falling reset trigger has on backward Euler integration.



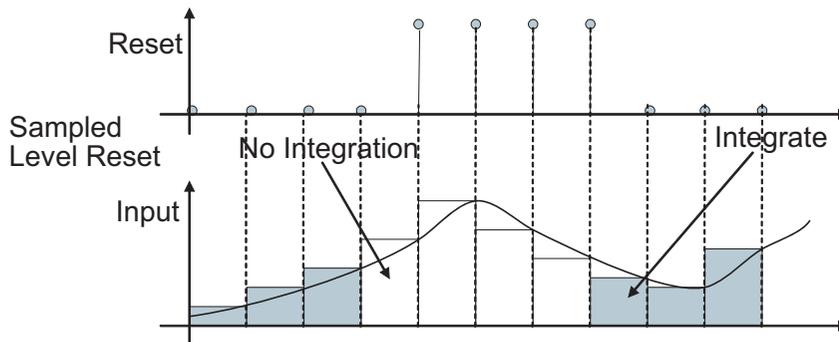
- `either` – Resets the state when the reset signal rises or falls. For example, the following figure shows the effect that an either reset trigger has on backward Euler integration.



- `level` – Resets and holds the output to the initial condition while the reset signal is nonzero. For example, this figure shows the effect that a level reset trigger has on backward Euler integration.



- `sampled level` – Resets the output to the initial condition when the reset signal is nonzero. For example, this figure shows the effect that a sampled level reset trigger has on backward Euler integration.



The sampled level reset option requires fewer computations, making it more efficient than the level reset option. However, the sampled level reset option can introduce a discontinuity when integration resumes.

Note For the discrete-time integrator block, all trigger detections are based on signals with positive values. For example, a signal changing from -1 to 0 is not considered a rising edge, but a signal changing from 0 to 1 is.

Behavior in Simplified Initialization Mode

Simplified initialization mode is enabled when you set **Underspecified initialization detection** to **Simplified** on the **Configuration Parameters** dialog box. If you use simplified initialization mode, the behavior of the Discrete-Time Integrator block differs from classic initialization mode. The new initialization behavior is more robust and provides more consistent behavior in these cases:

- In algebraic loops
- On enable and disable
- When comparing results using triggered sample time against explicit sample time, where the block is triggered at the same rate as the explicit sample time

Simplified initialization mode enables easier conversion from Continuous-Time Integrator blocks to Discrete-Time Integrator blocks, because the initial conditions have the same meaning for both blocks.

For more information on classic and simplified initialization modes, see “Underspecified initialization detection”.

When you use simplified initialization mode with **Initial condition setting** set to Output, the enable and disable behavior of the block is simplified as follows:

At disable time t_d :

$$y(t_d) = y(t_d-1)$$

At enable time t_e :

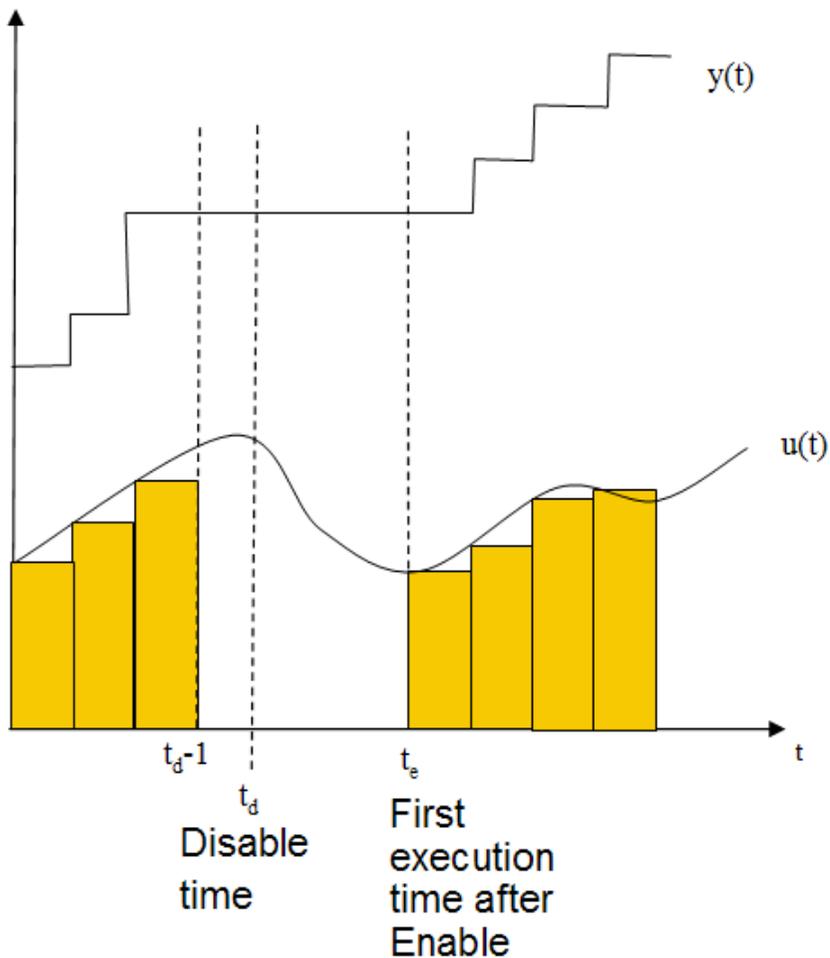
- If the parent subsystem control port has **States when enabling** set to reset:

$$y(t_e) = IC.$$

- If the parent subsystem control port has **States when enabling** set to held:

$$y(t_e) = y(t_d).$$

The following figure shows this condition.



When using simplified initialization mode, you cannot place the Discrete-Time Integrator block in an Iterator Subsystem.

In simplified initialization mode, Iterator subsystems do not maintain elapsed time. Thus, if a Discrete-Time Integrator block, which needs elapsed time, is placed inside an Iterator Subsystem block, Simulink reports an error.

Simulink does not support model simulation when all the following conditions are true:

- A Discrete-Time Integrator block is placed within a triggered subsystem or a function-call subsystem.
- The block's **Initial condition setting** parameter is set to `State` (most efficient).
- Simplified initialization mode is enabled.

Behavior in an Enabled Subsystem Inside a Function-Call Subsystem

Suppose that you have a function-call subsystem that contains an enabled subsystem, which contains a Discrete-Time Integrator block. The following behavior applies.

Integrator Method	Sample Time Type of Function-Call Trigger Port	Value of <code>delta T</code> When Function-Call Subsystem Executes for the First Time After Enabled	Reason for Behavior
Forward Euler	Triggered	$t - t_{\text{start}}$	When the function-call subsystem executes for the first time, the integrator algorithm uses t_{start} as the previous simulation time.
Backward Euler and Trapezoidal	Triggered	$t - t_{\text{previous}}$	When the function-call subsystem executes for the first time, the integrator algorithm uses t_{previous} as the previous simulation time.
Forward Euler, Backward Euler, and Trapezoidal	Periodic	Sample time of the function-call generator	In periodic mode, the Discrete-Time Integrator block uses sample time of the function-call generator for <code>delta T</code> .

Data Type Support

The Discrete-Time Integrator block accepts real signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name is resolved
- Min and Max values of the signal object

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder) in the Simulink Coder documentation.

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Integrator method

Specify the integration or accumulation method.

Default: Integration: Forward Euler

Integration: Forward Euler

Integrator method is Forward Euler.

Integration: Backward Euler

Integrator method is Backward Euler.

Integration: Trapezoidal

Integrator method is Trapezoidal.

Accumulation: Forward Euler

Accumulation method is Forward Euler.

Accumulation: Backward Euler

Accumulation method is Backward Euler.

Accumulation: Trapezoidal

Accumulation method is Trapezoidal.

Parameter: IntegratorMethod

Type: character vector

Value: 'Integration: Forward Euler' | 'Integration: Backward Euler' |
'Integration: Trapezoidal' | 'Accumulation: Forward Euler' |
'Accumulation: Backward Euler' | 'Accumulation: Trapezoidal'

Default: 'Integration: Forward Euler'

Gain value

Specify a scalar, vector, or matrix by which to multiply the integrator input. Each element of the gain must be a positive real number.

Default: 1.0

- Specifying a value other than 1.0 (the default) is semantically equivalent to connecting a Gain block to the input of the integrator.
- Valid entries include:
 - `double(1.0)`
 - `single(1.0)`
 - `[1.1 2.2 3.3 4.4]`
 - `[1.1 2.2; 3.3 4.4]`
- Using this parameter to specify the input gain eliminates a multiplication operation in the generated code. However, this parameter must be nontunable to realize this

benefit. If you want to tune the input gain, set this parameter to 1.0 and use an external Gain block to specify the input gain.

Parameter: gainval

Type: character vector

Value: '1.0'

Default: '1.0'

External reset

Reset the states to their initial conditions when a trigger event occurs in the reset signal.

Default: none

none

Do not reset the state to initial conditions.

rising

Reset the state when the reset signal has a rising edge.

falling

Reset the state when the reset signal has a falling edge.

either

Reset the state when the reset signal rises or falls.

level

Reset and holds the output to the initial condition while the reset signal is nonzero.

sampled level

Reset the output to the initial condition when the reset signal is nonzero.

Parameter: ExternalReset

Type: character vector

Value: 'none' | 'rising' | 'falling' | 'either' | 'level' | 'sampled level'

Default: 'none'

Initial condition source

Get the initial conditions of the states.

Default: `internal`

`internal`

Get the initial conditions of the states from the **Initial condition** parameter.

`external`

Get the initial conditions of the states from an external block.

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Selecting `internal` enables the **Initial condition** parameter.

Selecting `external` disables the **Initial condition** parameter.

Parameter: `InitialConditionSource`

Type: character vector

Value: `'internal' | 'external'`

Default: `'internal'`

Initial condition

Specify the states' initial conditions.

Default: 0

Minimum: value of **Output minimum** parameter

Maximum: value of **Output maximum** parameter

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Setting **Initial condition source** to `internal` enables this parameter.

Setting **Initial condition source** to `external` disables this parameter.

Parameter: InitialCondition

Type: scalar or vector

Value: '0'

Default: '0'

Initial condition setting

Specify whether to apply the **Initial condition** parameter to the block state or output. This initial condition is also used as the reset value. This parameter was named **Use initial condition as initial and reset value for** in Simulink before R2014a.

Default: State (most efficient)

State (most efficient)

Use this option in all situations except when the block is in a triggered subsystem or a function-call subsystem and **Integrator method** is set to an integration method.

Set the following initial conditions:

$$x(0) = IC$$

At reset:

$$x(n) = IC$$

Output

Use this option when the block is in a triggered subsystem or a function-call subsystem and **Integrator method** is set to an integration method.

Set the following initial conditions:

$$y(0) = IC$$

At reset:

$$y(n) = IC$$

Compatibility

This option is present to provide backward compatibility. You cannot select this option for Discrete-Time Integrator blocks in Simulink models but you can select it

for Discrete-Time Integrator blocks in a library. Use this option to maintain compatibility with Simulink models created before R2014a.

Prior to R2014a, the option `State` (most efficient) was known as `State only` (most efficient). The option `Output` was known as `State and output`. The behavior of the block with the option `Compatibility` is as follows.

- If **Underspecified initialization detection** is set to `Classic`, the **Initial condition setting** parameter behaves as `State (most efficient)`.
- If **Underspecified initialization detection** is set to `Simplified`, the **Initial condition setting** parameter behaves as `Output`.

Parameter: `InitialConditionSetting`

Type: character vector

Value: `'State (most efficient)'` | `'Output'` | `'Compatibility'`

Default: `'Output'`

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits.

Default: 1

By default, the block uses a discrete sample time of 1. To set a different sample time, enter another discrete value, such as 0.1.

See also “Specify Sample Time” in the online documentation for more information.

- Do not specify a sample time of 0. This value specifies a continuous sample time, which the Discrete-Time Integrator block does not support.
- Do not specify a sample time of `inf` or `NaN` because these values are not discrete.
- If you specify -1 to inherit the sample time from an upstream block, verify that the upstream block uses a discrete sample time. For example, the Discrete-Time Integrator block cannot inherit a sample time of 0.

Parameter: `SampleTime`

Type: character vector

Value: '1'
Default: '1'

Limit output

Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

Default: Off

On

Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

Off

Do not limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

This parameter enables **Upper saturation limit**.

This parameter enables **Lower saturation limit**.

Parameter: LimitOutput
Type: character vector
Value: 'off' | 'on'
Default: 'off'

Upper saturation limit

Specify the upper limit for the integral.

Default: `inf`

Minimum: value of **Output minimum** parameter

Maximum: value of **Output maximum** parameter

Limit output enables this parameter.

Parameter: UpperSaturationLimit

Type: scalar or vector

Value: 'inf'

Default: 'inf'

Lower saturation limit

Specify the lower limit for the integral.

Default: -inf

Minimum: value of **Output minimum** parameter

Maximum: value of **Output maximum** parameter

Limit output enables this parameter.

Parameter: LowerSaturationLimit

Type: scalar or vector

Value: '-inf'

Default: '-inf'

Show saturation port

Add a saturation output port to the block.

Default: Off

On

Add a saturation output port to the block.

Off

Do not add a saturation output port to the block.

Parameter: ShowSaturationPort

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Show state port

Add an output port to the block for the block's state.

Default: Off

On

Add an output port to the block for the block's state.

Off

Do not add an output port to the block for the block's state.

Parameter: ShowStatePort

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Ignore limit and reset when linearizing

Cause Simulink linearization commands to treat this block as not resettable and as having no limits on its output, regardless of the settings of the block reset and output limitation options.

Default: Off

On

Cause Simulink linearization commands to treat this block as not resettable and as having no limits on its output, regardless of the settings of the block reset and output limitation options.

Off

Do not cause Simulink linearization commands to treat this block as not resettable and as having no limits on its output, regardless of the settings of the block reset and output limitation options.

Ignoring the limit and resetting allows you to linearize a model around an operating point. This point may cause the integrator to reset or saturate.

Parameter: IgnoreLimit

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: LockScale

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: `Floor`

`Ceiling`

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

`Zero`

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Parameter: `RndMeth`

Type: character vector

Value: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`

Default: `'Floor'`

For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off

On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Parameter: SaturateOnIntegerOverflow

Type: character vector

Value: 'off' | 'on'

Default: 'off'

State name

Use this parameter to assign a unique name to each state.

Default: ' '

- If left blank, no name is assigned.
- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click the **Apply** button.

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder).

Parameter: StateName

Type: character vector

Value: ' '

Default: ' '

State name must resolve to Simulink signal object

Require that state name resolve to Simulink signal object.

Default: Off

On

Require that state name resolve to Simulink signal object.

Off

Do not require that state name resolve to Simulink signal object.

State name enables this parameter. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class**.

Parameter: `StateMustResolveToSignalObject`

Type: character vector

Value: `'off' | 'on'`

Default: `'off'`

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

To programmatically set this parameter, use `StateSignalObject`.

For examples and more information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

`Simulink.Signal`

Use custom storage classes from the built-in package `Simulink`.

Package.Class

Use custom storage classes from the package that defines the class that you select.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see “Apply Custom Storage Classes Directly to Signal Lines, Block States, and Output Blocks” (Embedded Coder).

Code generation storage class

Select state storage class for code generation.

Default: Auto

Auto

Auto is the appropriate storage class for states that you do not need to interface to external code.

StorageClass

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than Simulink.

State name enables this parameter.

Command-Line Information

Parameter: StateStorageClass

Type: character vector

Value: 'Auto' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'SimulinkGlobal' | 'Custom'

Default: 'Auto'

Code generation storage class

Select state storage class for code generation.

Default: Auto

Auto

Auto is the appropriate storage class for states that you do not need to interface to external code.

StorageClass

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Control Signals and States in Code by

Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than Simulink.

State name enables this parameter.

TypeQualifier

Note **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

- **Default:** ' ' (empty character vector)
- `const`
- `volatile`

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

Parameter Name: `RTWStateStorageTypeQualifier`

Value Type: character vector

Default: ' ' (empty character vector)

Output minimum

Lower value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMin

Type: character vector

Value: ' [] '

Default: ' [] '

Output maximum

Upper value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).

- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMax

Type: character vector

Value: ' [] '

Default: ' [] '

Output data type

Specify the output data type.

Default: `Inherit: Inherit via internal rule`

`Inherit: Inherit via internal rule`

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

Inherit: Inherit via back propagation

Use data type of the driving block.

double

Output data type is double.

single

Output data type is single.

int8

Output data type is int8.

uint8

Output data type is uint8.

int16

Output data type is int16.

uint16

Output data type is uint16.

int32

Output data type is int32.

uint32

Output data type is uint32.

fixdt(1,16,0)

Output data type is fixed point fixdt(1,16,0).

fixdt(1,16,2^0,0)

Output data type is fixed point fixdt(1,16,2^0,0).

<data type expression>

Use a data type object, for example, Simulink.NumericType.

Parameter: OutDataTypeStr

Type: character vector

Value: 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)'

Default: 'Inherit: Inherit via internal rule'

For more information, see “Control Signal Data Types”.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule (default)`
- `Inherit via back propagation`

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Fixed-point data types.

`Expression`

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

Parameter: OutDataTypeStr

Type: character vector

Value: 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)'

Default: 'Inherit: Inherit via internal rule'

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: Inherit

Inherit

Inherits the data type override setting from its context, that is, from the block, Simulink.Signal object or Stateflow chart in Simulink that is using the signal.

Off

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is Built in or Fixed point.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: Signed

Signed

Specify the fixed-point data as signed.

Unsigned

Specify the fixed-point data as unsigned.

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting Slope and bias enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > Binary point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

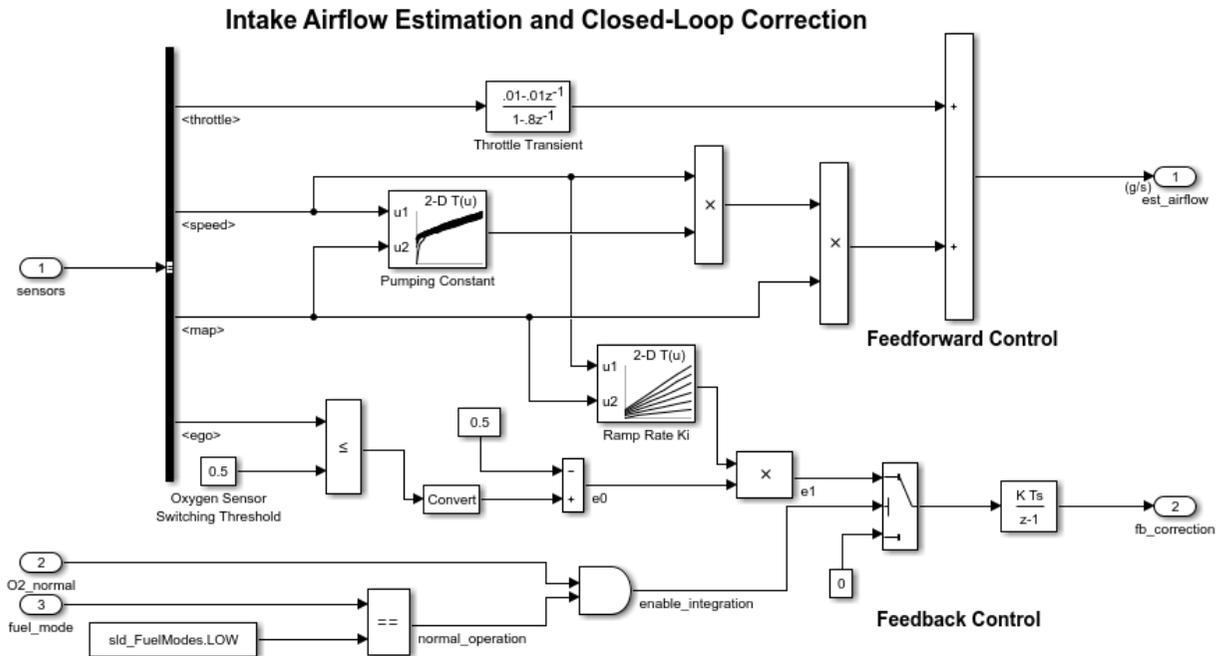
Specify any real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Examples

The `sldemo_fuelsys` model uses a Discrete-Time Integrator block in the `fuel_rate_control/airflow_calc` subsystem. This block uses the Forward Euler integration method.



When the Switch block feeds a nonzero value into the Discrete-Time Integrator block, integration occurs. Otherwise, integration does not occur.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Specified in the Sample time parameter
Direct Feedthrough	Yes, of the reset and external initial condition source ports. The input has direct feedthrough for every integration method except Forward Euler and accumulation Forward Euler.
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

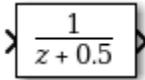
See Also

Integrator

Introduced before R2006a

Discrete Transfer Fcn

Implement discrete transfer function



Library

Discrete

Description

The Discrete Transfer Fcn block implements the z -transform transfer function:

$$H(z) = \frac{num(z)}{den(z)} = \frac{num_0 z^m + num_1 z^{m-1} + \dots + num_m}{den_0 z^n + den_1 z^{n-1} + \dots + den_n}$$

where $m+1$ and $n+1$ are the number of numerator and denominator coefficients, respectively. num and den contain the coefficients of the numerator and denominator in descending powers of z . num can be a vector or matrix, den must be a vector, and you specify both as parameters on the block dialog box. The order of the denominator must be greater than or equal to the order of the numerator.

Specify the coefficients of the numerator and denominator polynomials in descending powers of z . This block lets you use polynomials in z to represent a discrete system, a method that control engineers typically use. Conversely, the Discrete Filter block lets you use polynomials in z^{-1} (the delay operator) to represent a discrete system, a method that signal processing engineers typically use. The two methods are identical when the numerator and denominator polynomials have the same length.

The Discrete Transfer Fcn block applies the z -transform transfer function to each independent channel of the input. The **Input processing** parameter allows you to specify whether the block treats each element of the input as an individual channel (sample-based processing), or each column of the input as an individual channel (frame-based processing). To perform frame-based processing, you must have a DSP System Toolbox license.

Specifying Initial States

Use the **Initial states** parameter to specify initial filter states. To determine the number of initial states you must specify and how to specify them, see the following tables.

Frame-Based Processing

Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
<ul style="list-style-type: none"> • Column vector (K-by-1) • Unoriented vector (K) 	1	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1) • Row vector (1-by-M) 	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1)
<ul style="list-style-type: none"> • Row vector (1-by-N) • Matrix (K-by-N) 	N	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1) • Row vector (1-by-M) • Matrix (M-by-N) 	<ul style="list-style-type: none"> • Scalar • Matrix (M-by-N)

Sample-Based Processing

Input	Number of Channels	Valid Initial States (Dialog Box)	Valid Initial States (Input Port)
<ul style="list-style-type: none"> • Scalar 	1	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1) • Row vector (1-by-M) 	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1) • Row vector (1-by-M)
<ul style="list-style-type: none"> • Row vector (1-by-N) • Column vector (N-by-1) • Unoriented vector (N) 	N	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1) • Row vector (1-by-M) • Matrix (M-by-N) 	<ul style="list-style-type: none"> • Scalar
<ul style="list-style-type: none"> • Matrix (K-by-N) 	$K \times N$	<ul style="list-style-type: none"> • Scalar • Column vector (M-by-1) • Row vector (1-by-M) • Matrix (M-by-$(K \times N)$) 	<ul style="list-style-type: none"> • Scalar

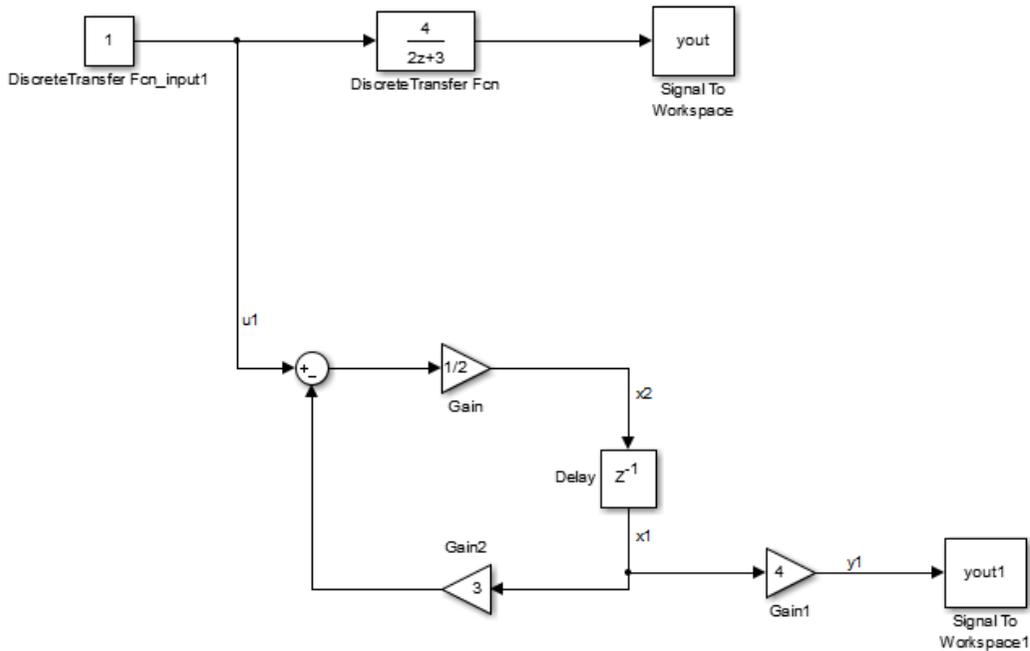
When the **Initial states** is a scalar, the block initializes all filter states to the same scalar value. Enter 0 to initialize all states to zero. When the **Initial states** is a vector or a matrix, each vector or matrix element specifies a unique initial state for a corresponding delay element in a corresponding channel:

- The vector length must equal the number of delay elements in the filter, $M = \max(\text{number of zeros}, \text{number of poles})$.
- The matrix must have the same number of rows as the number of delay elements in the filter, $M = \max(\text{number of zeros}, \text{number of poles})$. The matrix must also have one column for each channel of the input signal.

The following example shows the relationship between the initial filter output and the initial input and state. Given an initial input u_1 , the first output y_1 is related to the initial state $[x_1, x_2]$ and initial input by:

$$y_1 = 4x_1$$

$$x_2 = 1/2(u_1 - 3x_1)$$



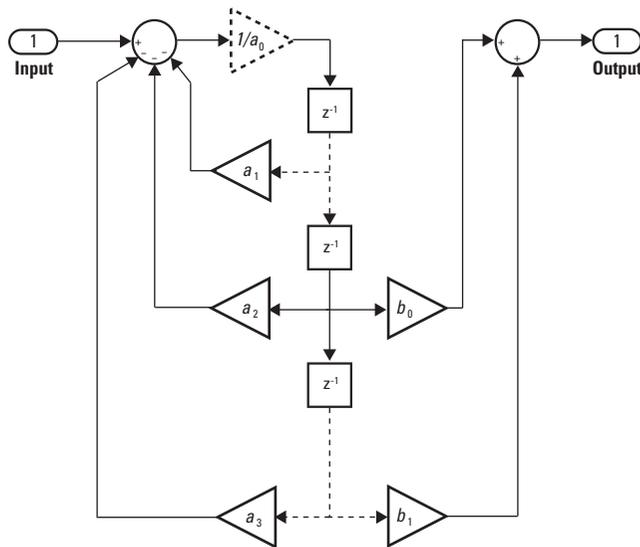
Data Type Support

The Discrete Transfer Function block accepts and outputs real and complex signals of any signed numeric data type that Simulink supports. The block supports the same types for the numerator and denominator coefficients.

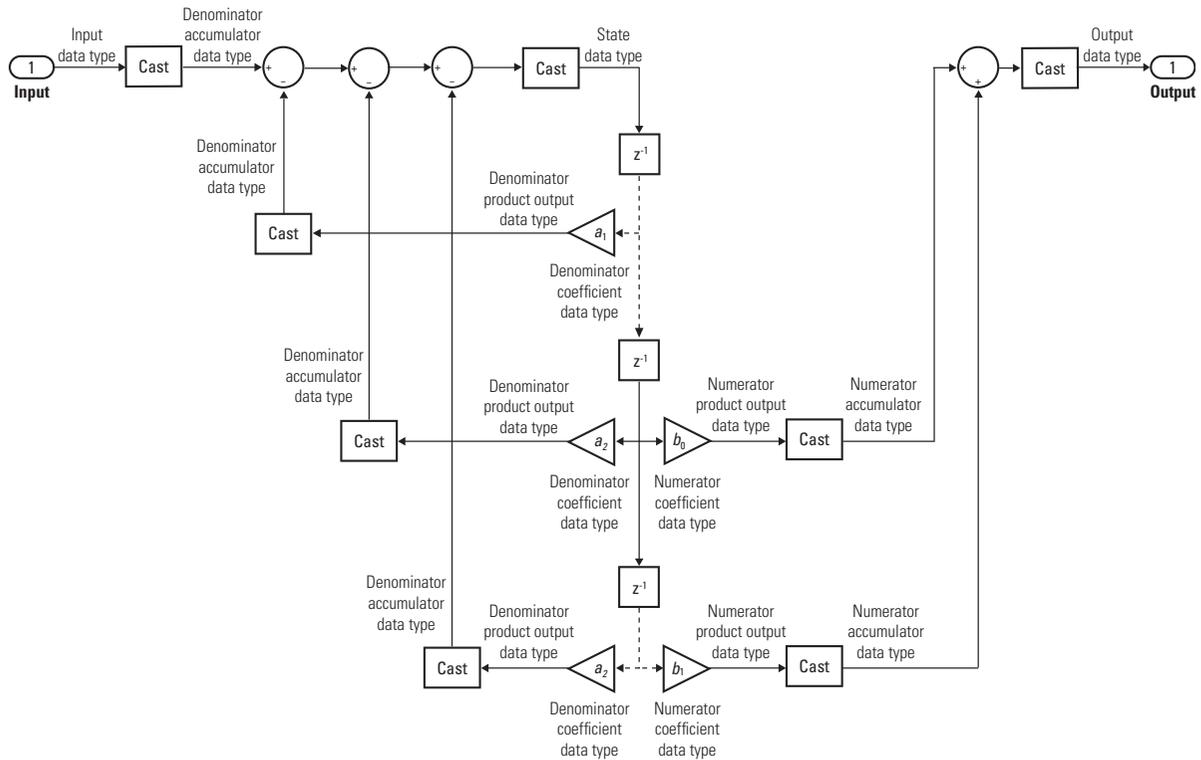
Numerator and denominator coefficients must have the same complexity. They can have different word lengths and fraction lengths.

States are complex when either the input or the coefficients are complex.

The following diagrams show the filter structure and the data types that the block uses for floating-point and fixed-point signals.



The block omits the dashed divide when you select the **Optimize by skipping divide by leading denominator coefficient (a0)** parameter.



Parameters

Numerator

Numerator coefficients of the discrete transfer function. To specify the coefficients, set the **Source** to `Dialog`. Then enter the coefficients in **Value** as descending powers of z . Use a row vector to specify the coefficients for a single numerator polynomial. Use a matrix to specify coefficients for multiple filters to be applied to the same input. Each matrix row represents a set of filter taps.

Denominator

Denominator coefficients of the discrete transfer function. To specify the coefficients, set the **Source** to `Dialog`. Then, enter the coefficients in **Value** as descending powers of z . Use a row vector to specify the coefficients for a single denominator

polynomial. Use a matrix to specify coefficients for multiple filters to be applied to the same input. Each matrix row represents a set of filter taps.

Initial states

If the **Source** is `Dialog`, then, in **Value**, specify the initial states of the filter states. To learn how to specify initial states, see “Specifying Initial States” on page 1-454.

If the **Source** is `Input port`, then there is nothing to be specified for **Value**.

External reset

Specify the trigger event to use to reset the states to the initial conditions.

Reset Mode	Behavior
None	No reset.
Rising	Reset on a rising edge.
Falling	Reset on a falling edge.
Either	Reset on either a rising or falling edge.
Level	Reset in either of these cases: <ul style="list-style-type: none"> • when the reset signal is nonzero at the current time step • when the reset signal value changes from nonzero at the previous time step to zero at the current time step
Level hold	Reset when the reset signal is nonzero at the current time step

The reset signal must be a scalar of type `single`, `double`, `boolean`, or `integer`. Fixed point data types, except for `ufix1`, are not supported.

Input processing

Specify whether the block performs sample- or frame-based processing.

- `Elements as channels` (sample based) — Process each element of the input as an independent channel.
- `Columns as channels` (frame based) — Process each column of the input as an independent channel.

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Optimize by skipping divide by leading denominator coefficient (a0)

Select when the leading denominator coefficient, a_0 , equals one. This parameter optimizes your code.

When you select this check box, the block does not perform a divide-by- a_0 either in simulation or in the generated code. An error occurs if a_0 is not equal to one.

When you clear this check box, the block is fully tunable during simulation, and performs a divide-by- a_0 in both simulation and code generation.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in “How Simulink Works” in the *Simulink User's Guide*.

State

Specify the state data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **State** parameter.

See “Specify Data Types Using Data Type Assistant” in the *Simulink User's Guide* for more information.

Numerator coefficients

Specify the numerator coefficient data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in integer, for example, `int8`

- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Numerator coefficients** parameter.

See “Specify Data Types Using Data Type Assistant” in the *Simulink User’s Guide* for more information.

Numerator coefficient minimum

Specify the minimum value that a numerator coefficient can have. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

Numerator coefficient maximum

Specify the maximum value that a numerator coefficient can have. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

Numerator product output

Specify the product output data type for the numerator coefficients. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Numerator product output** parameter.

See “Specify Data Types Using Data Type Assistant” in the *Simulink User's Guide* for more information.

Numerator accumulator

Specify the accumulator data type for the numerator coefficients. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Numerator accumulator** parameter.

See “Specify Data Types Using Data Type Assistant” in the *Simulink User's Guide* for more information.

Denominator coefficients

Specify the denominator coefficient data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Denominator coefficients** parameter.

See “Specify Data Types Using Data Type Assistant” in the *Simulink User's Guide* for more information.

Denominator coefficient minimum

Specify the minimum value that a denominator coefficient can have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)

- Automatic scaling of fixed-point data types

Denominator coefficient maximum

Specify the maximum value that a denominator coefficient can have. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

Denominator product output

Specify the product output data type for the denominator coefficients. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Denominator product output** parameter.

See “Specify Data Types Using Data Type Assistant” in the *Simulink User's Guide* for more information.

Denominator accumulator

Specify the accumulator data type for the denominator coefficients. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Denominator accumulator** parameter.

See “Specify Data Types Using Data Type Assistant” in the *Simulink User's Guide* for more information.

Output

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See “Control Signal Data Types” in the *Simulink User's Guide* for more information.

Output minimum

Specify the minimum value that the block can output. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value that the block can output. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer). in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</p>	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code> , which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code> , is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect

when overflow is not possible. In this case, the code generator does not produce saturation code.

State name

Use this parameter to assign a unique name to the block state. The default is ' '. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder).

State name must resolve to Simulink signal object

Select this check box to require that the state name resolve to a Simulink signal object. This check box is cleared by default.

State name enables this parameter. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class**.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see “Apply Custom Storage Classes Directly to Signal Lines, Block States, and Output Blocks” (Embedded Coder).

To programmatically set this parameter, use `StateSignalObject`.

For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage

classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

StorageClass

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

State name enables this parameter.

To programmatically set this parameter, use `StateStorageClass` or `StateSignalObject`. See “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder).

TypeQualifier

Note **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name resolves
- Minimum and maximum values of the signal object

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder) in the Simulink Coder documentation.

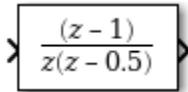
Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Specified in the Sample time parameter
Direct Feedthrough	Only when the leading numerator coefficient is not equal to zero and the numerator order equals the denominator order
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Discrete Zero-Pole

Model system defined by zeros and poles of discrete transfer function



Library

Discrete

Description

The Discrete Zero-Pole block models a discrete system defined by the zeros, poles, and gain of a z -domain transfer function. This block assumes that the transfer function has the following form:

$$H(z) = K \frac{Z(z)}{P(z)} = K \frac{(z - Z_1)(z - Z_2) \dots (z - Z_m)}{(z - P_1)(z - P_2) \dots (z - P_n)},$$

where Z represents the zeros vector, P the poles vector, and K the gain. The number of poles must be greater than or equal to the number of zeros ($n \geq m$). If the poles and zeros are complex, they must be complex conjugate pairs.

The block displays the transfer function depending on how the parameters are specified. See Zero-Pole for more information.

Data Type Support

The Discrete Zero-Pole block accepts and outputs real signals of type `double` and `single`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Zeros

Specify the matrix of zeros. The default is $[1]$.

Poles

Specify the vector of poles. The default is $[0 \ 0.5]$.

Gain

Specify the gain. The default is 1.

Sample time

Specify the time interval between samples. See [Specifying Sample Time](#) in the “How Simulink Works” chapter of the Simulink documentation.

State name

Use this parameter to assign a unique name to the block state. The default is ' '. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder).

State name must resolve to Simulink signal object

Select this check box to require that the state name resolve to a Simulink signal object. This check box is cleared by default.

State name enables this parameter. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class**.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-

in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see “Apply Custom Storage Classes Directly to Signal Lines, Block States, and Outport Blocks” (Embedded Coder).

To programmatically set this parameter, use `StateSignalObject`.

For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

StorageClass

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

State name enables this parameter.

To programmatically set this parameter, use `StateStorageClass` or `StateSignalObject`. See “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder).

TypeQualifier

Note **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use

an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name is resolved
- Min and Max values of the signal object

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder) in the Simulink Coder documentation.

Characteristics

Data Types	Double Single
Sample Time	Specified in the Sample time parameter
Direct Feedthrough	Yes, if the number of zeros and poles are equal
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Display

Display signal value during simulation

Library: Simulink / Dashboard

42

Description

The Display block connects to a signal in your model and displays its value during simulation. You can edit the parameters of the Display block while a simulation runs. Use the Display block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model.

Connection

Dashboard blocks do not use ports to connect to signals. To connect a signal to a Dashboard block, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting signals of interest in your model.
- 3 Mark the option button next to the signal you want to display.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.

- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined using the variable `engine(1)` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

Parameters

Connection — Select a signal to connect and display

signal connection options

Signal connection options. Populate the **Connection** table by selecting signals of interest in your model. Mark the check box next to the signal you want to display. Click **Apply** to connect the signal.

Format — Format for displaying numerical values

`long` (default) | `long_e`

Format for displaying numerical values.

- When you select `long`, the block displays numeric signal values using up to 15 digits with a fixed decimal point that displays up to four decimal places.
- When you select `long_e`, the block displays numeric signal values in scientific notation using up to 16 digits with a floating decimal point.

Alignment — Text alignment in block

'Center' (default) | 'Left' | 'Right'

Text alignment in Display block.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

See Also

Dashboard Scope | Gauge | Lamp | MultiStateImage

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2017b

Display

Show value of input



Library

Sinks

Description

Format Options

You control the display format using the **Format** parameter:

If you select...	The block displays...
short	A 5-digit scaled value with fixed decimal point
long	A 15-digit scaled value with fixed decimal point
short_e	A 5-digit value with a floating decimal point
long_e	A 16-digit value with a floating decimal point
bank	A value in fixed dollars and cents format (but with no \$ or commas)
hex (Stored Integer)	The stored integer value of a fixed-point input in hexadecimal format

If you select...	The block displays...
binary (Stored Integer)	The stored integer value of a fixed-point input in binary format
decimal (Stored Integer)	The stored integer value of a fixed-point input in decimal format
octal (Stored Integer)	The stored integer value of a fixed-point input in octal format

If the input to a Display block has an enumerated data type (see “Simulink Enumerations” and “Define Simulink Enumerations”):

- The block displays enumerated values, not the values of underlying integers.
- Setting **Format** to any of the `Stored Integer` settings causes an error.

Display Abbreviations

The following abbreviations appear on the Display block to help you identify the format of the value.

When you see...	The value that appears is...
(SI)	The stored integer value
	Note (SI) does not appear when the signal is of an integer data type.
hex	In hexadecimal format
bin	In binary format
oct	In octal format

Frequency of Data Display

The amount of data that appears and the time steps at which the data appears depend on the **Decimation** block parameter and the `SampleTime` property:

- The **Decimation** parameter enables you to display data at every n th sample, where n is the decimation factor. The default decimation, 1, displays data at every time step.

Note The Display block updates its display at the initial time, even when the **Decimation** value is greater than one.

- The `SampleTime` property, which you can set with `set_param`, enables you to specify a sampling interval at which to display points. This property is useful when you are using a variable-step solver where the interval between time steps is not the same. The default sample time, `-1`, causes the block to ignore the sampling interval when determining the points to display.

Note If the block inherits a sample time of `Inf`, the **Decimation** parameter is ignored.

Resizing Options

If the block input is an array, you can resize the block to show more than just the first element. You can resize the block vertically or horizontally. If the block input is a vector, the block sequentially adds display fields from left to right and top to bottom. The block displays as many values as possible. A black triangle indicates that the block is not displaying all input array elements.

The Display block shows the first 200 elements of a vector signal and the first 20 rows and 10 columns of a matrix signal.

Floating Display

To use the block as a floating display, select the **Floating display** check box. The block input port disappears and the block displays the value of the signal on a selected line.

If you select **Floating display**:

- Turn off signal storage reuse for your model. See “Signal storage reuse” in the Simulink documentation for more information.
- Do not connect a multidimensional signal to a floating display. Otherwise, you get a simulation error because the block does not support multidimensional signals.

Data Type Support

The Display block accepts real or complex signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Format

Specify the format of the data that appears, as discussed in “Format Options” on page 1-476. The default is `short`.

Decimation

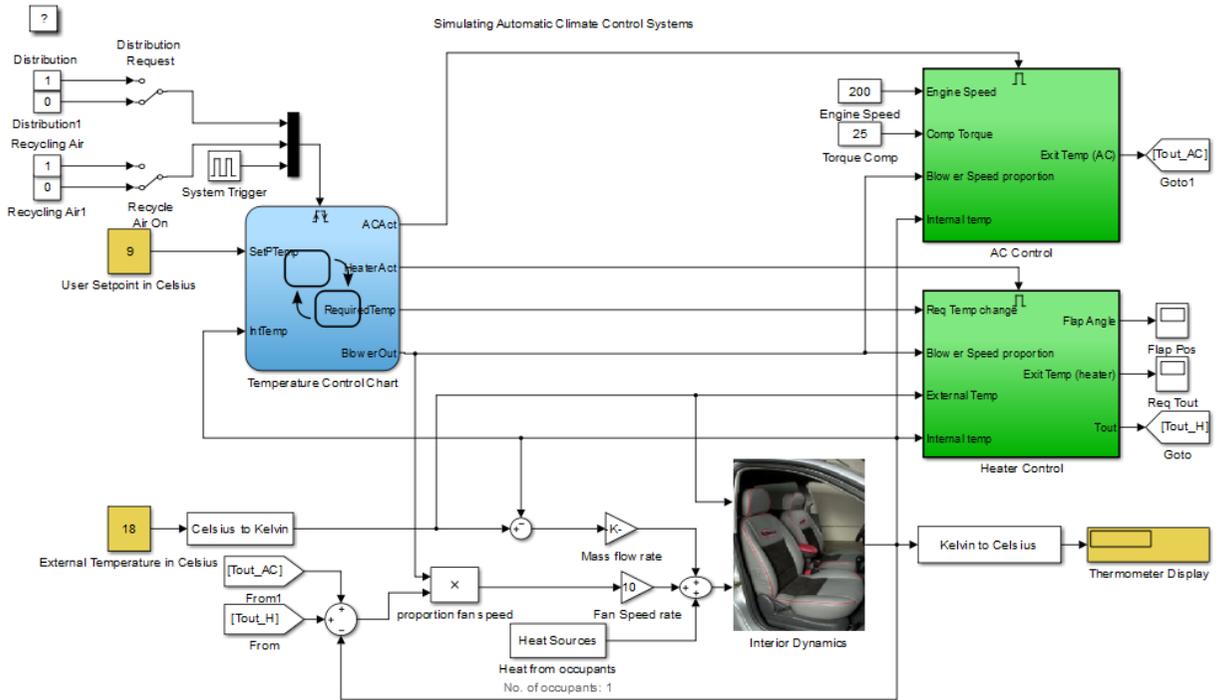
Specify how often to display data, as discussed in “Frequency of Data Display” on page 1-477. The default is 1.

Floating display

Select to use the block as a floating display, as discussed in “Floating Display” on page 1-478.

Examples

The `sldemo_auto_climatecontrol` model shows how you can use the Display block.



Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Use <code>set_param</code> to specify the <code>SampleTime</code> property
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	No

See Also

Scope

Introduced before R2006a

Divide

Divide one input by another



Library

Math Operations

Description

The Product and Product of Elements blocks are variants of the Divide block.

- For information on the Product block, see Product.
- For information on the Product of Elements block, see Product of Elements.

Supported Block Operations

The Divide block outputs the result of dividing its first input by its second. The inputs can be scalars, a scalar and a nonscalar, or two nonscalars that have the same dimensions. The Divide block is functionally a Product block that has two block parameter values preset:

- **Multiplication:** `Element-wise(.*)`
- **Number of Inputs:** `*/`

Setting non-default values for either of those parameters can change a Divide block to be functionally equivalent to a Product block or a Product of Elements block. See the documentation of those two blocks for more information.

Expected Differences Between Simulation and Code Generation

If any of the Divide block inputs contains a NaN or `inf` value, or if the block generates NaN or `inf` during execution, you might see different results when you compare the block simulation results with the generated code. This difference is due to the nonfinite NaN or `inf` values. In such cases, inspect your model configuration and eliminate the conditions that produce NaN or `inf`.

Code Optimizations

The Simulink Coder build process provides efficient code for matrix inverse and division operations. The following summary describes the benefits and when each benefit is available:

Benefit	Small matrices (2-by-2 to 5-by-5)	Medium matrices (6-by-6 to 20-by-20)	Large matrices (larger than 20-by-20)
Faster code execution time, compared to R2011a and earlier releases	Yes	No	Yes
Reduced ROM and RAM usage, compared to R2011a and earlier releases	Yes, for real values	Yes, for real values	Yes, for real values
Reuse of variables	Yes	Yes	Yes
Dead code elimination	Yes	Yes	Yes
Constant folding	Yes	Yes	Yes
Expression folding	Yes	Yes	Yes
Consistency with MATLAB Coder results	Yes	Yes	Yes

For blocks that have three or more inputs of different dimensions, the code might include an extra buffer to store temporary variables for intermediate results.

Parameters

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Number of inputs

Control two properties of the block:

- The number of input ports on the block
- Whether each input is multiplied or divided into the output

Default: * /

- **1 or * or /**

Has one input. In element-wise mode, processes the input as described for the Product of Elements block. In matrix mode, if the parameter value is 1 or *, the block outputs the input value. If the value is /, the input must be a square matrix (including a scalar as a degenerate case) and the block outputs the matrix inverse. See “Element-wise Mode” on page 1-1176 and “Matrix Mode” on page 1-1177 for more information.

- **Integer value > 1**

Has the number of inputs given by the integer value. The inputs are multiplied together in element-wise mode or matrix mode, as specified by the **Multiplication** parameter. See “Element-wise Mode” on page 1-1176 and “Matrix Mode” on page 1-1177 for more information.

- **Unquoted string of two or more * and / characters**

Has the number of inputs given by the length of the character vector. Each input that corresponds to a * character is multiplied into the output. Each input that corresponds to a / character is divided into the output. The operations occur in

element-wise mode or matrix mode, as specified by the **Multiplication** parameter. See “Element-wise Mode” on page 1-1176 and “Matrix Mode” on page 1-1177 for more information.

Setting **Number of inputs** to * and selecting `Element-wise(.*)` for **Multiplication** enables the parameter **Multiply over**.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Multiplication

Specify whether the Product block operates in Element-wise mode or Matrix mode.

Default: `Element-wise(.*)`

`Element-wise(.*)`

Operate in Element-wise mode.

`Matrix(*)`

Operate in Matrix mode.

Selecting `Element-wise(.*)` and setting **Number of inputs** to * enable the following parameter:

- **Multiply over**

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Multiply over

Affect multiplication on matrix input.

Default: All dimensions

All dimensions

Output a scalar that is product of all elements of the matrix, or the product of their inverses, depending on the value of **Number of inputs**.

Specified dimension

Output a vector, the composition of which depends on the value of the **Dimension** parameter.

- Enable this parameter by selecting `Element-wise (.*)` for **Multiplication** and setting **Number of inputs** to `*` or `1` or `/`.
- Setting this parameter to `Specified dimension` enables the **Dimension** parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Dimension

Affect multiplication on matrix input.

Default: 1

Minimum: 1

Maximum: 2

1

Output a vector that contains an element for each column of the input matrix.

2

Output a vector that contains an element for each row of the input matrix.

Each element of the output vector contains the product of all elements in the corresponding column or row of the input matrix, or the product of the inverses of those elements, depending on the value of **Number of inputs**:

- 1 or *

Multiply the values of the column or row elements

- /

Multiply the inverses of the column or row elements

Enable this parameter by selecting `Specified` dimension for **Multiply over**.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Require all inputs to have the same data type

Require that all inputs have the same data type.

Default: Off

On

Require that all inputs have the same data type.

Off

Do not require that all inputs have the same data type.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: LockScale

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: Floor

Ceiling

Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

Floor

Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

Nearest

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Rounding” (Fixed-Point Designer).

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off

On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.

- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Parameter: SaturateOnIntegerOverflow

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Output minimum

Lower value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMin
Type: character vector
Value: ' [] '
Default: ' [] '

Output maximum

Upper value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMax
Type: character vector
Value: ' [] '
Default: ' [] '

Output data type

Specify the output data type.

Default: `Inherit: Inherit via internal rule`

`Inherit: Inherit via internal rule`

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit: Same as first input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit: Inherit via back propagation`

Use data type of the driving block.

`Inherit: Same as first input`

Use data type of the first input signal.

`double`

Output data type is double.

`single`

Output data type is single.

`int8`

Output data type is int8.

`uint8`

Output data type is uint8.

`int16`

Output data type is `int16`.

`uint16`

Output data type is `uint16`.

`int32`

Output data type is `int32`.

`uint32`

Output data type is `uint32`.

`fixdt(1,16,0)`

Output data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`

Output data type is fixed point `fixdt(1,16,2^0,0)`.

<data type expression>

Use a data type object, for example, `Simulink.NumericType`.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Control Signal Data Types”.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule (default)`
- `Inherit via back propagation`
- `Same as first input`

Built in

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

Fixed point

Fixed-point data types.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

Off

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: Signed

Signed

Specify the fixed-point data as signed.

Unsigned

Specify the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: `Best precision`

`Binary point`

Specify binary point location.

`Slope and bias`

Enter slope and bias.

`Best precision`

Specify best-precision values.

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > Binary point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > Slope and bias enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

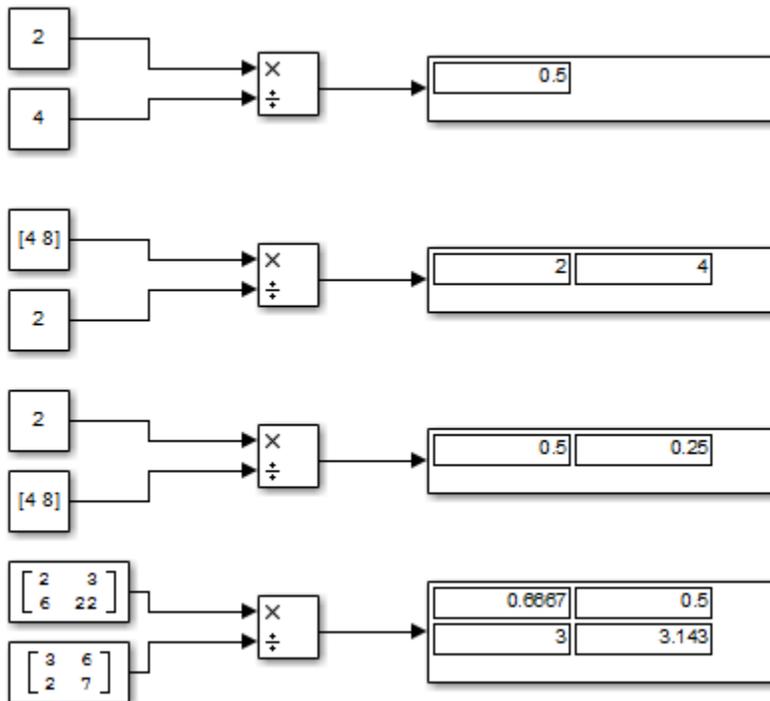
Specify any real number.

Selecting **Scaling** > Slope and bias enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Examples

The following examples show the output of the Divide block for some typical inputs using default block parameter values.



Introduced before R2006a

DocBlock

Create text that documents model and save text with model



Library

Model-Wide Utilities

Description

The DocBlock allows you to create and edit text that documents a model, and save that text with the model. Double-clicking an instance of the block creates a temporary file containing the text associated with this block and opens the file in an editor. Use the editor to modify the text and save the file. Simulink software stores the contents of the saved file in the model file.

The DocBlock supports HTML, Rich Text Format (RTF), and ASCII text document types. The default editors for these different document types are

- HTML — Microsoft® Word (if available). Otherwise, the DocBlock opens HTML documents using the editor specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.
- RTF — Microsoft Word (if available). Otherwise, the DocBlock opens RTF documents using the editor specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.
- Text — The DocBlock opens text documents using the editor specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.

Use the `docblock` command to change the default editors.

Data Type Support

Not applicable.

Parameters

Code generation template symbol (Embedded Coder license required)

Enter a template symbol name in this field. Embedded Coder software uses this symbol to add comments to the code generated from the model. For more information, see “Add Global Comments” (Embedded Coder).

Document type

Select the type of document associated with the DocBlock. The options are:

- Text (the default)
- RTF
- HTML

Note If you are using a DocBlock to add comments to your code during code generation, ensure that you set the **Document Type** as Text. If you set the **Document Type** as RTF or HTML, your comments will not appear in the code.

Characteristics

Data Types	Not applicable
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

Introduced before R2006a

Dot Product

Generate dot product of two vectors



Library

Math Operations

Description

The Dot Product block generates the dot product of the vectors at its inputs. The scalar output, y , is equal to the MATLAB operation

$$y = \text{sum}(\text{conj}(u1) .* u2)$$

where $u1$ and $u2$ represent the vectors at the block's top and bottom inputs, respectively. (See “Port Location After Rotating or Flipping” in the Simulink documentation for a description of the port order for various block orientations.) The inputs can be vectors, column vectors (single-column matrices), or scalars. If both inputs are vectors or column vectors, they must be the same length. If $u1$ and $u2$ are both column vectors, the block outputs the equivalent of the MATLAB expression $u1' * u2$.

The elements of the input vectors can be real- or complex-valued signals. The signal type (complex or real) of the output depends on the signal types of the inputs.

Input 1	Input 2	Output
real	real	real
real	complex	complex
complex	real	complex

Input 1	Input 2	Output
complex	complex	complex

To perform element-by-element multiplication without summing, use the Product block.

Data Type Support

The Dot Product block accepts and outputs signals of any numeric data type that Simulink supports, including fixed-point data types.

For more information, see “Data Types Supported by Simulink”.

Parameters

Require all inputs to have same data type

Select to require all inputs to have the same data type.

Output minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as

SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer). in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</p>	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code> , which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code> , is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect

when overflow is not possible. In this case, the code generator does not produce saturation code.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Product

Introduced before R2006a

Dual Port RAM

Dual port RAM with two output ports



Library

HDL Coder / HDL Operations

Description

The Dual Port RAM block models a RAM that supports simultaneous read and write operations, and has both a read data output port and write data output port. You can use this block to generate HDL code that maps to RAM in most FPGAs.

If you do not need to use the write output data, `wr_dout`, you can achieve better RAM inference with synthesis tools by using the Simple Dual Port RAM block.

Read-During-Write Behavior

During a write, new data appears at the output of the write port (`wr_dout`) of the Dual Port RAM block. If a read operation occurs simultaneously at the same address as a write operation, old data appears at the read output port (`rd_dout`).

Parameters

Address port width

Address bit width. Minimum bit width is 2, and maximum bit width is 29. The default is 8.

Ports

The block has the following ports:

`wr_din`

Write data input. The data can be any width. It inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

`wr_addr`

Write address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0

`wr_en`

Write enable.

Data type: Boolean

`rd_addr`

Read address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0

`wr_dout`

Output data from write address, `wr_addr`.

`rd_dout`

Output data from read address, `rd_addr`.

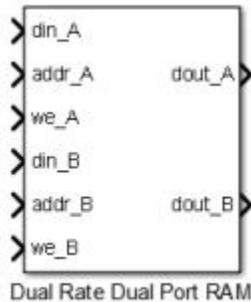
See Also

Dual Rate Dual Port RAM | Simple Dual Port RAM | Single Port RAM

Introduced in R2014a

Dual Rate Dual Port RAM

Dual Port RAM that supports two rates



Library

HDL Coder / HDL Operations

Description

The Dual Rate Dual Port RAM block models a RAM that supports simultaneous read and write operations to different addresses at two clock rates. Port A of the RAM can run at one rate, and port B can run at a different rate.

In high-performance hardware applications, you can use this block to access the RAM twice per clock cycle. If you generate HDL code, this block maps to a dual-clock dual-port RAM in most FPGAs.

Simultaneous Access

You can access different addresses from ports A and B simultaneously. You can also read the same address from ports A and B simultaneously.

However, do not access an address from one RAM port while it is being written from the other RAM port. During simulation, if you access an address from one RAM port at the same time as you write that address from the other RAM port, the software reports an error.

Read-During-Write Behavior

The RAM has write-first behavior. When you write to the RAM, the new write data is immediately available at the output port.

HDL Code Generation

For simulation results that match the generated HDL code, in the Solver pane of the Configuration Parameters dialog box, clear the checkbox for **Treat each discrete rate as a separate task**. When the checkbox is cleared, single-tasking mode is enabled.

If you simulate this block with **Treat each discrete rate as a separate task** selected, multitasking mode is enabled. The output data can update in the same cycle but in the generated HDL code, the output data is updated one cycle later.

Parameters

Address port width

Address bit width. Minimum bit width is 2, and maximum bit width is 28. The default value is 8.

Ports

The block has the following ports:

`din_A`

Write data input for RAM port A. The data can be any width. It inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

`addr_A`

Write address for RAM port A.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0

`we_A`

Write enable for RAM port A. Set `we_A` to `true` for a write operation, or `false` for a read operation.

Data type: Boolean

`din_B`

Write data input for RAM port B. The data can be of any width, and inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

`addr_B`

Write address for RAM port B.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0

`we_B`

Write enable for RAM port B. Set `we_B` to `true` for a write operation, or `false` for a read operation.

Data type: Boolean

`dout_A`

Output data from RAM port A address, `addr_A`.

`dout_B`

Output data from RAM port B address, `addr_B`.

See Also

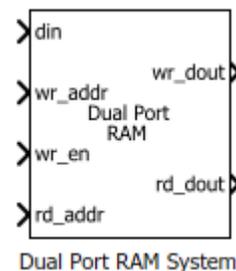
Dual Port RAM | HDL FIFO | Simple Dual Port RAM | Single Port RAM

Introduced in R2014a

Dual Port RAM System, Simple Dual Port RAM System, Single Port RAM System

RAM blocks based on the `hdl.RAM` system object with ability to provide initial value

Library: HDL Coder / HDL RAMs / Dual Port RAM System
 HDL Coder / HDL RAMs / Simple Dual Port RAM System
 HDL Coder / HDL RAMs / Single Port RAM System



Description

The blocks are MATLAB System blocks that use the `hdl.RAM System` object™. You can specify the RAM type as `Dual port`, `Simple dual port`, or `Single port`. In terms of simulation behavior, the Dual Port RAM System block behaves similar to the Dual Port RAM, the Single Port RAM System behaves similar to the Single Port RAM, and so on. With the MATLAB System blocks, you can:

- Specify an initial value for the RAM. In the Block Parameters dialog box, enter a value for **Specify the RAM initial value**.
- Obtain faster simulation results when you use these blocks in your Simulink model.

Ports

Input

din — Write data input

Scalar (default) | Vector

Data that you write into the RAM memory location when `wrEn` is true. This value can be double, single, integer, or a fixed-point (`fi`) object, and can be real or complex.

Data Types: `single` | `double` | `int8` | `int16` | `uint8` | `uint16` | `fixed point`

addr — Write or Read address

Scalar (default) | Vector

Address that you write the data into when `wrEn` is true. The RAM reads the value in memory location **addr** when `wrEn` is false. This value can be either fixed-point (`fi`) or integer, and must be real and unsigned.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to `Single port`.

Data Types: `uint8` | `uint16` | `fixed point`

wr_addr — Write address

Scalar (default) | Vector

RAM address that you write the data into. This value can be either fixed-point (`fi`) or integer, and must be real and unsigned.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to `Simple dual port` or `Dual port`.

Data Types: `uint8` | `uint16` | `fixed point`

wr_en — Write enable

Scalar (default) | Vector

When `wrEn` is true, the RAM writes the data into the memory location that you specify. If you set the **Specify the type of RAM** to `Single port`, the RAM reads the value in the memory location **addr** when `wrEn` is false.

Data Types: `Boolean`

rd_addr — Read address

Scalar (default) | Vector

Address that you read the data from the RAM. This value can be either `fixed-point` (`fi`) or `integer`, and must be real and unsigned.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to `Simple dual port` or `Dual port`.

Data Types: `uint8` | `uint16` | `fixed point`

Output

`dout` — Output data

`Scalar` (default) | `Vector`

Output data that the RAM reads from the memory location `addr` when `wrEn` is false.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to `Single port`.

`rd_dout` — Read data

`Scalar` (default) | `Vector`

Old output data that the RAM reads from the memory location `rd_addr`.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to `Simple dual port` or `Dual port`.

`wr_dout` — Write data output

`Scalar` (default) | `Vector`

New or old output data that the RAM reads from the memory location `wr_addr`.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to `Dual port`.

Parameters

Specify the type of RAM — RAM type

Dual port (default) | Simple dual port | Single port

Type of RAM, specified as either:

- `Single port` — Create a single port RAM with Write data, Address, and Write enable as inputs and Read data as the output.
- `Simple dual port` — Create a simple dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address as the output.
- `Dual port` — Create a dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address and write address as the outputs.

The code generator dynamically configures the input and output ports of the block based on the RAM type that you specify.

Specify the output data for a write operation — Write output behavior

New data (default) | Old data

Behavior for Write output, specified as either:

- `'New data'` — Send out new data at the address to the output.
- `'Old data'` — Send out old data at the address to the output.

Specify the RAM initial value — Initial simulation output of RAM

'0.0' (default) | Scalar | Vector

Initial simulation output of the System object, specified as either:

- A scalar value.
- A vector with one-to-one mapping between the initial value and the RAM words.

Block Characteristics

Data Types	double single base integer fixed point bus
Sample Time	Inherit
Direct Feedthrough	Yes
Multidimensional Signals	Scalar
Variable-Size Signals	Yes
Zero-Crossing Detection	No

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™.

If have HDL Coder installed, you can generate HDL code for the blocks. For more information, see Dual Port RAM System, Simple Dual Port RAM System, and Single Port RAM System.

Fixed-Point Conversion

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

See Also

Dual Port RAM | Dual Rate Dual Port RAM | Simple Dual Port RAM | Single Port RAM

Introduced in R2017b

Edit

Enter new value for parameter

Library: Simulink / Dashboard



Description

The Edit block allows you to type in new values for block parameters during simulation. Use the Edit block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model.

Double-clicking the Edit block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Connection

Dashboard block controls connect to variables and block parameters. To connect a Dashboard block to a variable or parameter, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting blocks of interest in your model.
- 3 Mark the option button next to the parameter or variable you want to adjust during simulation.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined using the variable `engine(1)` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

Parameters

Connection — Select a block parameter to connect

signal connection options

Select the block parameter to control using the **Connection** table. To open the dialog box, double-click the block. Then, populate the **Connection** table by selecting a block in your model. Select the variable parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram with **Ctrl+D**.

Align — Text alignment

'Center' (default) | 'Left' | 'Right'

Alignment of the text in the Edit block.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

See Also

Display | Knob | Slider

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2017b

Enable

Add enable port to subsystem or model

Library: Ports & Subsystems



Description

The Enable block allows an external signal to control execution of a subsystem or a model. To enable this functionality, add the block to a Subsystem block or at the root level of a model that is referenced in a Model block.

If you use an enable port at the root-level of a model:

- For multi-rate models, set the solver to single-tasking.
- For models with a fixed-step size, at least one block in the model must run at the specified fixed-step size rate.

Ports

Output

Enable signal — External enable signal for a subsystem or model

scalar

Enable signal attached externally to the outside of an Enable Subsystem block and passed to the inside of the subsystem. An enable signal port is added to an Enable block when you select the **Show output port** parameter.

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

Parameters

States when enabling — Select block states when subsystem or model is disabled

held (default) | reset

When a Subsystem block or Model block is disabled, select what happens to block states for the blocks within the subsystem or model.

held

Hold block states at their previous values.

reset

Reset block states to their initial conditions (zero if not defined).

Programmatic Use

Block parameter: StatesWhenEnabling

Type: character vector

Value: 'held' | 'reset'

Default: 'held'

Propagate sizes of variable-size signals — Select when to propagate a variable-size signal

Only when enabling (default) | During execution

Select when to propagate a variable-size signal.

Only when enabling

Propagate a variable-size signal when reenabling a Subsystem block or Model block containing an Enable port block. When you select this option, sample time must be periodic.

During execution

Propagate variable-size signals at each time step.

Programmatic Use

Block parameter: PropagateVarSize

Type: character vector

Value: 'Only when enabling' | 'During execution'

Default: 'Only when enabling'

Show output port — Control display of output port for enable signal

off (default) | on

The output port passes the enable signal attached externally to the outside of an Enable Subsystem block or enabled Model block to the inside.

off

Remove the output port on the Enable port block.

on

Display an output port on the Enable port block. Selecting this option allows the subsystem or model to process the enable signal.

Programmatic Use

Block parameter: ShowOutputPort

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Enable zero-crossing detection — Control zero-crossing detection

on (default) | off

Control zero-crossing detection for a model.

on

Detect zero crossings.

off

Do not detect zero crossings.

Programmatic Use

Block parameter: ZeroCross

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Port dimensions — Specify dimensions for the enable signal

1 (default) | [n] | [m n]

Specify dimensions for the enable signal attached externally to a Model block and passed to the inside of the block.

1

Scalar signal.

[n]

Vector signal of width n.

[m n]

Matrix signal having m rows and n columns.

Programmatic Use

Block parameter: PortDimensions

Type: character vector

Value: '1' | '[n]' | '[m n]'

Default: '1'

Sample time — Specify time interval

-1 (default) | Ts | [Ts, To]

Specify time interval between block method execution. See “Specify Sample Time”.

-1

Sample time inherited from the model.

Ts

Scalar where Ts is the time interval.

[Ts, To]

Vector where Ts is the time interval and To is the initial time offset.

Programmatic Use

Block parameter: SampleTime

Type: character vector

Value: '-1' | 'Ts' | '[Ts, To]'

Default: '-1'

Minimum — Specify minimum output value for the enable signal

[] (default) | real scalar

Specify minimum value for the enable signal attached externally to a Model block and passed to the inside of the block.

Simulink uses this value to perform:

- Simulation range checking. See “Signal Ranges”.
- Automatic scaling of fixed-point data types.
- Optimization of generated code. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. See “Optimize using the specified minimum and maximum values”.

[]

Unspecified minimum value.

real scalar

Real double scalar value.

Programmatic Use

Block parameter: OutMin

Type: character vector

Value: '[]' | '<real scalar>'

Default: '[]'

Maximum — Specify maximum output value for the enable signal

[] (default) | real scalar

Specify maximum value for the enable signal attached externally to a Model block and passed to the inside of the block.

Simulink uses this value to perform:

- Simulation range checking. See “Signal Ranges”.
- Automatic scaling of fixed-point data types.
- Optimization of generated code. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. See “Optimize using the specified minimum and maximum values”.

[]

Unspecified maximum value.

real scalar

Real double scalar value.

Programmatic Use

Block parameter: OutMax

Type: character vector

Value: '[]' | '<real scalar>'

Default: '[]'

Data type — Specify output data type for the enable signal

double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 |
boolean | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^,0) | <data type
expression>

Specify data type for the enable signal attached externally to a Model block and passed to the inside of the block.

double

Double-precision floating point.

single

Single-precision floating point.

int8

Signed 8-bit integer.

uint8

Unsigned 8-bit integer.

int16

Signed 16-bit integer.

uint16

Unsigned 16-bit integer.

int32

Signed 32-bit integer.

uint32

Unsigned 32-bit integer.

boolean

Boolean with a value of true or false.

fixdt(1,16)

Signed 16-bit fixed point number with binary point undefined.

fixdt(1,16,0)

Signed 16-bit fixed point number with binary point set to zero.

```
fixdt(1,16,2^0)
```

Signed 16-bit fixed point number with slope set to 2^0 and bias set to 0.

```
<data type expression>
```

Data type object, for example `Simulink.NumericType`. Do not specify a bus object as the expression.

Programmatic Use

b

Block parameter: `OutDataTypeStr`

Type: character vector

Value: `'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'boolean' | '<fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'`

Default: `'double'`

Mode — Select data type category

```
Build in (default) | Fixed point | Expression
```

Select data type category and display drop-down lists to help you define the data type.

Build in

Display drop-down lists for data type and Data type override.

Fixed point

Display drop-down lists for Signedness, Scaling, and Data type override.

Expression

Display text box for entering an expression.

Dependency

To enable this parameter, select the Show data type assistant button.

Programmatic Use

No equivalent command-line parameter.

Interpolate data — Specify value of missing workspace data

```
on (default) | off
```

Specify value of missing workspace data when loading data from the workspace.

on

Linearly Interpolate output at time steps for which no corresponding workspace data exists.

off

Do not interpolate output at time steps. The current output equals the output at the most recent time step for which data exists.

Programmatic Use

Block parameter: Interpolate

Type: character vector

Value: 'on' | 'off'

Default: 'on'

See Also

[Enabled Subsystem](#) | [Enabled and Triggered Subsystem](#) | [Subsystem](#)

Topics

“Conditional Subsystems”

“Enabled Subsystems”

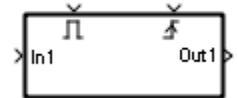
“Enabled and Triggered Subsystems”

Introduced before R2006a

Enabled and Triggered Subsystem

Subsystem whose execution is enabled and triggered by external input

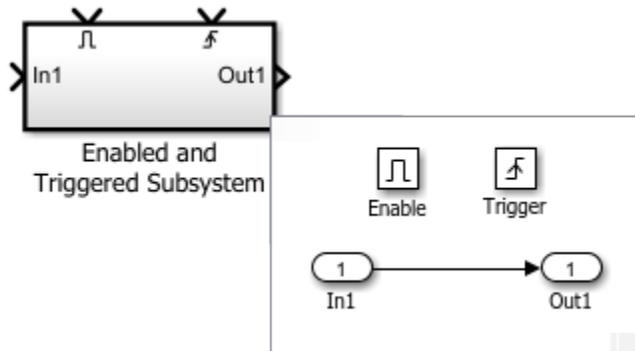
Library: Ports & Subsystems



Description

The Enabled and Triggered Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that executes when both of these conditions occur:

- Enable control signal has a positive value.
- Trigger control signal has a trigger event.



Use Enabled and Triggered Subsystem blocks to model:

- Optional functionality that runs with the detection of an event.
- Alternative functionality that runs with the detection of an event.

Ports

Input

In — Signal input to a subsystem block

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

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

Enable — Control signal input to a subsystem block

scalar

Placing an Enable block in a subsystem block adds an external input port to the block.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Trigger — Control signal input to a subsystem block

scalar

Placing a Trigger block in a subsystem block adds an external input port to the block.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

out — Signal output from a subsystem

scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

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

See Also

Blocks

[Enable](#) | [Enabled Subsystem](#) | [Function-Call Subsystem](#) | [Subsystem](#) | [Trigger](#) | [Triggered Subsystem](#)

Topics

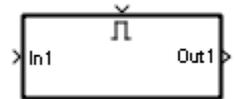
[“Conditional Subsystems”](#)
[“Enabled Subsystems”](#)
[“Triggered Subsystems”](#)
[“Enabled and Triggered Subsystems”](#)
[“Using Function-Call Subsystems”](#)

Introduced before R2006a

Enabled Subsystem

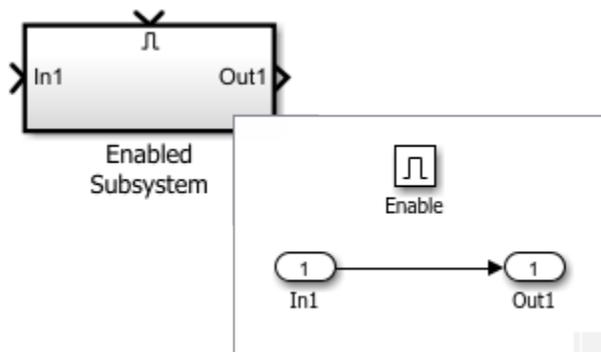
Subsystem whose execution is enabled by external input

Library: Ports & Subsystems



Description

The Enabled Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that executes when a control signal has a positive value.



Use Enable Subsystem blocks to model:

- Discontinuities
- Optional functionality
- Alternative functionality

Ports

Input

In — Signal input to a subsystem block

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

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

Enable — Control signal input to a subsystem block

scalar | vector | matrix

Placing an Enable block in a subsystem block adds an external input port to the block and changes the block to an Enable Subsystem block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Out — Signal output from a subsystem

scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

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

See Also

Blocks

Enable | Enabled and Triggered Subsystem | Function-Call Subsystem | Subsystem | Triggered Subsystem

Topics

“Conditional Subsystems”

“Enabled Subsystems”

“Triggered Subsystems”

“Enabled and Triggered Subsystems”

“Using Function-Call Subsystems”

Introduced before R2006a

Enabled Synchronous Subsystem

Represent enabled subsystem that has synchronous reset and enable behavior



Library

HDL Coder / HDL Subsystems

Description

An Enabled Synchronous Subsystem is an Enabled Subsystem that uses the Synchronous mode of the State Control block. If an **S** symbol appears in the subsystem, then it is synchronous.

To create an Enabled Synchronous Subsystem block, add the block to your Simulink model from the HDL Subsystems block library. You can also add a State Control block with **State control** set to *Synchronous* inside an Enabled subsystem.

For more information, see State Control and “Enabled Subsystems”.

Data Type Support

See Inport for information on the data types accepted by a subsystem's input ports. See Outport for information on the data types output by a subsystem's output ports.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Show port labels

Cause Simulink software to display labels for the subsystem's ports on the subsystem's icon.

Default: `FromPortIcon`

`none`

Does not display port labels on the subsystem block.

`FromPortIcon`

If the corresponding port icon displays a signal name, display the signal name on the subsystem block. Otherwise, display the port block's name.

`FromPortBlockName`

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

`SignalName`

If a name exists, display the name of the signal connected to the port on the subsystem block; otherwise, the name of the corresponding port block.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Read/Write permissions

Control user access to the contents of the subsystem.

Default: `ReadWrite`

`ReadWrite`

Enables opening and modification of subsystem contents.

`ReadOnly`

Enables opening but not modification of the subsystem. If the subsystem resides in a block library, you can create and open links to the subsystem and can make and

modify local copies of the subsystem but cannot change the permissions or modify the contents of the original library instance.

`NoReadOrWrite`

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

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Name of error callback function

Enter name of a function to be called if an error occurs while Simulink software is executing the subsystem.

Default: ' '

Simulink software passes two arguments to the function: the handle of the subsystem and a character vector that specifies the error type. If no function is specified, Simulink software displays a generic error message if executing the subsystem causes an error.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Permit hierarchical resolution

Specify whether to resolve names of workspace variables referenced by this subsystem.

Default: All

All

Resolve all names of workspace variables used by this subsystem, including those used to specify block parameter values and Simulink data objects (for example, `Simulink.Signal` objects).

`ExplicitOnly`

Resolve only names of workspace variables used to specify block parameter values, data store memory (where no block exists), signals, and states marked as “must resolve”.

None

Do not resolve any workspace variable names.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Treat as atomic unit

Causes Simulink software to treat the subsystem as a unit when determining the execution order of block methods.

Default: Off

On

Cause Simulink software to treat the subsystem as a unit when determining the execution order of block methods. For example, when it needs to compute the output of the subsystem, Simulink software invokes the output methods of all the blocks in the subsystem before invoking the output methods of other blocks at the same level as the subsystem block.

Off

Cause Simulink software to treat all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem when determining block method execution order. This can cause execution of methods of blocks in the subsystem to be interleaved with execution of methods of blocks outside the subsystem.

This parameter enables:

- “Minimize algebraic loop occurrences” on page 1-0 .
- “Sample time” on page 1-0
- “Function packaging” on page 1-0 (requires a Simulink Coder license)

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Treat as grouped when propagating variant conditions

Causes Simulink software to treat the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks.

Default: On

On

Simulink treats the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks. For example, when Simulink computes the variant condition of the subsystem, it propagates that condition to all the blocks in the subsystem.

Off

Simulink treats all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem itself when determining their variant condition.

“Treat as grouped when propagating variant conditions” on page 1-0 enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Function packaging

Specify the code format to be generated for an atomic (nonvirtual) subsystem.

Default: Auto

Auto

Simulink Coder software chooses the optimal format for you based on the type and number of instances of the subsystem that exist in the model.

Inline

Simulink Coder software inlines the subsystem unconditionally.

Nonreusable function

Simulink Coder software explicitly generates a separate function in a separate file. Subsystems with this setting generate functions that might have arguments

depending on the “Function interface” on page 1-0 parameter setting. You can name the generated function and file using parameters “Function name” on page 1-0 and “File name (no extension)” on page 1-0 . These functions are not reentrant.

Reusable function

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

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

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes
HDL Code Generation	Yes

See Also

Enable | Resettable Synchronous Subsystem | State Control | Synchronous Subsystem

Introduced in R2016a

Enumerated Constant

Generate enumerated constant value



Library

Sources

Description

The Enumerated Constant block outputs a scalar, array, or matrix of enumerated values. You can also use the Constant block to output enumerated values, but it provides block parameters that do not apply to enumerated types, such as **Output minimum** and **Output maximum**. When you need a block that outputs only constant enumerated values, preferably use Enumerated Constant rather than Constant. For more information, see “Simulink Enumerations”.

Data Type Support

The Enumerated Constant block supports only enumerated data types. Use the Constant block to output constant data of other types. For more information, see “Data Types Supported by Simulink”.

Parameters

Output data type

The **Output data type** field specifies the enumerated type from which you want the block to output one or more values. The initial value, Enum: SlDemoSign, is a dummy

enumerated type that prevents a newly cloned block from causing an error. To specify the desired enumerated type, select it from the pulldown or enter `Enum: ClassName` in the **Output data type** field, where *ClassName* is the name of the MATLAB class that defines the type.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Value

The **Value** field specifies the value(s) that the block outputs. The output of the block has the same dimensions and elements as the **Value** parameter. The initial value, `SlDemoSign.Positive`, is a dummy enumerated value that prevents a newly cloned block from causing an error.

To specify the desired enumerated value(s), select from the pulldown or enter any MATLAB expression that evaluates to the desired result, including an expression that uses tunable parameters. All values specified must be of the type indicated by the **Output data type**. To specify an array that includes every value in the enumerated type, use the `enumeration` function.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Sample time

Specify the interval between times that the Constant block output can change during simulation (for example, due to tuning the **Constant value** parameter).

Default: `inf`

This setting indicates that the block output can never change. This setting speeds simulation and generated code by avoiding the need to recompute the block output. See “Specify Sample Time” for more information.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Characteristics

Data Types	Enumerated
Sample Time	Specified in the Sample time parameter
Direct Feedthrough	Not applicable
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Topics

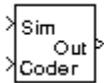
“Use Enumerated Data in Simulink Models”

“Simulink Enumerations”

Introduced in R2009b

Environment Controller

Create branches of block diagram that apply only to simulation or only to code generation



Library

Signal Routing

Description

This block outputs the signal at its Sim port only if the model that contains it is being simulated. It outputs the signal at its Coder port only if code is being generated from the model. This option enables you to create branches of a block diagram that apply only to simulation or code generation. The table below describes various scenarios where either the Sim or Coder port applies.

Scenario	Output
Normal mode simulation	Sim
Accelerator mode simulation	Sim
Rapid Accelerator mode simulation	Sim
Simulation of a referenced model (Normal or Accelerator modes)	Sim
Simulation of a referenced model (Processor-in-the-loop (PIL) mode)	Coder (uses the same code generated for a referenced model)
External mode simulation	Coder
Standard code generation	Coder
Code generation of a referenced model	Coder

Simulink Coder software does not generate code for blocks connected to the Sim port if these conditions hold:

- On the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box, you set **Default parameter behavior** to `Inlined`.
- The blocks connected to the Sim port do not have external signals.
- The Sim port input path does not contain an S-function or an Interpreted MATLAB Function block.

If you enable block reduction optimization, Simulink eliminates blocks in the branch connected to the Coder port when compiling the model for simulation. For information about block reduction, see “Block reduction” in the online Simulink documentation.

Note Simulink Coder code generation eliminates the blocks connected to the Sim branch only if the Sim branch has the same signal dimensions as the Coder branch. Regardless of whether it eliminates the Sim branch, Simulink Coder uses the sample times on the Sim branch as well as the Coder branch to determine the fundamental sample time of the generated code and might, in some cases, generate sample-time handling code that applies only to sample times specified on the Sim branch.

Data Type Support

The Environment Controller block accepts signals of any data type that Simulink supports. The output uses the same data type as the input.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

Introduced before R2006a

Event Listener

Add event port to a subsystem block

Library: Ports & Subsystems



Description

Add event port to a Subsystem block.

Parameters

Event type — Select event type for subsystem

`Initialize` (default) | `Terminate` | `Reset`

Select event type for subsystem to execute initialize, reset, or terminate algorithms.

`Initialize`

Select to trigger the execution of an Initialize Function block with an initialize event.

`Terminate`

Select to trigger the execution of a Terminate Function block with a terminate event.

`Reset`

Select to trigger the execution of an Initialize Function block reconfigured as a Reset Function block with a reset event.

Programmatic Use

Parameter: `EventType`

Type: character vector

Value: `'Initialize'` | `'Terminate'` | `'Reset'`

Default: `'Initialize'`

Event name — Specify event name

`Reset` (default) | event name

Specify event name for Reset Function block

Reset

Default name on the face of the Reset Function block.

event name

User entered name displayed on the face of the Reset Function block, and the name of the reset event port on the Model block containing the Reset Function block.

When typing the name for a reset function, the auto-completion list provides some suggestions. The list is not complete.

Dependency

To enable this parameter, set the **Event** parameter to `Reset`.

Programmatic Use

Parameter: `EventName`

Type: character vector

Value: `'reset' | '<event name>'`

Default: `'reset'`

Enable variant condition — Control activating the variant control (condition)

`off` (default) | `on`

Control activating the variant control (condition) defined with the **Variant Control** parameter.

off

Deactivate variant control of subsystem.

on

Activate variant control of subsystem.

Dependency

Selecting this parameter, enables the **Variant control** and **Generate preprocessor conditionals** parameters.

Programmatic Use

Parameter: `Variant`

Type: character vector

Value: `'off' | 'on'`

Default: 'off'

Variant control — Specify variant control (condition) expression

Variant (default) | logical expression

Specify variant control (condition) expression that executes a variant Initialize function, Reset function, or Terminate function block when the expression evaluates to true.

Variant

Default name for a logical (Boolean) expression.

logical expression

A logical (Boolean) expression or a Simulink.Variant object representing a boolean expression.

If you want to generate code for your model, define the variables in the expression as Simulink.Parameter objects.

Dependency

To enable this parameter, select the **Enable variant condition** parameter.

Programmatic Use

Parameter: VariantControl

Type: character vector

Value: 'Variant' | '<logical expression>'

Default: 'Variant'

Generate preprocessor conditionals — Select if variant choices are enclosed within C preprocessor conditional statements

off (default) | on

Select if variant choices are enclosed within C preprocessor conditional statements.

off

Does not enclose variant choices within C preprocessor conditional statements.

on

When generating code for an ERT target, encloses variant choices within C preprocessor conditional statements (#if).

Dependency

To enable this parameter, select the **Enable variant condition** parameter.

Programmatic Use

Parameter: `GeneratePreprocessorConditionals`

Type: character vector

Value: 'off' | 'on'

Default: 'off'

See Also

[Initialize Function](#) | [Reset Function](#) | [State Reader](#) | [State Writer](#) | [Terminate Function](#)

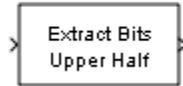
Topics

“Create Model to Initialize, Reset, and Terminate State”

“Create Test Harness to Generate Function Calls”

Extract Bits

Output selection of contiguous bits from input signal



Library

Logic and Bit Operations

Description

The Extract Bits block allows you to output a contiguous selection of bits from the stored integer value of the input signal. Use the **Bits to extract** parameter to define the method for selecting the output bits.

- Select `Upper half` to output the half of the input bits that contain the most significant bit. If there is an odd number of bits in the input signal, the number of output bits is given by the equation

$$\text{number of output bits} = \text{ceil}(\text{number of input bits}/2)$$

- Select `Lower half` to output the half of the input bits that contain the least significant bit. If there is an odd number of bits in the input signal, the number of output bits is given by the equation

$$\text{number of output bits} = \text{ceil}(\text{number of input bits}/2)$$

- Select `Range starting with most significant bit` to output a certain number of the most significant bits of the input signal. Specify the number of most significant bits to output in the **Number of bits** parameter.
- Select `Range ending with least significant bit` to output a certain number of the least significant bits of the input signal. Specify the number of least significant bits to output in the **Number of bits** parameter.

- Select `Range of bits` to indicate a series of contiguous bits of the input to output in the **Bit indices** parameter. You indicate the range in `[start end]` format, and the indices of the input bits are labeled contiguously starting at 0 for the least significant bit.

This block does not report wrap on overflow warnings during simulation. To report these warnings, see the `Simulink.restoreDiagnostic` reference page. The block does report errors due to wrap on overflow.

Data Type Support

The Extract Bits block accepts inputs of any numeric data type that Simulink supports, including fixed-point data types. Floating-point inputs are passed through the block unchanged. Boolean inputs are treated as `uint8` signals.

Note Performing bit operations on a signed integer is difficult. You can avoid difficulty by converting the data type of your input signals to unsigned integer types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Bits to extract

Select the method for extracting bits from the input signal.

Number of bits

(Not shown on dialog above.) Select the number of bits to output from the input signal. Signed integer data types can have no less than two bits in them. Unsigned data integer types can be as short as a single bit.

This parameter is only visible if you select `Range starting with most significant bit` or `Range ending with least significant bit` for the **Bits to extract** parameter.

Bit indices

(Not shown on dialog above.) Specify a contiguous range of bits of the input signal to output. Specify the range in `[start end]` format. The indices are assigned to the input bits starting with 0 at the least significant bit.

This parameter is only visible if you select `Range of bits` for the **Bits to extract** parameter.

Output scaling mode

Select the scaling mode to use on the output bits selection:

- When you select `Preserve fixed-point scaling`, the fixed-point scaling of the input is used to determine the output scaling during the data type conversion.
- When you select `Treat bit field as an integer`, the fixed-point scaling of the input is ignored, and only the stored integer is used to compute the output data type.

Example

Consider an input signal that is represented in binary by 110111001:

- If you select `Upper half` for the **Bits to extract** parameter, the output is 11011 in binary.
- If you select `Lower half` for the **Bits to extract** parameter, the output is 11001 in binary.
- If you select `Range starting with most significant bit` for the **Bits to extract** parameter, and specify 3 for the **Number of bits** parameter, the output is 110 in binary.
- If you select `Range ending with least significant bit` for the **Bits to extract** parameter, and specify 8 for the **Number of bits** parameter, the output is 10111001 in binary.
- If you select `Range of bits` for the **Bits to extract** parameter, and specify `[4 7]` for the **Bit indices** parameter, the output is 1011 in binary.

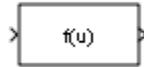
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Fcn

Apply specified expression to input



Library

User-Defined Functions

Description

The Fcn block applies the specified mathematical expression to its input. The expression can include one or more of these components:

- u — The input to the block. If u is a vector, $u(i)$ represents the i th element of the vector; $u(1)$ or u alone represents the first element.
- Numeric constants.
- Arithmetic operators (+ - * / ^).
- Relational operators (== != > < >= <=) — The expression returns 1 if the relation is true; otherwise, it returns 0.
- Logical operators (&& || !) — The expression returns 1 if the relation is true; otherwise, it returns 0.
- Parentheses.
- Mathematical functions — `abs`, `acos`, `asin`, `atan`, `atan2`, `ceil`, `cos`, `cosh`, `exp`, `floor`, `hypot`, `log`, `log10`, `power`, `rem`, `sgn` (equivalent to `sign` in MATLAB), `sin`, `sinh`, `sqrt`, `tan`, and `tanh`.

Note The Fcn block does not support `round` and `fix`. Use the Rounding Function block to apply these rounding modes.

- **Workspace variables** — Variable names that are not recognized in the preceding list of items are passed to MATLAB for evaluation. Matrix or vector elements must be specifically referenced (e.g., `A(1,1)` instead of `A` for the first element in the matrix).

The Fcn block observes the following rules of operator precedence:

- 1 ()
- 2 ^
- 3 + - (unary)
- 4 !
- 5 * /
- 6 + -
- 7 > < <= >=
- 8 == !=
- 9 &&
- 10 ||

The expression differs from a MATLAB expression in that the expression cannot perform matrix computations. Also, this block does not support the colon operator (:).

Block input can be a scalar or vector. The output is always a scalar. For vector output, consider using the Math Function block. If a block input is a vector and the function operates on input elements individually (for example, the `sin` function), the block operates on only the first vector element.

Limitations

The Fcn block has the following limitations:

- You cannot tune the expression during simulation in Normal or Accelerator mode (see “How Acceleration Modes Work”), or in generated code. To implement tunable expressions, tune the expression outside the Fcn block. For example, use the Relational Operator block to evaluate the expression outside.
- The Fcn block does not support custom storage classes. See “Custom Storage Classes” (Embedded Coder) in the Embedded Coder documentation.

Data Type Support

The Fcn block accepts and outputs signals of type `single` or `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Expression

Specify the mathematical expression to apply to the input. Expression components are listed above. The expression must be mathematically well-formed (uses matched parentheses, proper number of function arguments, and so on). The expression has restrictions on tunability (see “Limitations” on page 1-555)

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Examples

The following example models show how to use the Fcn block:

- `sldemo_absbrake`
- `sldemo_enginewc` (Throttle & Manifold/Throttle subsystem)

Characteristics

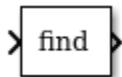
Data Types	Double Single
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No

Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Find

Find nonzero elements in array



Library

Math Operations

Description

The Find block locates all nonzero elements of the input signal and returns the linear indices of those elements. If the input is a multidimensional signal, the Find block can also return the subscripts of the nonzero input elements. In both cases, you can show an output port with the nonzero input values.

Data Type Support

The Find block accepts and outputs real values of any numeric data type that Simulink supports.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

- “Index output format” on page 1-559
- “Number of input dimensions” on page 1-559
- “Index mode” on page 1-560
- “Show output port for nonzero input values” on page 1-560

- “Sample time” on page 1-561
- “Output data type” on page 1-561
- “Mode” on page 1-562
- “Data type override” on page 1-563
- “Signedness” on page 1-564
- “Word length” on page 1-564
- “Scaling” on page 1-565

Index output format

Select the output format for the indices of the nonzero input values.

Default: Linear indices

Linear indices

Provides the element indices of any dimension signal in a vector form. For one dimension (vector) signals, indices correspond to the position of nonzero values within the vector. For signals with more than one dimension, the conversion of subscripts to indices is along the first dimension. You do not need to know the signal dimension of the input signal.

Subscripts

Provides the element indices of a two-dimension or larger signal in a subscript form. Because the block shows an output port for each dimension, this option requires you to know the number of dimensions for the input signal.

Selecting `Subscripts` from the **Index output format** list enables the **Number of input dimensions** parameter.

Parameter: `IndexOutputFormat`

Type: character vector

Value: `Linear indices` | `Subscripts`

Default: `Linear indices`

Number of input dimensions

Specify the number of dimensions for the input signal.

Default: 1

Minimum: 1

Maximum: 32

Selecting Subscripts from the **Index output format** list enables this parameter.

Parameter: NumberOfInputDimensions

Type: int

Value: positive integer value

Default: 1

Index mode

Specify the indexing mode.

Default: Zero-based

Zero-based

An index of 0 specifies the first element of the input vector. An index of 1 specifies the second element, and so on.

One-based

An index of 1 specifies the first element of the input vector. An index of 2, specifies the second element, and so on.

Parameter: IndexMode

Type: character vector

Value: Zero-based | One-based

Default: Zero-based

Show output port for nonzero input values

Show or hide the output port for nonzero input values.

Default: Off

On

Display the output port for nonzero input values. The additional output port provides the values of the nonzero input elements.

Off

Hide the output port for nonzero input values.

Parameter: ShowOutputPortForNonzeroInputValues

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Output data type

Specify the output data type.

Default: Inherit: Inherit via internal rule

Inherit: Inherit via internal rule

Output data type is defined by the target.

int8

Output data type is int8.

uint8

Output data type is uint8.

int16

Output data type is int16.

uint16

Output data type is uint16.

`int32`

Output data type is `int32`.

`uint32`

Output data type is `uint32`.

`fixdt(1,16)`

Output data type is fixed point, `fixdt(1,16)`.

`<data type expression>`

Use a data type object, for example, `Simulink.NumericType`.

Click the **Show data type assistant** button  to display additional parameters for the **Output data type** parameter.

Parameter: `OutDataTypeStr`

Type: character vector

Value: `'Inherit: Inherit via internal rule' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)' | '<data type expression>'`

Default: `'Inherit: Inherit via internal rule'`

“Control Signal Data Types”, “Specify Data Types Using Data Type Assistant”

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second list of the possible values:

- `Inherit via internal rule` (Discrete-Time Integrator, Gain, Product, Sum, Switch block default)

`Built in`

Built-in data types. Selecting `Built in` enables a second list of the possible values:

- int8
- uint8
- int16
- uint16
- int32
- uint32

Fixed point

Fixed-point data types.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second text box, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

Selecting `Fixed point` from the **Mode** list enables the following parameters:

- **Signed**
- **Scaling**
- **Word length**

“Specify Data Types Using Data Type Assistant”

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether the fixed-point data is signed or unsigned.

Default: `Signed`

`Signed`

Specifies the fixed-point data as signed.

`Unsigned`

Specifies the fixed-point data as unsigned.

Selecting `Fixed point` from the **Mode** list enables this parameter.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: `16`

Minimum: `0`

Maximum: `32`

Large word sizes represent large values with greater precision than small word sizes.

Selecting `Fixed point` from the **Mode** list enables this parameter.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Integer

Integer

Specifies a binary point location for fixed-point data and sets the fraction length to 0.

The **Scaling** list has only one item for you to select.

Selecting `Fixed point` from the **Mode** list enables this parameter.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced in R2010a

First-Order Hold

Implement first-order sample-and-hold



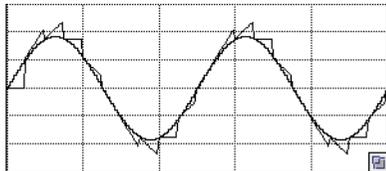
Library

Discrete

Description

The First-Order Hold block implements a first-order sample-and-hold that operates at the specified sampling interval. This block has little value in practical applications and is included primarily for academic purposes.

This figure compares the output from a Sine Wave block and a First-Order Hold block.



Data Type Support

The First-Order Hold block accepts and outputs signals of type `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Sample time

The time interval between samples. See “Specify Sample Time” in the online documentation for more information.

Characteristics

Data Types	Double
Sample Time	Specified in the Sample time parameter
Direct Feedthrough	No
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Zero-Order Hold

Introduced before R2006a

Fixed-Point State-Space

Implement discrete-time state space

$$\left. \begin{array}{l} y(n) = Cx(n) + Du(n) \\ x(n+1) = Ax(n) + Bu(n) \end{array} \right\}$$

Library

Additional Math & Discrete / Additional Discrete

Description

The Fixed-Point State-Space block implements the system described by

$$y(n) = Cx(n) + Du(n)$$

$$x(n+1) = Ax(n) + Bu(n)$$

where u is the input, x is the state, and y is the output. Both equations have the same data type.

The matrices A , B , C and D have the following characteristics:

- A must be an n -by- n matrix, where n is the number of states.
- B must be an n -by- m matrix, where m is the number of inputs.
- C must be an r -by- n matrix, where r is the number of outputs.
- D must be an r -by- m matrix.

In addition:

- The state x must be an n -by-1 vector.
- The input u must be an m -by-1 vector.

- The output y must be an r -by-1 vector.

The block accepts one input and generates one output. The block determines the input vector width by the number of columns in the B and D matrices. Similarly, the block determines the output vector width by the number of rows in the C and D matrices.

Data Type Support

The Fixed-Point State-Space block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

State Matrix A

Specify the matrix of states.

Input Matrix B

Specify the column vector of inputs.

Output Matrix C

Specify the column vector of outputs.

Direct Feedthrough Matrix D

Specify the matrix for direct feedthrough.

Initial condition for state

Specify the initial condition for the state.

Data type for internal calculations

Specify the data type for internal calculations.

Scaling for State Equation $AX+BU$

Specify the scaling for state equations.

Scaling for Output Equation CX+DU

Specify the scaling for output equations.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer). in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Discrete State-Space

Introduced before R2006a

Floating Scope and Scope Viewer

Display signals generated during simulation



Library

Sinks

Description

The Simulink Floating Scope block and Scope Viewer display time domain signals with respect to simulation time.

Input signal characteristics:

- **Signal** — Continuous (sample-based) or discrete (sample-based and frame-based).
- **Signal data type** — Any data type that Simulink supports including real, complex, fixed-point, and enumerated data types. See “Data Types Supported by Simulink”.
- **Signal dimension** — Scalar, one-dimensional (vector), two dimensional (matrix), or multidimensional. Display multiple channels within a signal depending on its dimension. See “Signal Dimensions” and “Determine Output Signal Dimensions”.

Floating Scope block characteristics:

- **Multiple y-axes (displays)** — Display multiple *y*-axes with multiple input ports. All of the *y*-axes have a common time range on the *x*-axis.
- **Multiple signals** — Show multiple signals on the same *y*-axis (display) from one or more input ports.
- **Modify parameters** — Modify scope parameter values before and during a simulation.

- **Display data after simulation** — If a Floating Scope is closed at the start of a simulation, scope data is still written to the scope during a simulation. As a result, if you open the Floating Scope after a simulation, the scope displays simulation results for input signals.

Note A Floating Scope block and Scope Viewer have the same functionality as a Scope block, but they are not connected to signal lines. Use the Signal Selector to add and display signals on a Floating Scope.

Note If you step back the simulation after adding or removing a signal, the Floating Scope clears the existing data. New data does not appear until the simulation steps forward again.

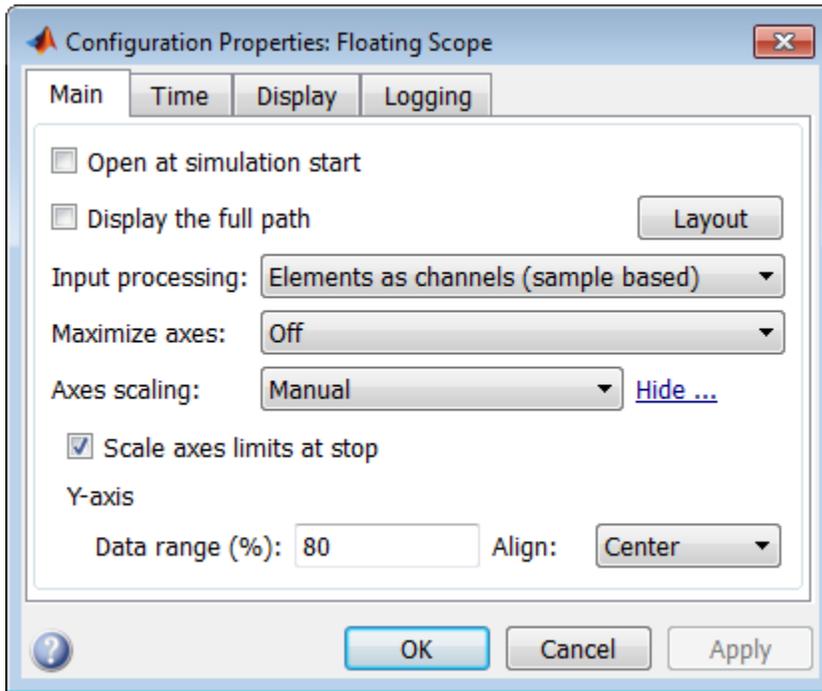
Note Scope Viewer may plot a single point when connected to a constant signal.

Note Simulink messages are not supported for Floating Scope block and Scope Viewer.

Note For information on controlling a Floating Scope programmatically, see “Control Scopes Programmatically” in the Simulink documentation.

When you use model configuration parameters such as **Signal storage reuse** and **Block reduction** for a faster simulation and optimized generated code, Simulink eliminates storage for some signals during simulation. You are unable to apply a Floating Scope to these eliminated signals. To work around this issue, configure an eliminated signal as a test point. You can then apply a Floating Scope to the signal regardless of optimization settings. To configure test points, see “Test Points”.

Configuration Properties



Open at simulation start

Specify when a Scope window opens.

Default: Clear for Scope block. Select for Time Scope block.

- Select
Open Scope window when simulation starts.
- Clear
Do not open a closed Scope at the start of a simulation.

Display the full path

Display full block path on Scope title bar.

Default: Clear

Select

Display block path and name.

Clear

Display block name.

Scope Configuration property: No corresponding property.

Number of input ports

Specify number of input ports on a Scope block, specified by a positive integer character vector. Maximum number of input ports is 96. This property does not apply to floating scopes and scope viewers.

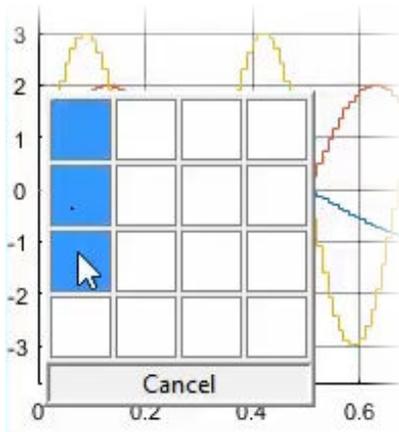
Default: 1

Scope Configuration property: NumInputPorts.

Layout button

Specify number and arrangement of displays. The maximum layout is 16 rows by 16 columns.

To expand the layout grid beyond 4 by 4, click within the dialog box and drag. Maximum of 16 rows by 16 columns.



- If the number of displays are equal to the number of ports, signals from each port appear on separate displays.
- If the number of displays are less than the number of ports, signals from additional ports appear on the last display.
- For layouts with multiple columns and rows, ports are mapped down then across.

Default: 1 display

Scope Configuration property: `LayoutDimensions`.

Sample time

Specify time interval between Scope block updates during a simulation, specified as a positive real character vector. This property does not apply to floating scopes and scope viewers.

Default: -1 for inherited

Scope Configuration property: `SampleTime`.

Input processing

Specify sample-based or frame-based processing of signals.

Default: Elements as channels (sample based) for Scope block. Columns as channels (frame based) for Time Scope block.

Elements as channels (sample based)

Process signal values in a channel at each time interval.

Columns as channels (frame based)

Process signal values in a channel as a group of values from multiple time intervals.
Frame-based processing is available only with discrete input signals.

Scope Configuration property: `FrameBasedProcessing`.

Maximize axes

Maximize size of signal plots. Each of the plots expands to fit the full display. Maximizing the size of signal plots removes the background area around the plots.

Default: `Off` for Scope block. `Auto` for Time Scope block.

`Auto`

If **Title** and **Y-label** properties are not specified, maximize all plots.

`On`

Maximize all plots. Values in **Title** and **Y-label** are hidden

`Off`

Do not maximize plots.

Scope Configuration property: `MaximizeAxes`.

Axes scaling

Specify when to scale *y*-axis to include all signal values.

Default: `Manual`

`Manual`

Manually scale *y*-axis range with **Scale Y-axis Limits** toolbar button.

Auto

Scale *y*-axis range during and after simulation. Selecting this option displays the **Do not allow Y-axis limits to shrink** check box.

If you want the *y*-axis range to increase and decrease with the maximum value of a signal, set **Axes scaling** to `Auto` and clear the **Do not allow Y-axis limits to shrink** check box.

After N Updates

Scale *y*-axis after the number of time steps specified in the Number of updates text box. Scaling occurs once during each run.

Do not allow Y-axis limits to shrink

Specify when *y*-axis range limits can change.

Default: Select



Select

Allow *y*-axis range limits to increase but not decrease during a simulation.



Clear

Allow *y*-axis range limits to increase and decrease.

Click the **Configure** link to the right of the **Axes scaling** property and set the **Axes scaling** property to `Auto` to display this property.

Number of updates

Specify the number of updates that occur during a simulation before a Scope scales the *y*-axes, specified as a positive integer character vector.

Default: 10

Display and activate this property by clicking the **Configure** link to the right of the **Axes scaling** property and set the **Axes scaling** property to `After N Updates`.

Scope Configuration property: `AxesScalingNumUpdates`.

Scale axes limits at stop

Specify when to scale axes.

Default: Select

- Select
Scale axes when simulation stops.
- Clear
Always scale axes.

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

The *y*-axes limits are always scaled. The *x*-axis limits are scaled only if you also select the **Autoscale X-axis limits** check box.

Y-axis Data range (%)

Specify percentage of *y*-axis range for plotting data. For example, if you set this property to 100, plotted data uses the entire *y*-axis range.

Default: 80

Values are 1 through 100.

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

Y-axis Align

Specify where to align plotted data along the *y*-axis data range when **Y-axis Data range** is set to less than 100 percent.

Default: Center

Top

Align signals with maximum values at top of y -axis range.

Center

Align signals with maximum and minimum values centered.

Bottom

Align signals with minimum values at bottom of y -axis range.

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

Autoscale X-axis limits

Scale x -axis range limits when scaling axes.

Default: Clear

Select

Scale x -axis range to fit all signal values. If **Axes scaling** is set to `Auto`, scales the data currently within the axes, not the entire signal in the data buffer.

Clear

Do not scale x -axis range.

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

X-axis Data range (%)

Specify percentage of x -axis range for plotting data. For example, if you set this property to 100, plotted data uses the entire x -axis range.

Default: 100

Values are 1 through 100.

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

X-axis Align

Specify where to align plotted data along the x -axis when **X-axis Data range** is set to less than 100 percent.

Default: Center

Top

Align signals with maximum values at top of x -axis range.

Center

Align signals with maximum and minimum values centered.

Bottom

Align signals with minimum values at bottom of x -axis range.

Click the **Configure** link to the right of the **Axes scaling** property to display this property.

Time span

Specify length of x -axis range to display.

The block calculates the beginning and end times of the time range using the **Time display offset** and **Time span** properties. For example, if you set the **Time display offset** to 10 and the **Time span** to 20, the scope sets the time range from 10 to 30.

Default: Auto

Auto

Difference between the simulation start and stop times.

User defined

Value less than the total simulation time.

Scope Configuration property: `TimeSpan`.

Time span overrun action

Specify how to display data beyond the visible x -axis range.

You can see the effects of this option only when plotting is slow with large models or small step sizes.

Default: `Wrap`

`Wrap`

Draw a full screen of data from left to right, clear the screen, and then restart drawing of data.

`Scroll`

Move data to the left as new data is drawn on the right. This mode is graphically intensive and can affect run-time performance.

Scope Configuration property: `TimeSpanOverrunAction`.

Time units

Display units for the x -axis.

Default: `None` for Scope block. `Metric` for Time Scope block.

`Metric`

Display time units based on the length of **Time span**.

`Seconds`

Display Time (seconds).

`None`

Do not display time units.

Scope Configuration property: `TimeUnits`.

Time display offset

Offset the x -axis by a specified time value, specified as a real number or vector of real numbers.

For input signals with multiple channels, you can enter a scaler or vector.

- Scaler — Offset all channels of an input signal by the same time value.
- Vector — Independently offset the channels.

Default: 0

Scope Configuration property: `TimeDisplayOffset`.

Time-axis labels

Specify how x -axis (time) labels display

Default: `Bottom Displays Only` for Scope block. `All` for Time Scope block.

All

Display x -axis labels on all y -axes.

None

Do not display labels. Selecting `None` also clears the **Show time-axis label** check box.

Bottom Displays Only

Display x -axis label on the bottom y -axis.

Set **Active display** before setting this property. Activate this property by selecting **Show time-axis label** and setting **Maximize axes** to off.

Scope Configuration property: `TimeAxisLabels`.

Show time-axis label

Display or hide *x*-axis (time) labels.

Default: Clear for Scope block. Select for Time Scope block.

- Select
Display *x*-axis label for the active display
- Clear
Hide *x*-axis labels.

Set **Active display** before setting this property. If you select this property and set the **Time-axis labels** is set to `None`, this property is deactivated.

Scope Configuration property: `ShowTimeAxisLabel`.

Active display

Display for setting display-specific properties, specified as a positive integer. The number of a display corresponds to its column-wise placement index.

Default: 1

Setting this property selects the display for setting the properties **Show Grid**, **Show legend**, **Title**, **Plot signals as magnitude and phase**, **Y-label**, and **Y-Limits**.

Scope Configuration property: `ActiveDisplay`.

Title

Specify a title for display, specified as a character vector. The default value `%<SignalLabel>` uses the input signal name for the title.

Default: `%<SignalLabel>`

Set **Active display** before setting this property.

Scope Configuration property: `Title`.

Show legend

Show signal legend. The names listed in the legend are the signal names from the model. For signals with multiple channels, a channel index is appended after the signal name. See the Scope block reference for an example.

The legend lets you modify what signals are shown. To show only one signal, click the signal name. To toggle a signal on/off, right-click the signal name.

Default: Clear

Select

Display signal legend. Continuous signals have straight lines before their names and discrete signals have step-shaped lines.

Clear

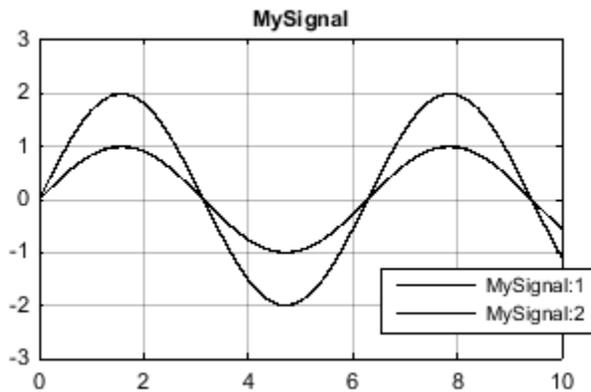
Hide signal legend.

Set **Active Display** before setting this property.

Scope Configuration property: `ShowLegend`.

Connect a Sine Wave block to a Scope. Set the **Amplitude** parameter for the Sine Wave to [1 2]. Select the **Legends** check box for the Scope. Set the **Signal name** property for the signal to `MySignal`.

After simulating this model, the Scope window displays a sine wave for the two signal channels in `MySignal`, `MySignal:1`, and `MySignal:2`.



Edit the name of any channel in the legend by double-clicking the current name and entering a new channel name.

See also “Signal Dimensions” and “Determine Output Signal Dimensions”.

Show grid

Show vertical and horizontal grid lines.

Default: Select

Select
Display grid lines.

Clear
Hide grid lines.

Set **Active Display** before setting this property.

Scope Configuration property: ShowGrid.

Plot signals as magnitude and phase

Split display into magnitude and phase plots.

Default: Clear

Select

Display magnitude and phase plots. If the signal is real, plots the absolute value of the signal for the magnitude. The phase is 0 degrees for positive values and 180 degrees for negative values.

Clear

Display signal plot. If the signal is complex, plots the real and imaginary parts on the same y -axis.

Set **Active Display** before setting this property.

Scope Configuration property: PlotAsMagnitudePhase.

Y-limits (Minimum)

Specify minimum value of y -axis.

Default: -10

Set **Active display** before setting this property. Selecting **Plot signals as magnitude and phase** applies this property value to the magnitude plot. The y -axis limits of the phase plot are always [-180 180].

Scope Configuration property: YLimits.

Y-limits (Maximum)

Specify maximum value of y -axis, specified as real number.

Default: +10

Set **Active display** before setting this property. Selecting **Plot signals as magnitude and phase** applies this property value to the magnitude plot. The y -axis limits of the phase plot are always $[-180 \ 180]$.

Scope Configuration property: `YLimits`.

Y-label

Specify y -axis label, specified as a character vector.

Default: No label for Scope block. `Amplitude` for Time Scope block.

Set **Active display** before setting this property. Selecting **Plot signals as magnitude and phase** hides this property and plots are labeled `Magnitude` and `Phase`.

Scope Configuration property: `YLabel`.

Limit data points to last

Specify to limit buffered data values before plotting and saving signals.

Default: Clear, 5000

Select

Save specified number of data values for each signal. If the signal is frame-based, the number of buffered data values is the specified number of data values multiplied by the frame size.

For simulations with **Stop time** set to `inf`, consider selecting **Limit data points to last**.

In some cases, for example where the sample time is small, selecting this parameter can have the effect of plotting signals for less than the entire time range of a simulation. If a scope plots a portion of your signals, consider increasing the number of data values to save.

Clear

Save and plot all data values. Clearing **Limit data points to last** can cause an out-of-memory error for simulations that generate a large amount of data or for systems without enough available memory.

If this property is selected, also specify the number of data points by entering a positive integer in the text box. This property limits the data values a scope plots and the data values saved in the MATLAB variable specified in **Variable name**. Data values are from the end of a simulation.

Scope Configuration properties: `DataLoggingLimitDataPoints` and `DataLoggingMaxPoints`.

Decimation

Reduce the amount of scope data to display and save.

Default: Clear, 2

Select

Plot and Log (save) scope data every Nth data point, where N is the decimation factor entered in the text box.

Clear

Save all scope data values.

If this property is selected, also specify the decimation factor by entering a positive integer in the text box. The scope buffers every Nth data point, where N is the decimation factor you specify. A value of 1 buffers all data values. This property limits the data values a scope plots and the data values saved in the MATLAB variable specified in **Variable name**.

Log/Unlog Viewed Signals to Workspace

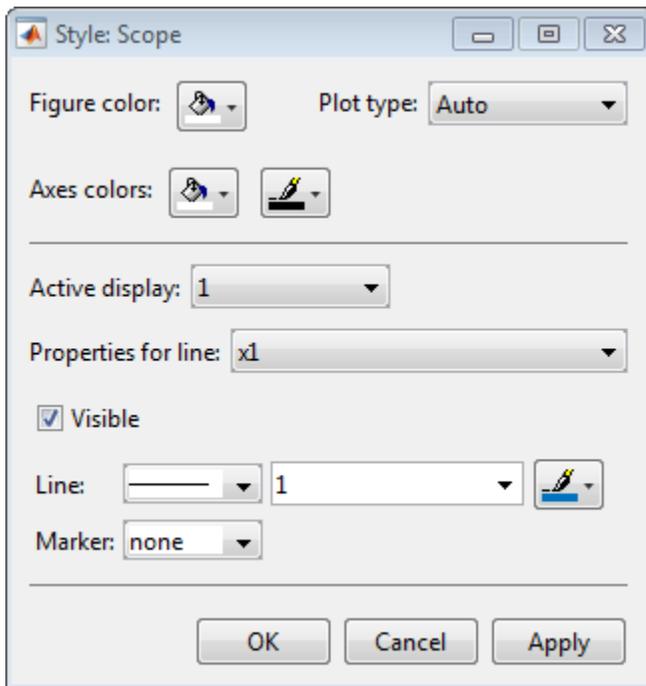
For signals selected with the Signal Selector, clicking this button selects the **Log signal data** check boxes in the Signals Properties dialog boxes.

Clicking the button a second time clears the **Log signal data** check boxes.

Style Properties

Open the Style dialog box:

- From the menu, select **View > Style**.
- From the Configuration Properties button arrow, select the Style button .



Select background color for display.

Specify how to plot a signal.

Default: Auto for Scope block. Line for Time Scope block.

- Line — Line graph.

- `Stairs` — Stair-step graph.
- `Auto` — Line graph if it is a continuous signal or a stair-step graph if it is a discrete signal.

Select active display for setting style properties.

Default: 1

Select the background color for axes (displays) with the first color pallet. Select the grid and label color with the second color pallet.

Select active line for setting line style properties.

Plot signal on active display.

Default: Select

Select
Plot signal.

Clear
Hide signal.

Select line style, width, and color.

Select marker style.

Default: None

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Multidimensional Signals	Yes

Variable-Size Signals	Yes
Code Generation	No

See Also

Scope

Topics

“Scope Blocks and Scope Viewer Overview”

“Simulate a Model Interactively”

“Step Through a Simulation”

“Common Scope Interactions”

“Floating Scope and Scope Viewer Tasks”

“Scope Triggers Panel”

“Cursor Measurements Panel”

“Scope Signal Statistics Panel”

“Scope Bilevel Measurements Panel”

“Peak Finder Measurements Panel”

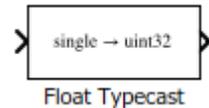
“Control Scopes Programmatically”

Introduced in R2015b

Float Typecast

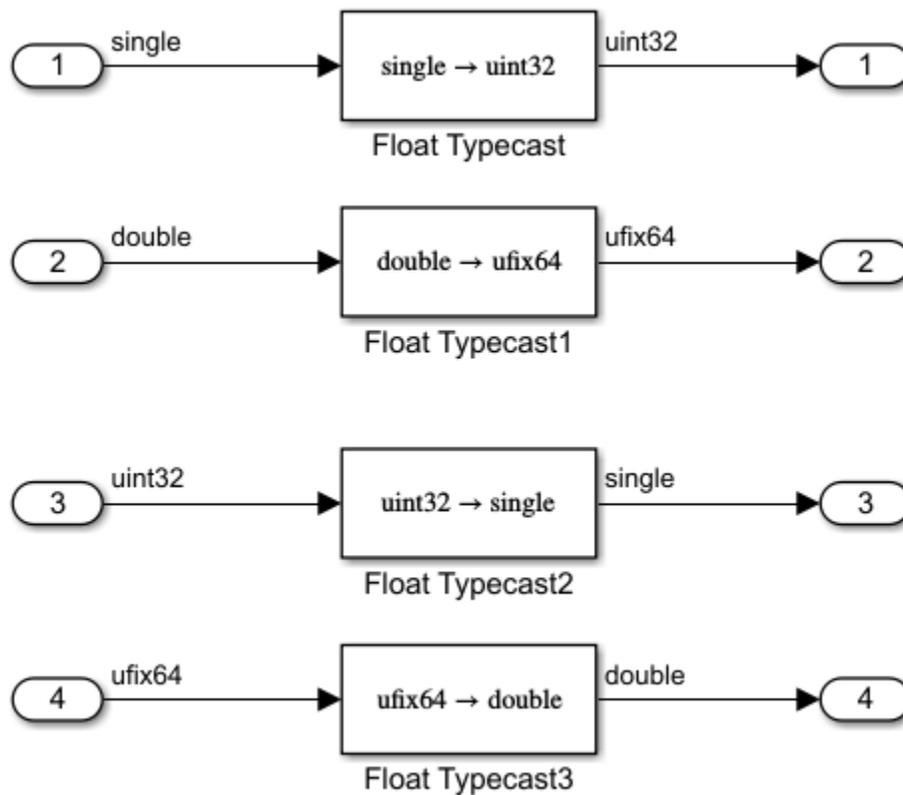
Typecast a floating-point type to an unsigned integer or vice versa

Library: HDL Coder / HDL Floating Point Operations / Float Typecast



Description

The block casts the underlying bits of the input to the corresponding fixed-point or floating point representation. The input and output of the block contain the same number of bits. This figure shows how the block mask, behavior, and output data type changes dynamically depending on the input data type that you specify.



Ports

Input

Port_1(u) — Input signal

`scalar | vector`

Port to provide input to the block.

Data Types: `single` | `double` | `uint32` | `fixed point`

Output

port_1(y) — Output signal

`scalar` | `vector`

Port to obtain calculated output from the block.

Data Types: `single` | `double` | `uint32` | `fixed point`

Model Examples

Block Characteristics

Data Types	<code>double</code> <code>single</code> <code>base integer</code> <code>fixed point</code> <code>bus</code>
Sample Time	<code>Inherit</code>
Direct Feedthrough	<code>Yes</code>
Multidimensional Signals	<code>Scalar</code>
Variable-Size Signals	<code>Yes</code>
Zero-Crossing Detection	<code>No</code>

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™.

If have HDL Coder installed, you can generate HDL code for the block in the `Native Floating Point` mode. For more information, see `Float Typecast`.

Fixed-Point Conversion

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

See Also

Functions

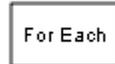
`typecast`

Introduced in R2017b

For Each

Process elements or subarrays of a mask parameter or input signal independently

Library: Ports & Subsystems



Description

The For Each block serves as a control block for the For Each Subsystem block. Specifically, the For Each block enables the blocks inside the For Each Subsystem to process the elements of input signals or mask parameters independently. Each block inside this subsystem that has states maintains a separate set of states for each element or subarray that it processes. As the set of blocks in the subsystem processes the elements or subarrays, the subsystem concatenates the results to form output signals.

You can use a For Each subsystem to iteratively compute output after changing inputs or mask parameters. To do so, you configure the partitioning of input signals or mask parameters in the For Each block dialog box.

Partition Input Signals to the Subsystem

In a For Each subsystem, you can specify which input signals to partition for each iteration using the **Input Partition** tab in the dialog box of the For Each block. When specifying a signal to be partitioned, you also have to specify the **Partition Dimension**, **Partition Width**, and **Partition Offset** parameters.

Partition Parameters in the For Each block

You can partition the mask parameters of a For Each Subsystem block. Partitioning is useful for systems that have identical structures in each iteration but different parameter values. In this case, changing the model to partition extra input signals for each parameter is cumbersome. Instead, add a mask parameter to a For Each subsystem. For more information, see “Create a Simple Mask”. To select the mask parameter for

partitioning, use the Parameter Partition tab on the For Each block dialog box. For more information, see “Select Partition Parameters” on page 1-597

Concatenate Output

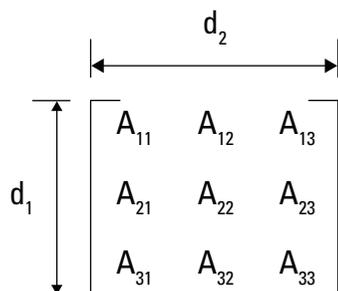
You define the dimension along which to concatenate the results by specifying the **Concatenation Dimension** in the Output Concatenation tab.

The results generated by the block for each subarray stack along the concatenation dimension, d_1 (y -axis). Whereas, if you specify d_2 by setting the concatenation dimension to 2, the results concatenate along the d_2 direction (x -axis). Thus if the process generates row vectors, then the concatenated result is a row vector.

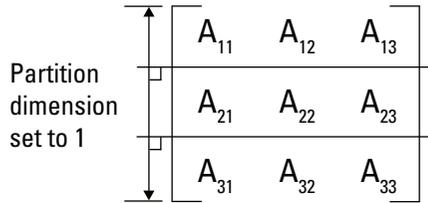
Select Partition Parameters

When selecting an input signal or subsystem mask parameter for partitioning, you need to specify how to decompose it into elements or subarrays for each iteration. Do this by setting integer values for the **Partition Dimension**, **Partition Width**, and **Partition Offset** parameters.

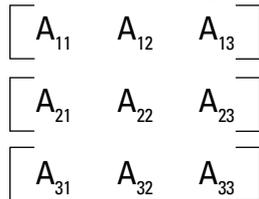
As an illustration, consider an input signal matrix A of the form:



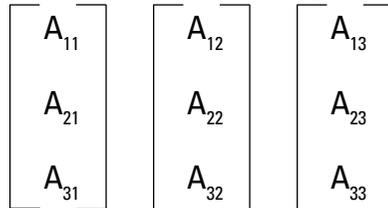
The labels d_1 and d_2 , respectively, define dimensions 1 and 2. If you retain the default setting of 1 for both the partition dimension and the partition width, and 0 for the partition offset, then Simulink slices perpendicular to partition dimension d_1 at a width equal to the partition width, that is one element.



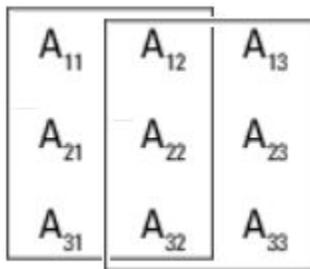
Matrix A decomposes into these three row vectors.



If instead you specify d_2 as the partition dimension by entering the value 2, Simulink slices perpendicular to d_2 to form three column vectors.



In addition to setting the **Partition Dimension** to 2, if you set the **Partition Width** to 2 and the **Partition Offset** to -1, Simulink uses two overlapping 3x2 partitions for processing.

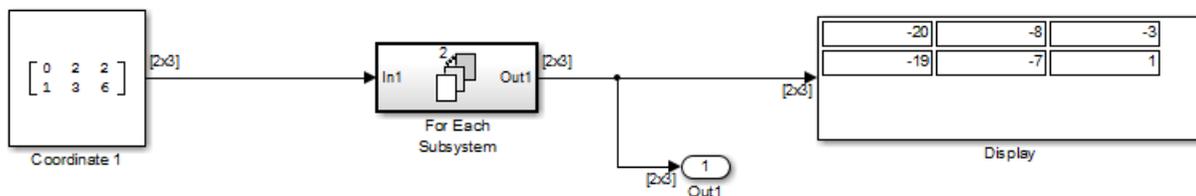


For an example using the **Partition Offset** parameter, open the Simulink model `.slexForEachOverlapExample`

Note Only signals are considered one-dimensional in Simulink. Mask parameters are row or column vectors, according to their orientation. To partition a row vector, specify the partition dimension as 2 (along the columns). To partition a column vector, specify the partition dimension as 1 (along the rows).

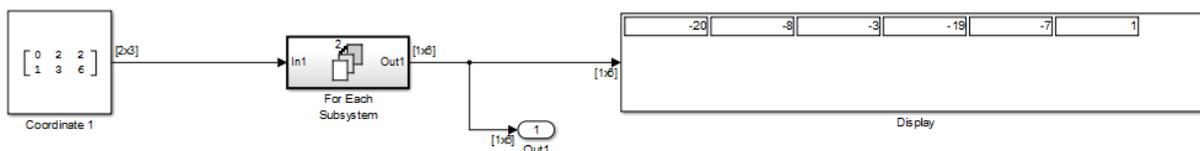
Examples

The following model demonstrates the partitioning of an input signal by a For Each block. Each row of this 2-by-3 input array contains three integers that represent the (x, y, z) -coordinates of a point. The goal is to translate each of these points based on a new origin at $(-20, -10, -5)$ and to display the results.



By placing the process of summing an input signal and the new origin inside of a For Each Subsystem block, you can operate on each set of coordinates by partitioning the input signal into two row vectors. To accomplish such partitioning, use the default settings of 1 for both the partition dimension and the partition width. If you also use the default concatenation dimension of 1, each new set of coordinates stacks in the d_1 direction, making your display a 2-by-3 array.

Alternatively, if you specify a concatenation dimension of 2, then you get a single row vector because each set of results stacks in the d_2 direction.



This example shows how to partition an input signal. To learn how the For Each block and subsystem handle a model with states, see the For Each Subsystem documentation.

Ports

Input

In — Signal input to a subsystem block

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

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

Output

Out — Signal output from a subsystem

scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

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

Parameters

Input Partition Tab

Select each input signal you want to partition and to specify the corresponding **Partition Dimension** and **Partition Width** parameters. See the Inport block reference page for more information.

Port — List of input ports

no default (default) | input port name

List of input ports connected to the For Each Subsystem block.

Partition — Select input port signals to partition

off (default) | on

Select input ports signals connected to the For Each Subsystem block to partition into subarrays or elements.

off

Clear input port signals.

on

Select input port signals to partition.

Dependency

Selecting this parameter enables the **Partition Dimension** and **Partition Width** parameters for the selected input port signal.

Programmatic Use

Parameter: InputPartition

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Partition Dimension — Specify dimension

1 (default) | integer

Specify the dimension through which to slice the input signal array. The resulting slices are perpendicular to the dimension that you specify. The slices also partition the array into subarrays or elements, as appropriate.

1

Specify a dimension of 1.

integer

Specify dimension. Minimum value 1.

Programmatic Use

Parameter: InputPartitionDimension

Type: character vector

Value: '1' | '<integer>'

Default: '1'

Partition Width — Specify width

1 (default) | integer

Specify the width of each partition slice of the input signal.

1

Width of one element.

integer

Specify width. Minimum value 1.

Programmatic Use

Parameter: InputPartitionWidth

Type: character vector

Value: '1' | '<integer>'

Default: '1'

Partition Offset — Specify partition offset

0 (default) | integer

Specify the offset for each partition slice of the input signal.

0

No offset between partition slices.

integer

Specify partition offset where the sum of the partition width and the partition offset is a positive integer.

For example, a **Partition Width** of 3 and a **Partition Offset** of -2 indicates that each 3 element slice overlaps its neighboring slices by 2 elements.

Programmatic Use

Parameter: SubsysMaskParameterOffset

Type: character vector

Value: '0' | '<integer>'

Default: '0'

Output Concatenation Tab

For each output port, specify the dimension along which to stack (concatenate) the For Each Subsystem block results. See the Outport block reference page for more information.

Port — List of output ports

none (default) | output port name

List of output ports connected to the For Each Subsystem block.

Concatenation Dimension — Specify dimension

1 (default) | integer

Specify the dimension along which to stack the results of the For Each Subsystem block.

1

The results stack in the d_1 direction. If the block generates column vectors, the concatenation process results in a single column vector.

integer

The results stack in the d_2 direction. If the block generates row vectors, the concatenation process results in a single row vector. Minimum value 1

Programmatic Use

Parameter: OutputConcatenationDimension

Type: character vector

Value: '1' | '<integer>'

Default: '1'

Parameter Partition Tab

Select each mask parameter to partition and to specify the corresponding **Partition Dimension** and **Partition Width** parameters. Parameters appear in the list only if you have added an editable parameter to the mask of the parent For Each subsystem.

Parameter — List of mask parameters

parameter name

List of mask parameters for the For Each Subsystem block.

Partition — Select mask parameters to partition

off (default) | on

Select mask parameters for the For Each Subsystem block to partition into subarrays or elements.

off

Clear mask parameters.

on

Select mask parameters to partition.

Dependency

Selecting this parameter enables the **Partition Dimension** and **Partition Width** parameters for the selected mask parameter.

Programmatic Use

Parameter: SubsysMaskParameterPartition

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Partition Dimension — Specify dimension

1 (default) | integer

Specify the dimension through which to slice the input signal array. The resulting slices are perpendicular to the dimension that you specify. The slices also partition the array into subarrays or elements, as appropriate.

Default: 1

1

Dimension of 1.

integer

Specify dimension. Minimum value 1

Programmatic Use

Parameter: SubsysMaskParameterPartitionDimension

Type: character vector

Value: '1' | '<integer>'

Default: '1'

Partition width — Specify partition width

1 (default) | integer

Specify the width of each partition slice of the input signal.

1

Width of one element.

integer

Specify width. Minimum value 1

Programmatic Use

Parameter: SubsysMaskParameterPartitionWidth

Type: character vector

Value: '1' | '<integer>'

Default: '1'

See Also

Blocks

For Each Subsystem | Subsystem

Topics

“Repeat an Algorithm Using a For Each Subsystem”

“Log Signals in For Each Subsystems”

Introduced in R2010a

For Each Subsystem

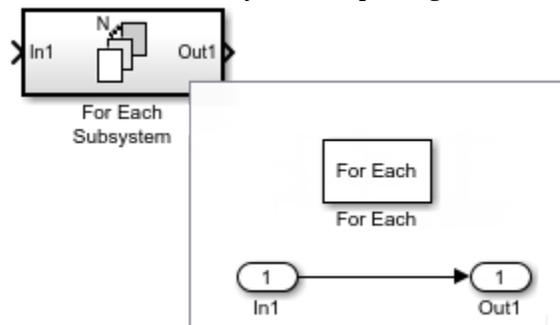
Subsystem that repeats execution on each element or subarray of input signal and concatenates results

Library: Ports & Subsystems



Description

The For Each Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that repeats execution during a simulation time step on each element or subarray of an input signal.



The set of blocks within the subsystem represents the algorithm applied to a single element or subarray of the original signal. The For Each block inside the subsystem allows you to configure the decomposition of the subsystem inputs into elements or subarrays, and to configure the concatenation of the individual results into output signals.

Inside this subsystem, each block that has states maintains separate sets of states for each element or subarray that it processes. Consequently, the operation of this subsystem is similar in behavior to copying the contents of the subsystem for each

element in the original input signal and then processing each element using its respective copy of the subsystem.

An additional benefit of the For Each Subsystem block is that, for certain models, it improves the code reuse in SimulinkCoder generated code. Consider a model containing two reusable Atomic Subsystem blocks with the same scalar algorithm applied to each element of the signal. If the input signal dimensions of these subsystems are different, Simulink Coder generated code includes two distinct functions. You can replace these two subsystems with two identical For Each Subsystem blocks that are configured to process each element of their respective inputs using the same algorithm. For this case, Simulink Coder generated code consists of a single function parameterized by the number of input signal elements. This function is invoked twice — once for each unique instance of the For Each Subsystem block in the model. For each of these cases, the input signal elements have different values.

S-Function Support

The For Each Subsystem block supports both C-MEX S-functions and Level-2 MATLAB S-functions, provided that the S-function supports multiple execution instances using one of the following techniques:

- A C-MEX S-function must declare `ssSupportsMultipleExecInstances(S, true)` in the `mdlSetWorkWidths` method.
- A Level-2 MATLAB S-function must declare `block.SupportsMultipleExecInstances = true` in the setup method.

If you use the above specifications:

- Do not cache run-time data, such as DWork and Block I/O, using global or persistent variables or within the userdata of the S-function.
- Every S-function execution method from `mdlStart` up to `mdlTerminate` is called once for each element processed by the S-function, when it is in a For Each Subsystem block. Consequently, you need to be careful not to free the same memory on repeated calls to `mdlTerminate`. For example, consider a C-MEX S-function that allocates memory for a run-time parameter within `mdlSetWorkWidths`. The memory only needs to be freed once in `mdlTerminate`. As a solution, set the pointer to be empty after the first call to `mdlTerminate`.

Limitations

The For Each Subsystem block has these limitations, and these are the workarounds.

Limitation	Workaround
<p>You cannot log bus or an array of bus signals directly in the For Each subsystem.</p>	<p>Use one of these approaches:</p> <ul style="list-style-type: none"> • Use a Bus Selector block to select the signals you want to log and mark those signals for signal logging. • Attach the signal to an Outport block and log the signal outside the For Each subsystem.
<p>You cannot log a signal inside a referenced model that is inside a For Each subsystem if either of these conditions exists:</p> <ul style="list-style-type: none"> • The For Each subsystem is in a model simulating in Rapid Accelerator mode. • The For Each subsystem itself is in a model referenced by a Model block in Accelerator mode. 	<p>For the first condition, use Accelerator mode.</p> <p>For the second condition, use Normal or Rapid Accelerator mode.</p>
<p>You cannot log the states of the blocks in a For Each subsystem .</p>	<p>Save and restore the simulation state.</p>
<p>You cannot use Normal mode to simulate a Model block inside a For Each subsystem.</p>	<p>Use Accelerator or Rapid Accelerator mode.</p>
<p>Reusable code is generated for two For Each Subsystems with identical contents if their input and output signals are vectors (1-D or 2-D row or column vector). For n-D input and output signals, reusable code is generated only when the dimension along which the signal is partitioned is the highest dimension.</p>	<p>Permute the signal dimensions to transform the partition dimension and the concatenation dimension to the highest nonsingleton dimension for n-D signals.</p>

The For Each Subsystem block does not support these features:

- You cannot include these blocks or S-functions inside a For Each Subsystem:

- Data Store Memory, Data Store Read, or Data Store Write blocks inside the subsystem
- The From Workspace block if the input is a `Structure` with `Time` that has an empty time field
- The To Workspace and To File data saving blocks
- Goto and From blocks that cross the subsystem boundary
- Model Reference block with simulation mode set to Normal
- Shadow Inports
- ERT S-functions

For a complete list of the blocks that support the For Each Subsystem, type `showblockdatatypetable` at the MATLAB command line.

- You cannot use these types of signals:
 - Test-pointed signals or signals with an external storage class inside the system
 - Frame signals on subsystem input and output boundaries
 - Variable-size signals
 - Function-call signals crossing the boundaries of the subsystem
- Creation of a linearization point inside the subsystem
- Propagating the Jacobian flag for the blocks inside the subsystem. You can check this condition in MATLAB using `J.Mi.BlockAnalyticFlags.jacobian`, where `J` is the Jacobian object. To verify the correctness of the Jacobian of the For Each Subsystem block, perform these steps:
 - Look at the tag of the For Each Subsystem Jacobian. If it is `"not_supported"`, then the Jacobian is incorrect.
 - Move each block out of the For Each Subsystem and calculate its Jacobian. If any block is `"not_supported"` or has a warning tag, the For Each Subsystem Jacobian is incorrect.
- You cannot perform these types of code generation:
 - Generation of a Simulink Coder S-function target
 - Simulink Coder code generation under both of the following conditions:
 - A Stateflow or MATLAB Function block resides in the subsystem.

- This block tries to access global data outside the subsystem, such as Data Store Memory blocks or Simulink.Signal objects of ExportedGlobal storage class.
- PLC code generation

Ports

Input

in — Signal input to a subsystem block

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

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

Output

out — Signal output from a subsystem

scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

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

See Also

Blocks

For Each | Subsystem

Topics

“Repeat an Algorithm Using a For Each Subsystem”

“Log Signals in For Each Subsystems”

Introduced in R2010a

For Iterator

Repeatedly execute contents of subsystem at current time step until iteration variable exceeds specified iteration limit



Library

Ports & Subsystems

Description

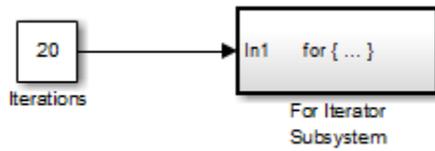
The For Iterator block, when placed in a subsystem, repeatedly executes the contents of the subsystem at the current time step until an iteration variable exceeds a specified iteration limit. You can use this block to implement the block diagram equivalent of a `for` loop in the C programming language.

The output of a For Iterator subsystem cannot be a function-call signal. Simulink software will display an error message if the simulation is run or the diagram updated.

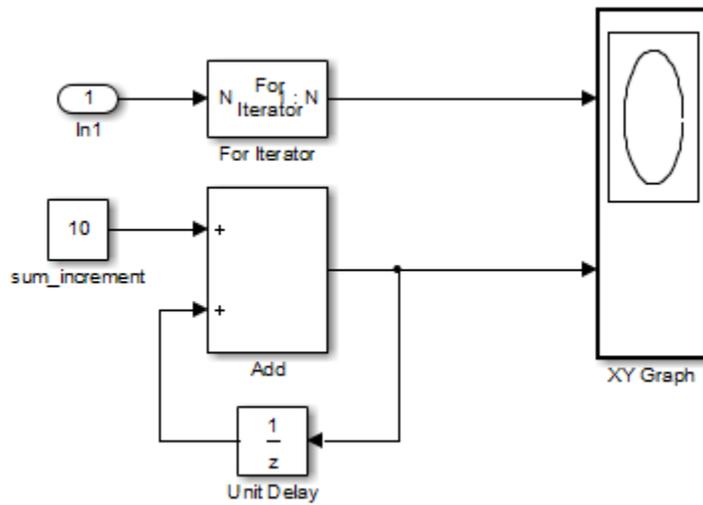
The block's parameter dialog allows you to specify the maximum value of the iteration variable or an external source for the maximum value and an optional external source for the next value of the iteration variable. If you do not specify an external source for the next value of the iteration variable, the next value is determined by incrementing the current value:

$$i_{n+1} = i_n + 1$$

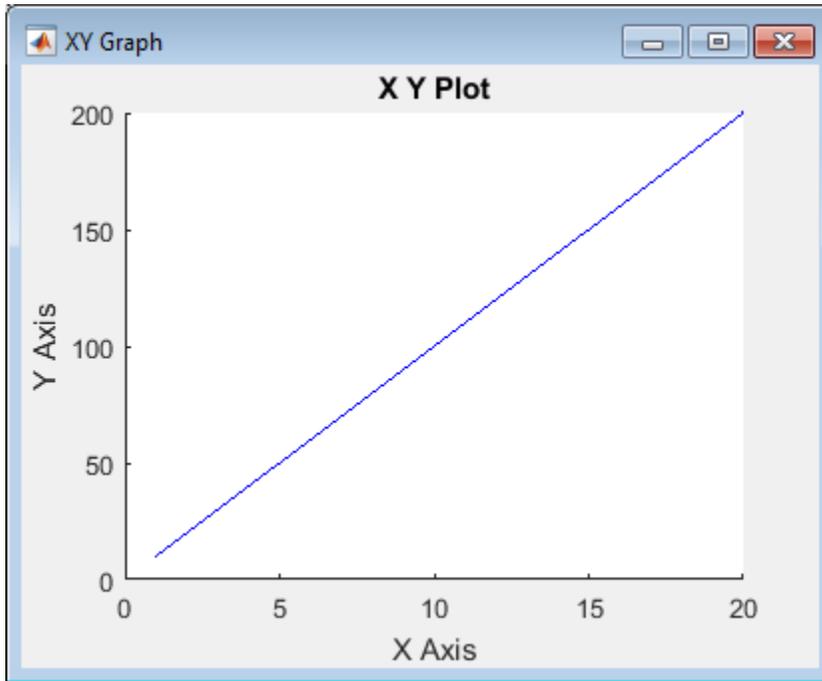
Suppose that you have the following model:



For 20 iterations, the For Iterator block increments a value by 10 at each time step:



The following figure shows the result.



The For Iterator subsystem in this example is equivalent to the following C code.

```
sum = 0;
iterations = 20;
sum_increment = 10;
for (i = 0; i < iterations; i++) {
    sum = sum + sum_increment;
}
```

Note Placing a For Iterator block in a subsystem makes it an atomic subsystem if it is not already an atomic subsystem.

Data Type Support

The following rules apply to the data type of the number of iterations (N) input port:

- The input port accepts data of mixed numeric types.
- If the input port value is noninteger, it is first truncated to an integer.
- Internally, the input value is cast to an integer of the type specified for the iteration variable output port.
- If no output port is specified, the input port value is cast to type `int32`.
- If the input port value exceeds the maximum value of the output port's type, it is truncated to that maximum value.

Data output for the iterator value can be selected as `double`, `int32`, `int16`, or `int8` in the block parameters dialog box.

The following rules apply to the iteration variable input port:

- It can appear only if the iteration variable output port is enabled.
- The data type of the iteration variable input port is the same as the data type of the iteration variable output port.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

States when starting

Set this field to `reset` if you want the states of the For subsystem to be reinitialized before the first iteration at each time step. Otherwise, set this field to `held` (the default) to make sure that these subsystem states retain their values from the last iteration at the previous time step.

Iteration limit source

If you set this field to `internal`, the value of the **Iteration limit** field determines the number of iterations. If you set this field to `external`, the signal at the For Iterator block's `N` port determines the number of iterations. The iteration limit source must reside outside the For Iterator subsystem.

Iteration limit

Set the number of iterations by specifying a number or a named constant. This field appears only if you selected `internal` for the **Iteration limit source** field. This parameter supports storage classes. You can define the named constant in the base

workspace of the Model Explorer as a `Simulink.Parameter` object of the built-in storage class `Define (custom)` type. For more information, see “Apply a Custom Storage Class from the Simulink Package Using Data Objects” (Embedded Coder) in the Embedded Coder documentation.

Set next i (iteration variable) externally

This option can be selected only if you select the **Show iteration variable** option. If you select this option, the For Iterator block displays an additional input for connecting an external iteration variable source. The value of the input at the current iteration is used as the value of the iteration variable at the next iteration.

Show iteration variable

If you select this check box, the For Iterator block outputs its iteration value.

Index mode

If you set this field to `Zero-based`, the iteration number starts at zero. If you set this field to `One-based`, the iteration number starts at one.

Iteration variable data type

Set the type for the iteration value output from the iteration number port to `double`, `int32`, `int16`, or `int8`.

Characteristics

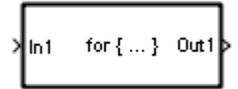
Direct Feedthrough	No
Sample Time	Inherited from driving blocks
Scalar Expansion	No
Dimensionalized	No
Zero Crossing	No

Introduced before R2006a

For Iterator Subsystem

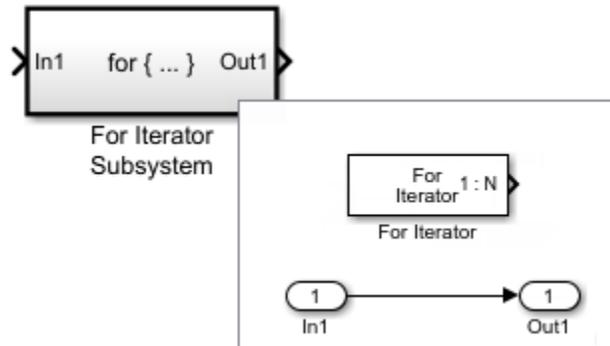
Subsystem that repeats execution during a simulation time step

Library: Ports & Subsystems



Description

The For Iterator Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that repeats execution during a simulation time step while a logical (Boolean) condition is true.



When using simplified initialization mode, if you place a block that needs elapsed time (such as a Discrete-Time Integrator block) in a While Iterator Subsystem block, Simulink displays an error.

Ports

Input

In — Signal input to a subsystem block

scalar | vector | matrix

Placing an Inport block in a Subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

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

Output

Out — Signal output from a subsystem

scalar | vector | matrix

Placing an Outport block in a Subsystem block adds an output port from the block. The port label on the Subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

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

See Also

Blocks

For Iterator | Subsystem | While Iterator | While Iterator Subsystem

Topics

“Use Control Flow Logic”

Introduced before R2006a

From

Accept input from Goto block



Library

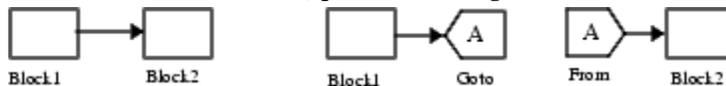
Signal Routing

Description

The From block accepts a signal from a corresponding Goto block, then passes it as output. The data type of the output is the same as that of the input from the Goto block. From and Goto blocks allow you to pass a signal from one block to another without actually connecting them. To associate a Goto block with a From block, enter the Goto block's tag in the **Goto Tag** parameter.

A From block can receive its signal from only one Goto block, although a Goto block can pass its signal to more than one From block.

This figure shows that using a Goto block and a From block is equivalent to connecting the blocks to which those blocks are connected. In the model at the left, Block1 passes a signal to Block2. That model is equivalent to the model at the right, which connects Block1 to the Goto block, passes that signal to the From block, then on to Block2.



The visibility of a Goto block tag determines the From blocks that can receive its signal. For more information, see Goto and Goto Tag Visibility. The block indicates the visibility of the Goto block tag:

- A local tag name is enclosed in brackets ([]).
- A scoped tag name is enclosed in braces ({}).
- A global tag name appears without additional characters.

The From block supports signal label propagation.

Data Type Support

The From block outputs real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see “Data Types Supported by Simulink”.

Parameters

Goto Tag

The tag of the Goto block that forwards its signal to this From block.

To change the tag, select a new tag from this control's drop-down list. The drop-down list displays the Goto tags that the From block can currently see. An item labeled `<More Tags . . . >` appears at the end of the list the first time you display the list in a Simulink session. Selecting this item causes the block to update the tags list to include the tags of Goto blocks residing in library subsystems referenced by the model containing this From block. Simulink software displays a progress bar while building the list of library tags. Simulink software saves the updated tags list for the duration of the Simulink session or until the next time you select the adjacent **Update Tags** button. You need to update the tags list again in the current session only if the libraries referenced by the model have changed since the last time you updated the list.

If you use multiple From and Goto Tag Visibility blocks to refer to the same Goto tag, you can simultaneously rename the tag in all of the blocks. Use the **Rename All** button in the Goto block dialog box. To find the Goto block, use the **Goto Source** hyperlink.

Update Tags

Updates the list of tags visible to this From block, including tags residing in libraries referenced by the model containing this From block.

Goto Source

Path of the Goto block connected to this From block. Clicking the path displays and highlights the Goto block.

Icon Display

Specifies the text to display on the From block's icon. The options are the block's tag, the name of the signal that the block represents, or both the tag and the signal name.

Examples

The following models show how to use the From block:

- `sldemo_auto_climatecontrol`
- `sldemo_hardstop`

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from the block driving the Goto block
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Goto

Introduced before R2006a

From File

Load data from MAT-file



Library

Sources

Description

The From File block loads data from a MAT-file to a model and outputs the data as a signal. The data is a sequence of samples. Each sample consists of a time stamp and an associated data value. The data can be in array format or MATLAB `timeseries` format.

The From File block icon shows the name of the MAT-file that supplies the data to the block.

You can have multiple From File blocks that load from the same MAT-file.

The supported MAT-file versions are Version 7.0 or earlier and Version 7.3. The From File block incrementally loads data from Version 7.3 files.

You can specify how the data is loaded, including:

- Sample time
- How to handle data for missing data points
- Whether to use zero-crossing detection

For more information, see “Load Data Using the From File Block”.

Code Generation Requirements

For a From File block, generating code that builds ERT or GRT targets or uses SIL or PIL simulation modes requires that:

- The MAT-file contains a nonempty, finite, real matrix with at least two rows.
 - Use a data type of `double` for the matrix.
 - Do not include any NaN, Inf, or -Inf elements in the matrix.
- In the From File block parameters dialog box:
 - Set the **Data extrapolation before first data point** and **Data extrapolation after last data point** parameters to `Linear extrapolation`.
 - Set the **Data interpolation within time range** parameter to `Linear interpolation`.
 - Clear the **Enable zero-crossing detection** parameter.

Data Type Support

For MATLAB `timeseries` data, the From File block loads real or complex signal data of any data type that Simulink supports. Fixed-point data cannot have a word length that exceeds 32 bits. The From File block supports loading nonvirtual bus signals in `timeseries` format.

For array data, the From File block loads only double signal data.

For more information, see “Create Data for a From File Block”.

Parameters

File name

The path or file name of the MAT-file that contains the input data. Specify a path or file name in one of these ways:

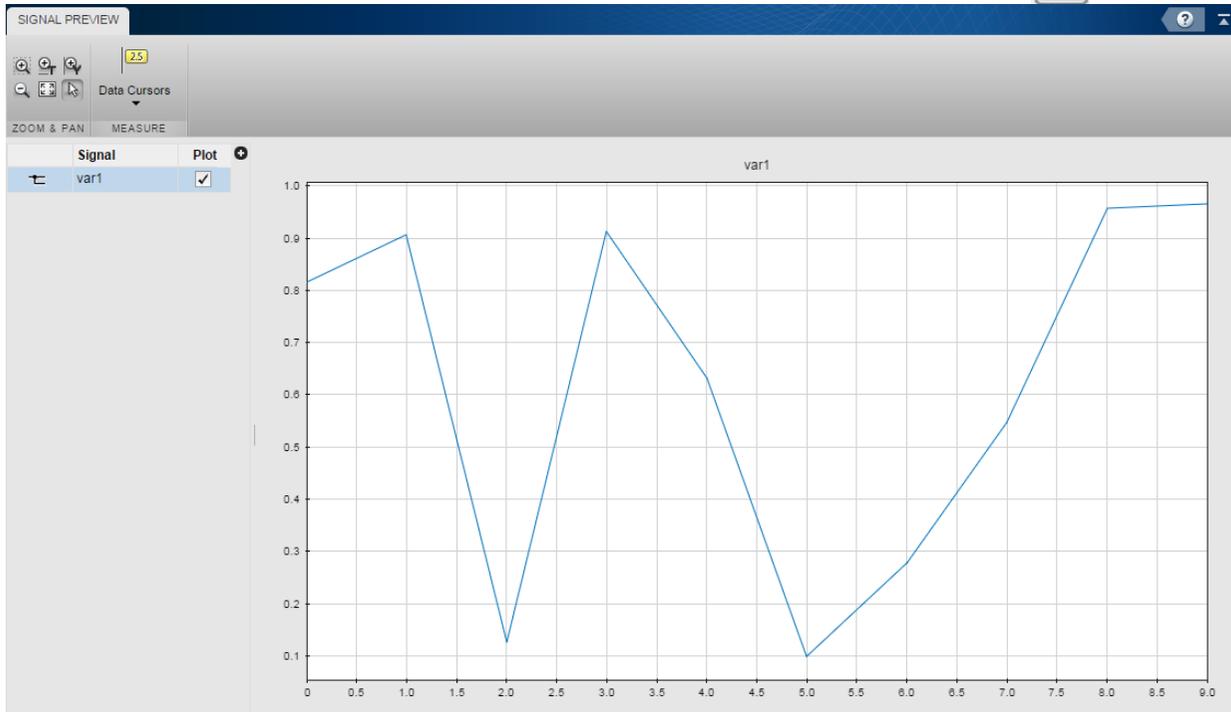
- Browse to a folder that contains a valid MAT-file.

On UNIX® systems, the path name can start with a tilde (~) character, which means your home folder.

- Enter the path for the file in the text box.

The default file name is `untitled.mat`. If you specify a file name without path information, Simulink loads the file in the current folder or on the MATLAB path. (To determine the current folder, at the MATLAB command prompt enter `pwd`.)

To preview the data in the Signal Preview window, click the view button .



Use the Signal Preview window to plot and inspect data.

- 1 To plot the signal, select the check box next to the signal. If the format is a bus, click the expander  to see and select the elements of the bus.
- 2 Explore the plots by using the **Measure** and **Zoom & Pan** sections of the toolbar.
 - In the **Measure** section, use the **Data Cursors** button to display one or two cursors for the plot. These cursors display the T and Y values of a data point in the plot. To view a data point, click a point on the line.
 - In the **Zoom & Pan** section, select how you want to zoom and pan the signal plots. Zooming is only for the selected axis.

Zoom Action	Button
Zoom in along the time and data value axes.	
Zoom in along the time axis. Click the button. On the graph, drag the cursor to select an area to enlarge.	
Zoom in along the data value axis. Click the button. On the graph, drag the cursor to select an area to enlarge.	
Zoom out from the graph.	
Fit the plot to the graph. Click the button. Click the graph to enlarge the plot to fill the graph.	
Pan the graph up, down, left, or right. Click the button. On the graph, drag the cursor to the area of the graph that you want to view.	

Parameter: FileName

Type: character vector

Default: 'untitled.mat'

Output data type

The data type for the data that the From File block outputs. For nonbus types, you can use `Inherit: auto` to skip any data type verification. If you specify an output data type, then the From File block verifies that the data in the file matches the specified data type. For more information, see “Control Signal Data Types”.

- `Inherit: auto` — Default
- `double`
- `single`
- `int8`
- `uint8`

- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`
- `fixdt(1,16,0)` — Data type is fixed point (1,16,0).
- `fixdt(1,16,2^0,0)` — Data type is fixed point (1,16,2^0,0).
- Enum: `<class_name>` — Data type is enumerated.
- Bus: `<bus_object>` — Data type is a bus object. For details, see the “Using Bus Data” section.
- `<data type expression>` — The name of a data type object, for example `Simulink.NumericType`. Do not specify a bus object as the expression.

If you set **Output data type** as a bus object, the bus object must be available when you compile the model. For each signal in bus data, the From File block verifies that data type, dimension, and complexity are the same for the data and for the bus object.

>> (Show data type assistant)

Displays the **Data Type Assistant**, to help you to set the **Output data type** parameter.

Mode

The category of data to specify. For more information, see “Control Signal Data Types”.

- `Inherit` — Inheritance rule for data types. Selecting `Inherit` enables a second menu and text box. (Default)
- `Built in` — Built-in data types. Selecting `Built in` enables a second menu and text box. Select one of the following choices:
 - `double` — Default
 - `single`
 - `int8`
 - `uint8`

- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`
- `Fixed point` — Fixed-point data types
- `Enumerated` — Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.
- `Bus` — Bus object. Enables a **Bus object** parameter, where you enter bus object name for defining the structure of the bus. If you create or change a bus object, open the Simulink Bus Editor by clicking **Edit**. For details, see “Create Bus Objects with the Bus Editor”.
- `Expression` — Expression that evaluates to a data type. Selecting `Expression` enables a second menu and text box to the right, where you enter the expression. Do not specify a bus object as the expression.

Sample time

The sample period and offset.

The From File block loads data from a MAT-file, using a sample time that either:

- You specify for the From File block.
- The From File block inherits from the blocks into which the From File block feeds data.

The default sample time is 0, which specifies a continuous sample time. The MAT-file is loaded at the base (fastest) rate of the model. For details, see “Specify Sample Time”.

Parameter: `SampleTime`

Type: character vector

Default: `'0'`

Data extrapolation before first data point

Extrapolation method for a simulation time hit that occurs before the initial time stamp in the MAT-file. Choose one of the following extrapolation methods.

Method	Description
<p>Linear extrapolation</p>	<p>(Default)</p> <p>If the MAT-file contains only one sample, then the From File block outputs the corresponding data value.</p> <p>If the MAT-file contains more than one sample, then the From File block linearly extrapolates using the first two samples:</p> <ul style="list-style-type: none"> • For double data, linearly extrapolates the value using the first two samples • For Boolean data, outputs the first data value • For a built-in data type other than double or Boolean, the From File block: <ul style="list-style-type: none"> • Upcasts the data to double • Performs linear extrapolation (as described for double data) • Downcasts the extrapolated data value to the original data type <p>You cannot use the Linear extrapolation option with enumerated (enum) data. All signals in a bus use the same extrapolation setting. If any signal in a bus uses enum data, then you cannot use the Linear extrapolation option.</p>
<p>Hold first value</p>	<p>Uses the first data value in the file</p>
<p>Ground value</p>	<p>Uses a value that depends on the data type of MAT-file sample data values:</p> <ul style="list-style-type: none"> • Fixed-point data types — uses the ground value • Numeric types other than fixed point — uses 0 • Boolean — uses false • Enumerated data types — uses default value

Parameter: ExtrapolationBeforeFirstDataPoint

Type: character vector

Value: 'Linear extrapolation' | 'Hold first value' | 'Ground value'
Default: 'Linear extrapolation'

Data interpolation within time range

The interpolation method that Simulink uses for a simulation time hit between two time stamps in the MAT-file. Choose one of the following interpolation methods.

Method	Description
Linear interpolation	<p>(Default)</p> <p>The From File block interpolates using the two corresponding MAT-file samples:</p> <ul style="list-style-type: none"> • For <code>double</code> data, linearly interpolates the value using the two corresponding samples • For <code>Boolean</code> data, uses <code>false</code> for the first half of the sample and <code>true</code> for the second half. • For a built-in data type other than <code>double</code> or <code>Boolean</code>, the From File block: <ul style="list-style-type: none"> • Upcasts the data to <code>double</code> • Performs linear interpolation, as described for <code>double</code> data • Downcasts the interpolated value to the original data type <p>You cannot use the <code>Linear interpolation</code> option with enumerated (<code>enum</code>) data. All signals in a bus use the same interpolation setting. If any signal in a bus uses <code>enum</code> data, then you cannot use the <code>Linear interpolation</code> option.</p>
Zero order hold	Uses the data from the first of the two samples

Parameter: `InterpolationWithinTimeRange`

Type: character vector

'Linear interpolation' | 'Zero order hold'

Default: 'Linear interpolation'

Data extrapolation after last data point

The extrapolation method for a simulation time hit that occurs after the last time stamp in the MAT-file. Choose one of the following extrapolation methods.

Method	Description
Linear extrapolation	<p>(Default)</p> <p>If the MAT-file contains only one sample, then the From File block outputs the corresponding data value.</p> <p>If the MAT-file contains more than one sample, then the From File block linearly extrapolates using data values of the last two samples:</p> <ul style="list-style-type: none"> • For <code>double</code> data, extrapolates the value using the last two samples. • For <code>Boolean</code> data, outputs the first data value. • For built-in data types other than <code>double</code> or <code>Boolean</code>: <ul style="list-style-type: none"> • Upcasts the data to <code>double</code> • Performs linear extrapolation, as described for <code>double</code> data • Downcasts the extrapolated value to the original data type <p>You cannot use the <code>Linear extrapolation</code> option with enumerated (<code>enum</code>) data. All signals in a bus use the same extrapolation setting. If any signal in a bus uses <code>enum</code> data, then you cannot use the <code>Linear extrapolation</code> option.</p>
Hold last value	Uses the last data value in the file

Method	Description
Ground value	<p>Uses a value that depends on the data type of MAT-file sample data values:</p> <ul style="list-style-type: none"> • Fixed-point data types — uses the ground value • Numeric types other than fixed point — uses 0 • Boolean — uses false • Enumerated data types — uses default value

Parameter: ExtrapolationAfterLastDataPoint

Type: character vector

'Linear extrapolation' | 'Hold last value' | 'Ground value'

Default: 'Linear extrapolation'

Enable zero-crossing detection

Enables zero-crossing detection.

The “Zero-Crossing Detection” parameter applies only if the **Sample time** parameter is set to 0 (continuous).

Simulink uses a technique known as zero-crossing detection to locate a discontinuity in time stamps, without resorting to excessively small time steps. “Zero-crossing” represents a discontinuity.

For the From File block, zero-crossing detection occurs only at time stamps in the file. Simulink examines only the time stamps, not the data values.

For bus signals, Simulink detects zero-crossings across all leaf bus elements.

If the input array contains duplicate time stamps (more than one entry with the same time stamp), Simulink detects a zero crossing at those time stamps. For example, suppose that the input array has this data:

```
time:      0 1 2 2 3
signal:    2 3 4 5 6
```

At time 2, there is a zero crossing from the input signal discontinuity.

For nonduplicate time stamps, zero-crossing detection depends on the settings of the following parameters:

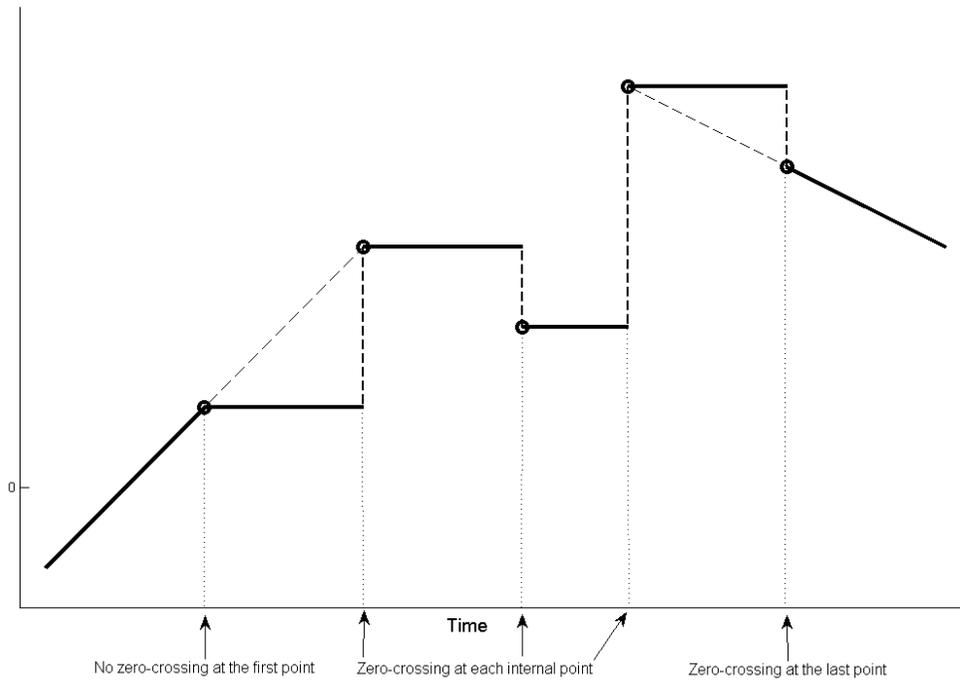
- **Data extrapolation before first data point**
- **Data interpolation within time range**
- **Data extrapolation after last data point**

The From File block determination of when zero-crossing occurs depends on the time stamp.

Time Stamp	Setting
First	Data extrapolation before first data point is set to Ground value.
Between first and last	Data interpolation within time range is set to Zero-order hold.
Last	One or both of these settings occur: <ul style="list-style-type: none"> • Data extrapolation after last data point is set to Ground value. • Data interpolation within time range is set to Zero-order hold.

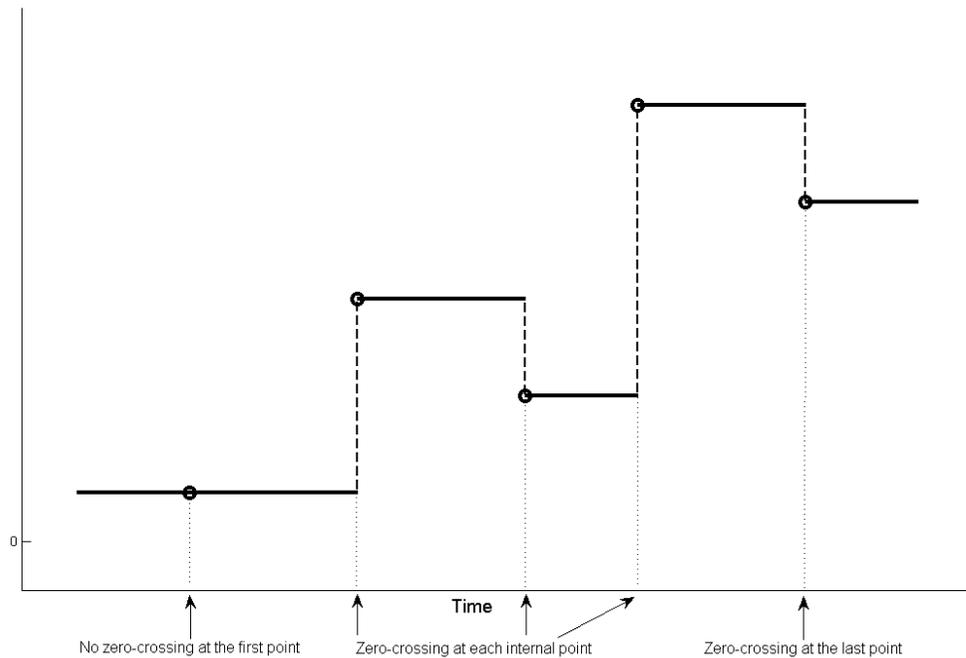
This figure illustrates zero-crossing detection for data accessed by a From File block that has the following settings:

- **Data extrapolation before first data point** — Linear extrapolation
- **Data interpolation within time range** (for internal points) — Zero order hold
- **Data extrapolation after last data point** — Linear extrapolation



This figure is another illustration of zero-crossing detection for data accessed by a From File block. The block has the following settings for the time stamps (points):

- **Data extrapolation before first data point** — Hold first value
- **Data interpolation within time range** — Zero order hold
- **Data extrapolation after last data point** — Hold last value



Examples

From File Block Loading Timeseries Data

Create a MATLAB® `timeseries` object with time and signal values. Save the `timeseries` object to a MAT-file and load into a model using a From File block.

Create an array with the time and signal data, specifying signal data for 10 time steps.

```
t = .1*(1:10);
d = .2*(1:10);
x = [t;d];
```

Create a MATLAB `timeseries` object.

```
ts = timeseries(x(2:end,:),x(1,:))
```

```
timeseries

Common Properties:
    Name: 'unnamed'
    Time: [10x1 double]
    TimeInfo: tsdata.timemetadata
    Data: [1x1x10 double]
    DataInfo: tsdata.datametadata
```

Save the `timeseries` object in a Version 7.3 MAT-file.

```
save('mySignals', 'ts', '-v7.3')
```

Add a From File block and set the **File name** parameter of that block to `mySignals.mat`.

Simulate the model. The Scope block reflects the data loaded from the `mySignals.mat` file.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed Point Enumerated Bus
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

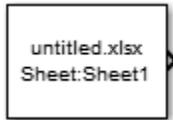
See Also

“Create Data for a From File Block”, “Load Data Using the From File Block”, “Load Signal Data That Uses Units”, “Comparison of Signal Loading Techniques”, To File, From Workspace, To Workspace, From Spreadsheet

Introduced before R2006a

From Spreadsheet

Read data from spreadsheet



Library

Sources

Description

The From Spreadsheet block reads data from Microsoft Excel® (all platforms) or CSV (Microsoft Windows® platform with Microsoft Office installed only) spreadsheets and outputs the data as a signal.

The From Spreadsheet icon displays the spreadsheet file name and sheet name specified in the block **File name** and **Sheet name** parameters.

Storage Formats

The data that the From Spreadsheet block reads from a spreadsheet must be appropriately formatted.

For Microsoft Excel spreadsheets:

- The From Spreadsheet block interprets the first row as a signal name. If you do not specify a signal name, the From Spreadsheet block assigns a default one with the format `Signal #`, where `#` increments with each additional unnamed signal.
- The From Spreadsheet block interprets the first column as time. In this column, the time values must monotonically increase.
- The From Spreadsheet block interprets the remaining columns as signals.

This example shows an acceptably formatted Microsoft Excel spreadsheet. The first column is Time and the first row contains signal names. Each worksheet contains a signal group.

1	Time	DC In	Trigger	AC In	
2	0	1	2	3	
3	1	2	3	4	
4	2	3	4	5	
5	3	4	5	6	
6	4	5	6	7	
7	5	6	7	8	
8	6	7	8	9	
9	7	8	9	10	
10	8	9	10	11	
11	9	10	11	12	
12	10	11	12	13	
13	11	12	13	14	
14	12	13	14	15	
15	13	14	15	16	
16	14	15	16	17	
17	15	16	17	18	
18					

Navigation: < > Group1 Group2 Group3

For CSV text files (Microsoft platform with Microsoft Office installed only):

- The From Spreadsheet block interprets the first column as time. In this column, the time values must increase.
- The From Spreadsheet block interprets the remaining columns as signals.
- Each column must have the same number of entries.
- The From Spreadsheet block interprets each file as one signal group.

This example shows an acceptably formatted CSV file. The contents represent one signal group.

```
0,0,0,5,0
1,0,1,5,0
```

```
2,0,1,5,0
3,0,1,5,0
4,5,1,5,0
5,5,1,5,0
6,5,1,5,0
7,0,1,5,0
8,0,1,5,1
9,0,1,5,1
10,0,1,5,0
```

Block Behavior During Simulation

The From Spreadsheet block incrementally reads data from the spreadsheet during simulation.

The **Sample time** parameter specifies the sample time that the From Spreadsheet block uses to read data from the spreadsheet. For details, see “Parameters” on page 1-641. The time stamps in the file must be monotonically nondecreasing.

For each simulation time hit for which the spreadsheet contains no matching time stamp, Simulink software interpolates or extrapolates to obtain the needed data using the selected method. For details, see “Simulation Time Hits That Have No Corresponding Spreadsheet Time Stamps” on page 1-638.

Simulation Time Hits That Have No Corresponding Spreadsheet Time Stamps

If the simulation time hit does not have a corresponding spreadsheet time stamp, the From Spreadsheet block output depends on:

- Whether the simulation time hit occurs before the first time stamp, within the range of time stamps, or after the last time stamp
- The interpolation or extrapolation methods that you select
- The data type of the spreadsheet data

For details about interpolation and extrapolation options, see the descriptions of the following parameters in “Parameters and Dialog Box”:

- **Data extrapolation before first data point**

- **Data interpolation within time range**
- **Data extrapolation after last data point**

Sometimes the spreadsheet includes two or more data values that have the same time stamp. In such cases, the From Spreadsheet block action depends on when the simulation time hit occurs, relative to the duplicate time stamps in the spreadsheet.

For example, suppose that the spreadsheet contains this data. Three data values have a time stamp value of 2.

```
time stamps:    0 1 2 2 2 3 4
data values:   2 3 6 4 9 1 5
```

The table describes the From Spreadsheet block output.

Simulation Time, Relative to Duplicate Time Stamp Values in Spreadsheet	From Spreadsheet Block Action
Before the duplicate time stamps	Performs the same actions as when the time stamps are distinct, using the first of the duplicate time stamp values as the basis for interpolation. (In this example, the time stamp value is 6.)
At or after the duplicate time stamps	Performs the same actions as when the times stamps are distinct, using the last of the duplicate time stamp values as the basis for interpolation. (In this example, that time stamp value is 9.)

Rounding Mode

The From Spreadsheet block rounds positive and negative numbers toward negative infinity. This mode is equivalent to the MATLAB `floor` function.

Saturation on Integer Overflow

For data type conversion, the From Spreadsheet block deals with saturation overflow by wrapping to the appropriate value that the data type can represent. For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126 .

Code Generation Requirements

Simulating in accelerator, rapid accelerator, model reference accelerator mode, or model reference rapid accelerator mode behaves the same way, and has the same requirements, as simulating in normal mode.

The From Spreadsheet block does not support generating code that involves building ERT or GRT targets, or using SIL or PIL simulation modes.

Data Type Support

The From Spreadsheet block supports these data types:

- Inherit: auto — Default
- double
- single
- int8
- uint8
- int16
- uint16
- int32
- uint32
- Boolean
- fixdt(1,16,0) — Data type is fixed point (1,16,0).
- fixdt(1,16,2^0,0) — Data type is fixed point (1,16,2^0,0).
- Enum: <class_name> — Data type is enumerated.
- data type expression — The name of a data type object, for example `Simulink.NumericType`. Do not specify a bus object as the expression.

The From Spreadsheet block accepts data type specifications at a block level.

If you want to specify different data types for each signal, consider selecting **Output Data Type > Inherit: Auto**. This option resolves back signal data types using back propagation. For example, assume that there are two signals in the From Spreadsheet block, In1 and In2, which the block sends to ports that have int8 and Boolean data types. With back propagation, the block recasts In1 as int8 and In2 as boolean.

Parameters

File name

Enter full path and file name of a spreadsheet file.

This block supports non-English full paths and file names only on Microsoft platforms.

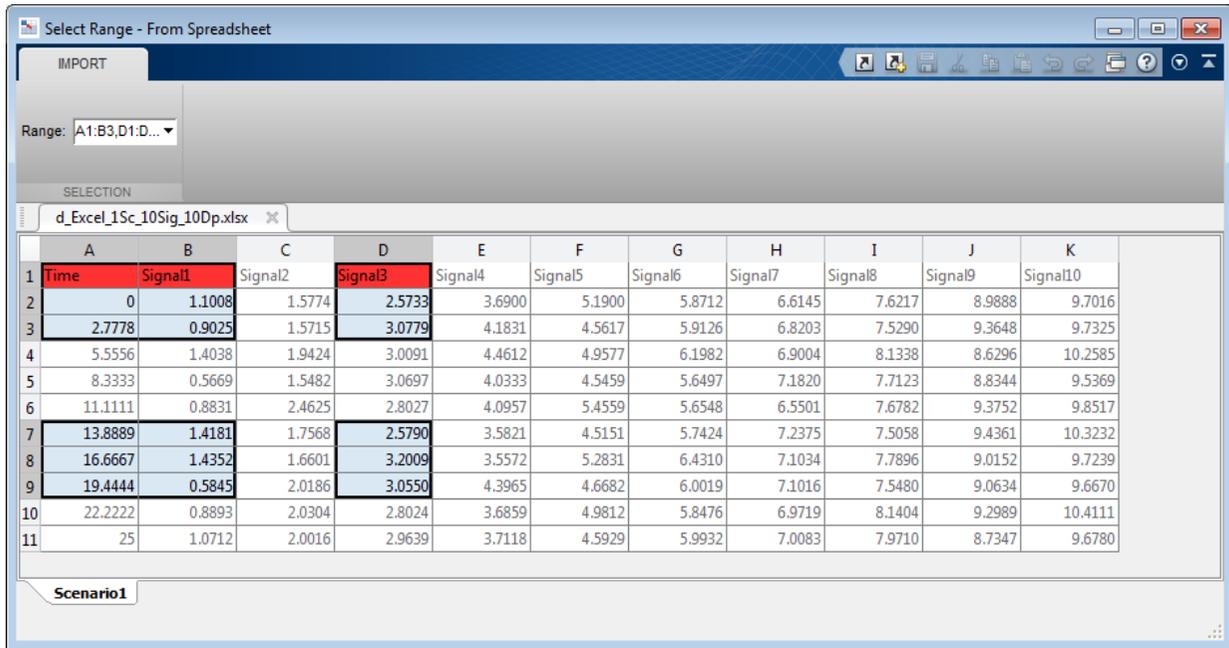
Default: `untitled.xlsx`

FileName	character vector
----------	------------------

Sheet name

Enter the name of the sheet in the spreadsheet. You can type the sheet name in this edit box, or select the sheet name after you open the sheet.

To open the sheet, click . In the sheet, you can select the range of data by dragging over the desired range of values.



Alternatively, you can select the range of data by specifying the range of values in the **Range** parameter.

Default: Sheet1

SheetName	character vector
-----------	------------------

Range

To specify the range, use the format *column:row*, with multiple specifications separated by commas. For example, A1:B3, D1:D3, A7:B9, D7:D9. If unspecified, or empty, the block automatically detects the used range, which is all the data in the sheet.

A	B	C	D	E	F	G	H	I	J	K
Time	Signal1	Signal2	Signal3	Signal4	Signal5	Signal6	Signal7	Signal8	Signal9	Signal10
0	1.1008	1.5774	2.5733	3.6900	5.1900	5.8712	6.6145	7.6217	8.9888	9.7016
2.7778	0.9025	1.5715	3.0779	4.1831	4.5617	5.9126	6.8203	7.5290	9.3648	9.7325
5.5556	1.4038	1.9424	3.0091	4.4612	4.9577	6.1982	6.9004	8.1338	8.6296	10.2585
8.3333	0.5669	1.5482	3.0697	4.0333	4.5459	5.6497	7.1820	7.7123	8.8344	9.5369
11.1111	0.8831	2.4625	2.8027	4.0957	5.4559	5.6548	6.5501	7.6782	9.3752	9.8517
13.8889	1.4181	1.7568	2.5790	3.5821	4.5151	5.7424	7.2375	7.5058	9.4361	10.3232
16.6667	1.4352	1.6601	3.2009	3.5572	5.2831	6.4310	7.1034	7.7896	9.0152	9.7239
19.4444	0.5845	2.0186	3.0550	4.3965	4.6682	6.0019	7.1016	7.5480	9.0634	9.6670
22.2222	0.8893	2.0304	2.8024	3.6859	4.9812	5.8476	6.9719	8.1404	9.2989	10.4111
25	1.0712	2.0016	2.9639	3.7118	4.5929	5.9932	7.0083	7.9710	8.7347	9.6780

If the selections overlap, the block resolves the selection information as appropriate. For example, if you specify multiple ranges that overlap, such as A1:B4, B1:E7, the block resolves the selection to A1 to E7, inclusive.

Default: Empty

Range	character vector
-------	------------------

Output data type

The data type for the From Spreadsheet block output. The From Spreadsheet block accepts spreadsheets that contain many data types. However, the block reads the spreadsheet data type as doubles. It then outputs the data type according to the value of **Output data type**.

To allow the block to cast the output data type to match that of the receiving block, use `Inherit: auto`.

For more information, see “Control Signal Data Types”.

Default: `Inherit: auto`

Cast output data type to match that of the receive block.

`Inherit: auto`

Cast output data type to match that of the receive block.

double

Output data type is double.

single

Output data type is single.

int8

Output data type is int8.

uint8

Output data type is uint8.

int16

Output data type is int16.

uint16

Output data type is uint16.

int32

Output data type is int32.

uint32

Output data type is uint32.

boolean

Output data type is boolean.

fixdt(1,16,0)

Output data type is fixed point fixdt(1,16,0).

fixdt(1,16,2^0,0)

Output data type is fixed point fixdt(1,16,2^0,0).

Enum: <class name>

Output data type is enumerated, for example, Enum: Basic Colors.

<data type expression>

The name of a data type object, for example Simulink.NumericType

OutDataTypeStr	<pre>character vector — {'Inherit: auto'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>'</pre>
----------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

>> (Show data type assistant)

Displays the **Data Type Assistant**, to help you to set the **Output data type** parameter.

Mode

The category of data to specify. For more information, see “Control Signal Data Types”.

Default: `Inherit`

Inheritance rule for data types. Selecting `Inherit` enables a second menu/text box to the right.

- `Inherit`

Inheritance rule for data types. Selecting `Inherit` enables a second menu/text box to the right.

- `Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double` — Default
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

- `boolean`
- `Fixed point` — Fixed-point data types
- `Enumerated` — Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.
- `Expression` — Expression that evaluates to a data type. Selecting `Expression` enables a second menu/text box to the right, where you enter the expression.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data as signed.

Unsigned

Specify the fixed-point data as unsigned.

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Binary point

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope` and `bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

See “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Treat first column as

Select how the block should treat the first column of the spreadsheet:

- `Time` — Treat first column as time.
- `Data` — Treat first column as data. Selecting this option disables:
 - **Data extrapolation before first data point**
 - **Data interpolation within time range**
 - **Data extrapolation after last data point**
- Selecting this option enables **Output after last data point**.

Sample time

The sample period and offset.

The From Spreadsheet block reads data from a spreadsheet using a sample time that either:

- You specify for the From Spreadsheet block
- The From Spreadsheet block inherits from the blocks into which the From Spreadsheet block feeds data

The default is 0, which specifies a continuous sample time. The spreadsheet is read at the base (fastest) rate of the model. For details, see “Specify Sample Time”.

SampleTime	character vector — {'0'}
------------	--------------------------

Data extrapolation before first data point

Extrapolation method that Simulink uses for a simulation time hit that is before the first time stamp in the spreadsheet. Choose one of the following extrapolation methods.

Method	Description
Linear extrapolation	<p>(Default)</p> <p>If the spreadsheet contains only one sample, the From Spreadsheet block outputs the corresponding data value.</p> <p>If the spreadsheet contains more than one sample, the From Spreadsheet block linearly extrapolates using the first two samples:</p> <ul style="list-style-type: none"> • For <code>double</code> data, linearly extrapolates the value using the first two samples • For <code>Boolean</code> data, outputs the first data value • For a built-in data type other than <code>double</code> or <code>Boolean</code>: <ul style="list-style-type: none"> • Upcasts the data to <code>double</code>. • Performs linear extrapolation (as described above for <code>double</code> data). • Downcasts the extrapolated data value to the original data type. <p>You cannot use the <code>Linear extrapolation</code> option with enumerated (<code>enum</code>) data.</p>
Hold first value	Uses the first data value in the file

Method	Description
Ground value	<p>Uses a value that depends on the data type of spreadsheet sample data values:</p> <ul style="list-style-type: none"> • Fixed-point data types — Uses the ground value • Numeric types other than fixed-point — Uses 0 • Boolean — Uses false • Enumerated data types — Uses default value
ExtrapolationBeforeFirstDataPoint	<p>character vector — {'Linear extrapolation'} 'Hold first value' 'Ground value'</p>

Data interpolation within time range

The interpolation method that Simulink uses for a simulation time hit between two time stamps in the spreadsheet. Choose one of the following interpolation methods.

Method	Description
Linear interpolation	<p>(Default)</p> <p>The From Spreadsheet block interpolates using the two corresponding spreadsheet samples:</p> <ul style="list-style-type: none"> • For <code>double</code> data, linearly interpolates the value using the two corresponding samples • For <code>Boolean</code> data, uses <code>false</code> for the first half of the sample and <code>true</code> for the second half. • For a built-in data type other than <code>double</code> or <code>Boolean</code>: <ul style="list-style-type: none"> • Upcasts the data to <code>double</code>. • Performs linear interpolation (as described above for <code>double</code> data). • Downcasts the interpolated value to the original data type. <p>You cannot use the <code>Linear interpolation</code> option with enumerated (<code>enum</code>) data.</p>
Zero order hold	Uses the data from the first of the two samples
<code>InterpolationWithinTimeRange</code>	character vector — {'Linear interpolation'} 'Zero order hold'

Data extrapolation after last data point

The extrapolation method that Simulink uses for a simulation time hit that is after the last time stamp in the spreadsheet. Choose one of the following extrapolation methods.

Method	Description
Linear extrapolation	<p>(Default)</p> <p>If the spreadsheet contains only one sample, the From Spreadsheet block outputs the corresponding data value.</p> <p>If the spreadsheet contains more than one sample, the From Spreadsheet block linearly extrapolates using data values of the last two samples:</p> <ul style="list-style-type: none"> • For double data, extrapolates the value using the last two samples • For Boolean data, outputs the last data value • For a built-in data type other than double or Boolean: <ul style="list-style-type: none"> • Upcasts the data to double. • Performs linear extrapolation (as described above for double data). • Downcasts the extrapolated value to the original data type. <p>You cannot use the Linear extrapolation option with enumerated (enum) data.</p>
Hold last value	Uses the last data value in the file
Ground value	<p>Uses a value that depends on the data type of spreadsheet sample data values:</p> <ul style="list-style-type: none"> • Fixed-point data types — uses the ground value • Numeric types other than fixed-point — uses 0 • Boolean — uses false • Enumerated data types — uses default value
ExtrapolationAfterLastDataPoint	character vector — {'Linear extrapolation'} 'Hold last value' 'Ground value'

Output after last data point

Select action after last data point:

- `Repeat sequence` — Repeat the sequence by reading the data from the first row of the range specified in **Range** (default)
- `Hold final value` — Output the last defined value for the remainder of the simulation.
- `Ground value` — Output a ground value depending on the data type value specified in **Output data type**.

Enable zero-crossing detection

Select to enable zero-crossing detection.

The “Zero-Crossing Detection” parameter applies only if the **Sample time** parameter is set to 0 (continuous).

Simulink uses a technique known as zero-crossing detection to locate accurately a discontinuity, without resorting to excessively small time steps. This section uses “zero-crossing” to represent discontinuities.

For the From Spreadsheet block, zero-crossing detection can only occur at time stamps in the file. Simulink examines only the time stamps, not the data values.

If the input array contains duplicate time stamps (more than one entry with the same time stamp), Simulink detects a zero crossing at that time stamp. For example, suppose that the input array has this data:

```
time:      0 1 2 2 3
signal:    2 3 4 5 6
```

At time 2, there is a zero crossing from the input signal discontinuity.

For data with nonduplicate time stamps, zero-crossing detection depends on the settings of the following parameters:

- **Data extrapolation before first data point**
- **Data interpolation within time range**
- **Data extrapolation after last data point**

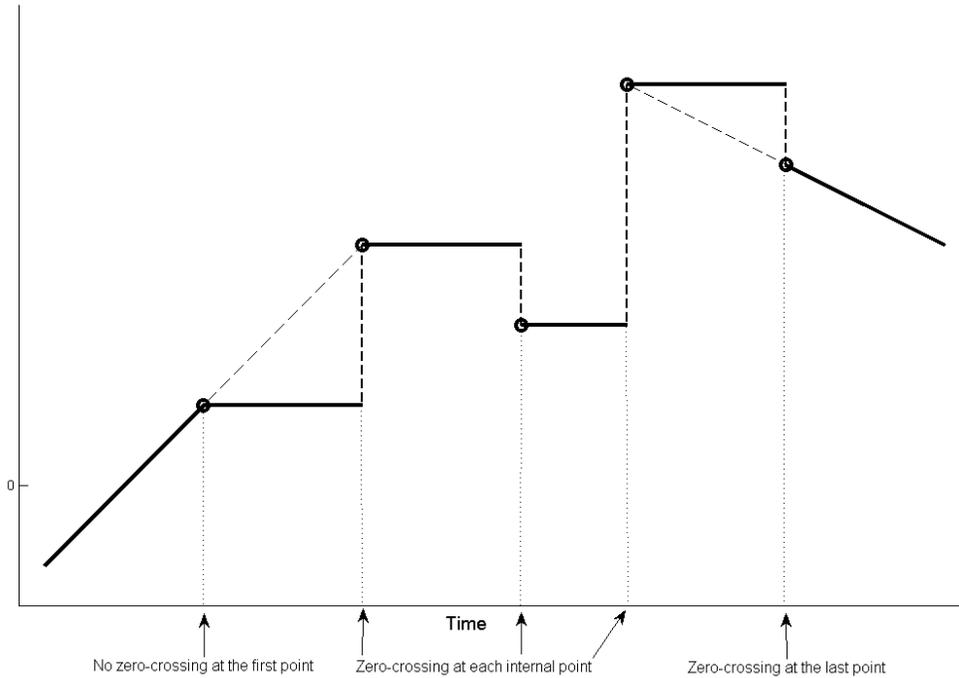
The block applies the following rules when determining when:

- Zero-crossing occurs for the first time stamp.
- For time stamps between the first and last time stamp.
- For the last time stamp.

Time Stamp	When Zero-Crossing Detection Occurs
First	Data extrapolation before first data point is set to Ground value.
Between first and last	Data interpolation within time range is set to Zero-order hold.
Last	One or both of these settings occur: <ul style="list-style-type: none"> • Data extrapolation after last data point is set to Ground value. • Data interpolation within time range is set to Zero-order hold.

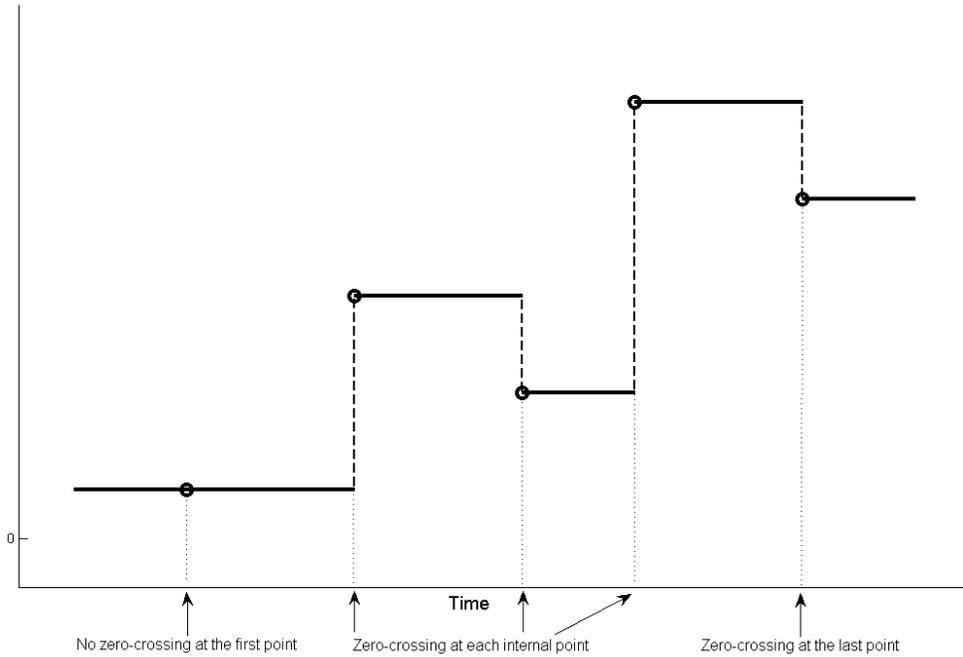
The following figure illustrates zero-crossing detection for data accessed by a From Spreadsheet block that has the following settings:

- **Data extrapolation before first data point** — Linear extrapolation
- **Data interpolation within time range** (for internal points) — Zero order hold
- **Data extrapolation after last data point** — Linear extrapolation



The following figure is another illustration of zero-crossing detection for data accessed by a From Spreadsheet block. The block has the following settings for the time stamps (points):

- **Data extrapolation before first data point** — Hold first value
- **Data interpolation within time range** — Zero order hold
- **Data extrapolation after last data point** — Hold last value



ZeroCross

character vector — 'off' | {'on'}

Characteristics

Data Types	Specified in “Data Type Support” on page 1-640
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes
Code Generation	No

See Also

From File

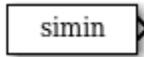
Topics

“Power Window”

Introduced in R2015b

From Workspace

Load signal data from workspace



Library

Sources

Description

The From Workspace block reads signal data from a workspace and outputs the data as a signal.

The From Workspace icon displays the expression specified in the **Data** parameter. For details about how Simulink software evaluates this expression, see “Symbol Resolution”.

You can specify how the data is loaded, including sample time, how to handle data for missing data points, and whether to use zero-crossing detection. For more information, see “Load Data Using the From Workspace Block”.

Specifying the Workspace Data

In the Block Parameters dialog box of the From Workspace block, in the **Data** parameter, specify the workspace data to load. Specify a MATLAB expression (for example, the name of a variable in the MATLAB workspace) that evaluates to one of the following:

- A MATLAB `timeseries` object
- A structure of MATLAB `timeseries` objects
- A structure, with or without time
- A two-dimensional matrix

For additional information, see “Load Data Using the From Workspace Block” .

Use With Data Dictionary

When you link a model to a data dictionary, you:

- Store design data, which contribute to the fundamental design of the model, in the Design Data section of the dictionary. Design data include numeric variables and `Simulink.Parameter` objects that you use to set block parameter values.
- Store simulation input data, which you use to stimulate and experiment with the model, in the base workspace. Typically, you create simulation input data as MATLAB `timeseries` objects.

For more information about storing variables, objects, and other data that a model uses, see “Determine Where to Store Variables and Objects for Simulink Models”.

- To access design data by using a From Workspace block, store the target variable in the Design Data section of the dictionary and set the **Data** parameter of the block to the name of the variable.
- To access simulation input data, store the target variable in the base workspace and set the **Data** parameter by using a call to the `evalin` function. In the call to `evalin`, specify the `ws` argument as `'base'` so that the block seeks the variable in the base workspace instead of the data dictionary. For example, if the name of the variable is `myTimeseriesObject`, set **Data** to `evalin('base', 'myTimeseriesObject')`.

Data Type Support

The From Workspace block accepts data from the workspace. The block outputs real or complex signals of any type that Simulink supports, including fixed-point and enumerated data types.

The From Workspace block also accepts a bus object as a data type. To load bus data, use a structure of MATLAB `timeseries` objects. For details, see “Load Bus Data to Root-Level Input Ports”.

Real signals of type `double` can be in any data format that the From Workspace block supports. For complex signals and real signals of a data type other than `double`, use any format except `Array`.

Parameters

Data

In the **Data** parameter, specify the workspace data to load. Specify a MATLAB expression (for example, the name of a variable in the MATLAB workspace) that evaluates to one of the following:

- A MATLAB `timeseries` object
- A structure of MATLAB `timeseries` objects
- A structure, with or without time
- A two-dimensional matrix

For additional information, see “Specify the Workspace Data”.

Parameter: `VariableName`

Type: character vector

Default: `'simin'`

Output data type

The required data type for the data for the workspace variable that the From Workspace block loads. For non-bus types, to skip any data type verification, you can use `Inherit: auto`. For more information, see “Control Signal Data Types”.

To load bus data, use a structure of MATLAB `timeseries` objects. For details, see “Load Bus Data to Root-Level Input Ports”.

- `Inherit: auto` — Default.
- `double`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`

- `int32`
- `uint32`
- `boolean`
- `fixdt(1,16,0)` — Data type is fixed point (1,16,0).
- `fixdt(1,16,2^0,0)` — Data type is fixed point (1,16,2^0,0).
- `Enum: <class_name>` — Data type is enumerated, for example, `Enum: Basic Colors`.
- `Bus: <bus_object>` — Data type is a bus object.
- `<data type expression>` — The name of a data type object, for example `Simulink.NumericType`. Do not specify a bus object as the expression.

Parameter: `OutDataTypeStr`

Type: character vector

`'Inherit: auto' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'boolean' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>' | 'Bus: <object name>'`

Default: `'Inherit: auto'`

>> (Show data type assistant)

Displays the **Data Type Assistant**, to help you to set the **Output data type** parameter.

Mode

The category of data to specify. For more information, see “Control Signal Data Types”.

- `Inherit` — Inheritance rule for data types. Selecting `Inherit` enables a second menu/text box to the right. (Default)
- `Built in` — Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:
 - `double` — Default
 - `single`
 - `int8`
 - `uint8`

- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`
- `Fixed point` — Fixed-point data types
- `Enumerated` — Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.
- `Bus object` — Bus object. Selecting `Bus` enables a **Bus object** parameter, where you enter the name of a bus object that you want to use to define the structure of the bus. To create or change a bus object, click **Edit** (to the right of the **Bus object** field) to open the Simulink Bus Editor. For details, see “Create Bus Objects with the Bus Editor”.
- `Expression` — Expression that evaluates to a data type. Selecting `Expression` enables a second menu/text box to the right, where you enter the expression. Do not specify a bus object as the expression.

Sample time

Sample rate of loaded data. For details, see “Specify Sample Time”.

Parameter: `SampleTime`

Type: character vector

Default: `'0'`

Interpolate data

To have the block linearly interpolate at time hits for which no corresponding workspace data exist, select this option. Otherwise, the current output equals the output at the most recent time for which data exists.

The From Workspace block interpolates by using the two corresponding workspace samples:

- For `double` data, linearly interpolates the value by using the two corresponding samples

- For `Boolean` data, uses `false` for the first half of the time between two time values and `true` for the second half
- For a built-in data type other than `double` or `Boolean`:
 - Upcasts the data to `double`
 - Performs linear interpolation (as described for `double` data)
 - Downcasts the interpolated value to the original data type

You cannot use linear interpolation with enumerated (`enum`) data.

The block uses the value of the last known data point as the value of time hits that occur after the last known data point.

To determine the block output after the last time hit for which workspace data is available, combine the settings of these parameters:

- **Interpolate data**
- **Form output after final data value by**

For details, see the **Form output after final data value by** parameter.

Parameter: Interpolate

Type: character vector

'off' | 'on'

Default: 'on'

Enable zero-crossing detection

If you select the **Enable zero-crossing detection** parameter, then when the input array contains multiple entries for the same time hit, Simulink detects a zero crossing. For example, suppose that the input array has this data:

```
time:      0 1 2 2 3
signal:    2 3 4 5 6
```

At time 2, there is a zero crossing from input signal discontinuity. For more information, see “Zero-Crossing Detection”.

For bus signals, Simulink detects zero crossings across all leaf bus elements.

Parameter: ZeroCross

Type: character vector

'off' | 'on'

Default: 'on'

Form output after final data value by

To determine the block output after the last time hit for which workspace data is available, combine the settings of these parameters:

- **Interpolate data**
- **Form output after final data value by**

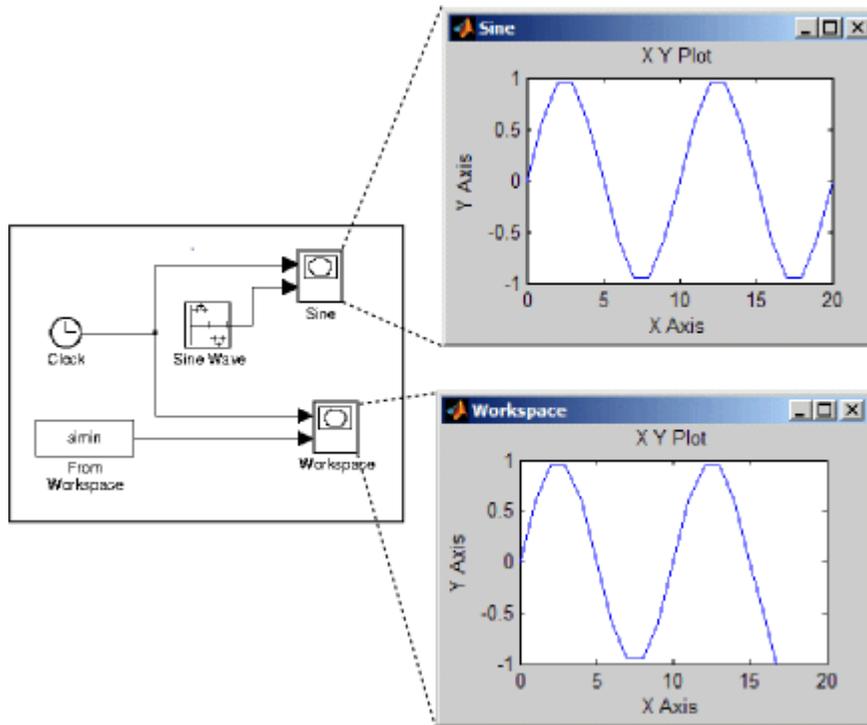
This table lists the block output, based on the values of the two options.

Setting for Form Output After Final Data Value By	Setting for Interpolate Data	Block Output After Final Data
Extrapolation	On	Extrapolated from final data value
	Off	Error
Setting to zero	On	Zero
	Off	Zero
Holding final value	On	Final value from workspace
	Off	Final value from workspace
Cyclic repetition	On	Error
	Off	Repeated from workspace if the workspace data is in structure-without-time format. Error otherwise.

For example, the block uses the last two known data points to extrapolate data points that occur after the last known point if you:

- Select **Interpolate data**.
- Set **Form output after final data value by** to `Extrapolation`.

Consider this model.



The From Workspace block reads data from the workspace. The data consists of the output of the Simulink Sine block sampled at one-second intervals. The workspace contains the first 16 samples of the output. The top and bottom X-Y plots display the output of the Sine Wave and From Workspace blocks, respectively, from 0 to 20 seconds. The straight line in the output of the From Workspace block reflects the linear extrapolation of missing data points at the end of the simulation.

Parameter: OutputAfterFinalValue

Type: character vector

'Extrapolation' | 'Setting to zero' | 'Holding final value' | 'Cyclic repetition'

Default: 'Extrapolation'

Examples

With the From Workspace block, you can read 1-D and 2-D signals into Simulink. The From Workspace block does not read n-D signals into Simulink.

Read 1-D Signals in Array Format

Create two signals x and y with a time vector t . Import the values into Simulink with an array.

- 1 In the MATLAB Command Window, enter:

```
t = 0.2 * [0:49]';  
x = sin(t);  
y = 10*sin(t);
```

The time vector must be a column vector.

- 2 Add a From Workspace block to your model.
- 3 For the From Workspace block, in **Data** parameter, enter the array $[t, x, y]$.

Read 1-D Signals in Structure Format

Create two signals x and y with a time vector t . Import the values into Simulink with a structure.

- 1 In the MATLAB Command Window, enter:

```
t = 0.2 * [0:49]';  
x = sin(t);  
y = 10*sin(t);  
wave.time = t;  
wave.signals.values = [x,y];  
wave.signals.dimensions =2;
```

The time vector must be a column vector. The `signals.dimensions` field for the signal is a scalar corresponding to the number of columns in the `signals.values` field.

- 2 Add a From Workspace block to your model.
- 3 In the From Workspace block parameters dialog box, in the **Data** field, enter the structure name.

Use Sample Time from Model

If you do not have a time vector, you can define the sample time in your model.

- 1 In the MATLAB Command Window, enter

```
wave.time = [];
```

- 2 In the From Workspace block parameters dialog box, in the **Sample time** field, enter a time interval. For example, enter 0.2. Clear the **Interpolate data** check box.
- 3 In the **Form output after final data value by**, select Setting to zero, Holding final value, or Cyclic repetition. Do not select Extrapolation.

Read 2-D Signals in Structure Format

To load 2-D signals from the MATLAB workspace into Simulink, you must have the signals in a structure format. This example creates a 10-by-10 matrix (2-D signal) by using the `magic` function, and then creates a 3-D matrix by adding a time vector.

- 1 In the MATLAB Command Window, enter:

```
t1 = 0.2 * [0:49]';  
m = magic(10);  
M = repmat(m, [1 1 length(t1)]);  
data.time=t1;  
data.signals.values = M;  
data.signals.dimensions=[10 10];
```

The time vector must be a column vector. The `signals.values` field is a 3-D matrix where the third dimension corresponds to time. The `signals.dimensions` field is a two-element vector. The first element is the number of rows and the second element is the number of columns in the `signals.values` field.

- 2 In the From Workspace block parameters dialog box, in the **Data** field, enter the name of the structure.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	Yes, if enabled
Code Generation	No

See Also

“Load Data Using the From Workspace Block”, “Use From Workspace Block for Test Case”, “Load Signal Data That Uses Units”, “Load Signal Data for Simulation”, To Workspace, From File, To File

Introduced before R2006a

Function-Call Feedback Latch

Break feedback loop involving data signals between function-call blocks

Library: Ports & Subsystems

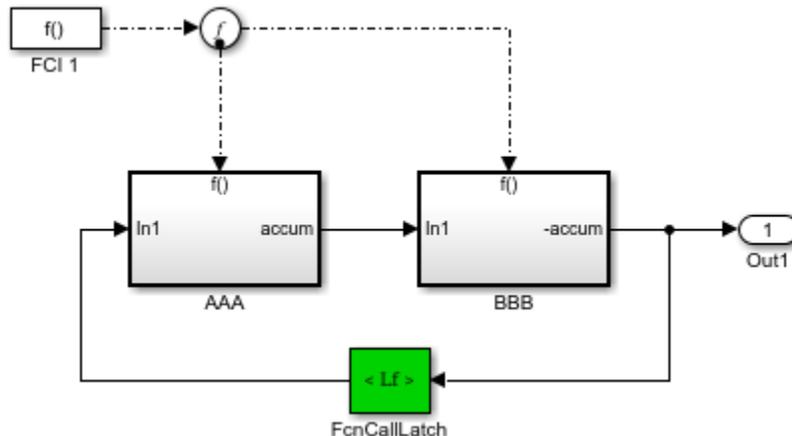


Description

Use the Function-Call Feedback Latch block to break a feedback loop of data signals between one or more function-call blocks. Specifically, break a feedback loop formed in one of the following ways.

- **When function-call blocks connect to branches of the same function-call signal**

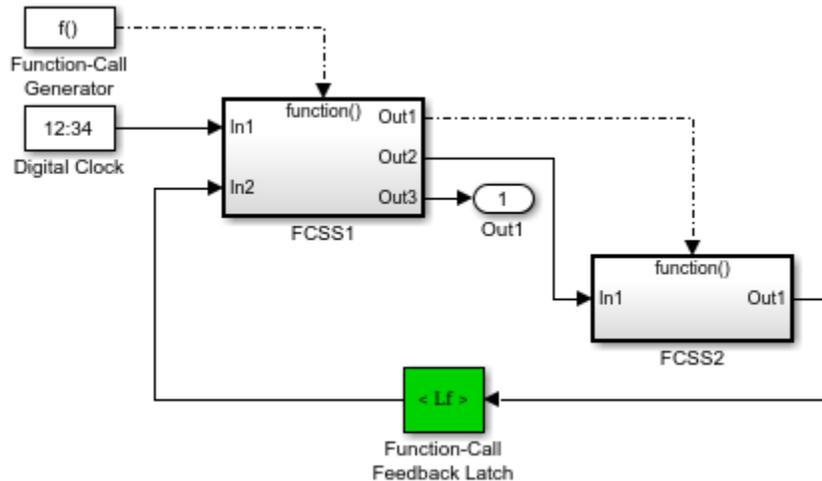
Place the Function-Call Feedback Latch block on the feedback signal between the branched blocks. As a result, the latch block delays the signal at the input of the destination function-call block, and the destination function-call block executes prior to the source function-call block of the latch block.



- **When the loop involves parent and child function-call blocks, where the child initiator is inside the parent**

Place the Function-Call Feedback Latch block on the feedback signal between the child and the parent. This arrangement prevents the signal value, read by the parent

(FCSS1), from changing during execution of the child. In other words, the parent reads the value from the previous execution of the child (FCSS2).



Using the latch block is equivalent to selecting the **Latch input for function-call feedback signals** check box on the Inport block in the destination function-call subsystem or model. However, an advantage of the latch block over using the dialog parameter is that one can design the destination function-call subsystem or model in a modular fashion and then use it either in or out of the context of loops.

The Function-Call Feedback Latch block is better suited than Unit Delay or Memory blocks in breaking function-call feedback loops for the following reasons:

- The latch block delays the feedback signal for exactly one execution of the source function-call block. This behavior is different from the Unit Delay or Memory blocks for cases where the function-call subsystem blocks may execute multiple times in a given simulation step.
- Unlike the Unit Delay or Memory blocks, the latch block may be used to break loops involving asynchronous function-call subsystems.
- The latch block can result in better performance, in terms of memory optimization, for generated code.

Ports

Input

In — Signal from a function-call subsystem block

scalar | vector | matrix

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

Output

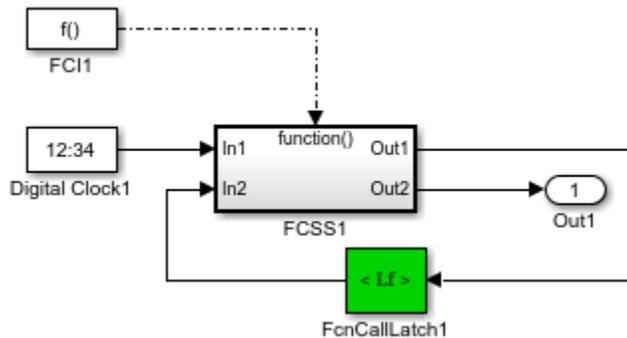
Out — Signal to a function-call subsystem

scalar | vector | matrix

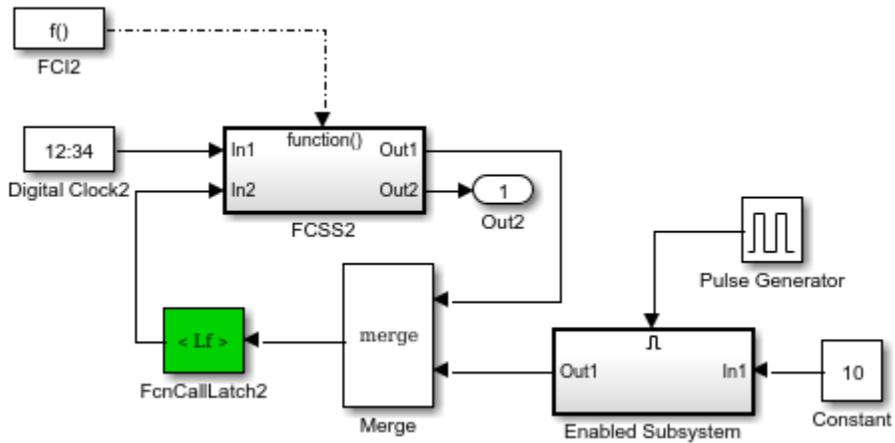
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

Examples

In the following model, a single function-call subsystem output serves as its own input.



A more complex case occurs when a merged signal serves as the input to a function-call subsystem. Latching is necessary if one of the signals entering the Merge block forms a feedback loop with the function-call subsystem. In this example, one of the output signals from FCSS2 combines with the output of an Enabled Subsystem block and then feeds back into an input of FCSS2.



See Also

Function-Call Feedback Latch | Function-Call Generator | Function-Call Split | Function-Call Subsystem | Trigger

Topics

“Using Function-Call Subsystems”

Introduced in R2011a

Function-Call Generator

Provide function-call signal to control execution of a subsystem or model

Library: Ports & Subsystems



Description

The Function-Call Generator block provides a function-call signal to execute a function-call subsystem or function-call model at the rate that you specify with the **Sample time** parameter. To iteratively execute each function-call block multiple times at each time step, use the **Number of Iterations** parameter.

To execute multiple function-call subsystems or models in a specified order, use the Function-Call Generator block with a Function-Call Split block. For an example, see the Function-Call Split block documentation.

Ports

Output

Function Call — Function-call signal to a function-call subsystem or function-call model
scalar

Parameters

Sample time — Specify time interval

-1 (default) | T_s | [T_s , T_o]

Specify the time interval between function calls to a subsystem or model containing a Trigger block with **Trigger type** set to `function-call`. If the actual calling rate for the

subsystem or model differs from the time interval this parameter specifies, Simulink displays an error.

Settings

-1

Inherit time interval from the trigger signal.

Ts

Scalar where Ts is the time interval.

[Ts, To]

Vector where Ts is the time interval and To is the initial time offset.

Programmatic Use

Block parameter: `sample_time`

Type: character vector

Value: '-1' | 'Ts' | '[Ts, To]'

Default: '-1'

Number of iterations — Specify number of times to provide a function-call at each time step

1 (default) | integer

The value of this parameter can be a vector where each element of the vector specifies a number of times to execute a function-call subsystem. The total number of times that a function-call subsystem executes per time step equals the sum of the values of the elements of the generator signal entering its control port.

Suppose that you specify the number of iterations to be [2 2] and connect the output of this block to the control port of a function-call subsystem. In this case, the function-call subsystem executes four times at each time step.

Settings

1

Provide function-call once during each time step.

integer

Signed or unsigned integer number. Provide the specified number of function calls at each time step.

Programmatic Use

Parameter: numberOfIterations

Type: character vector

Value: '1' | '<integer>'

Default: '1'

See Also

Function-Call Feedback Latch | Function-Call Generator | Function-Call Split |
Function-Call Subsystem | Trigger

Topics

“Using Function-Call Subsystems”

Introduced before R2006a

Function-Call Split

Provide junction for splitting function-call signal

Library: Ports & Subsystems



Description

The Function-Call Split block allows a function-call signal to branch and connect to several function-call subsystems or function-call models.

A Function-Call Split outputs multiple function-call signals to create multiple branches from a single function-call signal. In some cases, when you use this block, you do not need the function-call initiator to create multiple function-call signals to invoke a set of function-call subsystems or function-call models.

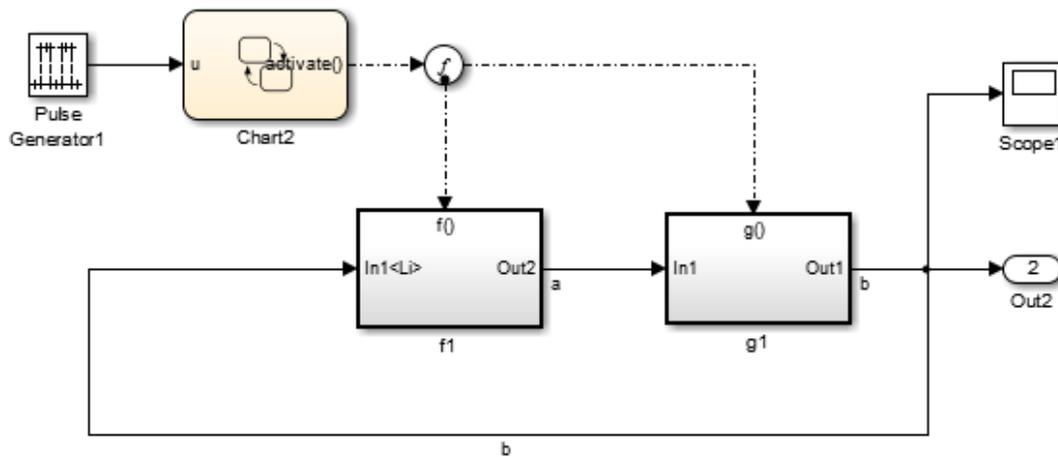
The function-call subsystem or function-call model connected to the output port of the Function-Call Split block that is marked with a dot execute before the subsystems or models connected to other output ports. If data dependencies between subsystems or models do not support the specified execution order, the Function-Call Split block returns an error. To eliminate this error, consider selecting the **Latch input for feedback signals of function-call subsystem outputs** parameter on one or more Inport blocks of the function-call subsystems models involved in a data-dependency loop. Selecting this option delays the corresponding input signal, thereby eliminating the data-dependency loop.

For a model to contain Function-Call Split blocks, you must set the following diagnostic to error: **Model Configuration Parameters > Diagnostics > Connectivity > Invalid function-call connection.**

If you select the model menu option **Display > Blocks > Sorted Execution Order**, then the execution order of function-call subsystems connected to branches of a given function-call signal appears on the blocks . Each subsystem has an execution order of the form $s: [B\#]$ where # is a number that ranges from 0 to one less than the total number of subsystems or models connected to branches of a given signal. The subsystems execute in ascending order based on this number.

The Function-Call Split block supports “Signal Label Propagation”.

The following model shows how to apply the **Latch input for feedback signals of function-call subsystem outputs** parameter to work around a data-dependency error caused by using a Function-Call Split block. By turning this parameter on in the f1 subsystem Inport block, the Function-Call Split block ignores the data dependency of signal b. The block breaks the loop of data dependencies between subsystems f1 and g1. The model achieves the behavior of consistently calling f1 to execute before g1. For a given execution step, subsystem f1 uses the g1 output computed at the previous execution step.



Limitations

The Function-Call Split block has these limitations:

- All function-call subsystems and models connected to a given function-call signal must reside within the same nonvirtual layer of the model hierarchy.
- You cannot connect branched function-call subsystems or models and their children directly back to the function-call initiator.
- Function-call subsystems and models connected to branches of a function-call signal cannot have multiple (muxed) initiators.
- A Function-Call Split block cannot have its input from a signal with multiple function-call elements.

Ports

Input

Function Call — Function-call signal

scalar

A Function-Call Generator block or a Stateflowchart can provide function-call signals.

Output

Function Call — Function-call signal

scalar

Function-call signal connected to a function-call subsystem or function-call model.

Parameters

Icon shape — Select block icon shape`distinctive (default) | round`

Select block icon shape.

Settings`distinctive`

Rectangular block icon.

`round`

Circular block icon.

Programmatic Use*Parameter:* IconShape*Type:* character vector*Value:* 'distinctive' | 'round'*Default:* 'distinctive'**Number of output ports — Specify number of output ports**`2 (default) | integer`

Specify number of function-call signal output ports.

Settings

2

Two function-call signal output ports.

integer

Integer number

Programmatic Use

Parameter: NumOutputPorts

Type: character vector

Value: '2' | '<integer>'

Default: '2'

Output port layout — Select order of output ports

default (default) | reverse

Select the order of output ports with respect to which port provides a function-call first.

Settings

default

Top port provides function-call signal first.

reverse

Bottom port provides function-call signal first.

Programmatic Use

Parameter: OutputPortLayout

Type: character vector

Value: 'default' | 'reverse'

Default: 'default'

See Also

Function-Call Feedback Latch | Function-Call Generator | Function-Call Subsystem

Topics

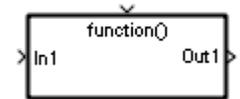
“Using Function-Call Subsystems”

Introduced in R2010a

Function-Call Subsystem

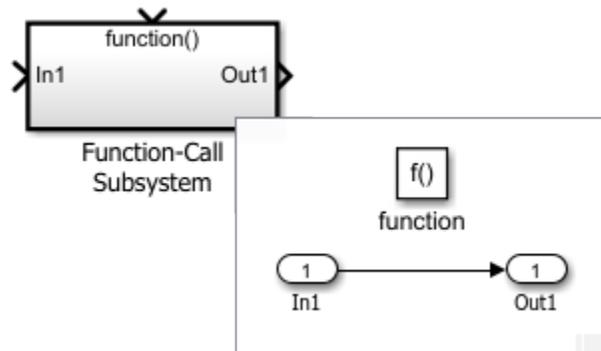
Subsystem whose execution is triggered by external function call input

Library: Ports & Subsystems



Description

The Function-Call Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that executes when a control signal has a function call event.



Use Function-Call Subsystem blocks to:

- Schedule the execution order of model components.
- Control the rate of model component execution.

For an explanation of Function-Call Subsystem blocks parameters, see [Subsystem](#), [Atomic Subsystem](#), [Nonvirtual Subsystem](#), [CodeReuse Subsystem](#)

Ports

Input

In — Signal input to a subsystem block

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

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

Function Call — Function-call control signal to a subsystem block

scalar

Placing a Trigger block in a subsystem block adds an external input port to the block. Selecting `function-call` from the **Trigger type** list, allows the Trigger block to accept function-call signals.

Output

Out — Signal output from a subsystem

scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

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

See Also

Function-Call Feedback Latch | Function-Call Generator | Function-Call Split | Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

Topics

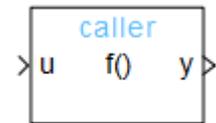
“Using Function-Call Subsystems”

Introduced before R2006a

Function Caller

Call Simulink or exported Stateflow function

Library: User-Defined Functions



Function Caller

Description

A Function Caller block calls and executes a function defined with a Simulink Function block or an exported Stateflow function. Using Function Caller blocks, you can call a function from anywhere in a model or chart hierarchy.

Ports

Input

Input argument — Input signal for an input argument

scalar | vector | matrix

Input signal for an input argument that is sent to the function.

The function prototype determines the number and name of input ports that appear on the Function Caller block. Connect signal lines to the input ports to send data to a function through the function input arguments.

For example, `y = myfunction(u)` creates one input port (`u`) on the Function Caller block.

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

Output

Output argument — Output signal for an output argument

scalar | vector | matrix

Output signal for an output argument that the function returns.

The function prototype determines the number and name of output ports that appear on the Function Caller block. Connect signal lines to the output ports to receive data from a function through the function output arguments.

For example, `y = myfunction(u)` creates one output port (`y`) on the Function Caller block.

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

Parameters

Function prototype — Specify function interface

`y=f(u)` (default) | `<function prototype>`

Specify the function interface between a Function Caller block and a Simulink function. A Simulink function can be a Simulink Function block, an exported Stateflow graphical function, or an exported Stateflow MATLAB function. For a call to a Simulink Function block:

- Function call argument names must match the function arguments.
- Function names, input arguments, and output arguments must be valid MATLAB identifiers.

Programmatic Use

Block Parameter: `FunctionPrototype`

Type: character vector

Values: `'y=f(u)'` | `'<function prototype>'`

Default: `'y=f(u)'`

Input argument specifications — Specify input argument data type, dimensions, and complexity

`<Enter example>` (default) | `<MATLAB expression>`

Specify a comma-separated list of MATLAB expressions that combine data type, dimensions, and complexity (real or imaginary) for each input argument. For examples, see “Argument Specification for Simulink Functions”.

This specification must match the Simulink Function block data type specified with the **Data type** parameter of the Argument Inport block.

Programmatic Use

Block Parameter: 'InputArgumentSpecifications'

Type: character vector

Values: '' | '<MATLAB expression>'

Default: ''

Output argument specifications — Specify output argument data type, dimensions, and complexity

<Enter example> (default) | <MATLAB Expression>

Specify a comma-separated list of MATLAB expressions that combine data type, dimensions, and complexity (real or imaginary) for each output argument. For examples, see “Argument Specification for Simulink Functions”.

This specification must match the Simulink Function block data type specified with the **Data type** parameter of the Argument Output Block.

Programmatic Use

Block Parameter: 'OutputArgumentSpecifications'

Type: character vector

Values: '' | '<MATLAB expression>'

Default: ''

Sample time — Time interval between function calls

-1 (default) | Ts | [Ts, To]

Specify the time interval between function calls to a subsystem or model containing this Trigger block. If the actual calling rate for the subsystem or model differs from the time interval this parameter specifies, Simulink displays an error.

Settings

-1

Inherit time interval from the trigger signal.

Ts

Scalar where Ts is the time interval.

[Ts, To]

Vector where Ts is the time interval and To is the initial time offset.

Programmatic Use

Block parameter: SampleTime

Type: character vector

Value: '-1' | 'Ts' | '[Ts, To]'

Default: '-1'

See Also

Argument Inport | Argument Outport | Chart | Function Caller | Function-Call
Subsystem | Inport | MATLAB Function | Outport | Subsystem | Trigger

Topics

“Simulink Functions”

“Simulink Functions in Models”

“Argument Specification for Simulink Functions”

“Simulink Functions in Referenced Models”

“Scoping Simulink Functions in Subsystems”

“Diagnostics Using a Client-Server Architecture”

Gain

Multiply input by constant

Library

Math Operations



Description

The Gain block multiplies the input by a constant value (gain). The input and the gain can each be a scalar, vector, or matrix.

You specify the value of the gain in the **Gain** parameter. The **Multiplication** parameter lets you specify element-wise or matrix multiplication. For matrix multiplication, this parameter also lets you indicate the order of the multiplicands.

The gain is converted from doubles to the data specified in the block mask offline using round-to-nearest and saturation. The input and gain are then multiplied, and the result is converted to the output data type using the specified rounding and overflow modes.

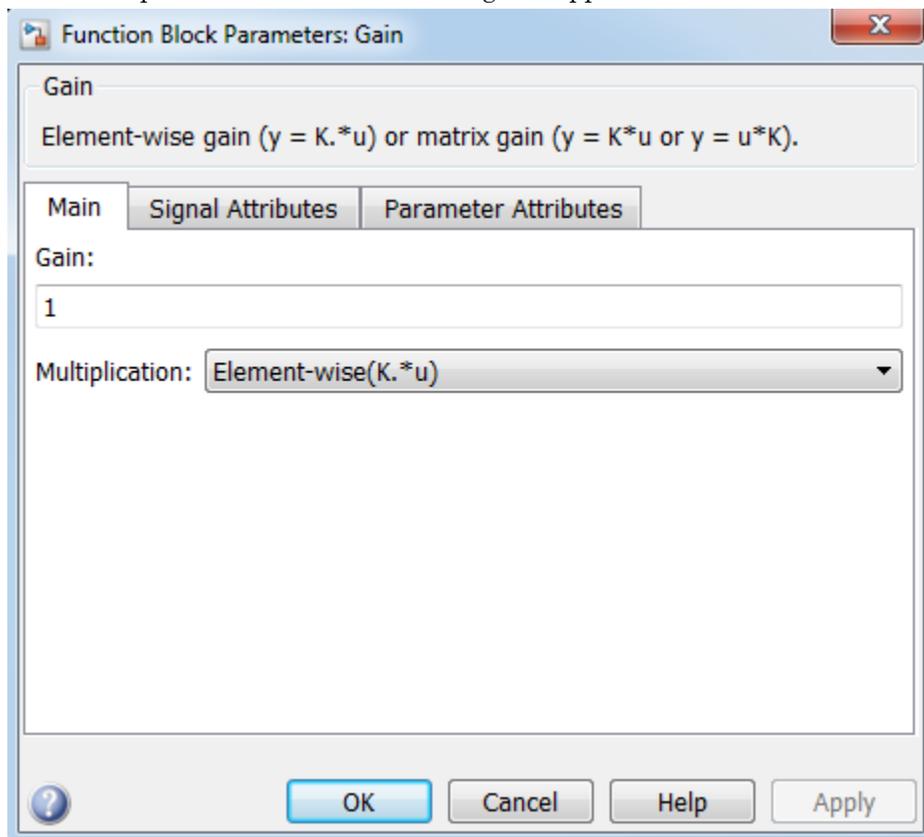
Data Type Support

The Gain block accepts a real or complex scalar, vector, or matrix of any numeric data type that Simulink supports. The Gain block supports fixed-point data types. If the input of the Gain block is real and the gain is complex, the output is complex.

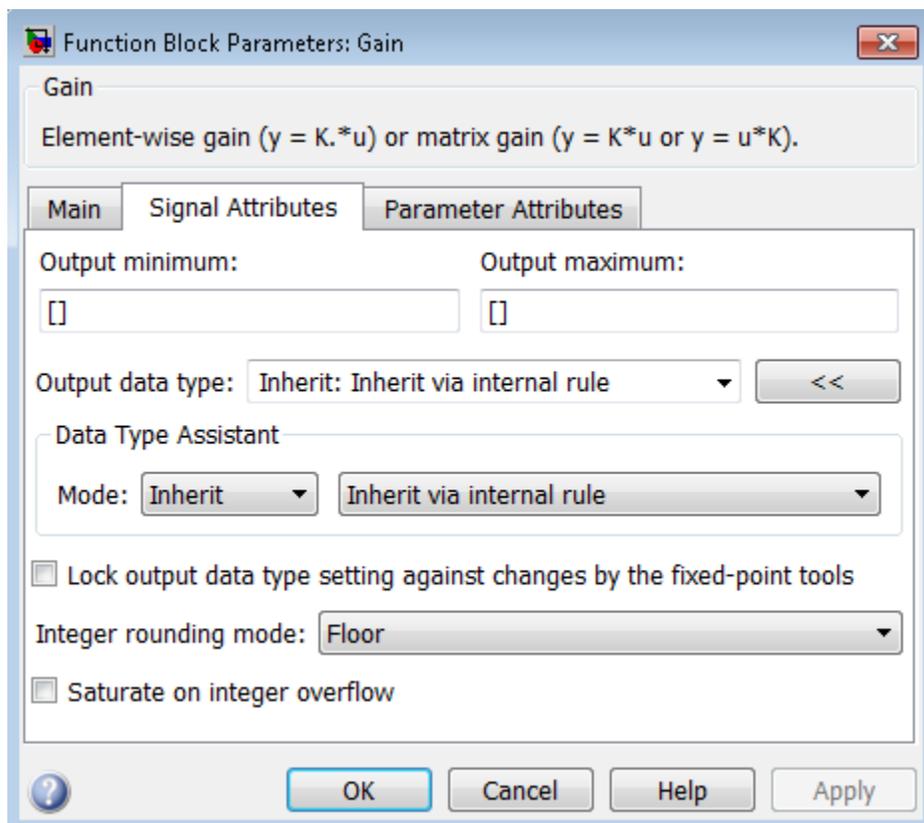
For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters and Dialog Box

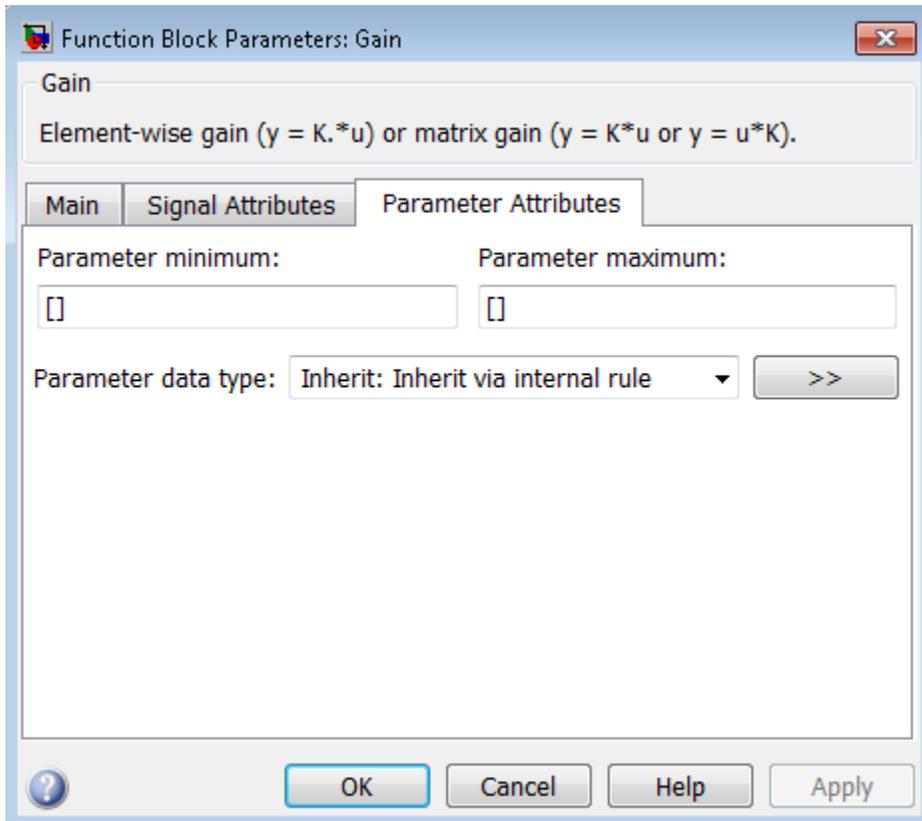
The **Main** pane of the Gain block dialog box appears as follows:



The **Signal Attributes** pane of the Gain block dialog box appears as follows:



The **Parameter Attributes** pane of the Gain block dialog box appears as follows:



Gain

Specify the value by which to multiply the input.

Default: 1

Minimum: value of **Parameter minimum** parameter

Maximum: value of **Parameter maximum** parameter

The gain can be a scalar, vector, or matrix.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Multiplication

Specify the multiplication mode.

Default: Element-wise ($K \cdot u$)

Element-wise ($K \cdot u$)

Each element of the input is multiplied by each element of the gain. The block performs expansions, if necessary, so that the input and gain have the same dimensions.

Matrix ($K \cdot u$)

The input and gain are matrix multiplied with the input as the second operand.

Matrix ($u \cdot K$)

The input and gain are matrix multiplied with the input as the first operand.

Matrix ($K \cdot u$) (u vector)

The input and gain are matrix multiplied with the input as the second operand. This mode is identical to `Matrix ($K \cdot u$)`, except for how dimensions are determined.

Suppose that K is an m -by- n matrix. `Matrix ($K \cdot u$) (u vector)` sets the input to a vector of length n and the output to a vector of length m . In contrast, `Matrix ($K \cdot u$)` uses propagation to determine dimensions for the input and output. For an m -by- n gain matrix, the input can propagate to an n -by- q matrix, and the output becomes an m -by- q matrix.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1 . To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Output minimum

Lower value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMin

Type: character vector

Value: ' [] '

Default: ' [] '

Output maximum

Upper value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.

- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMax

Type: character vector

Value: ' [] '

Default: ' [] '

Output data type

Specify the output data type.

Default: Inherit: Inherit via internal rule

Inherit: Inherit via internal rule

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. For example, if the block multiplies an input of type `int8` by a gain of `int16` and ASIC/FPGA is specified as the targeted hardware type, the output data type is `sfix24`. If `Unspecified` (assume 32-bit Generic), i.e., a generic 32-bit microprocessor, is specified as the target hardware, the output data type is `int32`. If none of the word lengths provided by the target microprocessor can accommodate the output range, Simulink software displays an error in the Diagnostic Viewer.

It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit: Same as input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit: Inherit via back propagation`

Use data type of the driving block.

`Inherit: Same as input`

Use data type of input signal.

`double`

Output data type is double.

`single`

Output data type is single.

`int8`

Output data type is int8.

`uint8`

Output data type is uint8.

`int16`

Output data type is int16.

`uint16`

Output data type is uint16.

`int32`

Output data type is int32.

`uint32`

Output data type is uint32.

`fixdt(1,16,0)`

Output data type is fixed point `fixdt(1,16,0)`.

```
fixdt(1,16,2^0,0)
```

Output data type is fixed point `fixdt(1,16,2^0,0)`.

<data type expression>

Use a data type object, for example, `Simulink.NumericType`.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Control Signal Data Types” for more information.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule (default)`
- `Inherit via back propagation`
- `Same as input`

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`

- uint32

Fixed point

Fixed-point data types.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: Signed

Signed

Specify the fixed-point data as signed.

Unsigned

Specify the fixed-point data as unsigned.

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Binary point

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting Slope and bias enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > Binary point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: `LockScale`

Type: character vector

Value: `'off'` | `'on'`

Default: `'off'`

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: `Floor`

`Ceiling`

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Parameter: `RndMeth`

Type: character vector

Value: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`

Default: `'Floor'`

For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off

On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Parameter: SaturateOnIntegerOverflow

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Parameter minimum

Specify the minimum value of the gain.

Default: []

The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Parameter maximum

Specify the maximum value of the gain.

Default: []

The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Parameter data type

Specify the data type of the **Gain** parameter.

Default: Inherit: Inherit via internal rule

Inherit: Inherit via internal rule

Use an internal rule to inherit the data type.

Inherit: Same as input

Use data type of sole input signal.

Inherit: Inherit from 'Gain'

Use data type of the **Gain** value. For example:

If you set Gain to...	The parameter data type inherits...
2	double
single(2)	single
int8(2)	int8

double

Data type is double.

single

Data type is single.

int8

Data type is int8.

uint8

Data type is uint8.

int16

Data type is int16.

`uint16`

Data type is `uint16`.

`int32`

Data type is `int32`.

`uint32`

Data type is `uint32`.

`fixdt(1,16)`

Data type is `fixdt(1,16)`.

`fixdt(1,16,0)`

Data type is `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`

Data type is `fixdt(1,16,2^0,0)`.

`<data type expression>`

Use a data type object, for example, `Simulink.NumericType`.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule (default)`
- `Same as input`
- `Inherit from 'Gain'`

Built in

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

Fixed point

Fixed-point data types.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant” in the Simulink documentation.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: `Best precision`

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting Slope and bias enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Examples

The following Simulink examples show how to use the Gain block:

- sldemo_bounce
- sldemo_tonegen_fixpt
- sldemo_hardstop
- sldemo_enginewc

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block

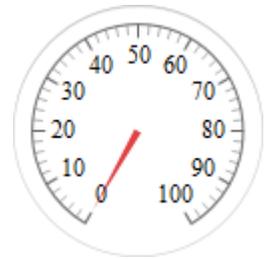
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Gauge

Display input value on circular scale

Library: Simulink / Dashboard



Description

The Gauge block displays the connected signal on a circular scale during simulation. You can use the Gauge block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. The Gauge block provides an indication of the instantaneous value of the connected signal throughout simulation. You can modify the range of the Gauge block to fit your data. You can also customize the appearance of the Gauge block to provide more information about your signal. For example, you can color-code in-specification and out-of-specification ranges.

Connection

Dashboard blocks do not use ports to connect to signals. To connect a signal to a Dashboard block, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting signals of interest in your model.
- 3 Mark the option button next to the signal you want to display.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- If you turn off logging for a signal connected to a Dashboard block, the model stops sending data from that signal to the block. To view the signal again, reconnect the signal.
- External mode simulation does not support visualization Dashboard blocks.

Parameters

Connection — Select a signal to connect and display

signal connection options

Signal connection options. Populate the **Connection** table by selecting signals of interest in your model. Mark the check box next to the signal you want to display. Click **Apply** to connect the signal.

Minimum — Minimum tick mark value

0 (default) | scalar

A finite, real, double, scalar value specifying the minimum tick mark value for the scale. The minimum must be less than the value entered for the maximum.

Maximum — Maximum tick mark value

100 (default) | scalar

A finite, real, double, scalar value specifying the maximum tick mark value for the scale. The maximum must be greater than the value entered for the minimum.

Tick Interval — Interval between major tick marks

auto (default) | scalar

A finite, real, positive, integer, scalar value specifying the interval of major tick marks on the scale. When set to `auto`, the block automatically adjusts the tick interval based on the minimum and maximum values.

Scale Colors — Color indications on Gauge scale

colors for scale ranges

Color specifications for ranges on the scale. Press the **+** button to add a color. For each color added, specify the minimum and maximum values of the range where you want to display that color.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

Model Examples

See Also

Half Gauge | Linear Gauge | Quarter Gauge

Topics

“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015a

Goto

Pass block input to From blocks



Library

Signal Routing

Description

The Goto block passes its input to its corresponding From blocks. The input can be a real- or complex-valued signal or vector of any data type. From and Goto blocks allow you to pass a signal from one block to another without actually connecting them.

A Goto block can pass its input signal to more than one From block, although a From block can receive a signal from only one Goto block. The input to that Goto block is passed to the From blocks associated with it as though the blocks were physically connected. Goto blocks and From blocks are matched by the use of Goto tags.

The **Tag Visibility** parameter determines whether the location of From blocks that access the signal is limited:

- `local`, the default, means that From and Goto blocks using the same tag must be in the same subsystem. A local tag name is enclosed in brackets (`[]`).
- `scoped` means that From and Goto blocks using the same tag must be in the same subsystem or at any level in the model hierarchy below the Goto Tag Visibility block that does not entail crossing a nonvirtual subsystem boundary, i.e., the boundary of an atomic, conditionally executed, or function-call subsystem or a model reference. A scoped tag name is enclosed in braces (`{}`).
- `global` means that From and Goto blocks using the same tag can be anywhere in the model except in locations that span nonvirtual subsystem boundaries.

The rule that From-Goto block connections cannot cross nonvirtual subsystem boundaries has the following exception. A Goto block connected to a state port in one conditionally executed subsystem is visible to a From block inside another conditionally executed subsystem.

Note A *scoped* Goto block in a masked system is visible only in that subsystem and in the nonvirtual subsystems it contains. Simulink generates an error if you run or update a diagram that has a Goto Tag Visibility block at a higher level in the block diagram than the corresponding *scoped* Goto block in the masked subsystem.

Use local tags when the Goto and From blocks using the same tag name reside in the same subsystem. You must use global or scoped tags when the Goto and From blocks using the same tag name reside in different subsystems. When you define a tag as global, all uses of that tag access the same signal. A tag defined as *scoped* can be used in more than one place in the model.

The Goto block supports signal label propagation.

Data Type Support

The Goto block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see “Data Types Supported by Simulink”.

Parameters

Goto Tag

The Goto block identifier. This parameter identifies the Goto block whose scope is defined in this block.

Rename All

Rename the Goto tag. The new name propagates to the From and Goto Tag Visibility blocks that are listed in the **Corresponding blocks** box.

Tag Visibility

The scope of the Goto block tag: `local`, `scoped`, or `global`. The default is `local`.

Corresponding blocks

List of the From blocks and Goto Tag Visibility blocks connected to this Goto block. Click an entry in the list to display and highlight the corresponding From or Goto Tag Visibility block.

Icon Display

Specifies the text to display on the block's icon. The options are the block's tag, the name of the signal that the block represents, or both the tag and the signal name.

Examples

The following models show how to use the Goto block:

- `sldemo_auto_climatecontrol`
- `sldemo_hardstop`

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from driving block
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

From, Goto Tag Visibility

Introduced before R2006a

Goto Tag Visibility

Define scope of Goto block tag



Library

Signal Routing

Description

The Goto Tag Visibility block defines the accessibility of Goto block tags that have scoped visibility. The tag specified as the **Goto tag** parameter is accessible by From blocks in the same subsystem that contains the Goto Tag Visibility block and in subsystems below it in the model hierarchy.

A Goto Tag Visibility block is required for Goto blocks whose **Tag Visibility** parameter value is `scoped`. No Goto Tag Visibility block is needed if the tag visibility is either `local` or `global`. The block shows the tag name enclosed in braces (`{}`).

Data Type Support

Not applicable.

Parameters

Goto tag

The Goto block tag whose visibility is defined by the location of this block.

If you use multiple From and Goto Tag Visibility blocks to refer to the same Goto tag, you can simultaneously rename the tag in all of the blocks. Use the **Rename All** button in the Goto block dialog box.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Not applicable
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

Introduced before R2006a

Ground

Ground unconnected input port



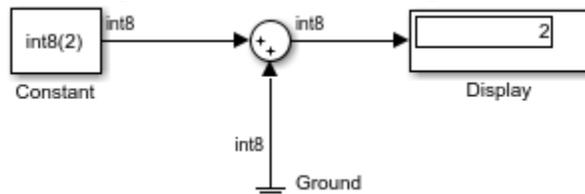
Library

Sources

Description

The Ground block connects to blocks whose input ports do not connect to other blocks. If you run a simulation with blocks having unconnected input ports, Simulink issues warnings. Using a Ground block to ground those unconnected blocks can prevent these warnings.

The Ground block outputs a signal of the same data type as the port to which it connects. For example, consider the following model:



In this example, the output of the Constant block determines the data type (`int8`) of the port to which the Ground block is connected. That port determines the output data type of the Ground block.

The Ground block outputs a signal with zero value. When the output data type cannot represent zero exactly, the Ground block outputs a nonzero value that is the closest possible value to zero. This behavior applies only to fixed-point data types with nonzero bias. The following expressions are examples of fixed-point data types that cannot represent zero:

- `fixdt(0, 8, 1, 1)` — an unsigned 8-bit type with slope of 1 and bias of 1
- `fixdt(1, 8, 6, 3)` — a signed 8-bit type with slope of 6 and bias of 3

If the output is an enumerated data type, the Ground block outputs the default value of the enumeration. This behavior applies whether or not:

- The enumeration can represent zero.
- The default value of the enumeration is zero.

If the enumerated type does not have a default value, the Ground block outputs the first enumeration value in the type definition.

Data Type Support

The Ground block supports all data types that Simulink supports, including fixed-point and enumerated data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Examples

The following Simulink examples show how to use the Ground block:

- `sldemo_doublebounce`

Characteristics

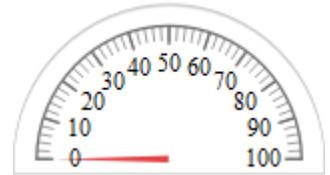
Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Constant (<i>inf</i>)
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

Introduced before R2006a

Half Gauge

Display input value on semicircular scale

Library: Simulink / Dashboard



Description

The Half Gauge block displays the connected signal on a semicircular scale during simulation. You can use the Half Gauge block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. The Half Gauge block provides an indication of the instantaneous value of the connected signal throughout simulation. You can modify the range of the Half Gauge block to fit your data. You can also customize the appearance of the Half Gauge block to provide more information about your signal. For example, you can color-code in-specification and out-of-specification ranges.

Connection

Dashboard blocks do not use ports to connect to signals. To connect a signal to a Dashboard block, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting signals of interest in your model.
- 3 Mark the option button next to the signal you want to display.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- If you turn off logging for a signal connected to a Dashboard block, the model stops sending data from that signal to the block. To view the signal again, reconnect the signal.
- External mode simulation does not support visualization Dashboard blocks.

Parameters

Connection — Select a signal to connect and display

signal connection options

Signal connection options. Populate the **Connection** table by selecting signals of interest in your model. Mark the check box next to the signal you want to display. Click **Apply** to connect the signal.

Minimum — Minimum tick mark value

0 (default) | scalar

A finite, real, double, scalar value specifying the minimum tick mark value for the scale. The minimum must be less than the value entered for the maximum.

Maximum — Maximum tick mark value

100 (default) | scalar

A finite, real, double, scalar value specifying the maximum tick mark value for the scale. The maximum must be greater than the value entered for the minimum.

Tick Interval — Interval between major tick marks

auto (default) | scalar

A finite, real, positive, integer, scalar value specifying the interval of major tick marks on the scale. When set to `auto`, the block automatically adjusts the tick interval based on the minimum and maximum values.

Scale Colors — Color indications on Gauge scale

colors for scale ranges

Color specifications for ranges on the scale. Press the + button to add a color. For each color added, specify the minimum and maximum values of the range where you want to display that color.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

Model Examples

See Also

Gauge | Linear Gauge | Quarter Gauge

Topics

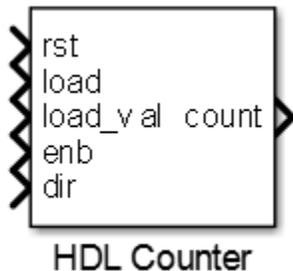
“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2015a

HDL Counter

Free-running or count-limited hardware counter



Library

HDL Coder / HDL Operations

Description

The HDL Counter block models a free-running or count-limited hardware counter that supports signed and unsigned integer and fixed-point data types.

The counter emits its value for the current sample time.

This block does not report wrap on overflow warnings during simulation. To report these warnings, see the `Simulink.restoreDiagnostic` reference page. The block does report errors due to wrap on overflow.

Control Ports

By default, the counter does not have input ports. Optionally, you can add control ports that enable, disable, load, reset or set the direction of the counter.

The table shows the priority of the control signals and how the counter value is updated in relation to the control signals.

Local reset, rst	Load trigger, load	Count enable, enb	Count direction, dir	Next Counter Value
1	–	–	–	initial value
0	1	–	–	load_val value
0	0	0	–	current value
0	0	1	1	current value + step value
0	0	1	0	current value - step value

Count direction

The **Step value** parameter and optional count direction port, `dir`, interact to determine the actual count direction.

dir Signal Value	Step Value Sign	Actual Count Direction
1	+ (positive)	Up
1	- (negative)	Down
0	+ (positive)	Down
0	- (negative)	Up

Parameters

Counter type

Counter behavior.

- **Free running (default):** The counter continues to increment or decrement by the **Step value** until reset.
- **Count limited:** The counter increments or decrements by the **Step value** until it is exactly equal to the **Count to value**.

Initial value

Counter value after reset. The default is 0.

Step value

Value added to counter at each sample time. The default is 1.

Count to value

When the count is exactly equal to **Count to value**, the count restarts at the **Initial value**. This option is available when **Counter type** is set to `Count limited`. The default is 100.

Count from

Specifies the parameter that sets the start value after rollover. When set to `Specify`, the **Count from value** parameter is the start value after rollover. The default is `Initial value`.

Count from value

Counter value after rollover when **Count from** is set to `Specify`. The default is 0.

Local reset port

When selected, creates a local reset port, `rst`.

Load ports

When selected, creates a load data port, `load_val`, and load trigger port, `load`.

Count enable port

When selected, creates a count enable port, `enb`.

Count direction port

When selected, creates a count direction port, `dir`.

Counter output data is

Output data type signedness. The default is `Unsigned`.

Word length

Bit width, including sign bit, for an integer counter; word length for a fixed-point data type counter. The minimum value if `Output data type` is `Unsigned` is 1, 2 if `Signed`. The maximum value is 125. The default is 8.

Fraction length

Fixed-point data type fraction length. The default is 0.

Sample time

Sample time. The default is 1.

This parameter is not available, and the block inherits its sample time from the input ports when any of these parameters is selected:

- **Local reset port**
- **Load ports**
- **Count enable port**
- **Count direction port**

Ports

The block has the following ports:

`rst`

Resets the counter value. Active-high.

This port is available when you select **Local reset port**.

Data type: Boolean

`load`

Sets the counter to the load value, `load_val`. Active-high.

This port is available when you select **Load ports**.

Data type: Boolean

`load_val`

Data value to load.

This port is available when you select **Load ports**.

Data type: Same as `count`.

`enb`

Enables counter operation. Active-high.

This port is available when you select **Count enable port**.

Data type: Boolean

`dir`

Count direction. This port interacts with **Step value** to determine count direction.

- 1: **Step value** is added to the current counter value to compute the next value.
- 0: **Step value** is subtracted from the current counter value to compute the next value.

This port is available when you select **Count direction port**.

Data type: Boolean

count

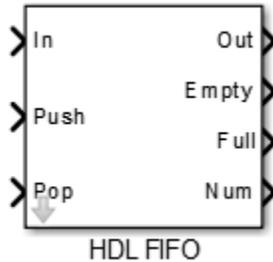
Counter value.

Data type: Determined automatically based on **Counter output data is**, **Word length**, and **Fraction length**.

Introduced in R2014a

HDL FIFO

Stores sequence of input samples in first in, first out (FIFO) register



Library

HDL Coder / HDL Operations

Description

The HDL FIFO block stores a sequence of input samples in a first in, first out (FIFO) register.

HDL Code Generation

For simulation results that match the generated HDL code, in the Solver pane of the Configuration Parameters dialog box, clear the checkbox for **Treat each discrete rate as a separate task**. When the checkbox is cleared, single-tasking mode is enabled.

If you simulate this block with **Treat each discrete rate as a separate task** selected, multitasking mode is enabled. The output data can update in the same cycle but in the generated HDL code, the output data is updated one cycle later.

Parameters

Register size

Specify the number of entries that the FIFO register can hold. The minimum is 4. The default is 10.

The ratio of output sample time to input sample time

Inputs (In, Push) and outputs (Out, Pop) can run at different sample times. Enter the ratio of output sample time to input sample time. Use a positive integer or $1/N$, where N is a positive integer. The default is 1.

For example:

- If you enter 2, the output sample time is twice the input sample time, meaning the outputs run slower.
- If you enter $1/2$, the output sample time is half the input sample time, meaning the outputs run faster.

The Full, Empty, and Num signals run at the faster rate.

Push onto full register

Response (Ignore, Error, or Warning) to a trigger received at the Push port when the register is full. The default is Warning.

Pop empty register

Response (Ignore, Error, or Warning) to a trigger received at the Pop port when the register is empty. The default is Warning.

Show empty register indicator port (Empty)

Enable the Empty output port, which is high (1) when the FIFO register is empty and low (0) otherwise.

Show full register indicator port (Full)

Enable the Full output port, which is high (1) when the FIFO register is full and low (0) otherwise.

Show number of register entries port (Num)

Enable the Num output port, which tracks the number of entries currently in the queue.

Ports

The block has the following ports:

In

Data input signal.

Push

Control signal. When this port receives a value of 1, the block pushes the input at the In port onto the end of the FIFO register.

Pop

Control signal. When this port receives a value of 1, the block pops the first element off the FIFO register and holds the Out port at that value.

Out

Data output signal.

Empty

The block asserts this signal when the FIFO register is empty. This port is optional.

Full

The block asserts this signal when the FIFO register is full. This port is optional.

Num

Current number of data values in the FIFO register. This port is optional.

If two or more of the control input ports are triggered in the same time step, the operations execute in the following order:

- 1 Pop
- 2 Push

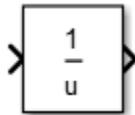
See Also

Dual Rate Dual Port RAM

Introduced in R2014a

HDL Reciprocal

Calculate reciprocal with Newton-Raphson approximation method



HDL Reciprocal

Library

HDL Coder / HDL Operations

Description

The HDL Reciprocal block uses the Newton-Raphson iterative method to compute the reciprocal of the block input. The Newton-Raphson method uses linear approximation to successively find better approximations to the roots of a real-valued function.

The reciprocal of a real number a is defined as a zero of the function:

$$f(x) = \frac{1}{x} - a$$

HDL Coder chooses an initial estimate in the range $0 < x_0 < \frac{2}{a}$ as this is the domain of convergence for the function.

To successively compute the roots of the function, specify the **Number of iterations** parameter in the Block Parameters dialog box. The process is repeated as:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i + (x_i - ax_i^2) = x_i \cdot (2 - ax_i)$$

$f'(x)$ is the derivative of the function $f(x)$.

Following table shows comparison of simulation behavior of HDL Reciprocal with Math Reciprocal block:

Math Reciprocal	HDL Reciprocal
Computes the reciprocal as $1/N$ by using the HDL divide operator (/) to implement the division.	Uses the Newton-Raphson iterative method. The block computes an approximate value of reciprocal of the block input and can yield different simulation results compared to the Math Reciprocal block. To match the simulation results with the Math Reciprocal block, increase the number of iterations for the HDL Reciprocal block.

Parameters

Number of iterations

Number of Newton-Raphson iterations. The default is 3.

Ports

The block has the following ports:

Input

- Supported data types: Fixed-point, integer (signed or unsigned), double, single
- Minimum bit width: 2
- Maximum bit width: 128

Output

Input data type	Output data type
double	double
single	single

Input data type	Output data type
built-in integer	built-in integer
built-in fixed-point	built-in fixed-point
$f_i(\text{value}, 0, \text{word_length}, \text{fraction_length})$	$f_i(\text{value}, 0, \text{word_length}, \text{word_length} - \text{fraction_length} - 1)$
$f_i(\text{value}, 1, \text{word_length}, \text{fraction_length})$	$f_i(\text{value}, 1, \text{word_length}, \text{word_length} - \text{fraction_length} - 2)$

See Also

Divide | Math Function

Introduced in R2014b

Hit Crossing

Detect crossing point



Library

Discontinuities

Description

The Hit Crossing block detects when the input reaches the **Hit crossing offset** parameter value in the direction specified by the **Hit crossing direction** property.

The block accepts one input of type double. If you select the **Show output port** check box, the block output indicates when the crossing occurs. If the input signal is exactly the value of the offset value after the hit crossing is detected, the block continues to output a value of 1. If the input signals at two adjacent points bracket the offset value (but neither value is exactly equal to the offset), the block outputs a value of 1 at the second time step. If the **Show output port** check box is *not* selected, the block ensures that the simulation finds the crossing point but does not generate output. If the input signal is constant and equal to the offset value, the block outputs 1 only if the **Hit crossing direction** property is set to either.

When the block's **Hit crossing direction** property is set to either, the block serves as an "Almost Equal" block, useful in working around limitations in finite mathematics and computer precision. Used for these reasons, this block might be more convenient than adding logic to your model to detect this condition.

When the block's **Hit crossing direction** property is set to either and the model uses a fixed-step solver, the block has the following behavior. If the output signal is 1, the block sets the output signal to 0 at the next time step, unless the input signal equals the offset value.

Data Type Support

The Hit Crossing block outputs a signal of type `Boolean` if Boolean logic signals are enabled (see “Implement logic signals as Boolean data (vs. double)”). Otherwise, the block outputs a signal of type `double`.

Parameters

Hit crossing offset

The value whose crossing is to be detected.

Hit crossing direction

The direction from which the input signal approaches the hit crossing offset for a crossing to be detected.

Show output port

If selected, draw an output port.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

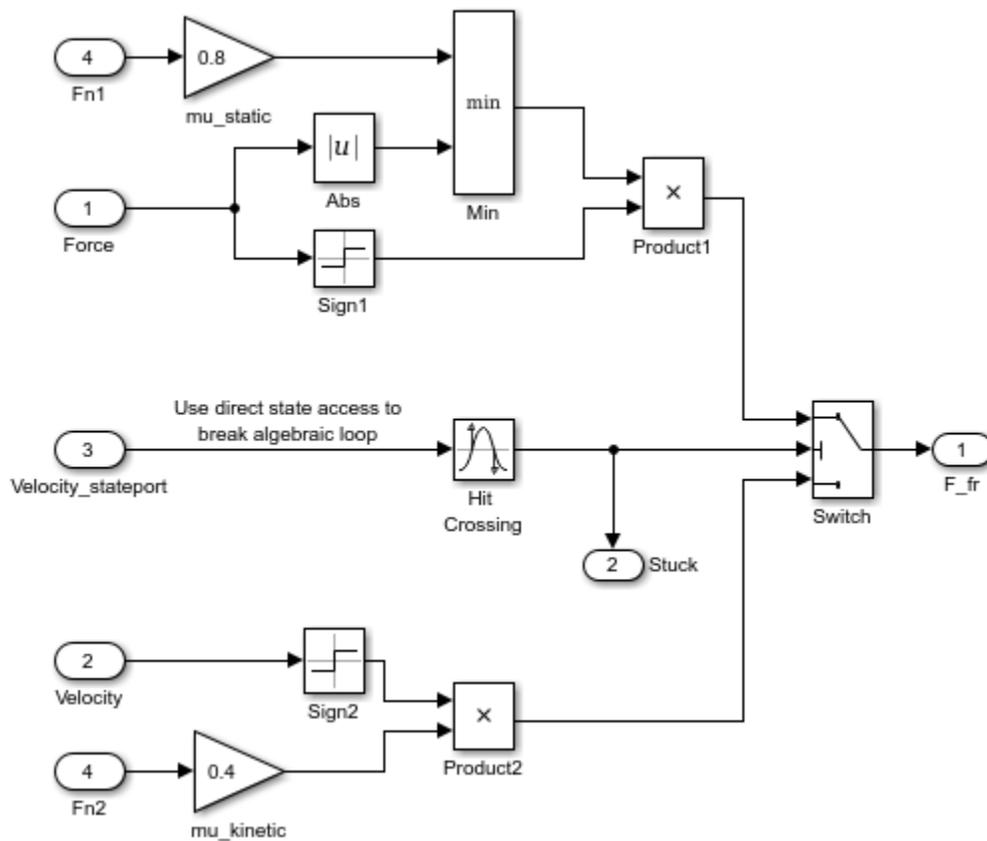
Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

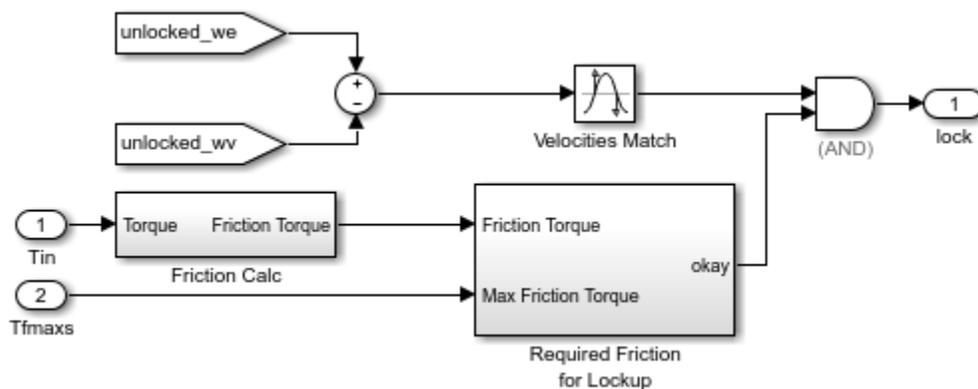
Examples

The `sldemo_hardstop` and `sldemo_clutch` models show how you can use the Hit Crossing block.

In the `sldemo_hardstop` model, the Hit Crossing block is in the Friction Model subsystem.



In the `sldemo_clutch` model, the Hit Crossing block is in the Friction Mode Logic/Lockup Detection subsystem.



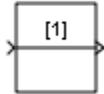
Characteristics

Data Types	Double
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

Introduced before R2006a

IC

Set initial value of signal



Library

Signal Attributes

Description

The IC block sets the initial condition of the signal at its input port, for example, the value of the signal at the simulation start time (t_{start}). The block does this by outputting the specified initial condition when you start the simulation, regardless of the actual value of the input signal. Thereafter, the block outputs the actual value of the input signal.

Note If an IC block has a nonzero sample time offset (t_{offset}), the IC block outputs its initial value at time t ,

$$t = n * t_{period} + t_{offset}$$

where n is the smallest integer such that $t \geq t_{start}$.

That is, the IC block outputs its initial value the first time blocks with sample time $[t_{period}, t_{offset}]$ execute, which can be after t_{start} .

The IC block is useful for providing an initial guess for the algebraic state variables in a loop. For more information, see “Algebraic Loops”.

Data Type Support

The IC block accepts and outputs signals of any Simulink built-in and fixed-point data type. The **Initial value** parameter accepts any built-in data type that Simulink supports. For more information, see “Data Types Supported by Simulink”.

Parameters

Initial value

Specify the initial value for the input signal.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1 . To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Examples

The following examples show how to use the IC block:

- `sldemo_bounce`
- `sldemo_hardstop`
- `sldemo_enginewc`

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	No

Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

If

Model if-else control flow



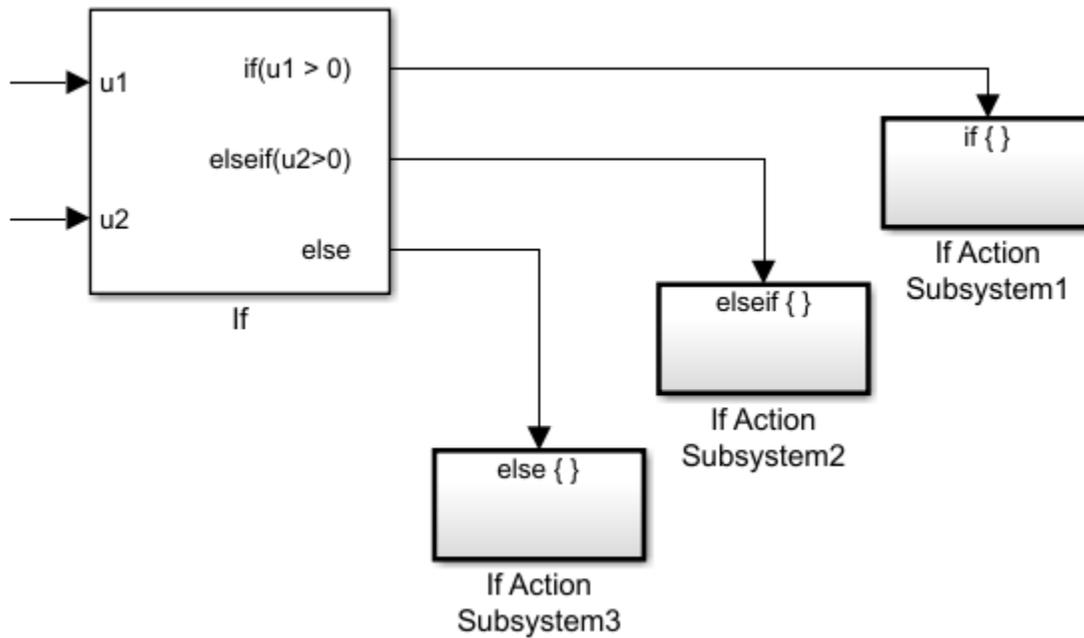
Library

Ports & Subsystems

Description

The If block, along with the If Action Subsystem block containing an Action Port, implements if-else logic.

The following shows a completed if-else control flow statement.



In this example, the inputs to the If block determine the values of conditions represented as output ports. Each output port is attached to an If Action Subsystem block. The conditions are evaluated top down starting with the `if` condition. If a condition is true, its If Action Subsystem is executed and the If block does not evaluate any remaining conditions.

The preceding if-else control flow statement can be represented by the following pseudocode.

```
if (u1 > 0) {
    If Action Subsystem1;
}
else if (u2 > 0){
    If Action Subsystem2;
}
else {
    If Action Subsystem3;
}
```

You construct a Simulink if-else control flow statement like the preceding example as follows:

- 1 Place an If block in the current system.
- 2 Open the dialog of the If block and enter as follows:
 - Enter the **Number of inputs** field with the required number of inputs necessary to define conditions for the `if-else` control flow statement.

Elements of vector inputs can be accessed for conditions using (row, column) arguments. For example, you can specify the fifth element of the vector `u2` in the condition `u2(5) > 0` in an **If expression** or **Elseif expressions** field.

- Enter the expression for the if condition of the `if-else` control flow statement in the **If expression** field.

This creates an if output port for the If block with a label of the form `if(condition)`. This is the only required If Action signal output for an If block.

- Enter expressions for any elseif conditions of the `if-else` control flow statement in the **Elseif expressions** field.

Use a comma to separate one condition from another. Entering these conditions creates an output port for the If block for each condition, with a label of the form `elseif(condition)`. elseif ports are optional and not required for operation of the If block.

- Check the **Show else condition** check box to create an else output port.

The else port is optional and not required for the operation of the If block.

- 3 Add If Action Subsystem blocks to connect to each of the if, else, and elseif ports.

These consist of a subsystem with an Action Port block. When you place an Action Port block inside each subsystem, an input port named Action is added to the subsystem.

- 4 Connect each if, else, and elseif port of the If block to the action port of an If Action Subsystem block.

When you make the connection, the icon for the If Action block is renamed to the type of the condition that it attaches to.

Note During simulation of an `if-else` control flow statement, the Action signal lines from the If block to the If Action subsystems turn from solid to dashed.

- 5 In each If Action Subsystem, enter the Simulink blocks appropriate to the body to be executed for the condition it handles.

Limitations

The If block has the following limitations:

- You cannot tune an if or elseif expression during simulation in Normal or Accelerator mode (see “How Acceleration Modes Work”), or in generated code. The If block does not support tunable parameters. To implement tunable if-else expressions, tune the expression outside the If block. For example, use the Relational operator block to evaluate the expression outside or add the tunable parameter as an input to the If block.
- The If block does not support custom storage classes. See “Custom Storage Classes” (Embedded Coder) in the Embedded Coder documentation.
- The **If expression** and **Elseif expressions** cannot accept certain operators, such as +, -, *, and /. See If Expression and Elseif Expressions in “Parameters” on page 1-744

Data Type Support

Inputs u_1, u_2, \dots, u_n can be scalars or vectors of any built-in Simulink data type and must all be of the same data type. The inputs cannot be of any user-defined type, such as an enumerated type. Outputs from the `if`, `else`, and `elseif` ports are action signals to If Action Subsystem blocks that you create by using Action Port blocks and subsystems.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Number of inputs

Specify the number of inputs to the If block. These appear as data input ports labeled with a 'u' character followed by a number, 1, 2, . . . , n, where n equals the number of inputs that you specify.

If expression

Specify the condition for the `if` output port. This condition appears on the If block adjacent to the `if` output port. The **If expression** can include only the operators `<`, `<=`, `==`, `~=`, `>`, `>=`, `&`, `|`, `~`, `()`, unary-minus, and cannot include

operators such as `+`, `-`, `*`, `/`, and `^`. The If Action Subsystem attached to the `if` port executes if its condition is true. The expression must not contain data type expressions, for example, `int8(6)`, and must not reference workspace variables whose data type is other than `double` or `single`.

Note There are limitations to the tunability of the **If expression**. See “Limitations” on page 1-744 for more information.

Elseif expressions

Specify a character vector list of `elseif` conditions delimited by commas. These conditions appear below the `if` port and above the `else` port when you select the **Show else condition** check box. **Elseif expressions** can include only the operators `<`, `<=`, `==`, `~=`, `>`, `>=`, `&`, `|`, `~`, `()`, unary-minus, and cannot include operators such as `+`, `-`, `*`, `/`, and `^`. The If Action Subsystem attached to an `elseif` port executes if its condition is true and all of the `if` and `elseif` conditions are false. The expression must not contain data type expressions, for example, `int8(6)`, and must not reference workspace variables whose data type is other than `double` or `single`.

Note There are limitations to the tunability of **Elseif expressions**. See “Limitations” on page 1-744 for more information.

Show else condition

If you select this check box, an `else` port is created. The If Action subsystem attached to the `else` port executes if the `if` port and all the `elseif` ports are false.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

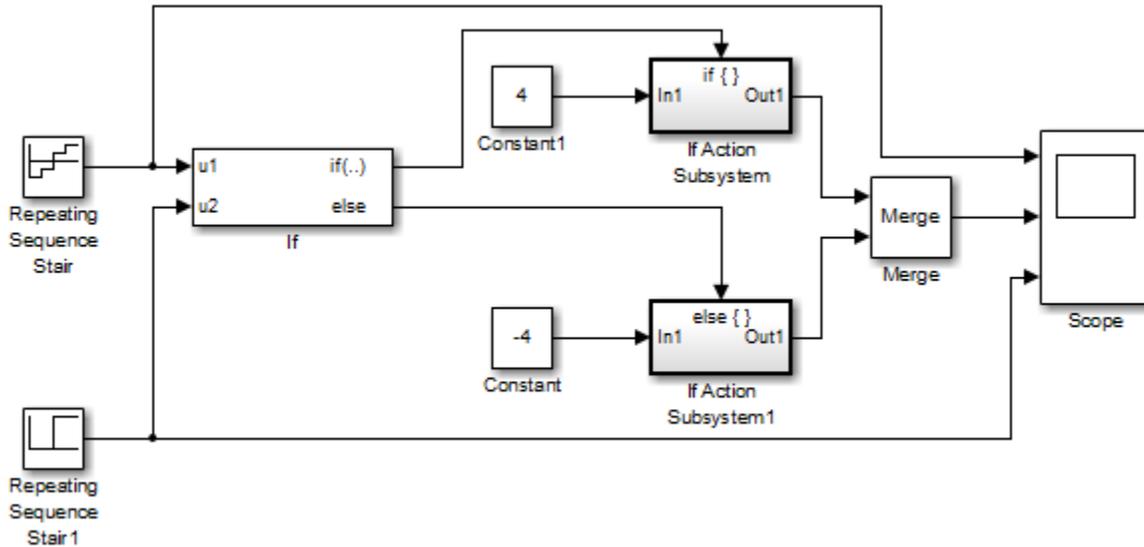
Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Examples

The If block does not directly support fixed-point data types. However, you can use the Compare To Constant block to work around this limitation.

For example, consider the following floating-point model:

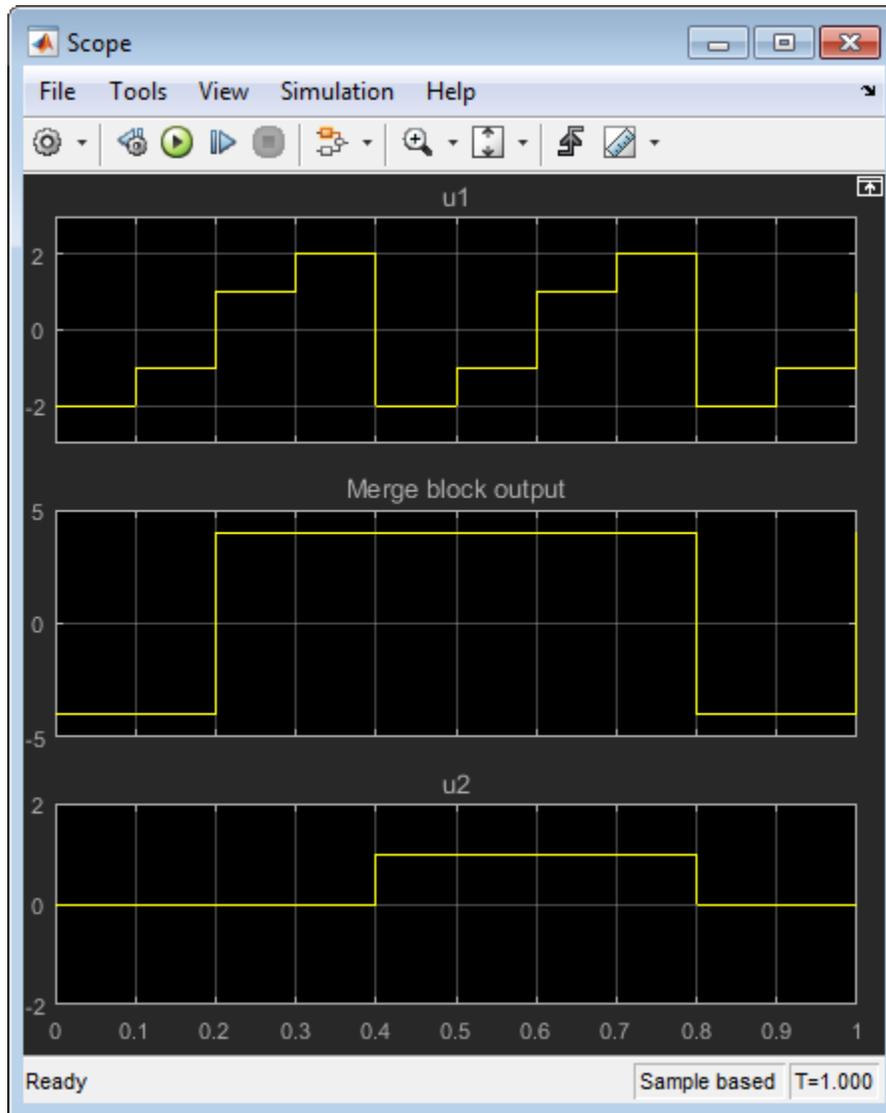


In this model, the If Action subsystems use their default configurations. The block and simulation parameters for the model are set to their default values except as follows:

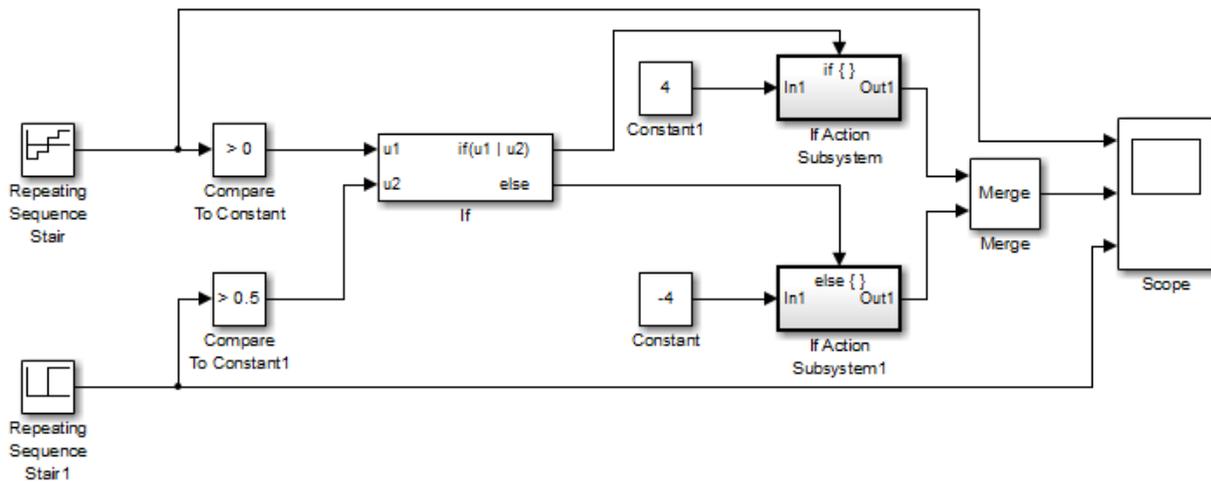
Block or Dialog Box	Parameter	Setting
Configuration Parameters — Solver pane	Start time	0.0
	Stop time	1.0
	Type	Fixed-step
	Solver	discrete (no continuous states)
	Fixed-step size	0.1
Repeating Sequence Stair	Vector of output values	[-2 -1 1 2].'

Block or Dialog Box	Parameter	Setting
Repeating Sequence Stair1	Vector of output values	[0 0 0 0 1 1 1 1].'
If	Number of inputs	2
	If expression	$(u1 > 0) \mid (u2 > 0.5)$
	Show else condition	selected
Constant	Constant value	-4
Constant1	Constant value	4
Scope	Number of axes	3
	Time range	1

For this model, if input $u1$ is greater than 0 or input $u2$ is greater than 0.5, the output is 4. Otherwise, the output is -4. The Scope block shows the output, $u1$, and $u2$:



You can implement this block diagram as a model with fixed-point data types:



The Repeating Sequence Stair blocks now output fixed-point data types.

The Compare To Constant blocks implement two parts of the **If expression** that is used in the If block in the floating-point version of the model, $(u1 > 0)$ and $(u2 > 0.5)$. The OR operation, $(u1 | u2)$, can still be implemented inside the If block. For a fixed-point model, the expression must be partially implemented outside of the If block as it is here.

The block and simulation parameters for the fixed-point model are the same as for the floating-point model with the following exceptions and additions:

Block	Parameter	Setting
Compare To Constant	Operator	>
	Constant value	0
	Output data type mode	Boolean
	Enable zero-crossing detection	off
Compare To Constant1	Operator	>
	Constant value	0.5
	Output data type mode	Boolean
	Enable zero-crossing detection	off

Block	Parameter	Setting
If	Number of inputs	2
	If expression	u1 u2

Characteristics

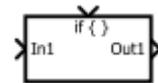
Data Types	Double Single Boolean Base Integer
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

Introduced before R2006a

If Action Subsystem

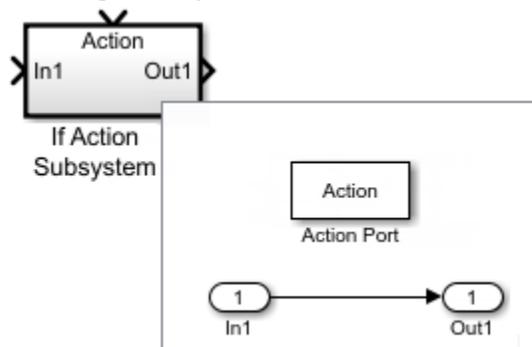
Subsystem whose execution is triggered by If block

Library: Ports & Subsystems



Description

The If Action Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem whose execution is triggered by an If block.



All blocks in an If Action Subsystem block must run at the same rate as the driving If block. You can achieve this requirement by setting each block's sample time parameter to be either inherited (-1) or the same value as the If block sample time.

Ports

Input

In_n — Signal input to a subsystem block

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

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

Action — Control signal input to a subsystem block

`scalar` | `vector` | `matrix`

Placing an Action Port block in a subsystem block adds an external input port to the block and changes the block to an If Action Subsystem.

Dot-dash lines from an If block to an If Action Subsystem block represent *Action* signals. Action signals control the execution of a subsystem.

Data Types: `action`

Output

out — Signal output from a subsystem

`scalar` | `vector` | `matrix`

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

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

See Also

Blocks

Action Port | If | Subsystem | Switch Case | Switch Case Action Subsystem

Topics

“Use Control Flow Logic”

“Action Subsystem”

Introduced before R2006a

In Bus Element

Select bus element or entire bus that is connected to subsystem input port

Library: Simulink / Ports & Subsystems
Simulink / Sources

InBus.signal1 

Description

Note This block has two different names, depending on the library in which it appears. The functionality of both blocks is the same.

- In the Sources library and the Ports & Subsystems library — In Bus Element
 - In the Signal Routing library — Bus Element In
-

Select an element from a bus (or the entire bus) that is connected to the input port of the subsystem. This block integrates into one block the functionality of using an Inport block and a Bus Selector block. The In Bus Element block is of the Inport block type. There are no specifications allowed on an In Bus Element block, which supports only an inherited workflow. You cannot use the Block Parameters dialog box of an In Bus Element block to specify bus element attributes, such as data type or dimensions.

To work with buses at subsystem interfaces, consider using In Bus Element and Out Bus Element blocks. This bus element port block combination:

- Reduces signal line complexity and clutter in a block diagram.
- Makes it easy to change the interface incrementally.
- Allows access to a bus element closer to the point of usage.
 - For input, avoid a duplicate Inport blocks and a Bus Selector, Goto, and From block configuration.
 - For output, avoid a Goto, From, and Bus Creator block configuration.

The In Bus Element block selects signals from a subsystem input port. Feed the output of the In Bus Element block to another block in the subsystem.

For bus input signals, either specify the signal that you want to select from the input port or to pass through the whole bus signal, leave the element empty. For a nonbus input signal for a subsystem, leave the element section of the block icon text empty. The block passes through the value of the nonbus signal. To select multiple signals from an input bus signal, create multiple In Bus Element blocks, one for each selected signal.

You can specify the background color for bus element port blocks, using the Block Parameters dialog box **Set color** option. This action sets the color of blocks associated with selected elements, or to all blocks if you do not select elements.

Ports

The block does not have an input port. Use the Block Parameters dialog box or Property Inspector to specify the subsystem input port from which the block receives its input signal.

Output

Port_1 — Pass selected signal to another block
signal

The output port passes the value of the selected input signal to another block. The signal can have a real or complex value of any data type that Simulink supports.

Parameters

Port name — Name of associated subsystem input port
InBus (default) | text

Specify a name for a subsystem port. That name appears on the Subsystem and In Bus Element block icons. If you specify a port name, that name cannot already be in use by another In Bus Element block or port. All In Bus Element blocks that access the same subsystem input port reflect the port name that you specify.

Programmatic Use

Block Parameter: PortName

Type: text

Default: InBus

Port number — Position in which port appears for subsystem input ports

1 (default) | integer

Specify the order in which the port appears on the subsystem, with 1 being the top port, 2 the second port down, and so on.

- If you specify a number that exceeds the number of subsystem input ports, new ports are added above the port associated with the In Bus Element block.
- If you add an In Bus Element block that creates another subsystem input port, the port number is the next available number.
- If you delete all In Bus Element blocks associated with a port, other port numbers are renumbered so that the blocks are in sequence and that no numbers are omitted.

Programmatic Use

Block Parameter: Port

Value: integer

Default: 1

Model Examples

Block Characteristics

Data Types	double single Boolean base integer fixed point enumerated bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes

Extended Capabilities

C/C++ Code Generation

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

Actual data type or capability support depends on block implementation.

Fixed-Point Conversion

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

Actual data type or capability support depends on block implementation.

See Also

Blocks

Bus Selector | Inport | Out Bus Element

Topics

“Simplify Subsystem Bus Interfaces”
“Composite Signal Techniques”
“Select a Composite Signal Technique”
“Getting Started with Buses”

Introduced in R2017a

Increment Real World

Increase real world value of signal by one

Library: Simulink / Additional Math & Discrete / Additional
Math: Increment — Decrement



Description

The Increment Real World block increases the real world value of the signal by one.

Overflows always wrap.

Ports

Input

Port_1 (u) — Input signal

Scalar

The Increment Real World block accepts signals of the data types listed below.

For more information, see “Data Types Supported by Simulink”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` |
`fixed point`

Output

Port_1 (y) — Calculated output signal

Scalar

Output data type always matches input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Block Characteristics

Data Types	<code>double</code> <code>single</code> <code>base integer</code> <code>fixed point</code>
Sample Time	<code>Inherit</code>
Direct Feedthrough	<code>Yes</code>
Multidimensional Signals	<code>Scalar</code>
Variable-Size Signals	<code>Yes</code>
Zero-Crossing Detection	<code>No</code>

Extended Capabilities

C/C++ Code Generation

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

The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.

HDL Code Generation

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

This block has a single, default HDL architecture.

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.

See also “ConstrainedOutputPipeline” (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.

See also “InputPipeline” (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.

See also “OutputPipeline” (HDL Coder).

Fixed-Point Conversion

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

See Also

Decrement Real World | Increment Stored Integer

Introduced before R2006a

Increment Stored Integer

Increase stored integer value of signal by one



Library

Additional Math & Discrete / Additional Math: Increment - Decrement

Description

The Increment Stored Integer block increases the stored integer value of a signal by one.

Floating-point signals also increase by one, and overflows always wrap.

Data Type Support

The Increment Stored Integer block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
------------	----------------------------------------------

Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

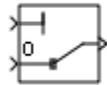
See Also

Decrement Stored Integer, Increment Real World

Introduced before R2006a

Index Vector

Switch output between different inputs based on value of first input



Library

Signal Routing

Description

The Index Vector block is an implementation of the Multiport Switch block. See Multiport Switch for more information.

Characteristics

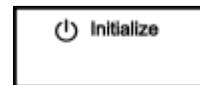
Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	Yes

Introduced before R2006a

Initialize Function

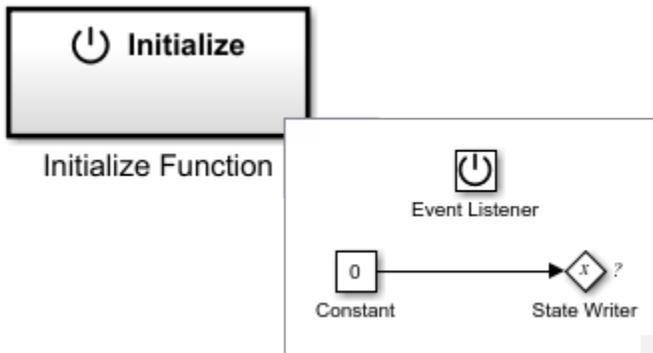
Executes contents on a model initialize event

Library: User-Defined Functions



Description

The Initialize Function block is a pre-configured subsystem block that executes on a model initialize event. By default, the Initialize Function block includes an Event Listener block with **Event** set to `Initialize`, a Constant block with **Constant value** set to 0, and a State Writer block.



Replace the Constant block with blocks that generate the state value for the State Writer block.

For a list of unsupported blocks and features, see “Initialize, Reset, and Terminate Function Limitations”.

The input and output ports of a component containing Initialize Function and Terminate Function blocks must connect to input and output port blocks.

The code generated from this block is part of the `model_initialize` function that is called once at the beginning of model execution.

See Also

[Event Listener](#) | [Reset Function](#) | [State Reader](#) | [State Writer](#) | [Terminate Function](#)

Topics

“Create Model to Initialize, Reset, and Terminate State”

“Create Test Harness to Generate Function Calls”

“Generate Code That Responds to Initialize, Reset, and Terminate Events” (Simulink Coder)

Inport

Create input port for subsystem or external input



Library

Ports & Subsystems, Sources

Description

Inport blocks are the links from outside a system into the system.

Simulink software assigns Inport block port numbers according to these rules:

- It automatically numbers the Inport blocks within a top-level system or subsystem sequentially, starting with 1.
- If you add an Inport block, the label is the next available number.
- If you delete an Inport block, other port numbers are automatically renumbered to ensure that the Inport blocks are in sequence and that no numbers are omitted.
- If you copy an Inport block into a system, its port number is *not* renumbered unless its current number conflicts with an Inport block already in the system. If the copied Inport block port number is not in sequence, renumber the block. Otherwise, you get an error message when you run the simulation or update the block diagram.

You can specify the dimensions of the input to the Inport block using the **Port dimensions** parameter. Entering a value of -1 lets Simulink determine the port dimension.

The **Sample time** parameter is the rate at which the signal is coming into the system. A value of -1 causes the block to inherit its sample time from the block driving it. You might need to set this parameter for:

- Inport blocks in a top-level system.
- Models with blocks where Simulink cannot determine the sample time, but these blocks drive Inport blocks.

For more information, see “Specify Sample Time”.

Inport Blocks in a Top-Level System

You can use an Inport block in a top-level system to:

- Supply external inputs from the workspace using one of these approaches. If no external outputs are supplied, then the default output is the ground value.
 - Use the **Configuration Parameters > Data Import/Export > Input** parameter. See “Load Data to Root-Level Input Ports”.

Tip To import many signals to root-level input ports, consider using the Root Inport Mapper tool. For more information, see “Map Data Using Root Inport Mapper Tool”.

- Use the `ut` argument of the `sim` command (see `sim`) to specify the inputs.
- Provide a means for perturbation of the model by the `linmod` and `trim` analysis functions.
 - Use Inport blocks to inject inputs into the system. See “Linearizing Models”.
- To load logged signal data using root Inport blocks, you can use the `createInputDataset` function to create a `Dataset` object that contains elements that correspond to root-level Inport blocks in the model.

Inport Blocks in a Subsystem

Inport blocks in a subsystem represent inputs to the subsystem. A signal arriving at an input port on a Subsystem block flows out of the associated Inport block in that subsystem. The Inport block associated with an input port on a Subsystem block is the block whose **Port number** parameter matches the relative position of the input port on the Subsystem block. For example, the Inport block whose **Port number** parameter is 1 gets its signal from the block connected to the topmost port on the Subsystem block.

If you renumber the **Port number** of an Inport block, the block becomes connected to a different input port, although the block continues to receive its signal from the same block outside the subsystem.

The Inport block name appears in the Subsystem icon as a port label. To suppress display of the label, select the Inport block and choose **Format > Hide Name**.

Inport blocks inside a subsystem support signal label propagation, but root-level Inport blocks do not.

You can use a subsystem inport to supply fixed-point data in a structure or any other format.

Tip For models that include bus signals composed of many bus elements that feed subsystems, consider using the In Bus Element and Out Bus Element blocks. You can use these bus element port blocks instead of Inport with Bus Selector blocks for inputs, and Outport with Bus Creator blocks for outputs. These bus element port blocks:

- Reduce signal line complexity and clutter in a block diagram.
- Make it easier to change the interface incrementally.
- Allow access to a bus element closer to the point of usage, avoiding the use of a Bus Selector and Goto block configuration.

The In Bus Element block is of block type Inport. However, there are no specifications allowed on bus element port blocks, which support inherited workflows. You cannot use the Block Parameters dialog box of an In Bus Element block to specify bus element attributes, such as data type or dimensions.

Creating Duplicate Inports

You can create any number of duplicates of an Inport block. The duplicates are graphical representations of the original intended to simplify block diagrams by eliminating unnecessary lines. The duplicate has the same port number, properties, and output as the original. Changing properties of a duplicate changes properties of the original and vice versa.

To create a duplicate of an Inport block:

- 1 In the block diagram, select the block that you want to duplicate.

- 2 In the Model Editor menu bar, select **Edit > Copy**.
- 3 In the block diagram, place your cursor where you want to place the duplicate.
- 4 Select **Edit > Paste Duplicate Inport**.

Connecting Buses to Root-Level Inports

If you want a root-level Inport of a model to produce a bus signal, you must set the **Data type** parameter to the name of a bus object that defines the bus that the Inport produces. For more information, see “When to Use Bus Objects”.

Data Type Support

The Inport block accepts complex or real signals of any data type that Simulink supports, including fixed-point and enumerated data types. The Inport block also accepts a bus object as a data type.

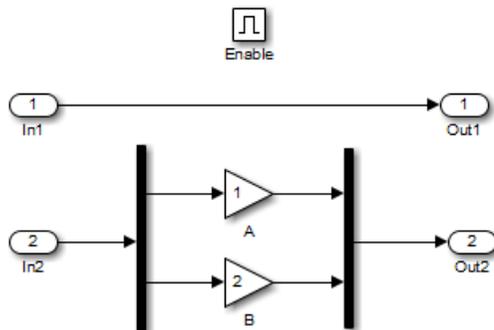
Note If you specify a bus object as the data type for this block, do not set the minimum and maximum values for bus data on the block. Simulink ignores these settings. Instead, set the minimum and maximum values for bus elements of the bus object specified as the data type. The values should be finite real double scalar.

For information on the Minimum and Maximum properties of a bus element, see `Simulink.BusElement`.

For more information, see “Data Types Supported by Simulink”.

The numeric and data types of the block output are the same as those of its input. You can specify the signal type and data type of an external input to a root-level Inport block using the **Signal type** and **Data type** parameters.

The elements of a signal array connected to a root-level Inport block must be of the same numeric and data types. Signal elements connected to a subsystem input port can be of differing numeric and data types, except in the following circumstance: If the subsystem contains an Enable, Trigger, or Atomic Subsystem block and the input port, or an element of the input port, connects directly to an output port, the input elements must be of the same type. For example, consider the following enabled subsystem:



In this example, the elements of a signal vector connected to In1 must be of the same type. The elements connected to In2, however, can be of differing types.

Parameters

- “Port number” on page 1-770
- “Icon display” on page 1-770
- “Latch input by delaying outside signal” on page 1-771
- “Latch input for feedback signals of function-call subsystem outputs” on page 1-772
- “Interpolate data” on page 1-773
- “Connect Input” on page 1-774
- “Output function call” on page 1-774
- “Minimum” on page 1-774
- “Maximum” on page 1-775
- “Data type” on page 1-776
- “Show data type assistant” on page 1-777
- “Mode” on page 1-777
- “Data type override” on page 1-779
- “Signedness” on page 1-779
- “Word length” on page 1-780
- “Scaling” on page 1-780

- “Fraction length” on page 1-781
- “Slope” on page 1-781
- “Bias” on page 1-782
- “Output as nonvirtual bus” on page 1-782
- “Lock output data type setting against changes by the fixed-point tools” on page 1-783
- “Unit (e.g., m, m/s², N*m)” on page 1-783
- “Port dimensions (-1 for inherited)” on page 1-784
- “Variable-size signal” on page 1-784
- “Sample time (-1 for inherited)” on page 1-785
- “Signal type” on page 1-785

Port number

Specify the port number of the block.

Default: 1

This parameter controls the order in which the port that corresponds to the block appears on the parent subsystem or model block.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Icon display

Specify the information to be displayed on the icon of this input port.

Default: Port number

Signal name

Display the name of the signal connected to this port (or signals if the input is a bus).

Port number

Display port number of this port.

Port number and signal name

Display both the port number and the names of the signals connected to this port.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Latch input by delaying outside signal

Output the value of the input signal at the previous time step.

Default: Off

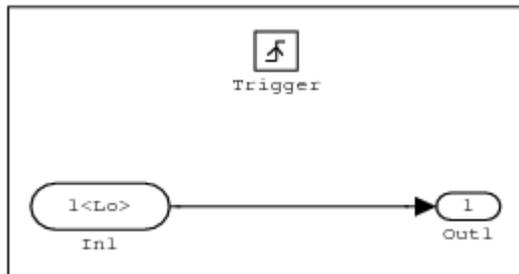
On

Output the value of the input signal at the previous time step.

Off

Do not output the value of the input signal at the previous time step.

- This option applies only to triggered subsystems and is enabled only if the Inport block resides in a triggered subsystem.
- Selecting this check box enables Simulink to resolve data dependencies among triggered subsystems that are part of a loop.
- Type `sl_subsys_semantics` at the MATLAB prompt for examples using latched inputs with triggered subsystems.
- The Inport block indicates that this option is selected by displaying `<Lo>`.



See “Block-Specific Parameters” on page 6-128 for the command-line information.

Latch input for feedback signals of function-call subsystem outputs

Latch the value of the input to this subsystem and prevent this value from changing during the execution of the subsystem. For a single function call that is branched to invoke multiple function-call subsystems, this option allows you to break a loop formed by a signal fed back from one of these function-call subsystems into the other. A second functionality of this option is to prevent any change to the values of a feedback signal from a function-call subsystem that is invoked during the execution of this subsystem.

Default: Off

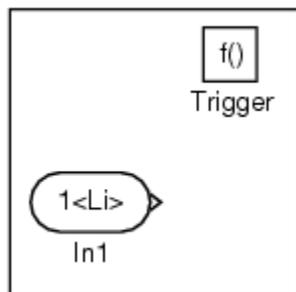
On

Latch the input value.

Off

Do not latch the input value.

- This parameter applies only to function-call subsystems and is enabled only if the Inport block resides in a function-call subsystem.
- This parameter ensures that the subsystem inputs, including those generated within the subsystem's context, do not change during execution of the subsystem.
- The Inport block indicates that this option is selected by displaying ``.



See “Block-Specific Parameters” on page 6-128 for the command-line information.

Interpolate data

When loading data from the workspace, cause the block to linearly interpolate and extrapolate output at time steps for which no corresponding data exists.

To load discrete signal data from the workspace, in the Inport block dialog box:

- 1 Set the **Sample time** parameter to a discrete value, such as 2.
- 2 Clear the **Interpolate data** parameter.

Specifying the discrete sample time causes the simulation to have hit times exactly at those instances when the discrete data is sampled. You only need to specify the data values, not time values.

Turning interpolation off avoids unexpected data values at other simulation time points as a result of double precision arithmetic processing. For more information, see “Load Data to Test a Discrete Algorithm”.

Default: On

On

When loading data from the workspace, cause the block to linearly interpolate and extrapolate output at time steps for which no corresponding data exists.

Off

When loading data from the workspace, do not cause the block to linearly interpolate or extrapolate output at time steps for which no corresponding data exists. Simulink uses the following interpolation and extrapolation:

- For time steps between the first specified data point and the last specified data point — zero-order hold
- For time steps before the first specified data point and after the last specified data point — ground value
- For variable-sized signals for time steps before the first specified data point — a NaN is logged for single or double data types and ground for other data types. For time steps after the last specified data point, uses ground values.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Connect Input

To import, visualize, and map signal and bus data to root-level inports, click this button. The Root Inport Mapper tool displays.

This button appears only if this block is a root inport block.

Output function call

Specify that the input signal is outputting a function-call trigger signal.

Default: Off



On

Input signal is a function-call trigger signal.



Off

Input signal is not a function-call trigger signal.

- Select this option if it is necessary for a current model to accept a function-call trigger signal when referenced in the top model.
- This feature is limited to an asynchronous function call.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Minimum

Specify the minimum value that the block should output.

Default: [] (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum

values for bus elements of the bus object specified as the data type. For information on the `Minimum` property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Maximum

Specify the maximum value that the block should output.

Default: [] (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the `Maximum` property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Data type

Specify the output data type of the external input.

Default: `Inherit: auto`

`Inherit: auto`

A rule that inherits a data type

`double`

Data type is `double`.

`single`

Data type is `single`.

`int8`

Data type is `int8`.

`uint8`

Data type is `uint8`.

`int16`

Data type is `int16`.

`uint16`

Data type is `uint16`.

`int32`

Data type is `int32`.

`uint32`

Data type is `uint32`.

`boolean`

Data type is `boolean`.

`fixdt(1,16,0)`

Data type is fixed point `fixdt(1,16,0)`.

```
fixdt(1,16,2^0,0)
```

Data type is fixed point `fixdt(1,16,2^0,0)`.

```
Enum: <class name>
```

Data type is enumerated, for example, `Enum: Basic Colors`.

```
Bus: <object name>
```

Data type is a bus object.

```
<data type expression>
```

The name of a data type object, for example `Simulink.NumericType`

Do not specify a bus object as the expression.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Mode

Select the category of data to specify.

Default: `Inherit`

```
Inherit
```

Inheritance rule for data types. Selecting `Inherit` enables a second menu/text box to the right.

```
Built in
```

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`

Fixed point

Fixed-point data types.

Enumerated

Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.

Bus object

Bus object. Selecting `Bus` enables a **Bus object** parameter to the right, where you enter the name of a bus object that you want to use to define the structure of the bus. If you need to create or change a bus object, click **Edit** to the right of the **Bus object** field to open the Simulink Bus Editor. For details, see “Create Bus Objects with the Bus Editor”.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Do not specify a bus object as the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data as signed.

`Unsigned`

Specify the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope` and `bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > Slope and bias enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Output as nonvirtual bus

Output a nonvirtual bus.

Default: Off



On

Output a nonvirtual bus.



Off

Output a virtual bus.

- All signals in a nonvirtual bus must have the same sample time, even if the elements of the associated bus object specify inherited sample times. Any bus operation that would result in a nonvirtual bus that violates this requirement generates an error. For referenced models, buses are single-rate. For details, see “Connect Multirate Buses to Referenced Models”.
- For the top model in a model reference hierarchy, code generation creates a C structure to represent the bus signal output by this block.
- For referenced models, select this option to create a C structure. Otherwise, code generation creates an argument for each leaf element of the bus used in the referenced model.

Selecting **Data type** > Bus: <object name> enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: LockScale

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Unit (e.g., m, m/s², N*m)

Specify physical unit of the input signal to the block.

Default: inherit

To specify a unit, begin typing in the text box. As you type, the parameter displays potential matching units. For a list of supported units, see Allowed Unit Systems.

To constrain the unit system, click the link to the right of the parameter:

- If a Unit System Configuration block exists in the component, its dialog box opens. Use that dialog box to specify allowed and disallowed unit systems for the component.
- If a Unit System Configuration block does not exist in the component, the model Configuration Parameters dialog box displays. Use that dialog box to specify allowed and disallowed unit systems for the model.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Port dimensions (-1 for inherited)

Specify the dimensions of the input signal to the block.

Default: -1

Valid values are:

-1	Dimensions are inherited from input signal
n	Vector signal of width n accepted
[m n]	Matrix signal having m rows and n columns accepted

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Variable-size signal

Specify the type of signals allowed into this port.

Default: `Inherit`

`Inherit`

Allow variable-size and fixed-size signals.

`No`

Do not allow variable-size signals.

`Yes`

Allow only variable-size signals.

When the signal at this port is a variable-size signal, the **Port dimensions** parameter specifies the maximum dimensions of the signal.

Parameter: VarSizeSig

Type: character vector

Value: 'Inherit' | 'No' | 'Yes'

Default: 'Inherit'

Sample time (-1 for inherited)

Specify the time interval between samples.

Default: -1

To inherit the sample time, set this parameter to -1.

See “Specify Sample Time” in the online documentation for more information.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Signal type

Specify the numeric type of the external input.

Default: auto

auto

Accept either `real` or `complex` as the numeric type.

real

Specify the numeric type as a real number.

complex

Specify the numeric type as a complex number.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Outport

`createInputDataset`

Asynchronous Task Specification

“Load Data to Root-Level Input Ports”

“Map Data Using Root Inport Mapper Tool”

“Load Big Data for Simulations”

Introduced before R2006a

Integrator, Integrator Limited

Integrate signal



Library

Continuous

Description

The Integrator block outputs the value of the integral of its input signal with respect to time.

The Integrator Limited block is identical to the Integrator block with the exception that the output of the block is limited based on the upper and lower saturation limits. See “Limiting the Integral” on page 1-789 for details.

Simulink treats the Integrator block as a dynamic system with one state. The block dynamics are given by:

$$\begin{cases} \dot{x}(t) = u(t) \\ y(t) = x(t) \end{cases} \quad x(t_0) = x_0$$

where:

- u is the block input.
- y is the block output.
- x is the block state.
- x_0 is the initial condition of x .

While these equations define an exact relationship in continuous time, Simulink uses numerical approximation methods to evaluate them with finite precision. Simulink can

use a number of different numerical integration methods to compute the Integrator block's output, each with advantages in particular applications. Use the **Solver** pane of the Configuration Parameters dialog box (see “Solver Pane”) to select the technique best suited to your application.

The selected solver computes the output of the Integrator block at the current time step, using the current input value and the value of the state at the previous time step. To support this computational model, the Integrator block saves its output at the current time step for use by the solver to compute its output at the next time step. The block also provides the solver with an initial condition for use in computing the block's initial state at the beginning of a simulation. The default value of the initial condition is 0. Use the block parameter dialog box to specify another value for the initial condition or create an initial value input port on the block.

Use the parameter dialog box to:

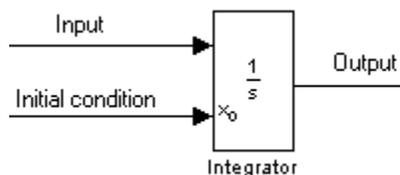
- Define upper and lower limits on the integral
- Create an input that resets the block's output (state) to its initial value, depending on how the input changes
- Create an optional state output so that the value of the block's output can trigger a block reset

Use the Discrete-Time Integrator block to create a purely discrete system.

Defining Initial Conditions

You can define the initial conditions as a parameter on the block dialog box or input them from an external signal:

- To define the initial conditions as a block parameter, specify the **Initial condition source** parameter as `internal` and enter the value in the **Initial condition** field.
- To provide the initial conditions from an external source, specify the **Initial condition source** parameter as `external`. An additional input port appears under the block input.



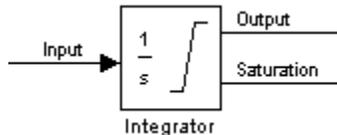
Note If the integrator limits its output (see “Limiting the Integral” on page 1-789), the initial condition must fall inside the integrator’s saturation limits. If the initial condition is outside the block saturation limits, the block displays an error message.

Limiting the Integral

To prevent the output from exceeding specifiable levels, select the **Limit output** check box and enter the limits in the appropriate parameter fields. This action causes the block to function as a limited integrator. When the output reaches the limits, the integral action is turned off to prevent integral wind up. During a simulation, you can change the limits but you cannot change whether the output is limited. The block determines output as follows:

- When the integral is less than or equal to the **Lower saturation limit**, the output is held at the **Lower saturation limit**.
- When the integral is between the **Lower saturation limit** and the **Upper saturation limit**, the output is the integral.
- When the integral is greater than or equal to the **Upper saturation limit**, the output is held at the **Upper saturation limit**.

To generate a signal that indicates when the state is being limited, select the **Show saturation port** check box. A saturation port appears below the block output port.



The signal has one of three values:

- 1 indicates that the upper limit is being applied.
- 0 indicates that the integral is not limited.
- -1 indicates that the lower limit is being applied.

When you select this check box, the block has three zero crossings: one to detect when it enters the upper saturation limit, one to detect when it enters the lower saturation limit, and one to detect when it leaves saturation.

Note For the Integrator Limited block, by default, **Limit output** is selected, **Upper saturation limit** is set to 1, and **Lower saturation limit** is set to 0.

Wrapping Cyclic States

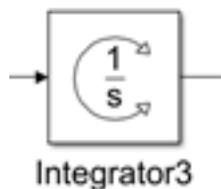
Several physical phenomena are cyclic, periodic, or rotary in nature. Objects or machinery that exhibit rotational movement and oscillators are examples of such phenomena.

Modeling these phenomena in Simulink involves integrating the rate of change of the periodic or cyclic signals to obtain the state of the movement.

The drawback with this approach, however, is that over long simulation time spans, the states representing periodic or cyclic signals integrate to large values. Further, computing the sine or cosine of these signals takes an increasingly large amount of time because of angle reduction. The large signals values also negatively impact solver performance and accuracy.

One approach for overcoming this drawback is to reset the angular state to 0 when it reaches 2π (or to $-\pi$ when it reaches π , for numerical symmetry). This approach improves the accuracy of sine and cosine computations and reduces angle reduction time. But it also requires zero-crossing detection and introduces solver resets, which slow down the simulation for variable step solvers, particularly in large models.

To eliminate solver resets at wrap points, the Integrator block supports wrapped states that you can enable by checking **Wrap state** on the block parameter dialog box. When you enable **Wrap state**, the block icon changes to indicate that the block has wrapping states.



Simulink allows wrapping states that are bounded by upper and lower values parameters of the wrapped state. The algorithm for determining wrapping states is given by:

$$y = \begin{cases} x \\ x - (x_u - x_l) \left\lfloor \frac{x - x_l}{x_u - x_l} \right\rfloor & x \in [x_l, x_u) \text{ otherwise} \end{cases}$$

where:

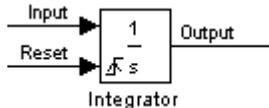
- x_l is the lower value of the wrapped state.
- x_u is the upper value of the wrapped state.
- y is the output.

The support for wrapping states provides these advantages.

- It eliminates simulation instability when your model approaches large angles and large state values.
- It reduces the number of solver resets during simulation and eliminates the need for zero-crossing detection, improving simulation time.
- It eliminates large angle values, speeding up computation of trigonometric functions on angular states.
- It improves solver accuracy and performance and enables unlimited simulation time.

Resetting the State

The block can reset its state to the specified initial condition based on an external signal. To cause the block to reset its state, select one of the **External reset** choices. A trigger port appears below the block's input port and indicates the trigger type.



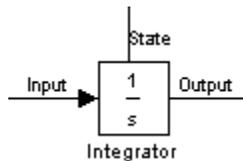
- Select `rising` to reset the state when the reset signal rises from a negative or zero value to a positive value.
- Select `falling` to reset the state when the reset signal falls from a positive value to a zero or negative value.
- Select `either` to reset the state when the reset signal changes from zero to a nonzero value, from a nonzero value to zero, or changes sign.
- Select `level` to reset the state when the reset signal is nonzero at the current time step or changes from nonzero at the previous time step to zero at the current time step.
- Select `level hold` to reset the state when the reset signal is nonzero at the current time step.

The reset port has direct feedthrough. If the block output feeds back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results (see “Algebraic Loops”). Use the Integrator block's state port to feed back the block's output without creating an algebraic loop.

Note To be compliant with the Motor Industry Software Reliability Association (MISRA®) software standard, your model must use Boolean signals to drive the external reset ports of Integrator blocks.

About the State Port

Selecting the **Show state port** check box on the Integrator block's parameter dialog box causes an additional output port, the state port, to appear at the top of the Integrator block.



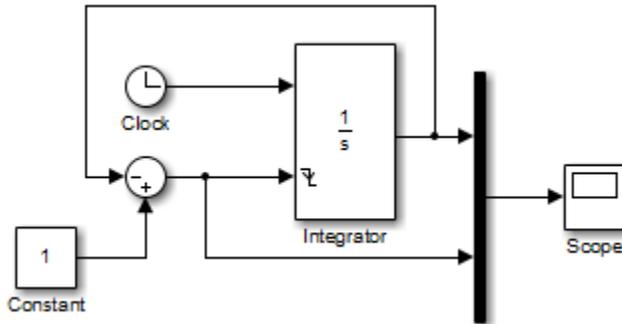
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset. The state port's output appears earlier in the time step than the output of the Integrator block's output port. Use the state port to avoid creating algebraic loops in these modeling scenarios:

- Self-resetting integrators (see “Creating Self-Resetting Integrators” on page 1-793)
- Handing off a state from one enabled subsystem to another (see “Handing Off States Between Enabled Subsystems” on page 1-794)

Note When updating a model, Simulink checks that the state port applies to one of these two scenarios. If not, an error message appears. Also, you cannot log the output of this port in a referenced model that executes in Accelerator mode. If logging is enabled for the port, Simulink generates a "signal not found" warning during execution of the referenced model.

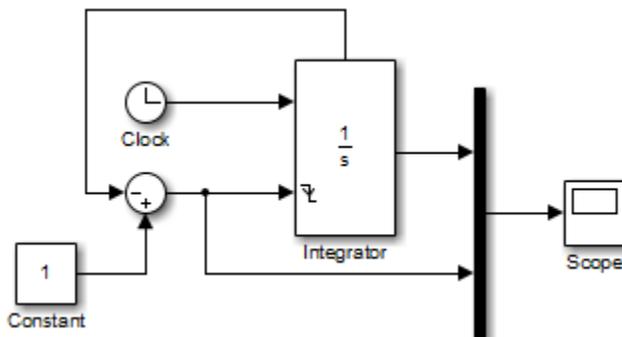
Creating Self-Resetting Integrators

The Integrator block's state port helps you avoid an algebraic loop when creating an integrator that resets itself based on the value of its output. Consider, for example, the following model.



This model tries to create a self-resetting integrator by feeding the integrator's output, subtracted from 1, back into the integrator's reset port. However, the model creates an algebraic loop. To compute the integrator block's output, Simulink software needs to know the value of the block's reset signal, and vice versa. Because the two values are mutually dependent, Simulink software cannot determine either. Therefore, an error message appears if you try to simulate or update this model.

The following model uses the integrator's state port to avoid the algebraic loop.

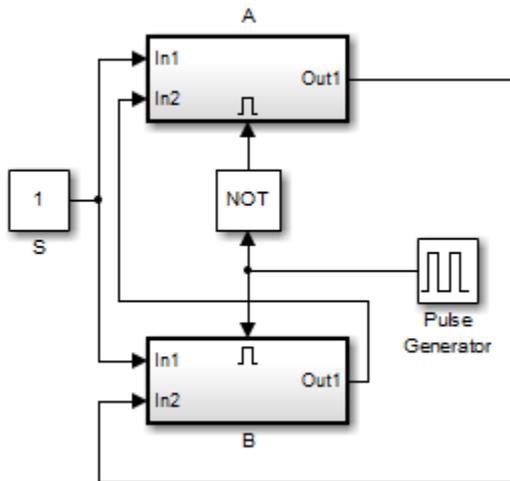


In this version, the value of the reset signal depends on the value of the state port. The value of the state port is available earlier in the current time step than the value of the integrator block's output port. Therefore, Simulink can determine whether the block

needs to be reset before computing the block's output, thereby avoiding the algebraic loop.

Handing Off States Between Enabled Subsystems

The state port helps you avoid an algebraic loop when passing a state between two enabled subsystems. Consider, for example, the following model.



The enabled subsystems, A and B, contain the following blocks:

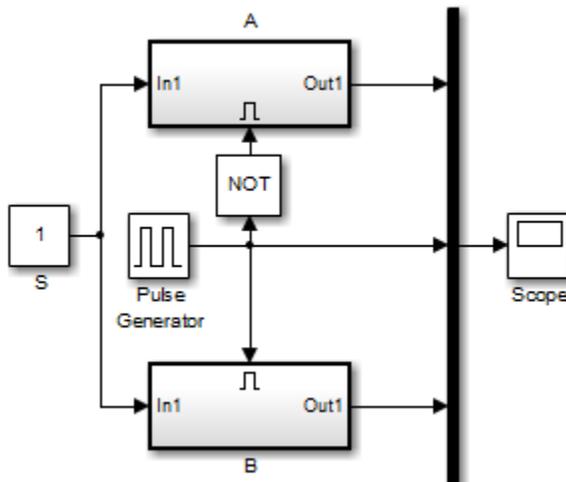
Subsystem A	Subsystem B

In this model, a constant input signal drives two enabled subsystems that integrate the signal. A pulse generator generates an enabling signal that causes execution to alternate

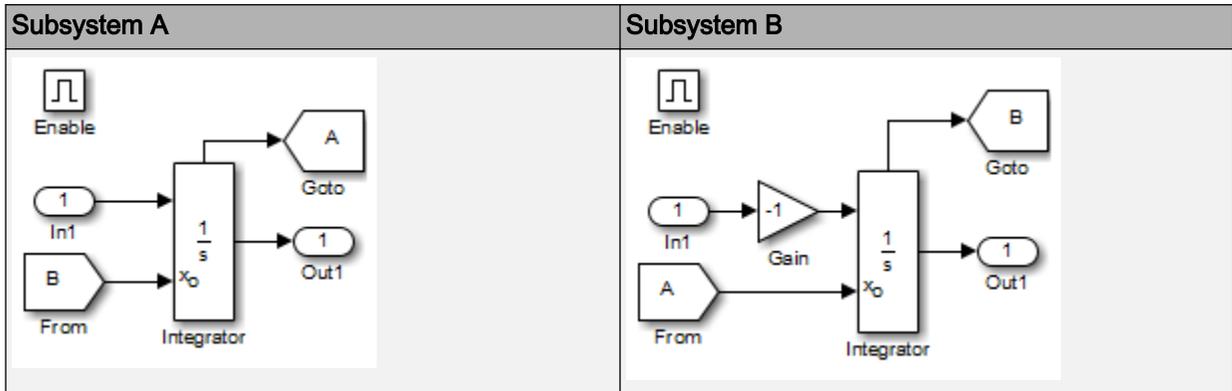
between the two subsystems. The enable port of each subsystem is set to reset, which causes the subsystem to reset its integrator when it becomes active. Resetting the integrator causes the integrator to read the value of its initial condition port. The initial condition port of the integrator in each subsystem is connected to the output port of the integrator in the other subsystem.

This connection is intended to enable continuous integration of the input signal as execution alternates between two subsystems. However, the connection creates an algebraic loop. To compute the output of A, Simulink needs to know the output of B, and vice versa. Because the outputs are mutually dependent, Simulink cannot compute the output values. Therefore, an error message appears if you try to simulate or update this model.

The following version of the same model uses the integrator state port to avoid creating an algebraic loop when handing off the state.



The enabled subsystems, A and B, contain the following blocks:



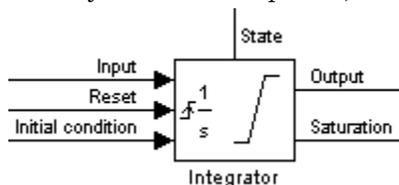
In this model, the initial condition of the integrator in A depends on the value of the state port of the integrator in B, and vice versa. The values of the state ports are updated earlier in the simulation time step than the values of the integrator output ports. Therefore, Simulink can compute the initial condition of either integrator without knowing the final output value of the other integrator. For another example of using the state port to hand off states between conditionally executed subsystems, see the `sldemo_clutch` model.

Specifying the Absolute Tolerance for the Block Outputs

By default Simulink software uses the absolute tolerance value specified in the Configuration Parameters dialog box (see “Error Tolerances for Variable-Step Solvers”) to compute the output of the Integrator block. If this value does not provide sufficient error control, specify a more appropriate value in the **Absolute tolerance** field of the Integrator block dialog box. The value that you specify is used to compute all the block outputs.

Selecting All Options

When you select all options, the block icon looks like this.



Data Type Support

The Integrator block accepts and outputs signals of type `double` on its data ports. The external reset port accepts signals of type `double` or `Boolean`.

Parameters

External reset

Reset the states to their initial conditions when a trigger event occurs in the reset signal.

Default: `none`

`none`

Do not reset the state to initial conditions.

`rising`

Reset the state when the reset signal rises from a zero to a positive value or from a negative to a positive value.

`falling`

Reset the state when the reset signal falls from a positive value to zero or from a positive to a negative value.

`either`

Reset the state when the reset signal changes from a zero to a nonzero value or changes sign.

`level`

Reset the state when the reset signal is nonzero at the current time step or changes from nonzero at the previous time step to zero at the current time step.

`level hold`

Reset the state when the reset signal is nonzero at the current time step.

Parameter: `ExternalReset`

Type: character vector

Value: `'none' | 'rising' | 'falling' | 'either' | 'level' | 'level hold'`

Default: 'none'

Initial condition source

Get the initial conditions of the states.

Default: `internal`

`internal`

Get the initial conditions of the states from the **Initial condition** parameter.

`external`

Get the initial conditions of the states from an external block.

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Selecting `internal` enables the **Initial condition** parameter.

Selecting `external` disables the **Initial condition** parameter.

Parameter: `InitialConditionSource`

Type: character vector

Value: `'internal' | 'external'`

Default: `'internal'`

Initial condition

Specify the states' initial conditions.

Default: 0

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Setting **Initial condition source** to `internal` enables this parameter.

Setting **Initial condition source** to `external` disables this parameter.

Parameter: `InitialCondition`

Type: scalar or vector

Value: `'0'`

Default: `'0'`

Limit output

Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

Default: Off



On

Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.



Off

Do not limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

This parameter enables **Upper saturation limit**.

This parameter enables **Lower saturation limit**.

Parameter: `LimitOutput`

Type: character vector

Value: `'off' | 'on'`

Default: `'off'`

Upper saturation limit

Specify the upper limit for the integral.

Default: `inf`

Minimum: value of **Output minimum** parameter

Maximum: value of **Output maximum** parameter

Limit output enables this parameter.

Parameter: UpperSaturationLimit

Type: scalar or vector

Value: 'inf'

Default: 'inf'

Lower saturation limit

Specify the lower limit for the integral.

Default: -inf

Minimum: value of **Output minimum** parameter

Maximum: value of **Output maximum** parameter

Limit output enables this parameter.

Parameter: LowerSaturationLimit

Type: scalar or vector

Value: '-inf'

Default: '-inf'

Wrap state

Enable wrapping of states between the **Wrapped state upper value** and **Wrapped state lower value** parameters. Enabling wrap states eliminates the need for zero-crossing detection, reduces solver resets, improves solver performance and accuracy, and increases simulation time span when modeling rotary and cyclic state trajectories.

Default: off

On

Enable wrap states between the **Wrapped state upper value** and **Wrapped state lower value** parameters.

If you specify **Wrapped state upper value** as `inf` and **Wrapped state lower value** as `-inf`, wrapping will never occur.

 Off

Do not enable wrap states.

This parameter enables **Wrapped state upper value**.

This parameter enables **Wrapped state lower value**.

Parameter: `WrapState`

Type: character vector

Value: `'off' | 'on'`

Default: `'off'`

Wrapped state upper value

Specify the upper value for the wrap state.

Default: `'pi'`

Wrap state enables this parameter.

Parameter: `WrappedStateUpperValue`

Type: scalar or vector

Value: `'2*pi'`

Default: `'pi'`

Wrapped state lower value

Specify the lower value for the wrap state.

Default: `-pi`

Wrap state enables this parameter.

Parameter: `WrappedStateLowerValue`

Type: scalar or vector

Value: `'0'`

Default: `'-pi'`

Show saturation port

Add a saturation output port to the block.

Default: Off

On

Add a saturation output port to the block.

Off

Do not add a saturation output port to the block.

Parameter: `ShowSaturationPort`

Type: character vector

Value: `'off' | 'on'`

Default: `'off'`

Show state port

Add an output port to the block for the block's state.

Default: Off

On

Add an output port to the block for the block's state.

Off

Do not add an output port to the block for the block's state.

Parameter: ShowStatePort

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Absolute tolerance

Specify the absolute tolerance for computing block states.

Default: auto

- You can enter `auto`, `-1`, a positive real scalar or vector.
- If you enter `auto` or `-1`, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute block states.
- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

Parameter: AbsoluteTolerance

Type: character vector, scalar, or vector

Value: 'auto' | '-1' | any positive real scalar or vector

Default: 'auto'

Ignore limit and reset when linearizing

Cause Simulink linearization commands to treat this block as unresettable and as having no limits on its output, regardless of the settings of the block's reset and output limitation options.

Default: Off

On

Cause Simulink linearization commands to treat this block as unresettable and as having no limits on its output, regardless of the settings of the block's reset and output limitation options.

Off

Do not cause Simulink linearization commands to treat this block as unresettable and as having no limits on its output, regardless of the settings of the block's reset and output limitation options.

Use this check box to linearize a model around an operating point that causes the integrator to reset or saturate.

Parameter: IgnoreLimit

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Default: On

On

Use zero crossings to detect and take a time step at any of the following events: reset, entering or leaving an upper saturation state, entering or leaving a lower saturation state.

Off

Do not use zero crossings to detect and take a time step at any of the following events: reset, entering or leaving an upper saturation state, entering or leaving a lower saturation state.

If you select this check box, **Limit output**, and zero-crossing detection for the model as a whole, the Integrator block uses zero crossings as described.

Parameter: ZeroCross

Type: character vector

Value: 'off' | 'on'

Default: 'on'

State Name (e.g., 'position')

Assign a unique name to each state.

Default: ' '

If this field is blank, no name assignment occurs.

- To assign a name to a single state, enter the name between quotes, for example, 'velocity'.
- To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.
- The state names apply only to the selected block.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Parameter: ContinuousStateAttributes

Type: character vector

Value: ' ' | user-defined

Default: ' '

Examples

The following example models show how to use the Integrator block:

- `sldemo_hardstop`
- `sldemo_suspn`
- `sldemo_wheelspeed_absbrake`

Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	Yes, of the reset and external initial condition source ports
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled and you select the Limit output check box, one for detecting reset, one each to detect upper and lower saturation limits, and one when leaving saturation
Code Generation	Yes

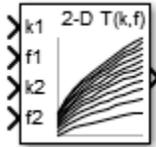
See Also

Discrete-Time Integrator

Introduced before R2006a

Interpolation Using Prelookup

Use precalculated index and fraction values to accelerate approximation of N-dimensional function



Library

Lookup Tables

Description

The Prelookup block calculates the index and interval fraction that specify how its input value u relates to the breakpoint data set. The Interpolation Using Prelookup block works best with the Prelookup block. Feed the resulting index and fraction values into an Interpolation Using Prelookup block to interpolate an n -dimensional table. These two blocks have distributed algorithms. When combined together, they perform the same operation as the integrated algorithm in the n -D Lookup Table block. However, the Prelookup and Interpolation Using Prelookup blocks offer greater flexibility that can provide more efficient simulation and code generation. For more information, see “Efficiency of Performance” in the Simulink documentation.

Supported Block Operations

To use the Interpolation Using Prelookup block, you specify a set of table data values directly on the dialog box or feed values into the T input port. Typically, these table values correspond to the breakpoint data sets specified in Prelookup blocks. The Interpolation Using Prelookup block generates output by looking up or estimating table values based on index and interval fraction values fed from Prelookup blocks. Labels for the index and interval fraction appear as k and f on the Interpolation Using Prelookup block icon.

When inputs for index and interval fraction...	The Interpolation Using Prelookup block...
Map to values in breakpoint data sets	Outputs the table value at the intersection of the row, column, and higher dimension breakpoints
Do not map to values in breakpoint data sets, but are within range	Interpolates appropriate table values, using the Interpolation method you select
Do not map to values in breakpoint data sets, and are out of range	Extrapolates the output value, using the Extrapolation method you select

How The Block Interpolates a Subset of Table Data

You can use the **Number of sub-table selection dimensions** parameter to specify that interpolation occur only on a subset of the table data. To activate this interpolation mode, set this parameter to a positive integer. This value defines the number of dimensions to select, starting from the highest dimension of table data. Therefore, the value must be less than or equal to the **Number of table dimensions**.

Suppose that you have 3-D table data in your Interpolation Using Prelookup block. This behavior applies.

Number of Selection Dimensions	Action by the Block	Block Appearance
0	Interpolates the entire table and does not activate subtable selection	Does not change
1	Interpolates the first two dimensions and selects the third dimension	Displays an input port with the label <code>sel1</code> that you use to select and interpolate 2-D tables
2	Interpolates the first dimension and selects the second and third dimensions	Displays two input ports with the labels <code>sel1</code> and <code>sel2</code> that you use to select and interpolate 1-D tables

Subtable selection uses zero-based indexing. For an example of interpolating a subset of table data, type `sldemo_bpcheck` at the MATLAB command prompt.

Data Type Support

The Interpolation Using Prelookup block accepts as input:

- Real signals of any numeric data type supported by Simulink software, except Boolean.
- Fixed-point data types for signals, table data, and intermediate results
- Bus data types

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

- “Main Tab” on page 1-809
- “Data Types Tab” on page 1-814

Main Tab

Number of dimensions

Specify the number of dimensions that the table data must have. This value defines the number of independent variables for the table. Enter an integer between 1 and 30 into this field.

Require index and fraction as bus

Input index and fraction signals to a bus port, with the index and fraction signals combined in a nonvirtual bus.

If you select this parameter, subtable selection ports continue to work in nonbus mode.

To enable the Prelookup block to supply input to the Interpolation Using Prelookup block, set:

- **Output selection** to Index and fraction as bus
- **Output** to Bus: <object name>, where <object name> must be a valid bus object name accessible to the model

Specification

Specify whether to enter table data directly or use a lookup table object. If you set this parameter to:

- `Explicit values`, the **Source** and **Value** parameters are visible on the dialog box.
- `Lookup table object`, the **Name** parameter is visible on the dialog box.

Source

Specify whether to enter table data in the dialog box or to inherit the data from an input port. This parameter is available when **Specification** is set to `Explicit values`. If you set **Source** to:

- `Dialog`, enter table data in the text box under **Value**.
- `Input port`, verify that an upstream signal supplies table data to the table input port.

Value

If you set **Specification** to `Explicit values` and **Source** to `Dialog`, explicitly specify the table data.

You can edit the block diagram without specifying a correctly dimensioned matrix by entering an empty matrix (`[]`) or an undefined workspace variable in the **Value** edit field. For information about how to construct multidimensional arrays in MATLAB, see “Multidimensional Arrays” (MATLAB).

If you set **Source** to `Input port`, verify that an upstream signal supplies table data to the T input port. The size of table data must match the **Number of table dimensions**. For this option, your block inherits table attributes from the T input port.

To edit lookup tables using the Lookup Table Editor, click **Edit** (see “Edit Lookup Tables” in the Simulink documentation).

Name

Specify the name of an existing `Simulink.LookupTable` object. An existing lookup table object references Simulink breakpoint objects.

Interpolation method

Select `Flat`, `Nearest`, or `Linear`. See “Interpolation Methods” in the Simulink documentation for more information.

Extrapolation method

Select `Clip` or `Linear`. See “Extrapolation Methods” in the Simulink documentation for more information. The **Extrapolation method** parameter is visible only when you select `Linear` as the **Interpolation method** parameter.

The Interpolation Using Prelookup block does not support `Linear` extrapolation when the input or output signals specify integer or fixed-point data types.

Valid index input may reach last index

Specify how block inputs for index (k) and interval fraction (f) access the last elements of n -dimensional table data. Index values are zero based.

Check Box	Block Behavior
Selected	Returns the value of the last element in a dimension of its table when: <ul style="list-style-type: none"> • k indexes the last table element in the corresponding dimension • f is 0
Cleared	Returns the value of the last element in a dimension of its table when: <ul style="list-style-type: none"> • k indexes the next-to-last table element in the corresponding dimension • f is 1

This check box is visible only when:

- **Interpolation method** is `Linear`.
- **Extrapolation method** is `Clip`.

Tip When you select **Valid index input may reach last index** for an Interpolation Using Prelookup block, you must also select **Use last breakpoint for input at or above upper limit** for all Prelookup blocks that feed it. This action allows the blocks

to use the same indexing convention when accessing the last elements of their breakpoint and table data sets.

Diagnostic for out-of-range input

Specify whether to produce a warning or error when the input `k` or `f` is out of range. Options include:

- `None` — Simulink software takes no action.
- `Warning` — Simulink software displays a warning and continues the simulation.
- `Error` — Simulink software terminates the simulation and displays an error.

Remove protection against out-of-range index in generated code

Specify whether or not to include code that checks for out-of-range index inputs.

Check Box	Result	When to Use
Selected	Generated code does not include conditional statements to check for out-of-range index inputs. When the input <code>k</code> or <code>f</code> is out-of-range, it may cause undefined behavior for generated code and simulations using accelerator mode.	For code efficiency
Cleared	Generated code includes conditional statements to check for out-of-range index inputs.	For safety-critical applications

If your input is not out-of-range, you can select the **Remove protection against out-of-range index in generated code** check box for code efficiency. By default, this check box is cleared. For safety-critical applications, do not select this check box. If you want to select the **Remove protection against out-of-range index in generated code** check box, first check that your model inputs are in range. For example:

- 1 Clear the **Remove protection against out-of-range index in generated code** check box.
- 2 Set the **Diagnostic for out-of-range input** parameter to `Error`.
- 3 Simulate the model in normal mode.
- 4 If there are out-of-range errors, fix them to be in range and run the simulation again.
- 5 When the simulation no longer generates out-of-range input errors, select the **Remove protection against out-of-range index in generated code** check box.

Note When you select the **Remove protection against out-of-range index in generated code** check box and the input `k` or `f` is out-of-range, the behavior is undefined for generated code and simulations using accelerator mode.

Depending on your application, you can run the following Model Advisor checks to verify the usage of this check box:

- **By Product > Embedded Coder > Identify lookup table blocks that generate expensive out-of-range checking code**
- **By Product > Simulink Check > Modeling Standards > DO-178C/DO-331 Checks > Check usage of lookup table blocks**

For more information about the Model Advisor, see “Run Model Checks” in the Simulink documentation.

This check box has no effect on the generated code when one of the following is true:

- The Prelookup block feeds index values to the Interpolation Using Prelookup block.

Because index values from the Prelookup block are always valid, no check code is necessary.

- The data type of the input `k` restricts the data to valid index values.

For example, unsigned integer data types guarantee nonnegative index values. Therefore, unsigned input values of `k` do not require check code for negative values.

Number of sub-table selection dimensions

Specify the number of dimensions of the subtable that the block uses to compute the output. Follow these rules:

- To enable subtable selection, enter a positive integer.

This integer must be less than or equal to the **Number of table dimensions**.

- To disable subtable selection, enter 0 to interpolate the entire table.

For more information, see “How The Block Interpolates a Subset of Table Data” on page 1-808.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Data Types Tab

Note The parameters for table attributes (data type, minimum, and maximum) are not available when you set **Source** to `Input port`. In this case, the block inherits all table attributes from the `T` input port.

Table data > Data Type

Specify the table data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as output`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the table data type.

Tip Specify a table data type different from the output data type for these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal
 - Sharing of prescaled table data between two Interpolation Using Prelookup blocks with different output data types
 - Sharing of custom storage table data in Simulink Coder generated code for blocks with different output data types
-

Table data > Minimum

Specify the minimum value for table data. The default value is [] (unspecified).

Table data > Maximum

Specify the maximum value for table data. The default value is [] (unspecified).

Intermediate results > Data Type

Specify the intermediate results data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as output`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the intermediate results data type.

Tip Use this parameter to specify higher precision for internal computations than for table data or output data.

Output > Data Type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`

- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the output data type.

See “Control Signal Data Types” in the *Simulink User's Guide* for more information.

Output > Minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output > Maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Internal rule priority

Specify the internal rule for intermediate calculations. Select `Speed` for faster calculations. If you do, a loss of accuracy might occur, usually up to 2 bits.

Lock data type settings against changes by the fixed-point tools

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function in the mask field.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.

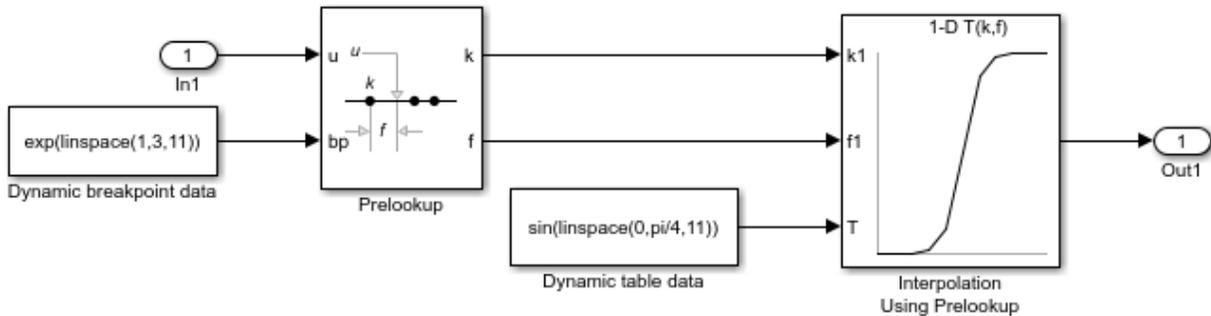
Action	Reasons for Taking This Action	What Happens for Overflows	Example
Do not select this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</p>	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code> , which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code> , is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Examples

Prelookup with External Breakpoint Specification

In the following model, a Constant block feeds the table data values to the `T` input port of the Interpolation Using Prelookup block.



The Interpolation Using Prelookup block inherits the following table attributes from the T input port:

Table Attribute	Value
Minimum	-Inf
Maximum	Inf
Data type	single

Similarly, a Constant block feeds the breakpoint data set to the bp input port of the Prelookup block, which inherits the following breakpoint attributes:

Breakpoint Attribute	Value
Minimum	-Inf
Maximum	Inf
Data type	single

Simulink uses double-precision, floating-point data to perform the computations in this model. However, the model stores the breakpoint and table data as single-precision, floating-point data. Using a lower-precision data type to store breakpoint and table data reduces the memory requirement.

For other examples, see “Prelookup and Interpolation Blocks” in the Simulink documentation.

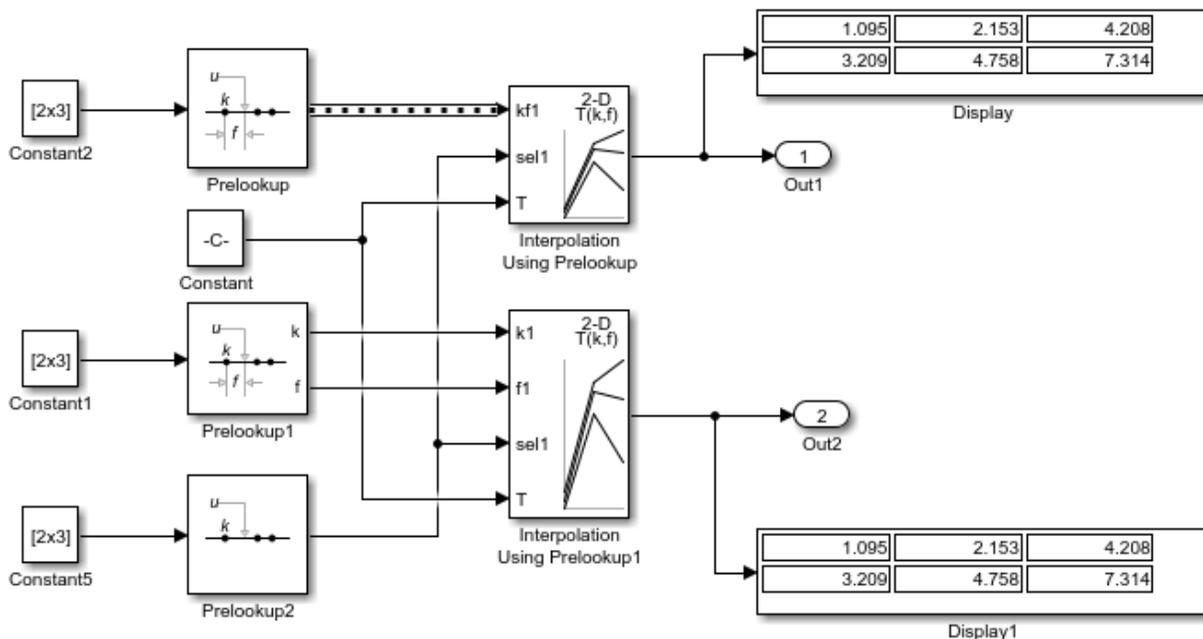
Use a Bus for Index and Fraction

You can use a bus object to specify the index (k) and fraction (f) in the Prelookup block and then use that bus as input to an Interpolation Using Prelookup block. This example

shows how to use a bus signal and how to get the same results without using a bus object.

Open the model and simulate it.

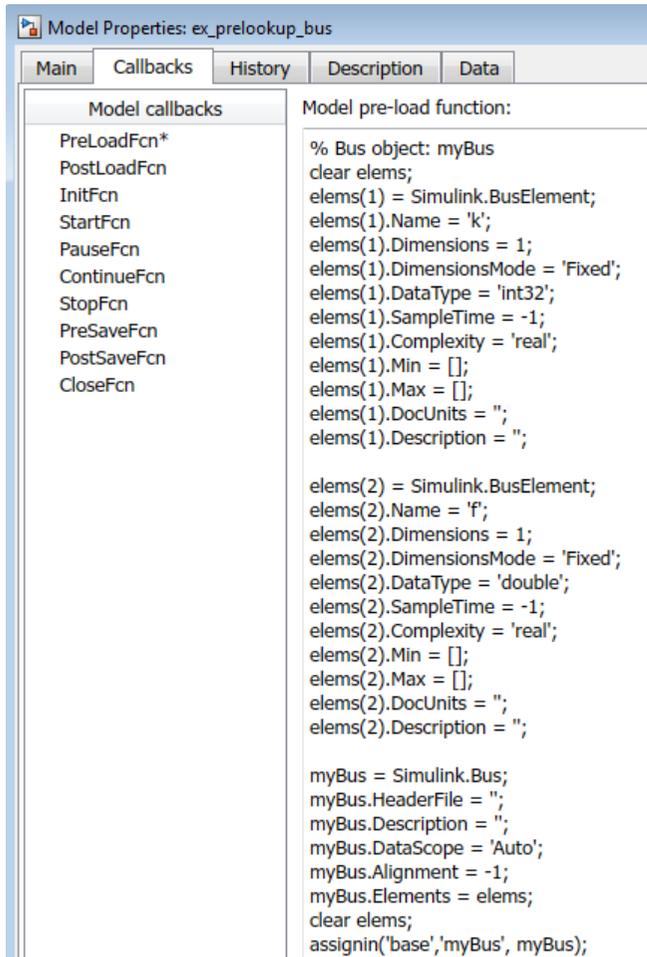
```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
    'examples', 'ex_prelookup_bus.slx')))
sim('ex_prelookup_bus')
```



At the top of the model, open the Prelookup block parameters dialog box. In the **Main** tab, note that **Output selection** is set to Index and fraction as bus.

In the **Data Types** tab, note that **Output** is set to Bus: myBus.

In the Simulink Editor, select **File > Model Properties** and open the **Callbacks** tab. The code defines the bus object myBus, which specifies the index as the first bus element and the fraction as the second element.



In the Interpolation Using Prelookup block parameters dialog box, in the **Main** tab, note that **Require index and fraction as bus** check box is selected. That option configures the block to use the bus output from the Prelookup block.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
------------	----------------------------------------------

Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

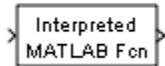
See Also

Prelookup

Introduced in R2006b

Interpreted MATLAB Function

Apply MATLAB function or expression to input



Library

User-Defined Functions

Description

The Interpreted MATLAB Function block applies the specified MATLAB function or expression to the input. The output of the function must match the output dimensions of the block.

Some valid expressions for this block are:

```
sin
atan2(u(1), u(2))
u(1)^u(2)
```

Note This block is slower than the Fcn block because it calls the MATLAB parser during each integration step. Consider using built-in blocks (such as the Fcn block or the Math Function block) instead. Alternatively, you can write the function as a MATLAB S-function or MEX-file S-function, then access it using the S-Function block.

Data Type Support

The Interpreted MATLAB Function block accepts one real or complex input of type `double` and generates real or complex output of type `double`, depending on the setting of the **Output signal type** parameter.

Parameters

MATLAB function

Specify the function or expression. If you specify a function only, it is not necessary to include the input argument in parentheses.

Output dimensions

Specify the dimensions of the block output signal, for example, 2 for a two-element vector. The output dimensions must match the dimensions of the value returned by the function or expression in the **MATLAB function** field.

Specify -1 to inherit the dimensions from the output of the specified function or expression. To determine the output dimensions, Simulink runs the function or expression once before simulation starts.

Note If you specify -1 for this parameter and your function has persistent variables, then the variables might update before the simulation starts. If you need to use persistent variables, consider setting this parameter to a value other than -1.

Output signal type

Specify the output signal type of the block as `real`, `complex`, or `auto`. A value of `auto` sets the output type to be the same as the type of the input signal.

Collapse 2-D results to 1-D

Select this check box to output a 2-D array as a 1-D array containing the 2-D array's elements in column-major order.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Characteristics

Data Types	Double
------------	--------

Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	No

Introduced in R2011a

Interval Test

Determine if signal is in specified interval



Library

Logic and Bit Operations

Description

The Interval Test block outputs TRUE if the input is between the values specified by the **Lower limit** and **Upper limit** parameters. The block outputs FALSE if the input is outside those values. The output of the block when the input is equal to the **Lower limit** or the **Upper limit** is determined by whether the boxes next to **Interval closed on left** and **Interval closed on right** are selected in the dialog box.

Data Type Support

The Interval Test block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

In this case, the **Upper limit** and **Lower limit** values must be of the same enumerated type.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Interval closed on right

When you select this check box, the **Upper limit** is included in the interval for which the block outputs TRUE.

Upper limit

The upper limit of the interval for which the block outputs TRUE.

Interval closed on left

When you select this check box, the **Lower limit** is included in the interval for which the block outputs TRUE.

Lower limit

The lower limit of the interval for which the block outputs TRUE.

Output data type

Select the output data type: `boolean` or `uint8`.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

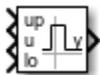
See Also

Interval Test Dynamic

Introduced before R2006a

Interval Test Dynamic

Determine if signal is in specified interval



Library

Logic and Bit Operations

Description

The Interval Test Dynamic block outputs TRUE if the input is between the values of the external signals up and lo. The block outputs FALSE if the input is outside those values. The output of the block when the input is equal to the signal up or the signal lo is determined by whether the boxes next to **Interval closed on left** and **Interval closed on right** are selected in the dialog box.

Data Type Support

The Interval Test Dynamic block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

In this case, all inputs must be of the same enumerated type.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Interval closed on right

When you select this check box, the value of the signal connected to the block's “up” input port is included in the interval for which the block outputs TRUE.

Interval closed on left

When you select this check box, the value of the signal connected to the block's “lo” input port is included in the interval for which the block outputs TRUE.

Output data type

Select the output data type: `boolean` or `uint8`.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

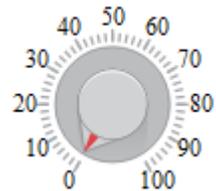
Interval Test

Introduced before R2006a

Knob

Tune parameter value with dial

Library: Simulink / Dashboard



Description

The Knob block tunes the value of the connected block parameter to during simulation. For example, you can connect the Knob block to a Gain block in your model and adjust its value during simulation. You can modify the range of the Knob block's scale to fit your data. Use the Knob block with other Dashboard blocks to create an interactive dashboard to control your model.

Connection

Dashboard block controls connect to variables and block parameters. To connect a Dashboard block to a variable or parameter, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting blocks of interest in your model.
- 3 Mark the option button next to the parameter or variable you want to adjust during simulation.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined using the variable `engine(1)` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

Parameters

Connection — Select a block parameter to connect

signal connection options

Select the block parameter to control using the **Connection** table. To open the dialog box, double-click the block. Then, populate the **Connection** table by selecting a block in your model. Select the variable parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram with **Ctrl+D**.

Scale Type — Type of scale

'Linear' (default) | 'Log'

Type of scale displayed on the control. **Linear** specifies a linear scale, and **Log** specifies a logarithmic scale.

Minimum — Minimum tick mark value

0 (default) | scalar

A finite, real, double, scalar value specifying the minimum tick mark value for the scale. The minimum must be less than the value entered for the maximum.

Maximum — Maximum tick mark value

100 (default) | scalar

A finite, real, double, scalar value specifying the maximum tick mark value for the scale. The maximum must be greater than the value entered for the minimum.

Tick Interval — Interval between major tick marks

auto (default) | scalar

A finite, real, positive, integer, scalar value specifying the interval of major tick marks on the scale. When set to `auto`, the block automatically adjusts the tick interval based on the minimum and maximum values.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

See Also

Rotary Switch | Slider

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2015a

Lamp

Display color reflecting input value

Library: Simulink / Dashboard



Description

The Lamp block displays a color indicating the value of the input signal. You can use the Lamp block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. You can specify pairs of input values and colors to provide the information you want during simulation.

Connection

Dashboard blocks do not use ports to connect to signals. To connect a signal to a Dashboard block, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting signals of interest in your model.
- 3 Mark the option button next to the signal you want to display.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.

- If you turn off logging for a signal connected to a Dashboard block, the model stops sending data from that signal to the block. To view the signal again, reconnect the signal.
- External mode simulation does not support visualization Dashboard blocks.

Parameters

Connection — Select a signal to connect and display

signal connection options

Signal connection options. Populate the **Connection** table by selecting signals of interest in your model. Mark the check box next to the signal you want to display. Click **Apply** to connect the signal.

States

state — Input value corresponding to a color

0 (default) | scalar

Input value that causes the specified color indication. The [undefined] state specifies the Lamp block's behavior when the input value is anything other than values specified in the **States** table. Click the + button to add another state.

Color — Color of lamp indicating input value

green (default)

Color indicating the input value specified in the corresponding **State**. You can select from a palette of standard colors or specify a custom color with RGB values. The Lamp block displays the color specified for the [undefined] state when the input's value does not correspond to a value in the **States** table. Click the + button to add another state.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

Model Examples

See Also

MultiStateImage

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2015a

Level-2 MATLAB S-Function

Use Level-2 MATLAB S-function in model

Library: User-Defined Functions



Description

This block allows you to use a Level-2 MATLAB S-function (see “Write Level-2 MATLAB S-Functions”) in a model. To do this, create an instance of this block in the model. Then enter the name of the Level-2 MATLAB S-function in the **S-function name** field of the block's parameter dialog box.

Note Use the S-Function block to include a Level-1 MATLAB S-function in a block.

If the Level-2 MATLAB S-function defines any additional parameters, you can enter them in the **Parameters** field of the block's parameter dialog box. Enter the parameters as MATLAB expressions that evaluate to their values in the order defined by the MATLAB S-function. Use commas to separate each expression.

If a model includes a Level-2 MATLAB S-Function block, and an error occurs in the S-function, the Level-2 MATLAB S-Function block displays MATLAB stack trace information for the error in a dialog box. Click **OK** to close the dialog box.

Parameters

S-Function Name — Specify S-Function name

no default (default)

Specify the name of a MATLAB function that defines the behavior of this block. The MATLAB function must follow the Level-2 standard for writing MATLAB S-functions (see “Write Level-2 MATLAB S-Functions” for details).

Parameters — Specify values of parameters for this block

no default (default)

Specify values of parameters for this block.

See Also

Blocks

MATLAB Function | MATLAB System | S-Function | S-Function Builder | Simulink Function | Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

Topics

“What Is an S-Function?”

“Write Level-2 MATLAB S-Functions”

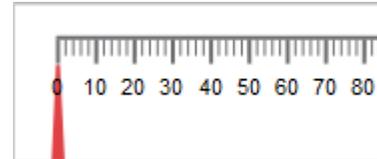
“Create a Custom Block”

Introduced in R2010b

Linear Gauge

Display input value on linear scale

Library: Simulink / Dashboard



Description

The Linear Gauge block displays the connected signal on a straight linear scale during simulation. Use the Linear Gauge block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. The Linear Gauge block provides an indication of the instantaneous value of the connected signal throughout simulation. You can modify the range of the Linear Gauge block to fit your data. You can also customize the appearance of the Linear Gauge block to provide more information about your signal. For example, you can color-code in-specification and out-of-specification ranges.

Connection

Dashboard blocks do not use ports to connect to signals. To connect a signal to a Dashboard block, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting signals of interest in your model.
- 3 Mark the option button next to the signal you want to display.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- If you turn off logging for a signal connected to a Dashboard block, the model stops sending data from that signal to the block. To view the signal again, reconnect the signal.
- External mode simulation does not support visualization Dashboard blocks.

Parameters

Connection — Select a signal to connect and display

signal connection options

Signal connection options. Populate the **Connection** table by selecting signals of interest in your model. Mark the check box next to the signal you want to display. Click **Apply** to connect the signal.

Minimum — Minimum tick mark value

0 (default) | scalar

A finite, real, double, scalar value specifying the minimum tick mark value for the scale. The minimum must be less than the value entered for the maximum.

Maximum — Maximum tick mark value

100 (default) | scalar

A finite, real, double, scalar value specifying the maximum tick mark value for the scale. The maximum must be greater than the value entered for the minimum.

Tick Interval — Interval between major tick marks

auto (default) | scalar

A finite, real, positive, integer, scalar value specifying the interval of major tick marks on the scale. When set to `auto`, the block automatically adjusts the tick interval based on the minimum and maximum values.

Scale Colors — Color indications on Gauge scale

colors for scale ranges

Color specifications for ranges on the scale. Press the + button to add a color. For each color added, specify the minimum and maximum values of the range where you want to display that color.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

See Also

Gauge | Half Gauge | Quarter Gauge

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2015a

Logical Operator

Perform specified logical operation on input



Library

Logic and Bit Operations

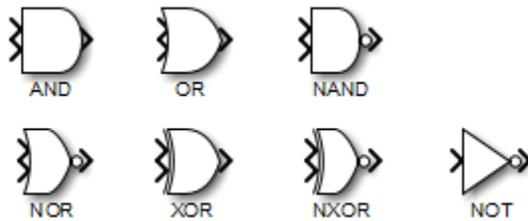
Description

The Logical Operator block performs the specified logical operation on its inputs. An input value is TRUE (1) if it is nonzero and FALSE (0) if it is zero.

You select the Boolean operation connecting the inputs with the **Operator** parameter list. If you select *rectangular* as the **Icon shape** property, the block updates to display the name of the selected operator. The supported operations are given below.

Operation	Description
AND	TRUE if all inputs are TRUE
OR	TRUE if at least one input is TRUE
NAND	TRUE if at least one input is FALSE
NOR	TRUE when no inputs are TRUE
XOR	TRUE if an odd number of inputs are TRUE
NXOR	TRUE if an even number of inputs are TRUE
NOT	TRUE if the input is FALSE

If you select *distinctive* as the **Icon shape**, the block's appearance indicates its function. Simulink software displays a distinctive shape for the selected operator, conforming to the IEEE Standard Graphic Symbols for Logic Functions:



The number of input ports is specified with the **Number of input ports** parameter. The output type is specified with the **Output data type** parameter. An output value is 1 if TRUE and 0 if FALSE.

Note The output data type should represent zero exactly. Data types that satisfy this condition include signed and unsigned integers, and any floating-point data type.

The size of the output depends on input vector size and the selected operator:

- If the block has more than one input, any nonscalar inputs must have the same dimensions. For example, if any input is a 2-by-2 array, all other nonscalar inputs must also be 2-by-2 arrays.

Scalar inputs are expanded to have the same dimensions as the nonscalar inputs.

If the block has more than one input, the output has the same dimensions as the inputs (after scalar expansion) and each output element is the result of applying the specified logical operation to the corresponding input elements. For example, if the specified operation is AND and the inputs are 2-by-2 arrays, the output is a 2-by-2 array whose top left element is the result of applying AND to the top left elements of the inputs, etc.

- For a single vector input, the block applies the operation (except the NOT operator) to all elements of the vector. The output is always a scalar.
- The NOT operator accepts only one input, which can be a scalar or a vector. If the input is a vector, the output is a vector of the same size containing the logical complements of the input vector elements.

When configured as a multi-input XOR gate, this block performs an addition- modulo-two operation as mandated by the IEEE Standard for Logic Elements.

Data Type Support

The Logical Operator block accepts real signals of any numeric data type that Simulink supports, including fixed-point data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Operator

Select logical operator to apply to block inputs.

Default: AND

AND

TRUE if all inputs are TRUE

OR

TRUE if at least one input is TRUE

NAND

TRUE if at least one input is FALSE

NOR

TRUE when no inputs are TRUE

XOR

TRUE if an odd number of inputs are TRUE

NXOR

TRUE if an even number of inputs are TRUE

NOT

TRUE if the input is FALSE

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Number of input ports

Specify number of block inputs.

Default: 2

- The value must be appropriate for the selected operator.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Icon shape

Specify shape of the block icon.

Default: rectangular

rectangular

Result in a rectangular block that displays the name of the selected operator.

distinctive

Use the graphic symbol for the selected operator as specified by the IEEE standard.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Require all inputs and output to have the same data type

Require all inputs and the output to have the same data type.

Default: Off

On

Require all inputs and the output to have the same data type.

Off

Do not require all inputs and the output to have the same data type.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Output data type

Specify the output data type.

Default: `boolean`

Inherit: `Logical` (see Configuration Parameters: Optimization)

Uses the **Implement logic signals as Boolean data** configuration parameter (see “Implement logic signals as Boolean data (vs. double)”) to specify the output data type.

Note This option supports models created before the `boolean` option was available. Use one of the other options, preferably `boolean`, for new models.

boolean

Specifies output data type is boolean.

fixdt(1,16)

Specifies output data type is fixdt(1,16).

<data type expression>

Uses the name of a data type object, for example, Simulink.NumericType.

Tip To enter a built-in data type (double, single, int8, uint8, int16, uint16, int32, or uint32), enclose the expression in single quotes. For example, enter 'double' instead of double.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Mode

Select the category of data to specify.

Default: Built in

Inherit

Specifies inheritance rules for data types. Selecting Inherit enables Logical (see Configuration Parameters: Optimization).

Built in

Specifies built-in data types. Selecting Built in enables boolean.

Fixed point

Specifies fixed-point data types.

Expression

Specifies expressions that evaluate to data types.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data as signed.

`Unsigned`

Specify the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Integer

Integer

Specify integer. This setting has the same result as specifying a binary point location and setting fraction length to 0.

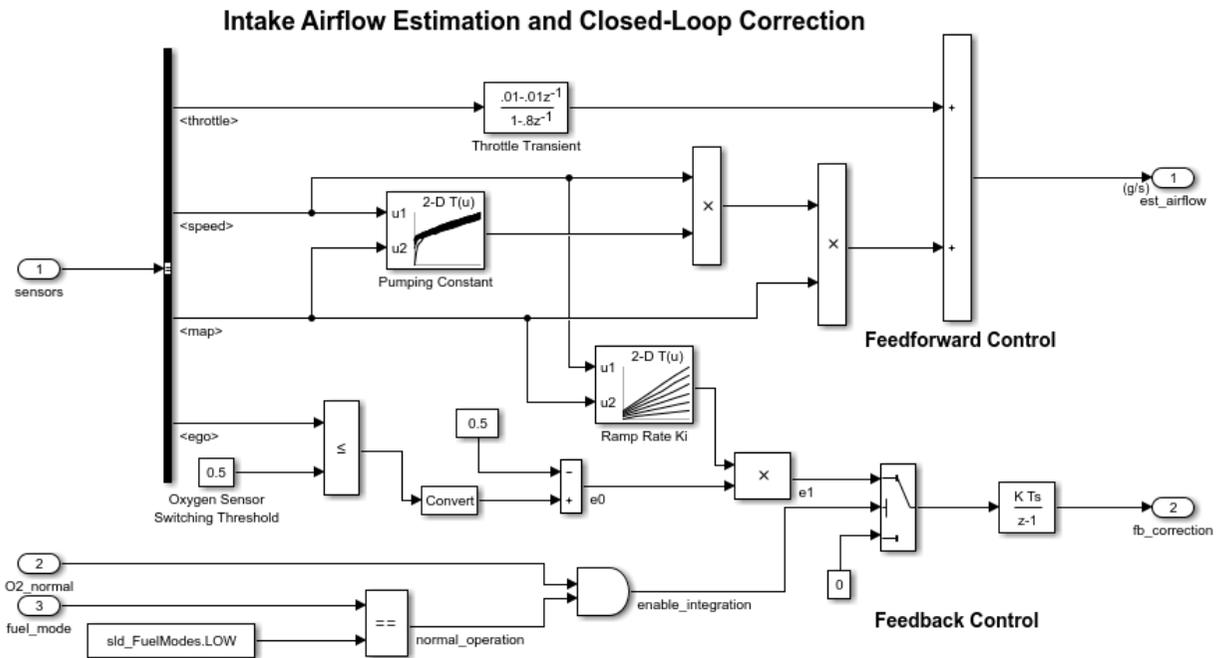
See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type”.

Examples

Logical Operator Block: AND Operator

In the `sldemo_fuelsys` model, the `fuel_rate_control/airflow_calc` subsystem uses a Logical Operator block as an AND operator:



The output of the Logical Operator block (the `enable_integration` signal) feeds into the control port of a Switch block that activates feedback control.

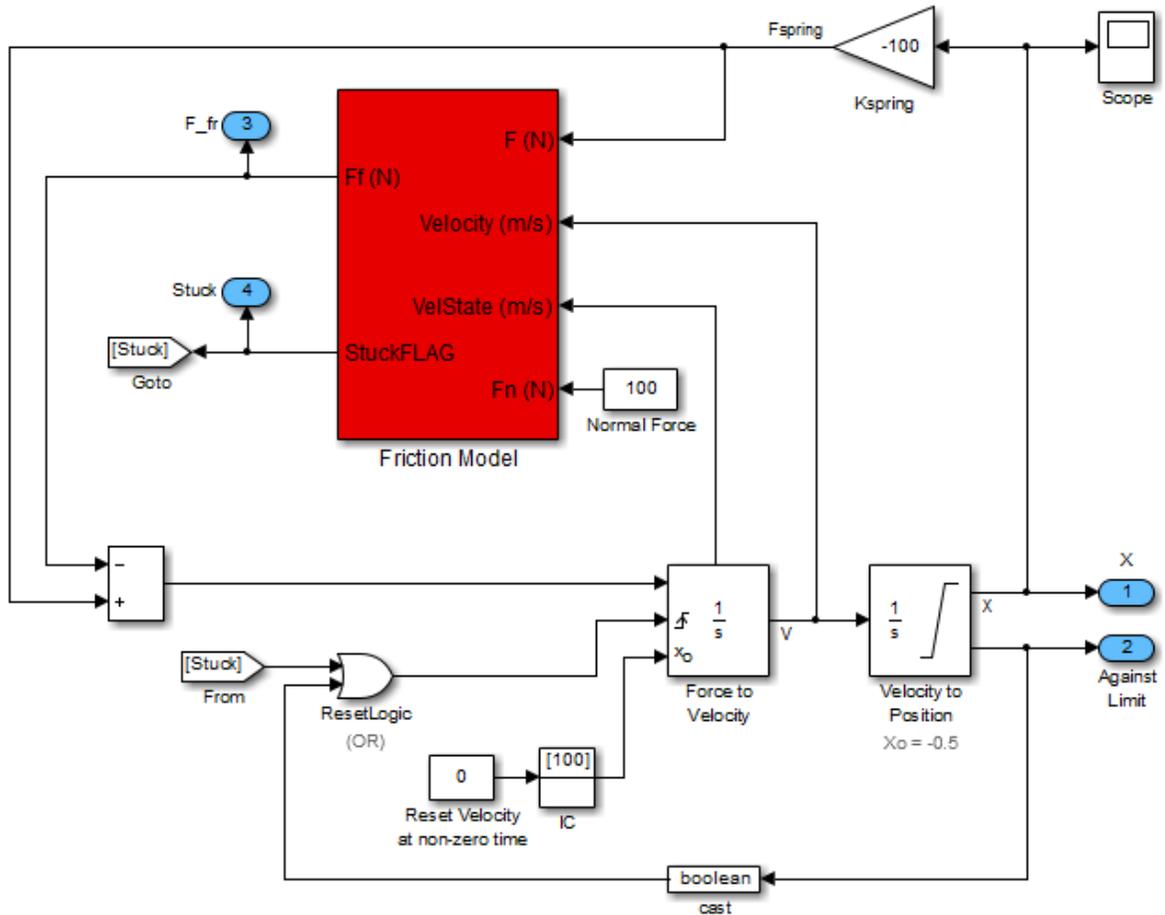
When the Logical Operator block output is...	Feedback control...
1	Occurs
0	Does not occur

Logical Operator Block: OR Operator

In the `sldemo_hardstop` model, the Logical Operator block appears as an OR operator:



Friction Model with Hard Stops



The output of the Logical Operator block feeds into the trigger port of an Integrator block to control whether velocity resets to the initial condition.

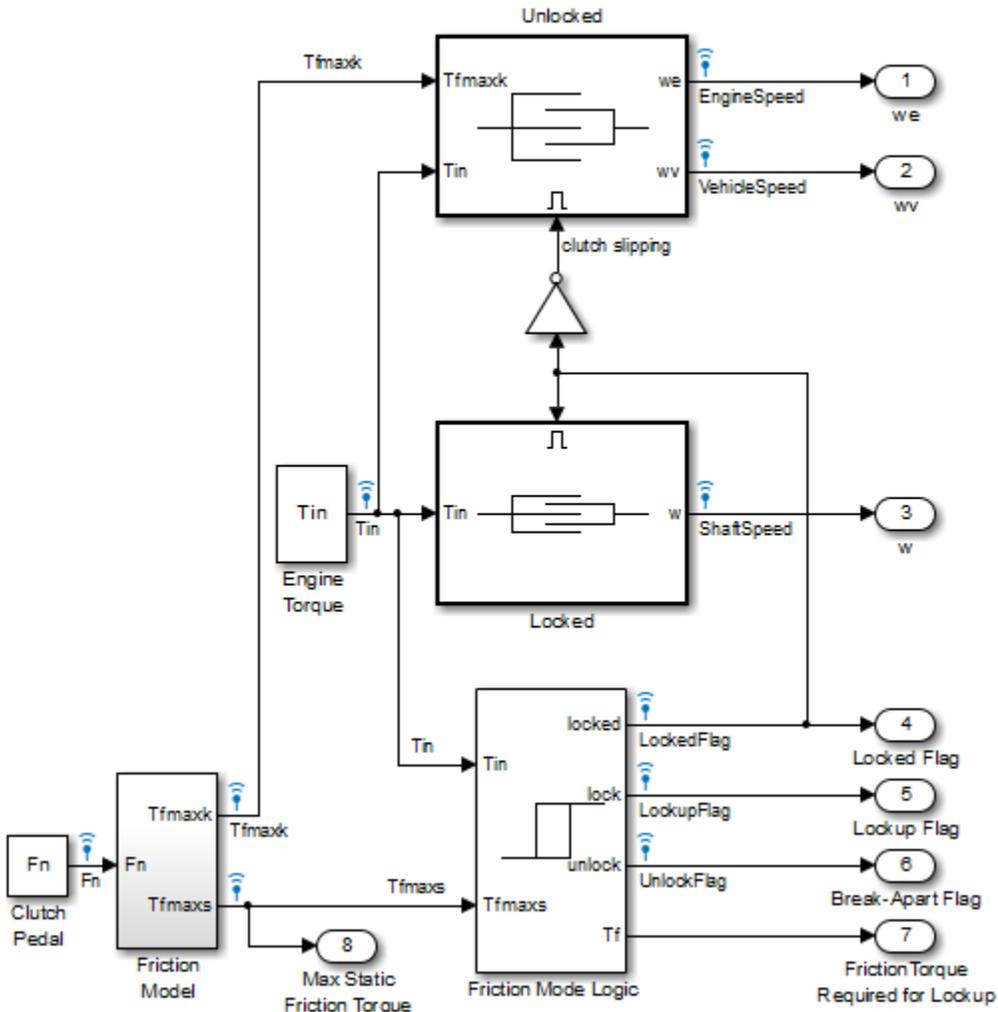
When the Logical Operator block output changes...	The Integrator block...
From 0 to 1	Resets the velocity
From 1 to 0	Does not reset velocity

Logical Operator Block: NOT Operator

In the `sldemo_clutch` model, the Logical Operator block appears as a NOT operator:



Building a Clutch Lock-Up Model An Example of Enabled Subsystems



The output of the Logical Operator block (the `clutch_slipping` signal) feeds into the trigger port of an enabled subsystem.

When the Logical Operator block outputs...	The Unlocked subsystem is...
1	Enabled
0	Disabled

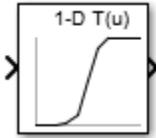
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

1-D Lookup Table

Approximate one-dimensional function



Library

Lookup Tables

Description

The 1-D Lookup Table block is a one-dimensional version of the n-D Lookup Table block.

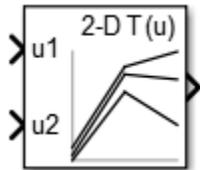
Characteristics

Data Types	Double Single Base Integer Fixed-Point
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

Introduced in R2011a

2-D Lookup Table

Approximate two-dimensional function



Library

Lookup Tables

Description

The 2-D Lookup Table block is a two-dimensional version of the n-D Lookup Table block.

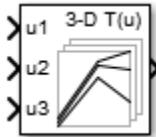
Characteristics

Data Types	Double Single Base Integer Fixed-Point
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

Introduced in R2011a

n-D Lookup Table

Approximate N-dimensional function



Library

Lookup Tables

Description

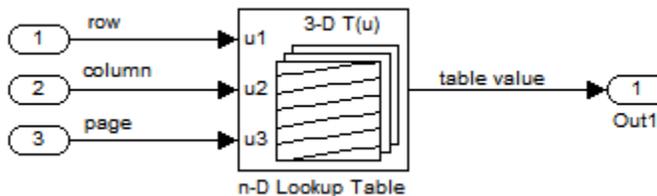
Supported Block Operations

The n-D Lookup Table block evaluates a sampled representation of a function in N variables

$$y = F(x_1, x_2, x_3, \dots, x_N)$$

where the function F can be empirical. The block maps inputs to an output value by looking up or interpolating a table of values you define with block parameters. The block supports flat (constant), linear, and cubic-spline interpolation methods. You can apply these methods to a table of any dimension from 1 through 30.

In the following block, the first input identifies the first dimension (row) breakpoints, the second input identifies the second dimension (column) breakpoints, and so on.



See “Port Location After Rotating or Flipping” in the Simulink documentation for a description of the port order for various block orientations.

Specification of Breakpoint and Table Data

The following block parameters define the breakpoint and table data.

Block Parameter	Purpose
Number of table dimensions	Specifies the number of dimensions of your lookup table.
Breakpoints	Specifies a breakpoint vector that corresponds to each dimension of your lookup table.
Table data	Defines the associated set of output values.

Tip Evenly spaced breakpoints can make the generated code division-free. For more information, see `fixpt_evenspace_cleanup` in the Simulink documentation and “Identify questionable fixed-point operations” (Embedded Coder) in the Simulink Coder documentation.

How the Block Generates Output

The n-D Lookup Table block generates output by looking up or estimating table values based on the input values:

When block inputs...	The n-D Lookup Table block...
Match the values of indices in breakpoint data sets	Outputs the table value at the intersection of the row, column, and higher dimension breakpoints
Do not match the values of indices in breakpoint data sets, but are within range	Interpolates appropriate table values, using the Interpolation method you select
Do not match the values of indices in breakpoint data sets, and are out of range	Extrapolates the output value, using the Extrapolation method you select

Other Blocks That Perform Equivalent Operations

You can use the Interpolation Using Prelookup block with the Prelookup block to perform the equivalent operation of one n-D Lookup Table block. This combination of blocks offers greater flexibility that can result in more efficient simulation performance for linear interpolations.

When the lookup operation is an array access that does not require interpolation, use the Direct Lookup Table (n-D) block. For example, if you have an integer value k and you want the k th element of a table, $y = \text{table}(k)$, interpolation is unnecessary.

Data Type Support

The n-D Lookup Table block supports all numeric data types that Simulink supports, including fixed-point data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

For cubic spline interpolation and linear extrapolation modes, the following parameters must use the same floating-point type:

- Table data
- Breakpoints
- Fraction
- Intermediate results
- Output

Inputs for indexing must be real, but table data can be complex.

Parameters

- “Table and Breakpoints Tab” on page 1-858
- “Algorithm Tab” on page 1-860
- “Data Types Tab” on page 1-865

Table and Breakpoints Tab

Number of table dimensions

Enter the number of dimensions of the lookup table by specifying an integer from 1 to 30. This parameter determines:

- The number of independent variables for the table and the number of block inputs
- The number of breakpoint sets to specify

Data specification

From the list, select:

- **Table and breakpoints** — Specify the table data and breakpoints. Selecting this option enables these parameters:
 - **Table data**
 - **Breakpoints specification**
 - **Breakpoints 1**
 - **Breakpoints 2**
 - **Breakpoints 3**
 - **Edit table and breakpoints**
- **Lookup table object** — Use an existing lookup table (`Simulink.LookupTable`) object. Selecting this option enables the **Name** field and **Edit table and breakpoints** button.

Name

Enter the name of the lookup table (`Simulink.LookupTable`) object.

Table data

Enter the table of output values.

During simulation, the matrix size must match the dimensions defined by the **Number of table dimensions** parameter. However, during block diagram editing, you can enter an empty matrix (specified as `[]`) or an undefined workspace variable. This technique lets you postpone specifying a correctly dimensioned matrix for the table data and continue editing the block diagram. For information about how to construct multidimensional arrays in MATLAB, see “Multidimensional Arrays” (MATLAB) in the MATLAB online documentation.

Breakpoints specification

Specify whether to enter data as explicit breakpoints or as parameters that generate evenly spaced breakpoints.

- To explicitly specify breakpoint data, set this parameter to `Explicit values` and enter breakpoint data in the text box next to the **Breakpoints** parameters.
- To specify parameters that generate evenly spaced breakpoints, set this parameter to `Even spacing` and enter values for the **First point** and **Spacing** parameters for each dimension of breakpoint data. The block calculates the number of points to generate from the table data.

First point

Specify the first point in your evenly spaced breakpoint data. This parameter is available when **Breakpoints specification** is set to `Even spacing`.

Spacing

Specify the spacing between points in your evenly-spaced breakpoint data. This parameter is available when **Breakpoints specification** is set to `Even spacing`.

Breakpoints

Specify the breakpoint data explicitly or as evenly-spaced breakpoints, based on the value of the **Breakpoints specification** parameter.

- If you set **Breakpoints specification** to `Even spacing`, enter the parameters **First point** and **Spacing** in each **Breakpoints** row to generate evenly-spaced breakpoints in the respective dimension. Your table data determines the number of evenly spaced points.
- If you set **Breakpoints specification** to `Explicit values`, enter the breakpoint set that corresponds to each dimension of table data in each **Breakpoints** row. For each dimension, specify breakpoints as a 1-by-n or n-by-1 vector whose values are strictly monotonically increasing.

Edit table and breakpoints

Click this button to open the Lookup Table Editor. For more information, see “Edit Lookup Tables” in the Simulink documentation.

Clicking this button for a lookup table object lets you edit the object and save the new values for the object.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Algorithm Tab

Interpolation method

Select `Flat`, `Nearest`, `Linear`, or `Cubic spline`. See “Interpolation Methods” in the Simulink documentation for more information.

If you select `Cubic spline`, the block supports only scalar signals. The other interpolation methods support nonscalar signals.

Extrapolation method

Select `Clip`, `Linear`, or `Cubic spline`. See “Extrapolation Methods” in the Simulink documentation for more information.

To select `Cubic spline` for **Extrapolation method**, you must also select `Cubic spline` for **Interpolation method**.

Use last table value for inputs at or above last breakpoint

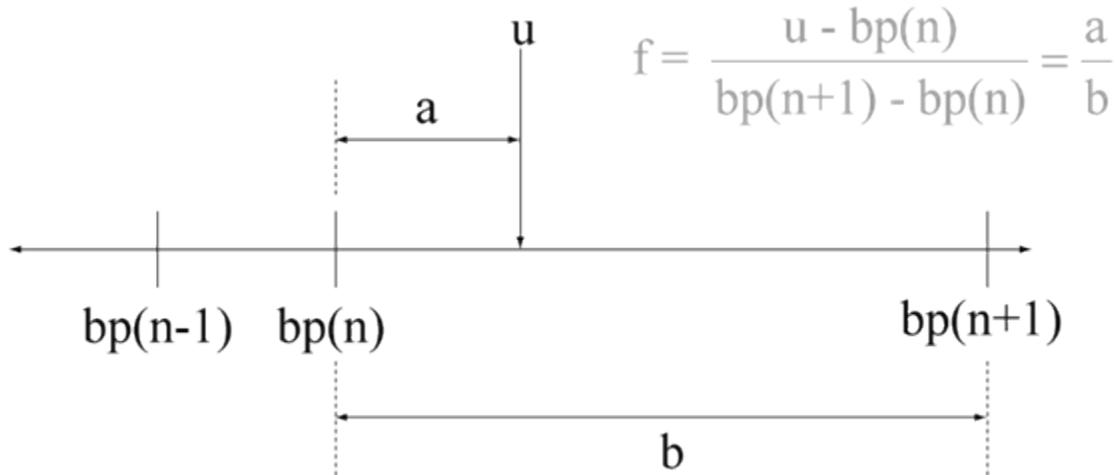
Using this check box, specify the indexing convention that the block uses to address the last element of a breakpoint set and its corresponding table value. This check box is relevant if the input is larger than the last element of the breakpoint data. This parameter is visible only when:

- **Interpolation method** is `Linear`.
- **Extrapolation method** is `Clip`.

Check Box	Block Uses Index Of The...	Interval Fraction
Selected	Last element of breakpoint data on the Table and Breakpoints tab	0

Check Box	Block Uses Index Of The...	Interval Fraction
Cleared	Next-to-last element of breakpoint data on the Table and Breakpoints tab	1

Given an input u within range of a breakpoint set bp , the interval fraction f , in the range $0 \leq f \leq 1$, is computed as shown below.



Suppose the breakpoint set is [1 4 5] and input u is 5.5. If you select this check box, the index is that of the last element (5) and the interval fraction is 0. If you clear this checkbox, the index is that of the next-to-last element (4) and the interval fraction is 1.

Diagnostic for out-of-range input

Specify whether to produce a warning or error when the input is out of range.

Options include:

- **None** — Simulink software takes no action.
- **Warning** — Simulink software displays a warning and continues the simulation.
- **Error** — Simulink software terminates the simulation and displays an error.

Remove protection against out-of-range input in generated code

Specify whether or not to include code that checks for out-of-range breakpoint input values.

Check Box	Result	When to Use
Selected	Generated code does not include conditional statements to check for out-of-range breakpoint inputs. When the input k or f is out-of-range, it may cause undefined behavior for generated code and simulations using accelerator mode.	For code efficiency
Cleared	Generated code includes conditional statements to check for out-of-range breakpoint inputs.	For safety-critical applications

If your input is not out-of-range, you can select the **Remove protection against out-of-range index in generated code** check box for code efficiency. By default, this check box is cleared. For safety-critical applications, do not select this check box. If you want to select the **Remove protection against out-of-range index in generated code** check box, first check that your model inputs are in range. For example:

- 1 Clear the **Remove protection against out-of-range index in generated code** check box.
- 2 Set the **Diagnostic for out-of-range input** parameter to `Error`.
- 3 Simulate the model in normal mode.
- 4 If there are out-of-range errors, fix them to be in range and run the simulation again.
- 5 When the simulation no longer generates out-of-range input errors, select the **Remove protection against out-of-range index in generated code** check box.

Note When you select the **Remove protection against out-of-range index in generated code** check box and the input `k` or `f` is out-of-range, the behavior is undefined for generated code and simulations using accelerator mode.

Depending on your application, you can run the following Model Advisor checks to verify the usage of this check box:

- **By Product > Embedded Coder > Identify lookup table blocks that generate expensive out-of-range checking code**
- **By Product > Simulink Check > Modeling Standards > DO-178C/DO-331 Checks > Check usage of lookup table blocks**

For more information about the Model Advisor, see “Run Model Checks” in the Simulink documentation.

Index search method

Select `Evenly spaced points`, `Linear search`, or `Binary search`. Each search method has speed advantages in different circumstances:

- For evenly spaced breakpoint sets (for example, 10, 20, 30, and so on), you achieve optimal speed by selecting `Evenly spaced points` to calculate table indices.

This algorithm uses only the first two breakpoints of a set to determine the offset and spacing of the remaining points.

Note Set **Index search method** to `Evenly spaced points` when using the `Simulink.LookupTable` object to specify table data and the **Breakpoints Specification** parameter of the referenced `Simulink.LookupTable` object is set to `Even spacing`.

- For unevenly spaced breakpoint sets, follow these guidelines:
 - If input signals do not vary much between time steps, selecting `Linear search` with **Begin index search using previous index result** produces the best performance.
 - If input signals jump more than one or two table intervals per time step, selecting `Binary search` produces the best performance.

A suboptimal choice of index search method can lead to slow performance of models that rely heavily on lookup tables.

Note The generated code stores only the first breakpoint, the spacing, and the number of breakpoints when:

- The breakpoint data is not tunable.
 - The index search method is `Evenly spaced points`.
-

Begin index search using previous index result

Select this check box when you want the block to start its search using the index found at the previous time step. For inputs that change slowly with respect to the interval size, enabling this option can improve performance. Otherwise, the linear search and binary search methods can take longer, especially for large breakpoint sets.

Use one input port for all input data

Select this check box to use only one input port that expects a signal that is N elements wide for an N -dimensional table. This option is useful for removing line clutter on a block diagram with many lookup tables.

Note When you select this check box, one input port with the label `u` appears on the block.

Support tunable table size in code generation

Select this check box to enable tunable table size in the generated code. This option enables you to change the size and values of the lookup table and breakpoint data in the generated code without regenerating or recompiling the code.

If you set **Interpolation method** to `Cubic spline`, this check box is not available.

Maximum indices for each dimension

Specify the maximum index values for each table dimension using zero-based indexing. You can specify a scalar or vector of positive integer values using the following data types:

- Built-in floating-point types: `double` and `single`
- Built-in integer types: `int8`, `int16`, `int32`, `uint8`, `uint16`, and `uint32`

Here are some examples of valid specifications:

- [4 6] for a 5-by-7 table
- [int8(2) int16(5) int32(9)] for a 3-by-6-by-10 table
- A `Simulink.Parameter` whose value on generating code is one less than the dimensions of the table data. For more information, see “Tunable Table Size in the Generated Code” on page 1-870.

This parameter is available when you select **Support tunable table size in code generation**. On tuning this parameter in the generated code, provide the new table data and breakpoints along with the tuned parameter value.

Data Types Tab

Note The dialog box can expand to show additional data type options. Up to 30 breakpoint data type specifications can appear.

Table data > Data Type

Specify the table data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as output`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the table data type.

Tip Specify a table data type different from the output data type for these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal
 - Sharing of prescaled table data between two n-D Lookup Table blocks with different output data types
 - Sharing of custom storage table data in the generated code for blocks with different output data types
-

Table data > Minimum

Specify the minimum value for table data. The default value is [] (unspecified).

Table data > Maximum

Specify the maximum value for table data. The default value is [] (unspecified).

Breakpoints > Data Type

Specify the data type for a set of breakpoint data. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as corresponding input`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the breakpoint data type.

See “Specify Data Types Using Data Type Assistant” in the Simulink documentation for more information.

Tip Specify a breakpoint data type different from the corresponding input data type for these cases:

- Lower memory requirement for storing breakpoint data that uses a smaller type than the input signal
 - Sharing of prescaled breakpoint data between two n-D Lookup Table blocks with different input data types
 - Sharing of custom storage breakpoint data in the generated code for blocks with different input data types
-

Breakpoints > Minimum

Specify the minimum value that a set of breakpoint data can have. The default value is [] (unspecified).

Breakpoints > Maximum

Specify the maximum value that a set of breakpoint data can have. The default value is [] (unspecified).

Fraction > Data Type

Specify the fraction data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the fraction data type.

See “Specify Data Types Using Data Type Assistant” in the Simulink documentation for more information.

Intermediate results > Data Type

Specify the intermediate results data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as output`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the intermediate results data type.

Tip Use this parameter to specify higher (or lower) precision for internal computations than for table data or output data.

Output > Data Type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the output data type.

See “Control Signal Data Types” for more information.

Output > Minimum

Specify the minimum value that the block outputs. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output > Maximum

Specify the maximum value that the block outputs. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Internal rule priority

Specify the internal rule for intermediate calculations. Select *Speed* for faster calculations. If you do, a loss of accuracy might occur, usually up to 2 bits.

Require all inputs to have the same data type

Select to require all inputs to have the same data type.

Lock data type settings against changes by the fixed-point tools

Select to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Fixed-Point Tool” (Fixed-Point Designer) and “Preparation for Fixed-Point Conversion” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Integer rounding mode

Specify the rounding mode for fixed-point lookup table calculations that occur during simulation or execution of code generated from the model. For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

This option does not affect rounding of values of block parameters. Simulink rounds such values to the nearest representable integer value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the edit field on the block dialog box.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Do not select this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</p>	Overflows wrap to the appropriate value that is representable by the data type.	The number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tip If you save your model as version R2009a or earlier, this check box setting has no effect and no saturation code appears. This behavior preserves backward compatibility.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Examples

Interpolation and Extrapolation Behavior

For an example that illustrates linear interpolation and extrapolation methods of this block, see “Create a Logarithm Lookup Table” in the Simulink documentation.

For an example of entering breakpoint and table data, see “Entering Data in a Block Parameter Dialog Box” in the Simulink documentation.

Tunable Table Size in the Generated Code

Suppose that you have a lookup table and want to make the size tunable in the generated code. When you use `Simulink.LookupTable` and `Simulink.Breakpoint` objects to

configure lookup table data for calibration in the generated code, use the `SupportTunableSize` property of the objects to enable a tunable table size. When you do not use these classes, use the **Support tunable table size in code generation** parameter in an n-D Lookup Table block to enable a tunable table size.

Assume that:

- You define a `Simulink.Parameter` structure in the preload function of your model:

```
p = Simulink.Parameter;
p.Value.MaxIdx = [2 2];
p.Value.BP1 = [1 2 3];
p.Value.BP2 = [1 4 16];
p.Value.Table = [4 5 6; 16 19 20; 10 18 23];
p.DataType = 'Bus: slLookupTable';
p.CoderInfo.StorageClass = 'ExportedGlobal';

% Create bus object slBus1 from MATLAB structure
Simulink.Bus.createObject(p.Value);
slLookupTable = slBus1;
slLookupTable.Elements(1).DataType = 'uint32';
```

- The following block parameters apply in the n-D Lookup Table block dialog box:

Parameter	Value
Number of table dimensions	2
Table data	<code>p.Table</code>
Breakpoints 1	<code>p.BP1</code>
Breakpoints 2	<code>p.BP2</code>
Support tunable table size in code generation	on
Maximum indices for each dimension	<code>p.MaxIdx</code>

The generated `model_types.h` header file contains a type definition that looks something like this:

```
typedef struct {
    uint32_T MaxIdx[2];
    real_T BP1[3];
    real_T BP2[3];
    real_T Table[9];
} slLookupTable;
```

The generated `model.c` file contains code that looks something like this:

```
/* Exported block parameters */
slLookupTable p = {
    { 2U, 2U },

    { 1.0, 2.0, 3.0 },

    { 1.0, 4.0, 16.0 },

    { 4.0, 16.0, 10.0, 5.0, 19.0, 18.0, 6.0, 20.0, 23.0 }
};

/* More code */

/* Model output function */
static void ex_lut_nd_tunable_table_output(int_T tid)
{
    /* Lookup_n-D: '<Root>/n-D Lookup Table' incorporates:
     * Inport: '<Root>/In1'
     * Inport: '<Root>/In2'
     */
    Y = look2_binlcpw(U1, U2, p.BP1, p.BP2, p.Table, ...
p.MaxIdx, p.MaxIdx[0] + 1U);

    /* Outport: '<Root>/Out1' */
    ex_lut_nd_tunable_table_Y.Out1 = Y;

    /* tid is required for a uniform function interface.
     * Argument tid is not used in the function. */
    UNUSED_PARAMETER(tid);
}
```

The highlighted line of code specifies a tunable table size for the lookup table. You can change the size and values of the lookup table and breakpoint data without regenerating or recompiling the code.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Inherited from driving block

Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Prelookup, Interpolation Using Prelookup, `Simulink.LookupTable`,
`Simulink.Breakpoint`

See Also

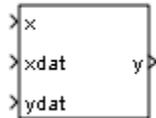
Topics

“Import Lookup Table Data from MATLAB”

Introduced in R2011a

Lookup Table Dynamic

Approximate one-dimensional function using dynamic table



Library

Lookup Tables

Description

How This Block Differs from Other Lookup Table Blocks

The Lookup Table Dynamic block computes an approximation of a function $y = f(x)$ using `xdat` and `ydat` vectors. The lookup method can use interpolation, extrapolation, or the original values of the input.

Using the Lookup Table Dynamic block, you can change the table data without stopping the simulation. For example, you can incorporate new table data if the physical system you are simulating changes.

Inputs for Breakpoint and Table Data

The `xdat` vector is the breakpoint data, which must be *strictly monotonically increasing*. The value of the next element in the vector must be greater than the value of the preceding element after conversion to a fixed-point data type. Due to quantization, `xdat` can be strictly monotonic for a floating-point data type, but not after conversion to a fixed-point data type.

The `ydat` vector is the table data, which is an evaluation of the function at the breakpoint values.

Note The inputs to `xdat` and `ydat` cannot be scalar (one-element array) values. If you provide a scalar value to either of these inputs, you see an error upon simulation. Provide a 1-by-n vector to both the `xdat` and `ydat` inputs.

Lookup Table Definition

You define the lookup table by feeding `xdat` and `ydat` as 1-by-n vectors to the block. To reduce ROM usage in the generated code for this block, you can use different data types for `xdat` and `ydat`. However, these restrictions apply:

- The `xdat` breakpoint data and the `x` input vector must have the same sign, bias, and fractional slope. Also, the precision and range for `x` must be greater than or equal to the precision and range for `xdat`.
- The `ydat` table data and the `y` output vector must have the same sign, bias, and fractional slope.

Tip Breakpoints with even spacing can make Simulink Coder generated code division-free. For more information, see `fixpt_evenspace_cleanup` in the Simulink documentation and “Identify questionable fixed-point operations” (Embedded Coder) in the Simulink Coder documentation.

How the Block Generates Output

The block uses the input values to generate output using the method you select for **Lookup Method**:

Lookup Method	Block Action
Interpolation-Extrapolation	<p>Performs linear interpolation and extrapolation of the inputs.</p> <ul style="list-style-type: none"> • If the input matches a breakpoint, the output is the corresponding element in the table data. • If the input does not match a breakpoint, the block performs linear interpolation between two elements of the table to determine the output. If the input falls outside the range of breakpoint values, the block extrapolates using the first two or last two points. <p>Note If you select this lookup method, Simulink Coder software cannot generate code for this block.</p>
Interpolation-Use End Values (default)	Performs linear interpolation but does not extrapolate outside the end points of the breakpoint data. Instead, the block uses the end values.
Use Input Nearest	Finds the element in <code>xdat</code> nearest the current input. The corresponding element in <code>ydat</code> is the output.
Use Input Below	Finds the element in <code>xdat</code> nearest and below the current input. The corresponding element in <code>ydat</code> is the output. If there is no element in <code>xdat</code> below the current input, the block finds the nearest element.
Use Input Above	Finds the element in <code>xdat</code> nearest and above the current input. The corresponding element in <code>ydat</code> is the output. If there is no element in <code>xdat</code> above the current input, the block finds the nearest element.

Note The Use Input Nearest, Use Input Below, and Use Input Above methods perform the same action when the input `x` matches a breakpoint value.

Some continuous solvers subdivide the simulation time span into major and minor time steps. A minor time step is a subdivision of the major time step. The solver produces a result at each major time step and uses results at minor time steps to improve the accuracy of the result at the major time step. For continuous solvers, the output of the

Lookup Table Dynamic block can appear like a stair step because the signal is fixed in minor time step to avoid incorrect results. For more information about the effect of solvers on block output, see “Solvers” in the Simulink documentation.

Data Type Support

The Lookup Table Dynamic block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

- “Main tab” on page 1-877
- “Signal Attributes tab” on page 1-877

Main tab

Lookup Method

Specify the lookup method. For details, see “How the Block Generates Output” on page 1-875.

Signal Attributes tab

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`

- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt('double')`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer), in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Examples

For an example of...	See...
Breakpoint and table data entry	“Entering Data Using Inports of the Lookup Table Dynamic Block”
Block output for different lookup methods	“Example Output for Lookup Methods”

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
------------	--------------------------------------------------------

Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

n-D Lookup Table

Introduced before R2006a

Magnitude-Angle to Complex

Convert magnitude and/or a phase angle signal to complex signal



Library

Math Operations

Description

Supported Operations

The Magnitude-Angle to Complex block converts magnitude and phase angle inputs to a complex output. The angle input must be in radians.

The block supports the following combinations of input dimensions when there are two block inputs:

- Two inputs of equal dimensions
- One scalar input and the other an n-dimensional array

If the block input is an array, the output is an array of complex signals. The elements of a magnitude input vector map to the magnitudes of the corresponding complex output elements. Similarly, the elements of an angle input vector map to the angles of the corresponding complex output elements. If one input is a scalar, it maps to the corresponding component (magnitude or angle) of all the complex output signals.

Effect of Out-of-Range Input on CORDIC Approximations

If you use the CORDIC approximation method (see “Definitions” on page 1-881), the block input for phase angle has the following restrictions:

- For signed fixed-point types, the input angle must fall within the range $[-2\pi, 2\pi)$ radians.
- For unsigned fixed-point types, the input angle must fall within the range $[0, 2\pi)$ radians.

The following table summarizes what happens for an out-of-range input:

Block Usage	Effect of Out-of-Range Input
Simulation	An error appears.
Generated code	Undefined behavior occurs.
Accelerator modes	

Ensure that you use an in-range input for the Magnitude-Angle to Complex block when you use the CORDIC approximation. Avoid relying on undefined behavior for generated code or Accelerator modes.

Definitions

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

Data Type Support

The block accepts real input signals of the following data types:

- Floating point
- Fixed point (only when **Approximation method** is CORDIC)

The following restrictions also apply:

- If one input uses a floating-point type, the other input must use the same data type. For example, both signals must be `double` or `single`.
- If one input uses a fixed-point type, the other input must also use a fixed-point type.

Parameters

Input

Specify the kind of input: a magnitude input, an angle input, or both.

Angle (Magnitude)

Input	What to Specify
Magnitude	The constant phase angle of the output signal in radians
Angle	The constant magnitude of the output signal

This parameter is not available when **Input** is `Magnitude` and `angle`.

Approximation method

Specify the type of approximation for computing output.

Approximation Method	Data Types Supported	When to Use This Method
None (default)	Floating point	You want to use the default Taylor series algorithm.
CORDIC	Floating point and fixed point	You want a fast, approximate calculation.

When you use the CORDIC approximation, follow these guidelines:

- For signed fixed-point types, the input angle must fall within the range $[-2\pi, 2\pi)$ radians.
- For unsigned fixed-point types, the input angle must fall within the range $[0, 2\pi)$ radians.

The block uses the following data type propagation rules:

Data Type of Magnitude Input	Approximation Method	Data Type of Complex Output
Floating point	None or CORDIC	Same as input
Signed, fixed point	CORDIC	$\text{fixdt}(1, WL + 2, FL)$ where WL and FL are the word length and fraction length of the magnitude
Unsigned, fixed point	CORDIC	$\text{fixdt}(1, WL + 3, FL)$ where WL and FL are the word length and fraction length of the magnitude

Number of iterations

Specify the number of iterations to perform the CORDIC algorithm. The default value is 11.

Data Type of Block Inputs	Value You Can Specify
Floating point	A positive integer
Fixed point	A positive integer that does not exceed the word length of the magnitude input or the word length of the phase angle input, whichever value is smaller

Entering a value that is not a positive integer causes an error.

This parameter is available when you set **Approximation method** to CORDIC.

Scale output by reciprocal of gain factor

Select this check box to scale the real and imaginary parts of the complex output by a factor of $(1/\text{CORDIC gain})$. This value depends on the number of iterations you specify. As the number of iterations goes up, the value approaches 1.647.

This check box is selected by default, which leads to a more numerically accurate result for the complex output, $X + iY$. However, scaling the output adds two extra multiplication operations, one for X and one for Y .

This parameter is available when you set **Approximation method** to CORDIC.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1 . To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Characteristics

Data Types	Double Single Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Complex to Magnitude-Angle

References

- [1] Volder, JE. “The CORDIC Trigonometric Computing Technique.” *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.
- [2] Andraka, R. “A survey of CORDIC algorithm for FPGA based computers.” *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. Feb. 22–24, 1998, pp. 191–200.

- [3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf
- [4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

Introduced before R2006a

Manual Switch

Switch between two inputs



Library

Signal Routing

Description

The Manual Switch block is a toggle switch that selects one of its two inputs to pass through to the output. To toggle between inputs, double-click the block. The block propagates the selected input to the output, while the block discards the unselected input. You can interactively control the signal flow by setting the switch before you start the simulation or by changing the switch while the simulation is executing. The Manual Switch block retains its current state when you save the model.

Data Type Support

The Manual Switch block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

- “Allow the two inputs to differ in size” on page 1-887
- “Sample time” on page 1-887

Allow the two inputs to differ in size

Select this check box to allow input signals with different sizes.

Default: Off

On

Block allows input signals with different sizes, and propagates the input signal size to the output signal.

Off

Block expands scalar inputs to have the same dimensions as nonscalar inputs. For more information, see “Scalar Expansion of Inputs and Parameters”.

Parameter: `varsize`

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Examples

The following models show how to use the Manual Switch block:

- `sldemo_auto_climatecontrol`
- `sldemo_fuelsys`
- `sldemo_doublebounce`

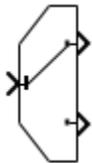
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Manual Variant Sink

Switch amongst multiple variant choices at output



Library

Signal Routing

Description

The Manual Variant Sink block is a toggle switch that activates one of its variant choices at the output to pass the input.

It can have two or more output ports and has one input port. Each output port is associated with a variant control.

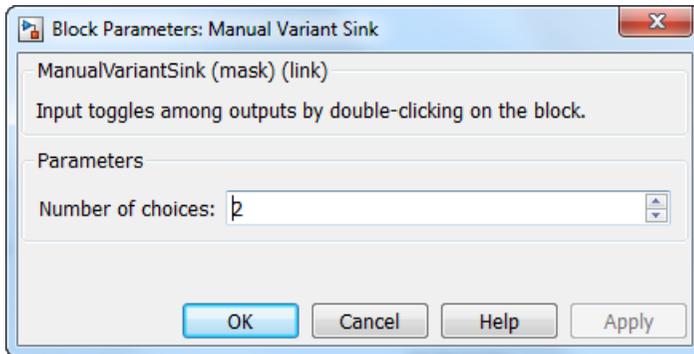
To toggle between the variant choices at output, double-click the block. The block propagates the active variant choice at output and discards the blocks connected to inactive output ports during simulation.

Data Type Support

The Manual Variant Sink block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Dialog Box and Parameters

To open the block dialog box, right-click the block and select **Mask > Mask Parameters**.



By default, the number of output choice is 2. You can change the number of outputs by typing a value in the **Number of choices** box. The maximum number of output choices can be 25.

Note Double-clicking the Manual Variant Sink block does not open the block dialog box instead it toggles the output choice.

Command-Line Information

Parameter: NumChoices

Type: character vector

Value: 2-25

Default: 2

Examples

To view how to use the Manual Variant Sink block, click Manual Variant Source and Sink Blocks.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from driving block

Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

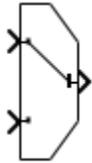
See Also

Variant Sink | Variant Source

Introduced in R2016b

Manual Variant Source

Switch amongst multiple variant choices at input



Library

Signal Routing

Description

The Manual Variant Source block is a toggle switch that activates one of its variant choices at the input to pass through to the output.

A Manual Variant Source block can have two or more input ports and has one output port. Each input port is associated with a variant control.

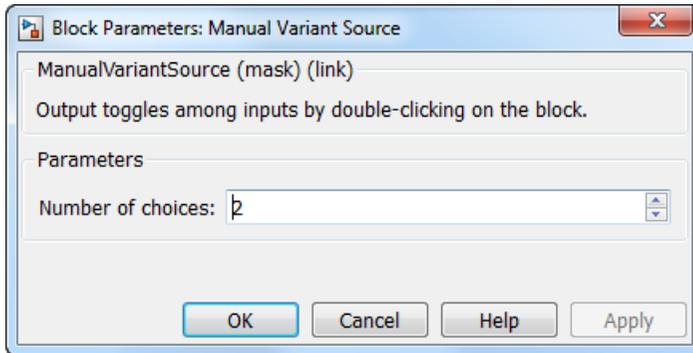
To toggle between the variant choices at input, double-click the block. The block propagates the active variant choice at input directly to the output and discards the blocks connected to inactive input ports during simulation.

Data Type Support

The Manual Variant Source block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Dialog Box and Parameters

To open the block dialog box, right-click the block and select **Mask > Mask Parameters**.



By default, the number of input choices is 2. You can change the number of inputs by typing a value in the **Number of choices** box. The maximum number of input choices can be 25.

Note Double-clicking the Manual Variant Source block does not open the block dialog box instead it toggles the input choice.

Command-Line Information

Parameter: NumChoices

Type: character vector

Value: 2-25

Default: 2

Examples

To view how to use the Manual Variant Source block, click Manual Variant Source and Sink Blocks.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from driving block

Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Variant Sink | Variant Source

Introduced in R2016b

Math Function

Perform mathematical function



Library

Math Operations

Description

The Math Function block performs numerous common mathematical functions.

Tip To perform square root calculations, use the Sqrt block.

You can select one of the following functions from the **Function** parameter list.

Function	Description	Mathematical Expression	MATLAB Equivalent
exp	Exponential	e^u	exp
log	Natural logarithm	$\ln u$	log
10^u	Power of base 10	10^u	10.^u (see power)
log10	Common (base 10) logarithm	$\log u$	log10
magnitude^2	Complex modulus	$ u ^2$	(abs(u)).^2 (see abs and power)

Function	Description	Mathematical Expression	MATLAB Equivalent
square	Power 2	u^2	<code>u.^2</code> (see <code>power</code>)
pow	Power	u^v	<code>power</code>
conj	Complex conjugate	\bar{u}	<code>conj</code>
reciprocal	Reciprocal	$1/u$	<code>1./u</code> (see <code>rdivide</code>)
hypot	Square root of sum squares	$(u^2+v^2)^{0.5}$	<code>hypot</code>
rem	Remainder after division	—	<code>rem</code>
mod	Modulus after division	—	<code>mod</code>
transpose	Transpose	u^T	<code>u.'</code> (see “Array vs. Matrix Operations” (MATLAB))
hermitian	Complex conjugate transpose	u^H	<code>u'</code> (see “Array vs. Matrix Operations” (MATLAB))

The block output is the result of the operation of the function on the input or inputs. The functions support the following types of operations.

Function	Scalar Operations	Element-Wise Vector and Matrix Operations	Vector and Matrix Operations
<code>exp</code>	yes	yes	—
<code>log</code>	yes	yes	—
<code>10^u</code>	yes	yes	—
<code>log10</code>	yes	yes	—
<code>magnitude^2</code>	yes	yes	—
<code>square</code>	yes	yes	—
<code>pow</code>	yes	yes	—

Function	Scalar Operations	Element-Wise Vector and Matrix Operations	Vector and Matrix Operations
conj	yes	yes	—
reciprocal	yes	yes	—
hypot	yes, on two inputs	yes, on two inputs (two vectors or two matrices of the same size, a scalar and a vector, or a scalar and a matrix)	—
rem	yes, on two inputs	yes, on two inputs (two vectors or two matrices of the same size, a scalar and a vector, or a scalar and a matrix)	—
mod	yes, on two inputs	yes, on two inputs (two vectors or two matrices of the same size, a scalar and a vector, or a scalar and a matrix)	—
transpose	yes	—	yes
hermitian	yes	—	yes

The name of the function appears on the block. The appropriate number of input ports appears automatically.

Tip Use the Math Function block instead of the Fcn block when you want vector or matrix output, because the Fcn block produces only scalar output.

Data Type Support

The following table shows the input data types that each function of the block can support.

Function	single	double	boolean	built-in integer	fixed point
exp	yes	yes	—	—	—

Function	single	double	boolean	built-in integer	fixed point
log	yes	yes	—	—	—
10 ^u	yes	yes	—	—	—
log10	yes	yes	—	—	—
magnitude ²	yes	yes	—	yes	yes
square	yes	yes	—	yes	yes
pow	yes	yes	—	—	—
conj	yes	yes	—	yes	yes
reciprocal	yes	yes	—	yes	yes
hypot	yes	yes	—	—	—
rem	yes	yes	—	yes	—
mod	yes	yes	—	yes	—
transpose	yes	yes	yes	yes	yes
hermitian	yes	yes	—	yes	yes

All supported modes accept both real and complex inputs, except for reciprocal, which does not accept complex fixed-point inputs.

The block output is real or complex, depending on what you select for **Output signal type**.

Parameters

Function

Specify the mathematical function. See Description on page 1-895 for more information about the options for this parameter.

Output signal type

Specify the output signal type of the Math Function block as auto, real, or complex.

Function	Input Signal Type	Output Signal Type		
		Auto	Real	Complex
exp, log, 10u, log10, square, pow, reciprocal, conjugate, transpose, hermitian	real	real	real	complex
	complex	complex	error	complex
magnitude squared	real	real	real	complex
	complex	real	real	complex
hypot, rem, mod	real	real	real	complex
	complex	error	error	error

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1 . To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Note Some parameters on this pane are available only when the function you select in the **Function** parameter supports fixed-point data types.

Output minimum

Specify the minimum value that the block can output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value that the block can output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)

- Automatic scaling of fixed-point data types

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” in *Simulink User's Guide* for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer). in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</p>	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code> , which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code> , is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect

when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes, of the input when the function requires two inputs
Dimensionalized	Yes
Multidimensionalized	Yes, for all functions except <code>hermitian</code> and <code>transpose</code>
Zero-Crossing Detection	No

See Also

Sqrt, Trigonometric Function

Introduced before R2006a

MATLAB Function

Include MATLAB code in models that generate embeddable C code



Library

User-Defined Functions

Description

With a MATLAB Function block, you can write a MATLAB function for use in a Simulink model. The MATLAB function you create executes for simulation and generates code for a Simulink Coder target. If you are new to the Simulink and MATLAB products, see “What Is a MATLAB Function Block?” and “Create Model That Uses MATLAB Function Block” for an overview.

Double-clicking the MATLAB Function block opens its editor, where you write the MATLAB function, as in this example:

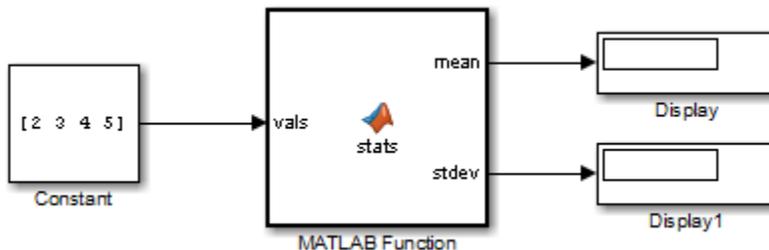
```

1  function [mean,stdev] = stats(vals)
2  % #codegen
3
4  % calculates a statistical mean and a standard
5  % deviation for the values in vals.
6
7  - len = length(vals);
8  - mean = avg(vals,len);
9  - stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
10 - coder.extrinsic('plot');
11 - plot(vals,'-+');
12 |
13 function mean = avg(array,size)
14 - mean = sum(array)/size;

```

To learn more about this editor, see “MATLAB Function Block Editor”.

You specify input and output data to the MATLAB Function block in the function header as arguments and return values. The argument and return values of the preceding example function correspond to the inputs and outputs of the block in the model:



You can also define data, input triggers, and function call outputs using the Ports and Data Manager, which you access from the MATLAB Function Block Editor by selecting **Edit Data**. See “Ports and Data Manager”.

The MATLAB Function block generates efficient embeddable code based on an analysis that determines the size, class, and complexity of each variable. This analysis imposes the following restrictions:

- The first assignment to a variable defines its, size, class, and complexity.

See “Best Practices for Defining Variables for C/C++ Code Generation”.

- You cannot reassign variable properties after the initial assignment except when using variable-size data or reusing variables in the code for different purposes.

See “Reassignment of Variable Properties”.

In addition to language restrictions, the MATLAB Function block supports a subset of the functions available in MATLAB. A list of supported functions is given in “Functions and Objects Supported for C/C++ Code Generation — Alphabetical List”. These functions include functions in common categories, such as:

- Arithmetic operators like `plus`, `minus`, and `power`. For more information, see “Array vs. Matrix Operations” (MATLAB).
- Matrix operations like `size`, and `length`
- Advanced matrix operations like `lu`, `inv`, `svd`, and `chol`
- Trigonometric functions like `sin`, `cos`, `sinh`, and `cosh`

See “Functions and Objects Supported for C/C++ Code Generation — Category List” for a complete list of function categories.

Note Although the code for this block attempts to produce exactly the same results as MATLAB, differences might occur due to rounding errors. These numerical differences, which might be a few `eps` initially, can magnify after repeated operations. Reliance on the behavior of `nan` is not recommended. Different C compilers can yield different results for the same computation.

Note In the MATLAB Function block, the `%#codegen` directive is included to emphasize that the block’s MATLAB algorithm is always intended for code generation. The `%#codegen` directive, or the absence of it, does not change the error checking behavior in the context of the MATLAB Function block. For more information see “Compilation Directive `%#codegen`”.

To support visualization of data, the MATLAB Function block supports calls to MATLAB functions for simulation only. See “Extrinsic Functions” to understand some of the limitations of this capability, and how it integrates with code analysis for this block. If

these function calls do not directly affect any of the Simulink inputs or outputs, the calls do not appear in Simulink Coder generated code.

From MATLAB Function blocks, you can also call functions defined in a Simulink Function block. You can call Stateflow functions with **Export Chart Level Functions (Make Global)** and **Allow exported functions to be called by Simulink** checked in the chart Properties dialog box.

In the Ports and Data Manager, you can declare a block input to be a Simulink parameter instead of a port. The MATLAB Function block also supports inheritance of types and size for inputs, outputs, and parameters. You can also specify these properties explicitly. See “Type Function Arguments”, “Size Function Arguments”, and “Add Parameter Arguments” for descriptions of variables that you use in MATLAB Function blocks.

Recursive calls are not allowed in MATLAB Function blocks.

By default, MATLAB Function blocks have direct feedthrough enabled. To disable it, in the Ports and Data Manager, clear the **Allow direct feedthrough** check box. Nondirect feedthrough enables semantics to ensure that outputs rely only on current state. Using nondirect feedthrough enables you to use MATLAB Function blocks in a feedback loop and prevent algebraic loops.

Data Type Support

The MATLAB Function block accepts inputs of any type that Simulink supports, including fixed-point and enumerated types. For more information, see “Data Types Supported by Simulink”.

Data types supported by MATLAB but not supported by Simulink may not be passed between the Simulink model and the function within the MATLAB Function block. These types may be used within the MATLAB Function block.

For more information on fixed-point support for this block, refer to “Fixed-Point Data Types with MATLAB Function Block” (Fixed-Point Designer) and “MATLAB Function Block with Data Type Override” (Fixed-Point Designer).

Parameters

See the reference page for the Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem blocks for information about each block parameter.

Examples

The following models shows how to use the MATLAB Function block:

- `sldemo_radar_eml`
- `sldemo_eml_galaxy`

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Specified in the Sample time parameter
Direct Feedthrough	Yes (default). To disable, in the Ports and Data Manager, clear the Allow direct feedthrough check box.
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced in R2011a

MATLAB System

Include System object in model



Library

User-Defined Functions

Description

The MATLAB System block brings existing System objects (based on `matlab.System`) into Simulink. It also enables you to use System object APIs to develop new blocks for Simulink. For more information on this block, see “MATLAB System Block”.

For interpreted execution, the model simulates the block using the MATLAB execution engine.

For code generation, the model simulates the block using code generation (using the subset of MATLAB® code supported for code generation). The MATLAB System block supports only a subset of the functions available in MATLAB. See “Functions and Objects Supported for C/C++ Code Generation — Alphabetical List” for a complete list of functions. These functions include those in common categories, such as:

- “Array vs. Matrix Operations” (MATLAB), like `plus`, `minus`, and `power`
- Matrix operations, like `size` and `length`
- Advanced matrix operations, like `lu`, `inv`, `svd`, and `chol`
- Trigonometric functions, like `sin`, `cos`, `sinh`, and `cosh`

System Objects

To use the MATLAB System block, you must first have a new System object or use an existing one. For more information, see “System Object Integration”.

Data Type Support

The MATLAB System block accepts inputs of most types that Simulink supports. It does not support virtual buses as input or output. It does not support nonvirtual buses that contain variable-size signals. For more information, see “Data Types Supported by Simulink”.

For information on fixed-point support for this block, see “Code Acceleration and Code Generation from MATLAB” (Fixed-Point Designer).

The MATLAB System block supports Simulink frames. For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

Parameters

System object name

Specify the full name of the user-defined System object class without the file extension. This entry is case sensitive. The class name must exist on the MATLAB path.

You can specify a System object name in one of these ways:

- Enter the name in the text box.
- Click the list arrow attached to the text box. If valid System objects exist in the current folder, the names appear in the list. Select a System object from this list.
- Browse to a folder that contains a valid System object. If the folder is not on your MATLAB path, the software prompts you to add it.

If you need to create a System object, you can create one from a template by clicking **New**.

After you save the System object, the name appears in the **System object name** text box.

Default: None

Use the full name of the user-defined System object class name. The block does not accept a MATLAB variable that you have assigned to a System object class name.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

New

Click this button to create a System object from a template.

Select one of these options.

- Basic

Starts MATLAB Editor and displays a template for a simple System object using the fewest System object methods.

- Advanced

Starts MATLAB Editor and displays a template for a more advanced System object using most of the System object methods.

- Simulink Extension

Starts MATLAB Editor and displays a file that contains utilities for customizing the block for Simulink. This is the same file available in MATLAB when you select **New > System Object > Simulink Extension**.

After you save the System object, you can enter the name in the **System object name** text box.

Default: Basic

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Simulate using

Select the simulation mode.

Default: Code generation

Code generation

On the first model run, simulate and generate code for MATLAB System block using only MATLAB functions supported for code generation. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is Code generation, system objects accept a maximum of 32 inputs.

Interpreted execution

Simulate model using all supported MATLAB functions. Choosing this option can slow simulation performance.

After you assign a valid System object class name to the block, the next time you open the block dialog box, the parameter is visible. This parameter appears for every MATLAB System block. You cannot remove it.

- If the block has no tabs, this parameter appears at the bottom of the dialog box.
- If the block has multiple tabs, this parameter appears at the bottom of the first tab of the dialog box.

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off

On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

This check box appears when you use the `showFiSettingsImpl` method in the `System` object.

Parameter: `SaturateOnIntegerOverflow`

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Treat these inherited Simulink signal types as fi objects

Specify which inherited data types to treat as fi data types.

Default: Fixed-point

Fixed-point

Treat fixed-point data types as fi data types.

Fixed-point & Integer

Treat fixed-point and integer data types as fi data types.

This check box appears when you use the `showFiSettingsImpl` method in the `System` object.

MATLAB System `fimath`

Specify which fixed-point math settings to use.

Default: Same as MATLAB

Fixed-point

Use the current MATLAB fixed-point math settings.

Specify Other

Enable the edit box for specifying the desired fixed-point math settings. For information on setting fixed-point math, see `fimath`.

This check box appears when you use the `showFiSettingsImpl` method in the System object.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

MATLAB Function

Topics

“MATLAB System Block”

“What Are System Objects?” (MATLAB)

Introduced in R2013b

Memory

Output input from previous time step



Library

Discrete

Description

The Memory block holds and delays its input by one major integration time step. When placed in an iterator subsystem, it holds and delays its input by one iteration. This block accepts continuous and discrete signals. The block accepts one input and generates one output. Each signal can be scalar or vector. If the input is a vector, the block holds and delays all elements of the vector by the same time step.

You specify the block output for the first time step using the **Initial condition** parameter. Careful selection of this parameter can minimize unwanted output behavior. However, you cannot specify the sample time. This block's sample time depends on the type of solver used, or you can specify to inherit it. The **Inherit sample time** parameter determines whether sample time is inherited or based on the solver.

Tip Avoid using the Memory block when both these conditions are true:

- Your model uses the variable-step solver `ode15s` or `ode113`.
 - The input to the block changes during simulation.
-

When the Memory block inherits a discrete sample time, the block is analogous to the Unit Delay block. However, the Memory block does not support state logging. If logging the final state is necessary, use a Unit Delay block instead.

Comparison with Similar Blocks

Blocks with Similar Functionality

The Unit Delay, Memory, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different. The sections that follow highlight some of these differences.

Recommended Usage for Each Block

Block	Purpose of the Block	Reference Examples
Unit Delay	Implement a delay using a discrete sample time that you specify. The block accepts and outputs signals with a discrete sample time.	<ul style="list-style-type: none"> sldemo_enginewc (Compression subsystem)
Memory on page 1-915	Implement a delay by one major integration time step. Ideally, the block accepts continuous (or fixed in minor time step) signals and outputs a signal that is fixed in minor time step.	<ul style="list-style-type: none"> sldemo_bounce sldemo_clutch (Friction Mode Logic/Lockup FSM subsystem)
Zero-Order Hold	Convert an input signal with a continuous sample time to an output signal with a discrete sample time.	<ul style="list-style-type: none"> sldemo_radar_eml aero_dap3dof

Overview of Block Capabilities

Capability	Block		
	Unit Delay	Memory	Zero-Order Hold
Specification of initial condition	Yes	Yes	No, because the block output at time $t = 0$ must match the input value.

Capability	Block		
	Unit Delay	Memory	Zero-Order Hold
Specification of sample time	Yes	No, because the block can only inherit sample time (from the driving block or the solver used for the entire model).	Yes
Support for frame-based signals	Yes	No	Yes
Support for state logging	Yes	No	No

Data Type Support

The Memory block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Initial condition

Specify the output at the initial integration step. This value must be 0 when you do not use a built-in input data type. Simulink does not allow the initial output of this block to be `inf` or `NaN`.

Inherit sample time

Select to inherit the sample time from the driving block:

- If the driving block has a discrete sample time, the block inherits the sample time.
- If the driving block has a continuous sample time, selecting this checkbox has no effect. The sample time depends on the type of solver used for simulating the model.

When this check box is cleared, the block sample time depends on the type of solver used for simulating the model:

- If the solver is a variable-step solver, the block sample time is continuous but fixed in minor time step: $[0, 1]$.
- If the solver is a fixed-step solver, the $[0, 1]$ sample time converts to the solver step size after sample-time propagation.

Direct feedthrough of input during linearization

Select to output the input during linearization and trim. This selection sets the block mode to direct feedthrough.

Selecting this check box can cause a change in the ordering of states in the model when using the functions `linmod`, `dlinmod`, or `trim`. To extract this new state ordering, use the following commands.

First compile the model using the following command, where `model` is the name of the Simulink model.

```
[sizes, x0, x_str] = model([], [], [], 'lincompile');
```

Next, terminate the compilation with the following command.

```
model([], [], [], 'term');
```

The output argument, `x_str`, which is a cell array of the states in the Simulink model, contains the new state ordering. When passing a vector of states as input to the `linmod`, `dlinmod`, or `trim` functions, the state vector must use this new state ordering.

Treat as a unit delay when linearizing with discrete sample time

Select to linearize the Memory block to a unit delay when the Memory block is driven by a signal with a discrete sample time.

State name

Use this parameter to assign a unique name to the block state. The default is ' '. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.

- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder).

State name must resolve to Simulink signal object

Select this check box to require that the state name resolve to a Simulink signal object. This check box is cleared by default.

State name enables this parameter. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class**.

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see “Apply Custom Storage Classes Directly to Signal Lines, Block States, and Output Blocks” (Embedded Coder).

To programmatically set this parameter, use `StateSignalObject`.

For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

StorageClass

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than Simulink.

State name enables this parameter.

To programmatically set this parameter, use `StateStorageClass` or `StateSignalObject`. See “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder).

TypeQualifier

Note **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

During simulation, the block uses the following values:

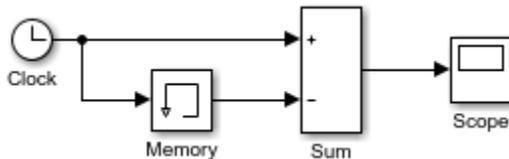
- The initial value of the signal object to which the state name is resolved
- Min and Max values of the signal object

See “Discrete Block State Naming in Generated Code” (Simulink Coder) in the Simulink Coder documentation for more information.

Examples of Memory Block Usage

Usage with the Clock Block

The following model shows how to display the step size in a simulation. The Sum block subtracts the time at the previous step, which the Memory block generates, from the current time, which the Clock block generates.

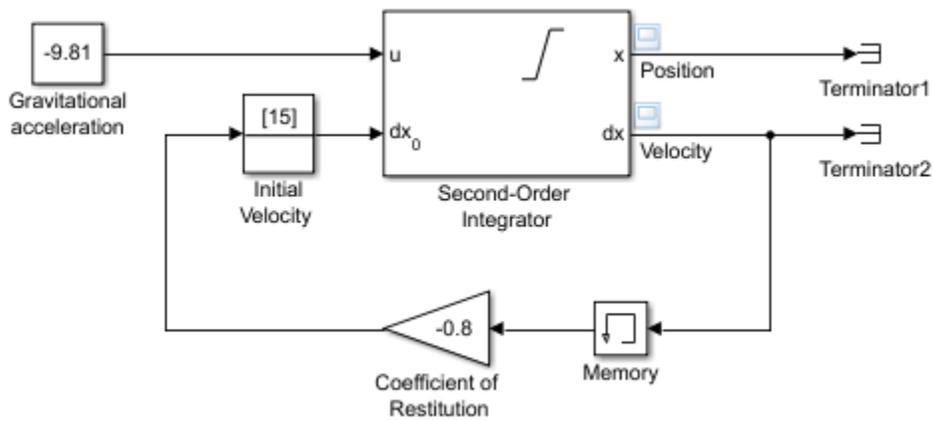


Because **Inherit sample time** is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a fixed-step solver. Therefore, the sample time of the Memory block is the solver step size, or 1.

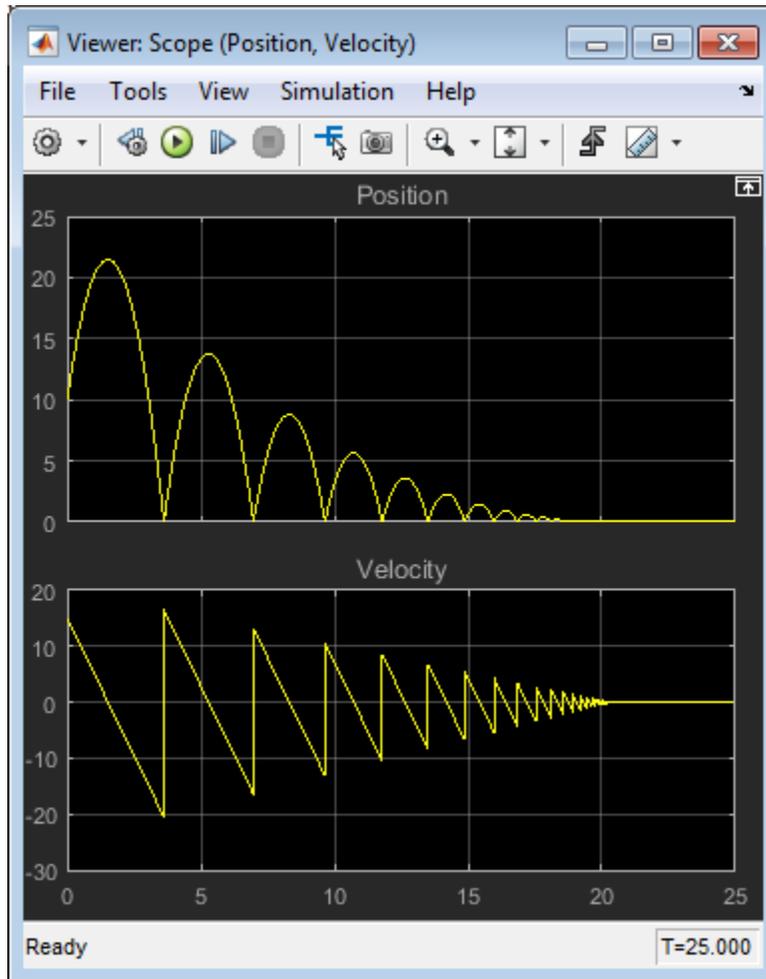
If you replace the Memory block with a Unit Delay block, you get the same results. The Unit Delay block inherits a discrete sample time of 1.

Usage with the Second-Order Integrator Block

The `sldemo_bounce` model shows how a bouncing ball reacts after being tossed into the air. The **dx** port of the Second-Order Integrator block and the Memory block capture the velocity of the ball just before it hits the ground.



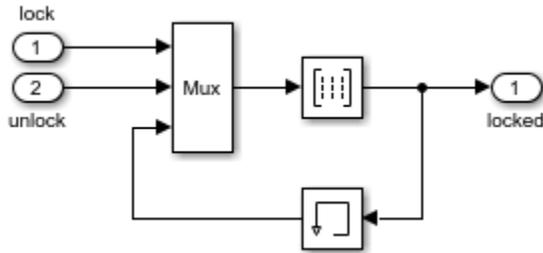
Because **Inherit sample time** is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a variable-step (*ode23*) solver. Therefore, the sample time of the Memory block is continuous but fixed in minor time step: $[0, 1]$. When you run the model, you get the following results:



If you replace the Memory block with a Unit Delay block, you get the same results. However, a warning also appears due to the discrete Unit Delay block inheriting a continuous sample time.

Usage with the Combinatorial Logic Block

The `sldemo_clutch` model shows how you can use the Memory block with the Combinatorial Logic block to implement a finite-state machine. This construct appears in the Friction Mode Logic/Lockup FSM subsystem.



Because **Inherit sample time** is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a variable-step (`ode23`) solver. Therefore, the sample time of the Memory block is continuous but fixed in minor time step: $[0, 1]$.

Bus Support

The Memory block is a bus-capable block. The input can be a virtual or nonvirtual bus signal subject to the following restrictions:

- **Initial condition** must be zero, a nonzero scalar, or a finite numeric structure.
- If **Initial condition** is zero or a structure, and you specify a **State name**, the input cannot be a virtual bus.
- If **Initial condition** is a nonzero scalar, you cannot specify a **State name**.

For information about specifying an initial condition structure, see “Specify Initial Conditions for Bus Signals”.

All signals in a nonvirtual bus input to a Memory block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “Specify Bus Signal Sample Times” and Bus-Capable Blocks for more information.

You can use an array of buses as an input signal to a Memory block. You can specify the **Initial condition** parameter with:

- The value 0. In this case, all of the individual signals in the array of buses use the initial value 0.
- An array of structures that specifies an initial condition for each of the individual signals in the array of buses.
- A single scalar structure that specifies an initial condition for each of the elements that the bus type defines. Use this technique to specify the same initial conditions for each of the buses in the array.

For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Depends on the type of solver used. If you select the Inherit sample time check box, the block inherits sample time from the driving block.
Direct Feedthrough	No, except when you select Direct feedthrough of input during linearization
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Unit Delay, Zero-Order Hold

Introduced before R2006a

Merge

Combine multiple signals into single signal



Library

Signal Routing

Description

The Merge block combines its inputs into a single output line whose value at any time is equal to the most recently computed output of its driving blocks. You can specify any number of inputs by setting the block's **Number of inputs** parameter.

Use Merge blocks only to interleave input signals that update at different times into a combined signal in which the interleaved values retain their separate identities and times. To combine signals that update at the same time into an array or matrix signal, use a Concatenate block.

Merge blocks assume that all driving signals share the same signal memory. The shared signal memory should be accessed only in mutually exclusive fashion. Therefore, always use alternately executing subsystems to drive Merge blocks.

All signals that connect to a Merge block, or exist anywhere in a network of Merge blocks, are functionally the same signal, and are therefore subject to the restriction that a given signal can have at most one associated signal object. See `Simulink.Signal` for more information.

Run-time diagnostics do not run if the inputs to a merge block are from a single initiator. For example, a single initiator could be a Stateflow chart executing function-call subsystems that are connected to a merge block.

Guidelines for Using the Merge Block

When you use the Merge block, follow these guidelines:

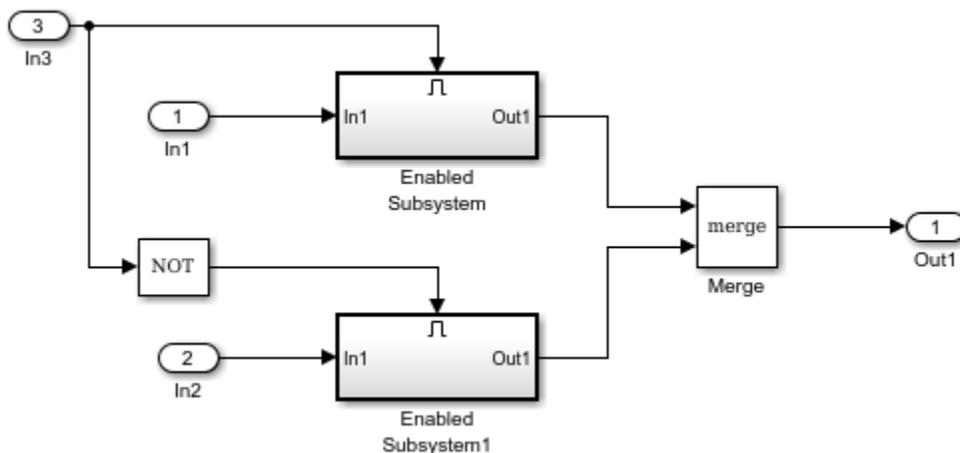
- Always use conditionally-executed subsystems to drive Merge blocks.
- Write your control logic to ensure that at most one of the driving conditionally-executed subsystems executes at any time step.
- Do not connect more than one input of a Merge block to the same conditionally-executed subsystem.
- Always connect a Merge block to at least two input signals.
- Ensure that all input signals have the same sample time.
- Always set the **Initial output** parameter of the Merge block, unless the output port of the Merge block connects to another Merge block.
- Do not branch a signal that inputs to a Merge block, if you use the default setting of `Classic` for the **Model Configuration Parameters > Diagnostics > Underspecified initialization detection** parameter. See the last example in “Merge Block Usage” on page 1-928 for additional usage guidelines relating to branched signals.
- For all conditionally-executed subsystem Output blocks that drive Merge blocks, set the **Output when disabled** parameter to `held`.
- If the output of a Model block is coming from a MATLAB Function block or a Stateflow chart, do not connect that output port to the input port of the Merge block.
- In the code generation workflow, when the Merge block receives a constant value and non-constant sample times, one of these conditions must hold. Otherwise Simulink displays an error.
 - The source of the constant value is a grounded signal.
 - The source of the constant value is a constant block with a non-tunable parameter.
 - There is only one constant block that feeds the Merge block.
 - All other input signals to the Merge block are from conditionally executed subsystems.
 - The Merge block and outport blocks of all conditionally executed subsystems should not specify any initial outputs.

Merge Block Usage

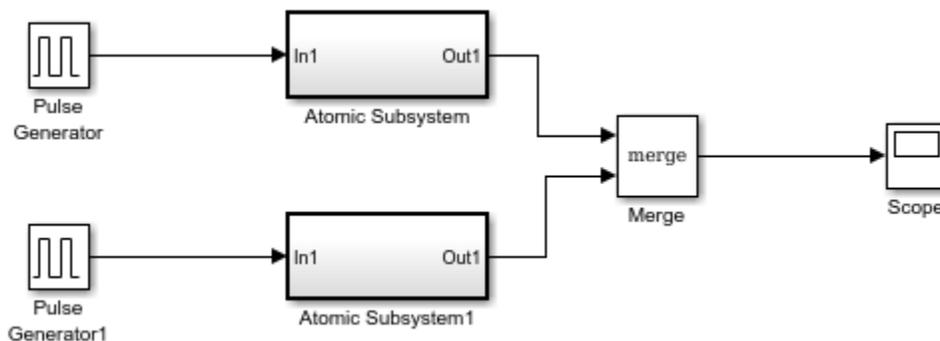
For each input of a Merge block, the topmost non-atomic and nonvirtual source must be a conditionally-executed subsystem that is not an Iterator Subsystem.

You can use the Model Advisor to check Merge block usage in your model. For more information, see “Check usage of Merge blocks” on page 8-32.

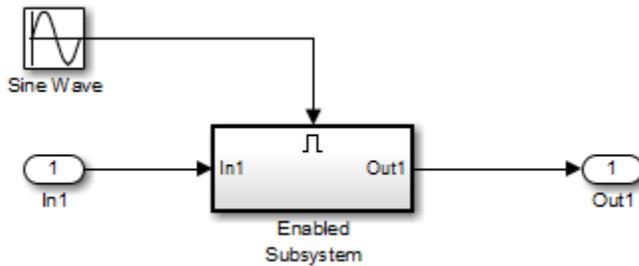
The following schematic shows valid Merge block usage, merging signals from two conditionally-executed subsystems.



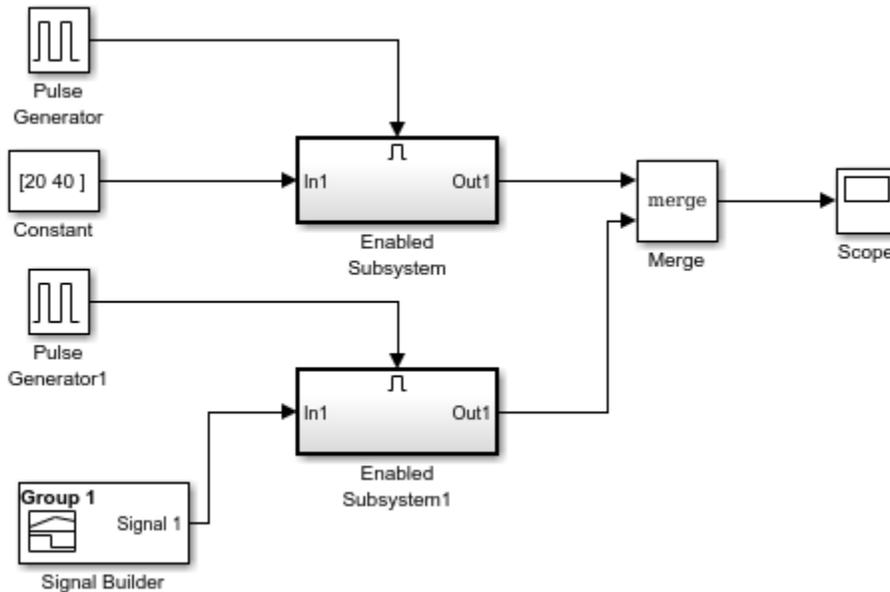
The following example is also a valid Merge block usage, where the topmost nonatomic, nonvirtual source is a conditionally executed subsystem.



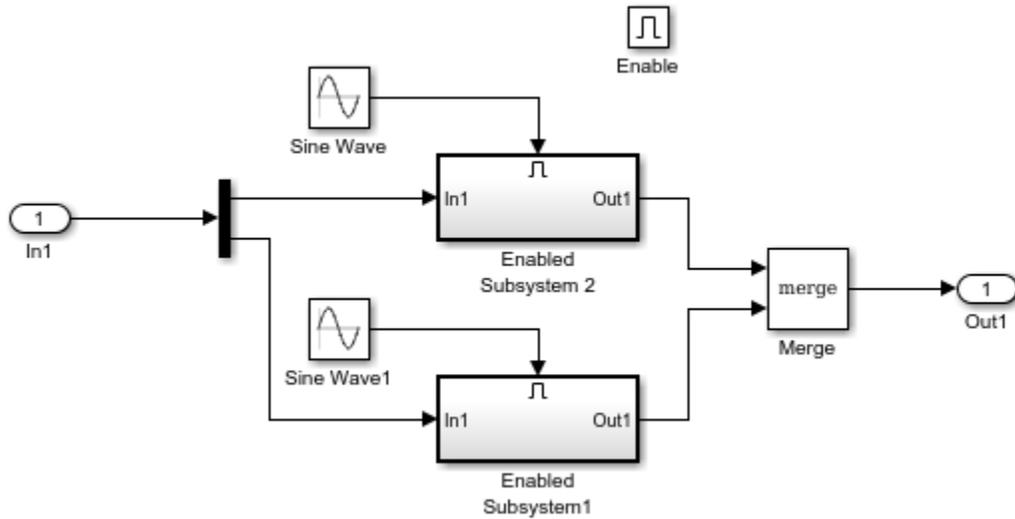
Each Atomic Subsystem block contains an enabled subsystem.



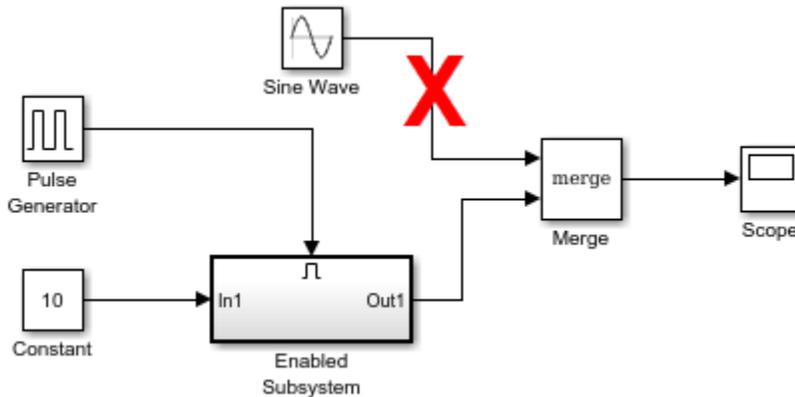
You can also use multiple Merge blocks at different levels of the model hierarchy. The following example contains a Merge block at the model root.



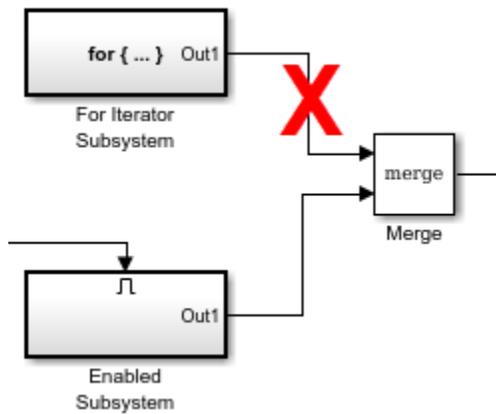
A Merge block is also located inside the Enabled Subsystem block, one level down.



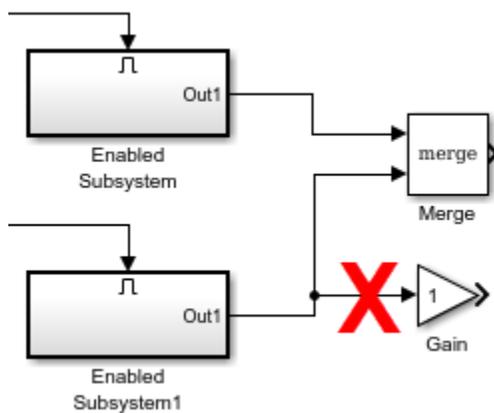
A Merge block *cannot* connect to a Sine Wave block because a Sine Wave block is not a conditionally-executed subsystem.



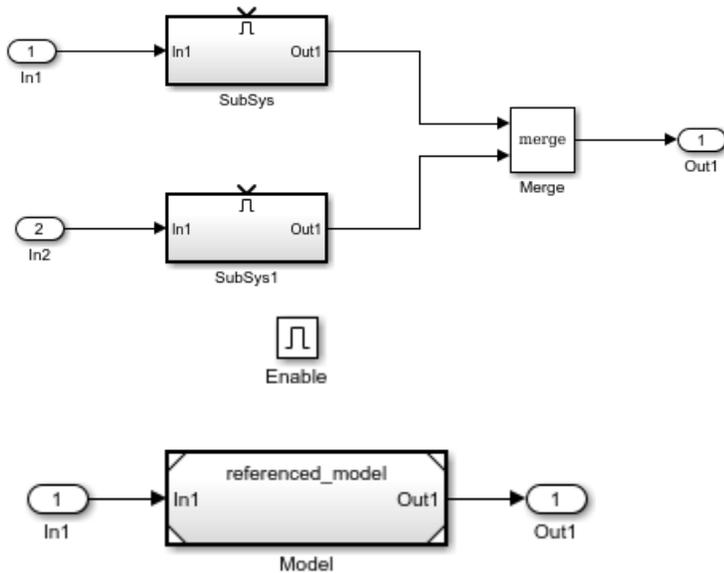
A Merge block *cannot* connect to a For Iterator Subsystem.



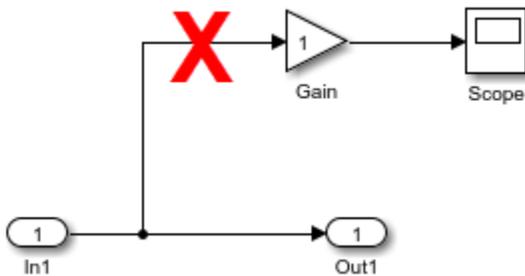
A Merge block *cannot* connect to a branched signal.



In the following model, the referenced model has a signal that branches. The subsystem `Subsys1` includes a Model block that references `referenced_model`. It includes a block that inputs to a block in the referenced model and also inputs to the Merge block that is outside of the referenced model.

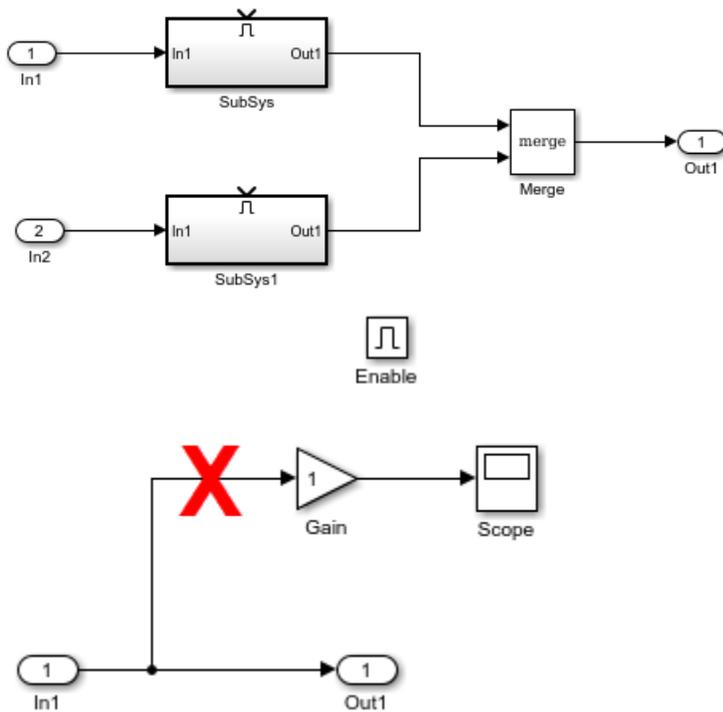


The referenced model includes a signal that incorrectly branches to a Gain block and to the Out1 Outputport block, which connects to the Merge block that is outside of the referenced model.



The following example also shows a branched signal in a subsystem that connects to a Merge block, which is *not allowed* if you use the default setting of Classic for the **Model Configuration Parameters > Diagnostics > Underspecified initialization detection** parameter.

If you set the **Underspecified initialization detection** parameter to Simplified, then the following example does *not* generate an error. For more information on simplified initialization mode, see “Underspecified initialization detection”.



You can specify an initial output value for the Merge block by setting the **Initial output** parameter.

If you do not specify an initial output value, the block's initial output depends on the initialization mode and the driving blocks. In Simplified initialization mode, for an unspecified (empty matrix `[]`) value of **Initial output**, the block uses the default initial value of the output data type. For information on the default initial value, see “Initializing Signal Values”. In Classic initialization mode, for an unspecified (empty matrix `[]`) value of **Initial output**, the initial output of the block equals the most recently evaluated initial output of the driving blocks. Since the initialization ordering for these sources may vary, initialization may be inconsistent for the simulation and the code generation of a model. For example, the following model can produce inconsistent initialization:

- The model contains a Merge block with two inputs: one driven by a Stateflow chart and the other driven by a conditionally executed subsystem (such as an Enabled Subsystem).
- The Merge block **Initial output** parameter is unspecified (that is, specified as empty matrix ([])) and the model uses Classic initialization mode.
- The Stateflow chart initializes the output being merged to `val1`.
- The conditionally executed subsystem initializes the output being merged to different value `val2`.
- Both the Stateflow chart and the conditionally executed subsystem do not execute at the first time step.

Because the initialization ordering may vary, the output of the Merge block at the first time step is `val1` if the Stateflow chart initializes last and `val2` if the conditionally executed subsystem initializes last. The initialization ordering is different for simulation and code generation.

To address this issue, use one of the following approaches:

- Set the **Initial output** parameter of the Merge block, unless the output port of the Merge block connects to another Merge block.
- Turn on simplified initialization mode. Set **Underspecified initialization detection** to `Simplified` on the **Configuration Parameters** dialog box.

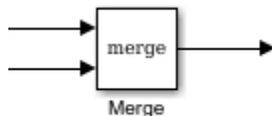
To use the `Simplified` initialization setting, specify the **Initial output** value for all root Merge blocks. A root Merge block is any Merge block with an output port that does not connect to another Merge block.

To upgrade your model to simplified initialization mode, use the Model Advisor check “Check usage of Merge blocks” on page 8-32.

For more information on simplified initialization mode, see “Underspecified initialization detection”.

Single-input merge is not supported. Each Merge block must have at least two inputs.

Use Merge blocks only for signals that require merging. If you were previously connecting a Merge block input to a Mux block, use a multi-input Merge block instead.



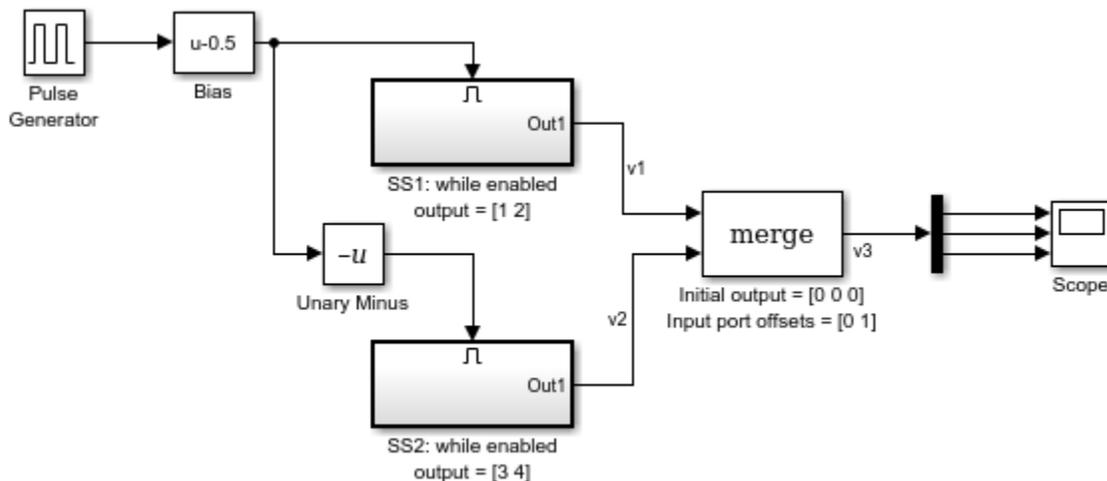
The Merge block accepts only inputs of equal dimensions and outputs a signal of the same dimensions as the inputs, unless you select the **Allow unequal port widths** parameter.

If you select **Allow unequal port widths**, the block accepts scalars and vectors (but not matrices) having differing numbers of elements. Further, the block allows you to specify an offset for each input signal relative to the beginning of the output signal. The width of the output signal is

$$\max(w_1+o_1, w_2+o_2, \dots, w_n+o_n)$$

where w_1, \dots, w_n are the widths of the input signals and o_1, \dots, o_n are the offsets for the input signals.

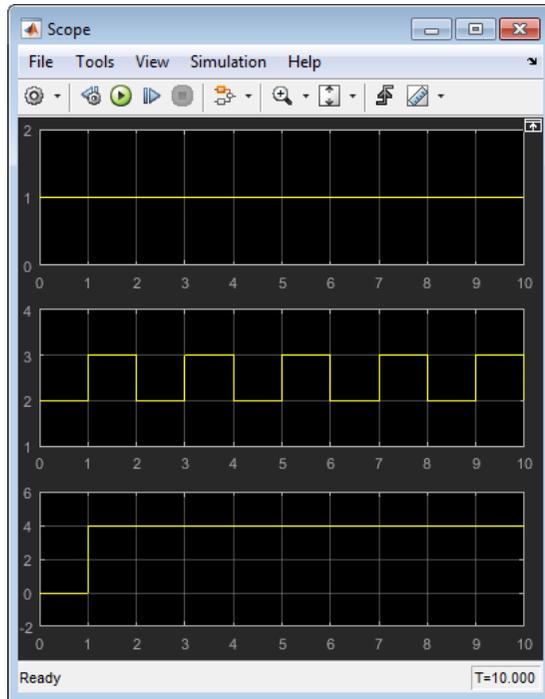
Suppose that you have the following block diagram:



The Merge block has the following output width:

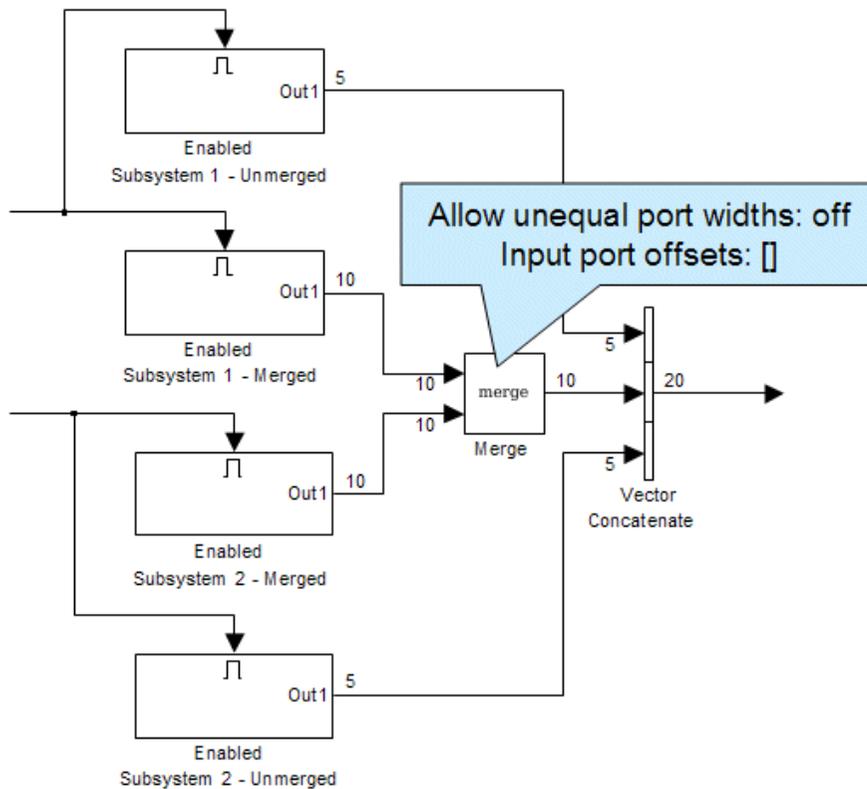
$\max(2+0, 2+1) = 3$

In this example, the offset of v_1 is 0 and the offset of v_2 is 1. The Merge block maps the elements of v_1 to the first two elements of v_3 and the elements of v_2 to the last two elements of v_3 . Only the second element of v_3 is effectively merged, as shown in the scope output:



If you use Simplified Initialization Mode, you must clear the **Allow unequal port widths** check box. The input port offsets for all input signals must be zero.

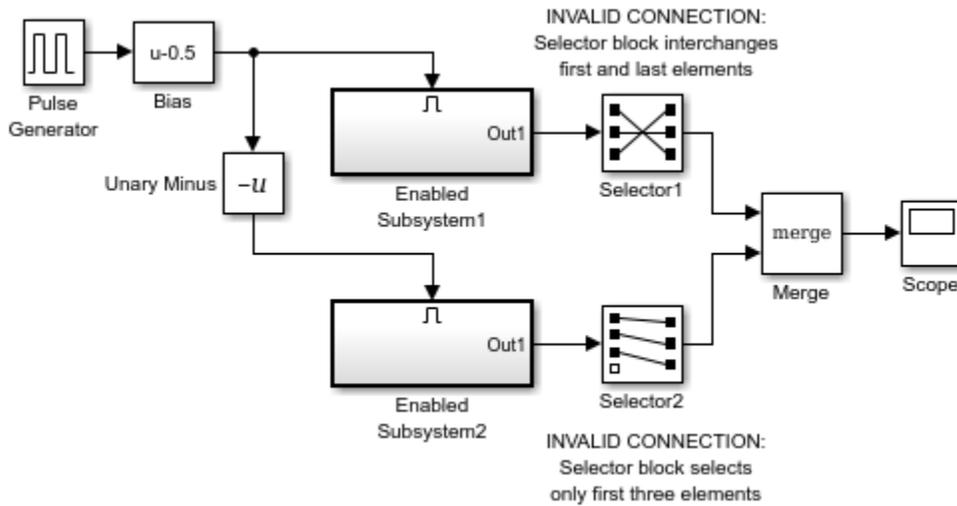
Consider using Merge blocks only for signal elements that require true merging. Other elements can be combined with merged elements using the Concatenate block, as shown in the following example.



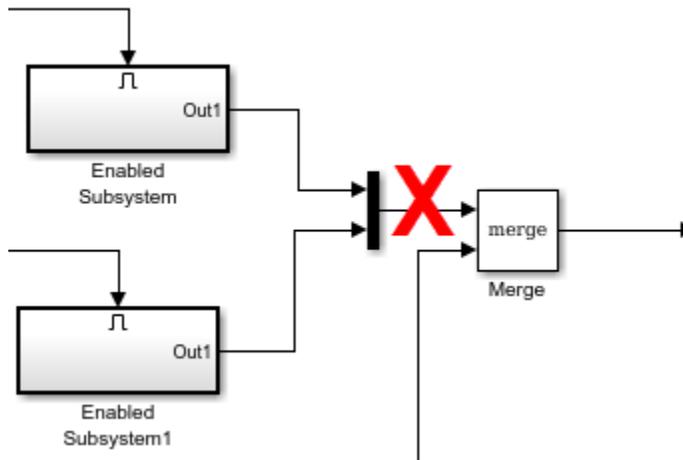
For more information on simplified initialization mode, see “Underspecified initialization detection”.

A Merge block does not accept input signals whose elements have been reordered or partially selected. In addition, you should not connect input signals to the Merge block that have been combined outside of a conditionally-executed subsystem.

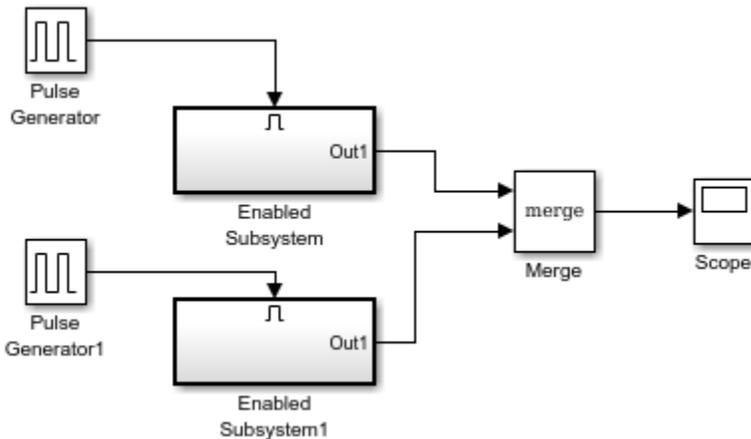
For example, in the following block diagram, the Merge block does not accept the output of the first Selector block because the Selector block interchanges the first and last elements of the vector signal. Similarly, the Merge block does not accept the output of the second Selector block because the Selector block selects only the first three elements.



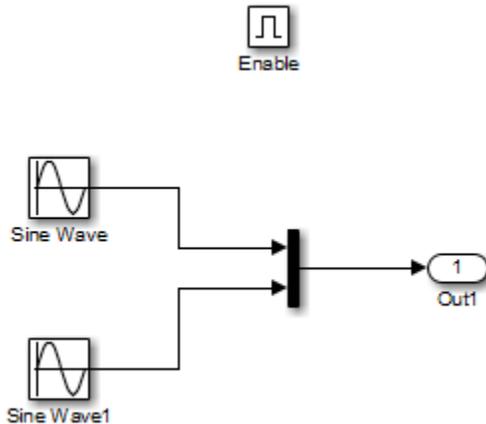
If you use simplified initialization mode, the following arrangement is *not allowed* because two signals are being combined outside of a conditionally-executed subsystem.



You can, however, combine or reorder Merge block input signals within a conditionally-executed subsystem. For example, the following model is valid.



Each Enabled Subsystem contains the following blocks.



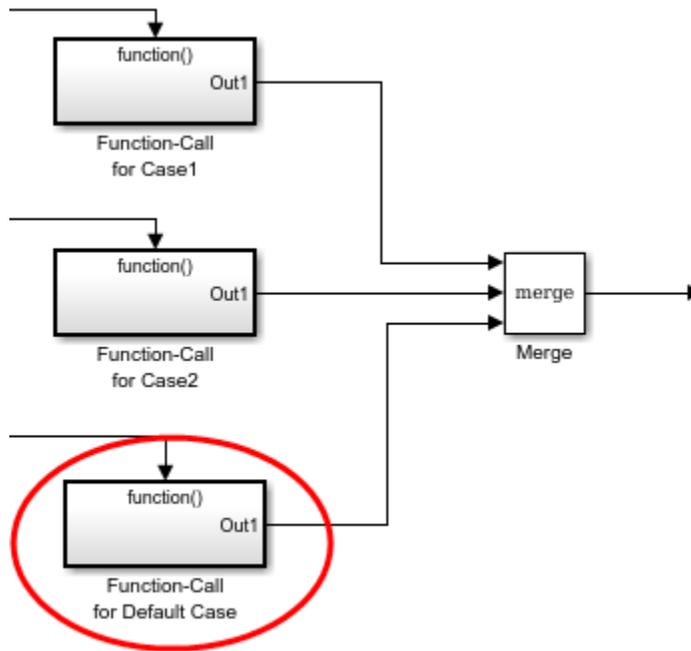
For more information on simplified initialization mode, see “Underspecified initialization detection”.

The Outports of conditionally-executed subsystems being merged should not reset when disabled. This action can cause multiple subsystems to update the Merge block at the same time. Specifically, the disabled subsystem updates the Merge block by resetting its output, while the enabled subsystem updates the Merge block by computing its output.

To prevent this behavior, set the Output block parameter **Output when disabled** to held for each conditionally-executed subsystem being merged.

Note If you are using Simplified Initialization Mode, you *must* set the Output block parameter **Output when disabled** to held.

Instead of resetting the subsystem output when it is disabled, add an additional subsystem for the default case, and use control logic to run this subsystem if nothing else runs. For example, see the following block layout:



For more information on simplified initialization mode, see “Underspecified initialization detection”.

Merging S-Function Outputs

The Merge block can merge a signal from an S-Function block only if the memory used to store the S-Function block's output is reusable. Simulink software displays an error message if you attempt to update or simulate a model that connects a nonreusable port of an S-Function block to a Merge block. See `ssSetOutputPortOptimOpts` for more information.

Data Type Support

The Merge block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types. All inputs must be of the same data type and numeric type.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Number of inputs

Specify the number of input ports to merge.

Initial output

Specify the initial value of output For more information, see “Initial Output Value” on page 1-933.

Allow unequal port widths

Select this check box to allow the block to accept inputs having different numbers of elements.

Input port offsets

Enter a vector to specify the offset of each input signal relative to the beginning of the output signal.

Bus Support

The Merge block is a bus-capable block. The inputs can be virtual or nonvirtual bus signals subject to the following restrictions:

- The number of inputs must be greater than one.
- **Initial output** must be zero, a nonzero scalar, or a finite numeric structure.
- **Allow unequal port widths** must be disabled.
- All inputs to the merge must be buses and must be equivalent (same hierarchy with identical names and attributes for all elements).

For information about specifying an initial condition structure, see “Specify Initial Conditions for Bus Signals”.

All signals in a nonvirtual bus input to a Merge block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “Composite Signals” and Bus-Capable Blocks for more information.

You can use an array of buses as an input signal to a Merge block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”. Using an array of buses with a Merge block involves these limitations:

- **Allow unequal port widths** — Clear this parameter.
- **Number of inputs** — Set to a value of 2 or greater.
- **Initial condition** — You can specify this parameter with:
 - The value 0. In this case, all of the individual signals in the array of buses use the initial value 0.
 - An array of structures that specifies an initial condition for each of the individual signals in the array of buses.
 - A single scalar structure that specifies an initial condition for each of the elements that the bus type defines. Use this technique to specify the same initial conditions for each of the buses in the array.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from driving block
Direct Feedthrough	Yes

Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

MinMax

Output minimum or maximum input value



Library

Math Operations

Description

The MinMax block outputs either the minimum or the maximum element or elements of the inputs. You can choose the function to apply by selecting one of the choices from the **Function** parameter list.

If the block has one input port, the input must be a scalar or a vector. The block outputs a scalar equal to the minimum or maximum element of the input vector.

If the block has multiple input ports, all nonscalar inputs must have the same dimensions. The block expands any scalar inputs to have the same dimensions as the nonscalar inputs. The block outputs a signal having the same dimensions as the input. Each output element equals the minimum or maximum of the corresponding input elements.

The MinMax block ignores any input value that is NaN, except when every input value is NaN. When all input values are NaN, the output is NaN, either as a scalar or the value of each output vector element.

Data Type Support

The MinMax block accepts and outputs real signals of the following data types:

- Boolean (If one input is a boolean, all inputs must be booleans. In this case, all outputs are also booleans.)
- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Function

Specify whether to apply the function `min` or `max` to the input.

Number of input ports

Specify the number of inputs to the block.

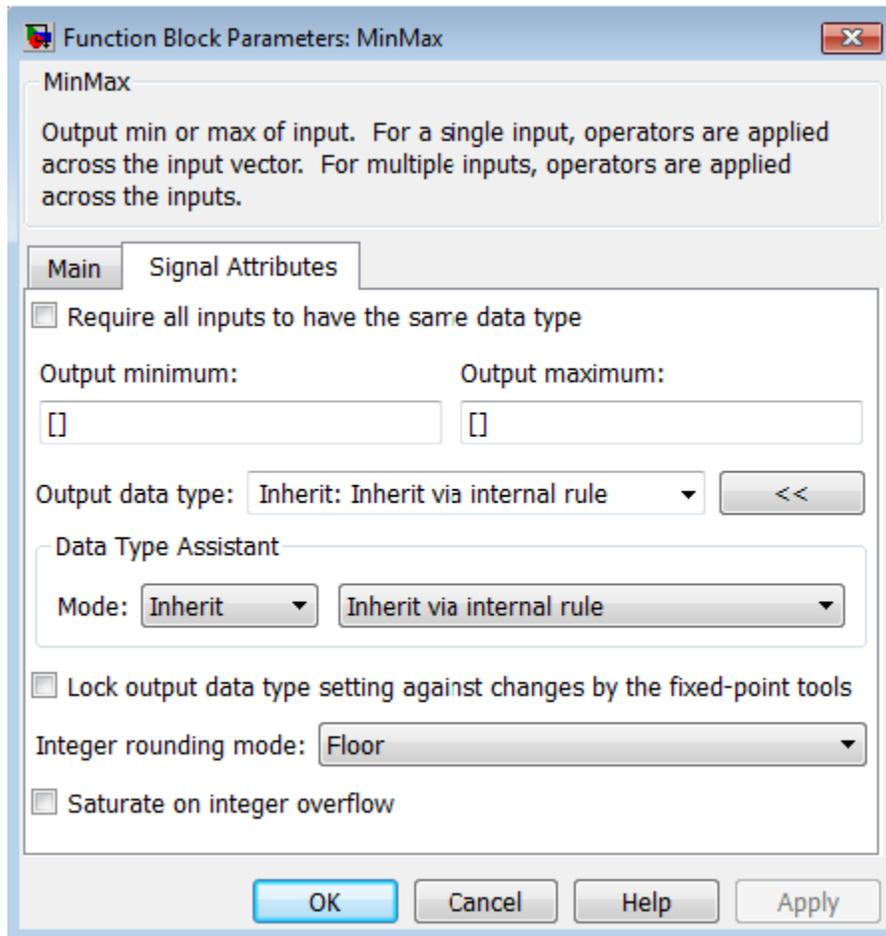
Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

The **Signal Attributes** pane of the MinMax block dialog box appears as follows:



Require all inputs to have the same data type

Select this check box to require that all inputs have the same data type.

Output minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.

- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” in the *Simulink User's Guide* for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer). in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</p>	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code> , which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code> , is -126.

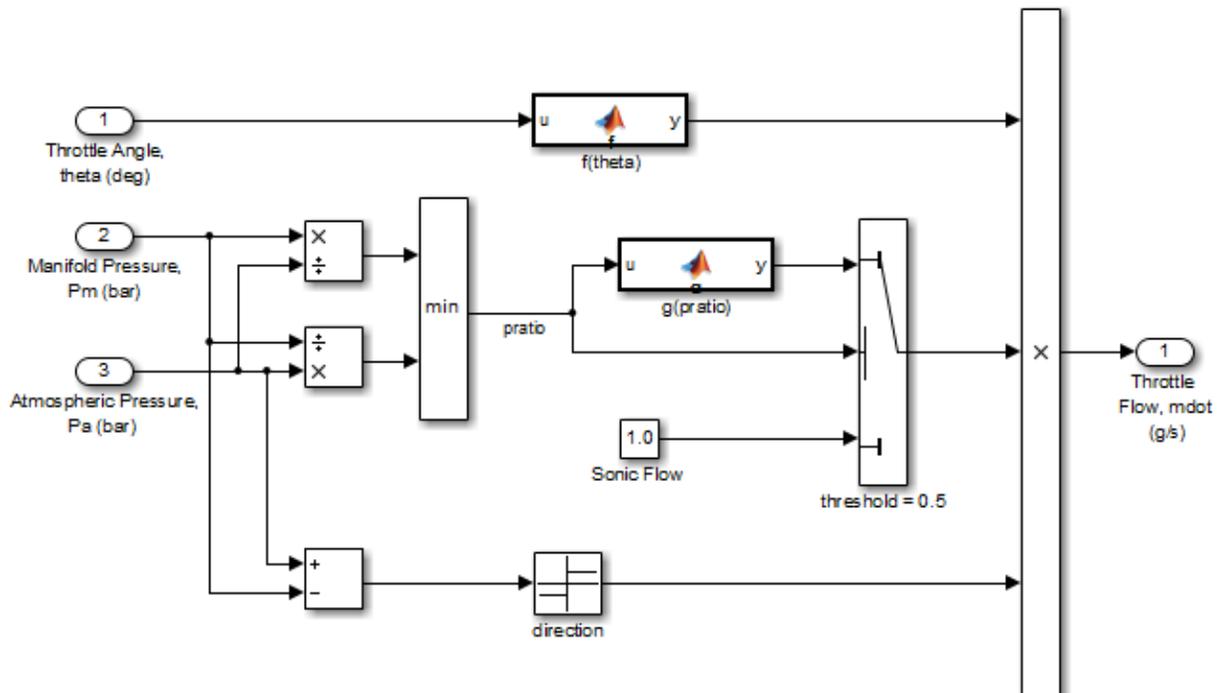
When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect

when overflow is not possible. In this case, the code generator does not produce saturation code.

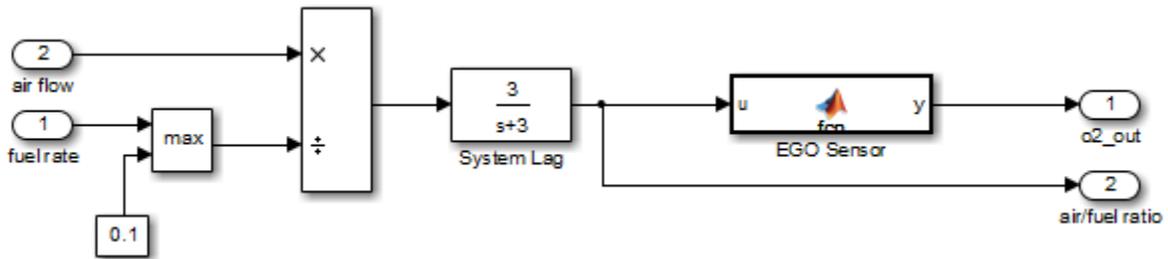
Examples

The `sldemo_fuelsys` model shows how to use the MinMax block.

In the Engine Gas Dynamics/Throttle & Manifold/Throttle subsystem, the MinMax block uses the min operator:



In the Engine Gas Dynamics/Mixing & Combustion subsystem, the MinMax block uses the max operator:



Characteristics

Data Types	Boolean Double Single Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

See Also

MinMax Running Resettable

Introduced before R2006a

MinMax Running Resettable

Determine minimum or maximum of signal over time



Library

Math Operations

Description

The MinMax Running Resettable block outputs the minimum or maximum of all past inputs u . You specify whether the block outputs the minimum or the maximum with the **Function** parameter.

The block can reset its state based on an external reset signal R . When the reset signal R is `TRUE`, the block resets the output to the value of the **Initial condition** parameter.

The input can be a scalar, vector, or matrix signal. If you specify a scalar **Initial condition** parameter, the block expands the parameter to have the same dimensions as a nonscalar input. The block outputs a signal having the same dimensions as the input. Each output element equals the running minimum or maximum of the corresponding input elements.

Data Type Support

The MinMax Running Resettable block accepts and outputs real signals of any numeric data type that Simulink supports, except `boolean`. The MinMax Running Resettable block supports fixed-point data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Function

Specify whether the block outputs the minimum or the maximum.

Initial condition

Specify initial condition.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

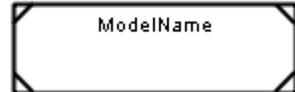
MinMax

Introduced before R2006a

Model

Include multiple model implementations as block in another model

Library: Ports & Subsystems



Description

The Model block allows you to include a model as a block in another model. The included model is called a *referenced model*, and the model containing it (using the Model block) is called the *parent model*.

The Model block displays input and output ports corresponding to the top-level input and output ports of the referenced model. Using these ports allow you to connect the referenced model to other blocks in the parent model. See “Model Referencing” for more information.

As an alternative to specifying a particular referenced model in the **Main** tab, you can dynamically reference one of several model variants, depending on a model workspace value. A *variant* describes one possible mode in which a Model block can operate. Each variant of the Model block references a specific model with optional model-specific arguments. Only one variant is active for simulation. To create model variants, click **Enable Variants** button, which opens a dialog box for implementing model variants. For more information about how to specify a referenced model for multiple specifications, see “Set up Model Variants Using a Model Block”.

Note For new models, use a Model block for model variants only if you need to use variants that are conditionally executed models (models with control ports). Model variants are supported for backward compatibility. However, support for model variants will be removed in a future release. Use of a Variant Subsystem block provides these advantages:

- Allows you to mix Model and Subsystem blocks as variant systems
- Supports flexible I/O, so that all variants do not need to have the same number of input and output ports

To convert a Model block that contains variant models to a Variant Subsystem block that contains Model blocks that reference the variant models, right-click the Model block and select **Subsystems & Model Reference > Convert to > Variant Subsystem**.

Alternatively, you can use the `Simulink.VariantManager.convertToVariant` function. Specify the Model block path or block handle. The converted model produces the same results as the original model.

Alternatively, to set up model variants for a new model, you can use the Variant Model block, which is a subsystem variant template block that contains two Model blocks that you can use for variants.

By default, the contents of a referenced model are user-visible by double-clicking the Model block. However, you can hide the contents of a referenced model by making the model a protected model.

Depending on whether you want to reference a single model or variant models, and whether you want to use model arguments, you need to use different parts of the Model block interface.

Modeling Goal	Model Block Interface to Use
Reference a single model (nonvariant model)	Use the Main tab.
Reference variant models	On the Main tab, click Enable Variants , which opens the model variants dialog box.
Specify values for model arguments	For a nonvariant model reference, use the Arguments tab. For variant models, use the variant dialog box.

To manage various variants that are modeled using variant blocks in a model, in the model variants dialog box click **Open block in Variant Manager** link.

Ports

Input

Input_Port_1 — Input port corresponding to root-level Inport, Enable, and Trigger blocks of referenced model

real or complex values of any data type supported by Simulink

The Model block has an input port for each root-level Inport, Enable, or Trigger block in the referenced model. The name of the Model block port matches the name of the corresponding referenced model input block. The Model block input signals must be valid for the corresponding referenced model input blocks. See “Define Referenced Model Inputs and Outputs”.

Input signals can have real or complex values of any data type supported by Simulink, including bus objects, arrays of buses, fixed-point, and enumerated data types. For details about data types, see Simulink, “Data Types Supported by Simulink”.

Output

Output_Port_1 — Output port corresponding to root-level Outport block of referenced model

real or complex values of any data type supported by Simulink

The Model block has an output port for each root-level Outport block in the referenced model. The name of the Model block port matches the name of the corresponding Outport block. The Model block output signals are the signals from the corresponding referenced model Outport blocks. See “Define Referenced Model Inputs and Outputs”.

Output signals can have real or complex values of any data type supported by Simulink, including bus objects, arrays of buses, fixed-point, and enumerated data types. For details about data types, see Simulink, “Data Types Supported by Simulink”.

Parameters

Main Tab

Model name — File name of referenced model

' ' (default) | character vector

Path to the referenced model. The file name must be a valid MATLAB identifier. The extension, for example, `.slx`, is optional. The file name must contain fewer than 60 characters, exclusive of the `.slx` or `.mdl` suffix.

To navigate to the model that you want to reference, click **Browse**.

To view the model that you specified, click **Open Model**.

Programmatic Use

Parameter: `ModelFile`

Type: character vector

Value: ' ' | '<file name>'

Default: ' '

Simulation mode — Simulation mode for referenced model

Normal (default) | Accelerator | Software-in-the-loop (SIL) | Processor-in-the-loop (PIL)

Specify the simulation mode for the referenced model. The simulation mode for models in a model referencing hierarchy do not have to match. For information about model reference simulation modes and precedence of the simulation mode in a hierarchy, see “Simulate Model Reference Hierarchies”.

- `Accelerator` — Create a MEX-file for the referenced model and then executes the referenced model by running the S-function.
- `Normal` — Execute the referenced model interpretively, as if the referenced model is an atomic subsystem implemented directly within the parent model.
- `Software-in-the-loop (SIL)` — This option requires the Embedded Coder software. Generate production code based on the **Code Interface** parameter setting. The code is compiled for, and executed on, the host platform.
- `Processor-in-the-loop (PIL)` — This option requires the Embedded Coder software. Generate production code based on the **Code Interface** parameter setting.

This code is compiled for, and executed on, the target platform. A documented target connectivity API supports exchange of data between the host and target at each time step during the PIL simulation.

The corners of the Model block reflect the simulation mode for the referenced model. For normal mode, the corners have empty triangles. For accelerator mode, the corner triangles are filled in. For SIL and PIL modes, the corners are filled in and the word (SIL) or (PIL) appears on the block icon.

Programmatic Use

Parameter: `SimulationMode`

Type: character vector

Value: 'Normal' | 'Accelerator' | 'Software-in-the-loop' | 'Processor-in-the-loop'

Default: 'Normal'

Code interface — Generate code from top model or referenced model

`Model reference (default) | Top model`

Specify whether to generate the code from the top model or the referenced model for SIL and PIL simulation modes. To deploy the generated code as part of a larger application that uses the referenced model, specify `Model reference`. To deploy the generated code as a standalone application, specify `Top model`.

`Model reference`

Code generated from referenced model as part of a model reference hierarchy. Code generation uses the `slbuild('model', 'ModelReferenceRTWTarget')` command.

`Top model`

Code generated from top model with the standalone code interface. Code generation uses the `slbuild('model')` command.

Dependencies

To display and enable this parameter, select either `Software-in-the-loop (SIL)` or `Processor-in-the-loop (SIL)` from the **Simulation mode** drop-down list.

Programmatic Use

Parameter: `CodeInterface`

Type: character vector

Value: 'Model reference' | 'Top model'

Default: 'Model reference'

Show model initialize port — Control display of initialize event port

off (default) | on

Control display of initialize event port on the Model block.

off

Remove port.

on

Display model initialize event port.

Programmatic Use

Block parameter: ShowModelInitializePort

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Show model reset port — Control display of reset event ports

off (default) | on

Control display of reset event ports on the Model block.

off

Remove port.

on

Display model reset event ports.

Programmatic Use

Block parameter: ShowModelResetPorts

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Show model terminate port — Control display of terminate event port

off (default) | on

Control display of terminate event port on Model block.

off

Remove port.

on

Display model block port.

Programmatic Use

Block parameter: ShowModelTerminatePort

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Show model periodic event ports — Control display of periodic event ports

off (default) | on

Control display of periodic event ports on Model block.

off

Hide ports.

on

Display ports for rate-based models. A rate-based model is a model with the **Sample time** for a connected Inport block specified.

If you want to manually specify the port rates, set the parameter `AutoFillPortDiscreteRates` to 'off', and then add the port rates to the parameter `PortDiscreteRates`.

Programmatic Use

Block parameter: ShowModelPeriodicEventPorts

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Enable variants — Enable variants and open dialog box for defining model variants

off (default) | on

Opens a dialog box for defining model variants. The dialog box includes a **Variant choices** table, with a separate row for defining each variant.

Programmatic Use

Block parameter: Variants

Type: array

Value: array of variant structures where each element specifies one variant. The structure fields are:

- `variant.Name` (character vector) — The variant control can be a boolean condition expression, or a `Simulink.Variant` object representing a boolean condition expression. If you want to generate code for your model, you must define the control variables as `Simulink.Parameter` objects.
- `variant.ModelName` (character vector) — The name of the referenced model associated with the specified variant control in the Model block.
- `variant.ParameterArgumentNames` (character vector) — Read-only character vector containing the names of the model arguments for which the Model block must supply values.
- `variant.ParameterArgumentValues` (character vector) — The values to supply for the model arguments when this variant is the active variant.
- `variant.SimulationMode` (character vector) — The execution mode to use when this variant is the active variant. Possible values are 'Accelerator' | 'Normal' | 'Software-in-the-loop (SIL)' | 'Processor-in-the-loop (PIL)'

Arguments Tab

Model arguments — Display model arguments and specify argument values for referenced model

number | workspace variable | mathematical expression | structure or structure field

Display model arguments and specify model argument values. Model arguments enable the referenced model to use a different value for a variable used by a referenced model. To specify model argument values, use the **Value** column in the table. For more information about configuring model arguments in a referenced model and specifying argument values, see “Parameterize Instances of a Reusable Referenced Model”.

Programmatic Use

Block parameter: `ParameterArgumentValues` (corresponding to the **Value** column in the table)

Type: structure

Value: Structure with one field for each model argument that the referenced model defines. Fields of the structure are names of the model arguments. Values are character vector representations of the values.

Default: Structure with no fields

Model Variants Dialog Box

Variant choices — Table of variant models

row for the model specified in **Model name** parameter in the **Main** tab (default)

Table containing a row for each variant object in the base workspace. The **Variant choices** table includes the **Model name**, associated **Variant control**, and **Condition (read-only)** columns. For details, see the descriptions of each of the parameters.

To modify the table, use buttons to the left of the **Variant choices** table.

Function	Button
Add a new variant: Add a new, empty row below the currently selected row	
Delete selected variant: Delete the currently selected row. Models and objects are not affected.	
Create/Edit selected variant object: Creates a Simulink.Variant object in the base workspace and opens the Simulink.Variant object parameter dialog box for specifying the variant Condition . This button is enabled only for valid Simulink.Variant objects.	
Move variant up: Move up the currently selected row one slot in the table	
Move variant down: Move the currently selected row down one slot in the table	

Dependency

Enable variants on the **Main** tab opens the variant model dialog box that contains this parameter.

Programmatic Use

Block parameter: Variants

Type: array

Value: array of variant structures where each element specifies one variant. The structure fields are:

- `variant.Name` (character vector) — The variant control can be a Boolean condition expression, or a `Simulink.Variant` object representing a Boolean condition expression. To generate code for your model, define the control variables as `Simulink.Parameter` objects.
- `variant.ModelName` (character vector) — The name of the referenced model associated with the specified variant control in the Model block.
- `variant.ParameterArgumentNames` (character vector) — Read-only character vector containing the names of the model arguments for which the Model block must supply values.
- `variant.ParameterArgumentValues` (character vector) — The values to supply for the model arguments when this variant is the active variant.
- `variant.SimulationMode` (character vector) — The execution mode to use when this variant is the active variant.
 - Possible values are 'Accelerator' | 'Normal' | 'Software-in-the-loop (SIL)' | 'Processor-in-the-loop (PIL)'

Model name — Display or enter name of model associated with variant control

' ' (default) | character vector

Path to the variant model, specified as a character vector. The name must be a valid MATLAB identifier. The extension, for example, `.slx`, is optional. The model name must contain fewer than 60 characters, exclusive of the `.slx` or `.mdl` suffix.

To navigate to the model that you want to reference for the selected variant in the table, click **Browse**.

To view the model that you specified is the model you want to reference, click **Open Model**.

Dependency

Enable variants on the **Main** tab opens the variant model dialog box that contains this parameter.

Programmatic Use**Parameter:** ModelName**Type:** character vector**Value:** '' | '<file name>'**Default:** ''**Variant control — Variant activation condition or variant control that contains expression for variant activation**

Variant (default) | Simulink.Variant object representing a Boolean condition expression | Simulink.Parameter object

The variant control can be a boolean condition expression or a Simulink.Variant object representing a boolean condition expression. If you want to generate code for your model, define control variables as Simulink.Parameter objects.

The variant condition must be a Boolean expression that references at least one base workspace variable or parameter. For example, `FUEL == 2 && EMIS == 1`. Do not surround the condition with parentheses or single quotes. The expression can include:

- MATLAB variables defined in the base workspace
- Simulink parameter objects defined in the base workspace
- Scalar variables
- Enumerated values
- Operators `==`, `~=`, `&&`, `||`, `~`
- Parentheses for grouping

During model compilation, Simulink evaluates variant objects before calling the `InitFcn` callback. Therefore, do not modify the condition of the variant object in the `InitFcn` callback.

Dependency

Enable variants on the **Main** tab opens the variant model dialog box that contains this parameter.

Programmatic Use**Structure field:** Represented by the `variant.Name` field in the `Variants` parameter structure**Type:** character vector

Value: Variant control associated with the variant

Default: ''

Condition (read-only) — Display condition for Simulink.Variant object

a Boolean expression that references at least one base workspace variable or parameter

This read-only field displays the condition for the associated model variant in the base workspace. To specify the condition for the selected variant object, click the **Edit selected variant object** button.

The variant condition must be a Boolean expression that references at least one base workspace variable or parameter. For example, `FUEL == 2 && EMIS == 1`. Do not surround the condition with parentheses or single quotes. The expression can include:

- MATLAB variables defined in the base workspace
- Simulink parameter objects defined in the base workspace
- Scalar variables
- Enumerated values
- Operators `==`, `~=`, `&&`, `||`, `~`
- Parentheses for grouping

Dependency

Enable variants on the **Main** tab opens the variant model dialog box that contains this parameter.

Override variant conditions and use the following variant — Override variant conditions and make variant the active variant

off (default) | on

By default, Simulink determines the active variant by the value of the variant conditions. To override the variant conditions and set the active variant to the value of the **Variant**, enable the **Override variant conditions and use the following variant** parameter.

Dependency

Selecting this parameter, enables the **Variant** parameter.

Programmatic Use

Parameter: `OverrideUsingVariant`

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Variant — Specify variant

varObject(<Enter Model Name>) (default) | character vector

Specify overriding variant.

Dependency

To enable this parameter, select the **Override variant conditions and use the following variant** parameter.

Programmatic Use

Parameter: ActiveVariant

Type: character vector

Value: '' if no overriding variant control is specified.

Default: ''

Model name — Display or enter name of model associated with variant control

'' (default) | character vector

Path to the variant model, specified as a character vector. The name must be a valid MATLAB identifier. The extension, for example, .slx, is optional. The model name must contain fewer than 60 characters, exclusive of the .slx or .mdl suffix.

To navigate to the model that you want to reference for the selected variant in the table, click **Browse**.

To view the model that you specified is the model you want to reference, click **Open Model**.

Dependency

Enable variants on the **Main** tab opens the variant model dialog box that contains this parameter.

Model arguments — Display model arguments

comma-separated list of argument names

Display model arguments for the selected referenced model. For more information about configuring model arguments in a referenced model and specifying argument values, see “Parameterize Instances of a Reusable Referenced Model”.

Dependency

Enable variants enables this parameter.

Programmatic Use

Structure field: Represented by the `variant.ParameterArgumentNames` field in the Variants parameter structure `OneArgName`

Type: character vector

Value: Enter model arguments as a comma separated list

Default: ''

Model argument values (for this instance) — Specify model argument values for referenced model variant

number | workspace variable | mathematical expression | structure or structure field

Enter the argument values in this parameter as a list, in the same order as the corresponding argument names in the **Model arguments** parameter. To separate the items in the list, use commas, spaces, or semicolons. For information about valid values, see “Set Block Parameter Values”.

To define model arguments, see “Model Arguments for Model Blocks That Contain Model Variants”.

Dependency

Enable variants on the **Main** tab opens the variant model dialog box that contains this parameter.

Programmatic Use

Structure field: Represented by the `variant.ParameterArgumentValues` field in the Variants parameter structure `OneArgName`

Type: character vector

Value: Any valid value

Default: ''

Simulation mode — Simulation mode for model variant

Accelerator (default) | Normal | Software-in-the-loop (SIL) | Processor-in-the-loop (PIL)

Simulation mode for the highlighted model variant control.

- `Accelerator` — Creates a MEX-file for the referenced model and then executes the referenced model by running the S-function.
- `Normal` — Executes the referenced model interpretively, as if the referenced model is an atomic subsystem implemented directly within the parent model.
- `Software-in-the-loop (SIL)` — This option requires the Embedded Coder software. Generates production code using model reference target for the referenced model. This code is compiled for, and executed on, the host platform.
- `Processor-in-the-loop (PIL)` — This option requires the Embedded Coder software. Generates production code using a model reference target for the referenced model. This code is compiled for, and executed on, the target platform. A documented target connectivity API supports exchange of data between the host and target at each time step during the PIL simulation.

Generate preprocessor conditionals — For ERT target, specify whether variant choices are enclosed within C preprocessor conditional statements

off (default) | on

When generating code for an ERT target, this parameter determines whether variant choices are enclosed within C preprocessor conditional statements (`#if`). When you enable this option, Simulink analyzes all variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness of all variant choices.

Dependencies

- The check box is available for generating only ERT targets.
- **Override variant conditions and use following variant** is cleared.

Programmatic Use

Parameter: `GeneratePreprocessorConditionals`

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Disable variants — Disable model reference variants and display Main tab

off (default) | on

Disable model reference variants and display the **Main** tab. The block retains any information you have entered and approved by clicking **Apply** or **OK**.

Programmatic Use

Parameter: Variant

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Model Examples

Component-Based Modeling with Model Reference Converting Subsystems to Model Reference Introduction to Managing Data with Model Reference Model Reference Variants

See Also

`Simulink.SubSystem.convertToModelReference` |
`Simulink.VariantManager.convertToVariant` | `find_mdrefs` | `view_mdrefs`

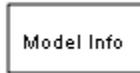
Topics

- “Create a Referenced Model”
- “Simulate Model Reference Hierarchies”
- “Set up Model Variants Using a Model Block”
- “Parameterize Instances of a Reusable Referenced Model”

Introduced before R2006a

Model Info

Display model properties and text in model



Library

Model-Wide Utilities

Description

The Model Info block displays model properties and text about a model on the mask of the block. Use the Model Info block dialog box to specify the content and format of the text that the block displays. You can select model properties to display on the block. In the text displayed on the block mask, Simulink replaces the property name with the current value of the property in the model.

Data Type Support

Not applicable.

Parameters

Specify Text and Properties to Display

Use the **Enter text and tokens to display on Model Info block** edit box to specify the text and properties to display.

- In the edit box, enter any text you want to display on the block mask. Edit the default text `Model Info`.
- To display a model property on the block mask, select a property in the **Model properties** list and click the right arrow button.

The block adds a token of the form `%<modelpropertyname>` to the edit box. In the text the block mask displays, Simulink will replace the token with the value of the property.

- 1 For example, if you select `Description` in the **Model properties** list and click the right arrow button, then the token

`%<Description>`

appears in the right edit box.

- 2 You could add some explanatory text before the model property, e.g. “`Model description:`”.
- 3 When you click **Apply** or **OK**, Simulink displays your new text and the current value of the model property on the block mask in the Model Editor.

See “Version Information Properties” for descriptions of the model properties.

If you are interested in source control information, for a flexible interface to source control tools, use Simulink Project. See “Source Control in Simulink Project”.

Tip To add text, images, links, and equation annotations to a model, see instead “Describe Models Using Annotations”.

Characteristics

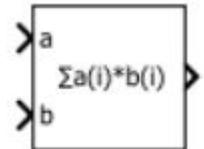
Data Types	Not applicable
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

Introduced before R2006a

Multiply-Accumulate

Perform a multiply-accumulate operation on the inputs

Library: HDL Coder / HDL Operations / Multiply-Accumulate



Multiply-Accumulate

Description

The Multiply-Accumulate block performs this operation on the inputs u_1 and u_2 , and the bias k , to compute the result y .

$$y = \text{sum}(u_1 .* u_2) + k$$

The inputs u_1 and u_2 can be scalars or vectors. By default, the bias k is equal to zero, and the block computes the dot product of the inputs u_1 and u_2 . You can specify a nonzero value for k using `Dialog` or `Input port` as the **Source**. The block adds this bias to the dot product of u_1 and u_2 . The multiplication operation is full precision irrespective of the **Output data type** setting. The **Output data type** and **Integer rounding mode** settings apply to the addition operation.

Ports

Input

a — Input signal

vector | matrix | array | bus

Port to provide input to the block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

b — Input signal

scalar | vector | matrix | array | bus

Port to provide input to the block.

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

c — Bias signal

scalar | vector | matrix | array | bus

Port to provide the bias signal to the block. The block adds this bias to the inputs. Make sure that the bias signal data type matches that of the dot product of the inputs.

Dependencies

To enable this port, set **Source** to external.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

y — Output signal

scalar | vector | matrix | array | bus

Port that generates output of the multiply-accumulate operation.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

Bias — Offset to add to the input dot product

{ '0.0' } (default)

You can specify the bias with:

- **Source** as Dialog. Then, specify the **Value**.
- **Source** as Input port. This setting creates an external input port c to input the bias signal to the block.

Output data type — Data type of the block output

Inherit: `Inherit` via internal rule (default)

Set the output data type to:

- A rule that inherits a data type, such as `Inherit: Same as first input`.
- A built-in data type, such as `single` or `int16`.
- The name of a data type object. for instance, a `Simulink.NumericType` object.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

When you set the **Output data type**, you can use the **Data Type Assistant**. To display

the assistant, click the **Show data type assistant** .

Programmatic Use

Block parameter: `OutDataTypeStr`

Type: character vector

Default: `{'Inherit: Inherit via internal rule'}`

To see possible values that you can specify for this parameter, see “Block-Specific Parameters” on page 6-128.

Integer rounding mode — Rounding mode for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding action as:

`Ceiling`

Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`

Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`

Rounds the number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Programmatic Use

Block parameter: `RndMeth`

Type: character vector

Default: `{'Floor'}`

To see possible values that you can specify for this parameter, see “Block-Specific Parameters” on page 6-128.

Block Characteristics

Data Types	<code>double single base integer fixed point bus</code>
Sample Time	<code>Inherit</code>
Direct Feedthrough	<code>Yes</code>
Multidimensional Signals	<code>Scalar</code>
Variable-Size Signals	<code>Yes</code>
Zero-Crossing Detection	<code>No</code>

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™.

If have HDL Coder installed, you can map the generated HDL code for the block efficiently to DSP slices on the FPGA when you synthesize your design. For more information, see Multiply-Accumulate.

You can use the data types listed above for the ports when you simulate the block. To generate HDL code, make sure that you use `vector` inputs and a `scalar` bias signal. If you use `single` or `double` data types for the inputs, the block may not map the generated code to DSP slices on the FPGA.

Fixed-Point Conversion

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

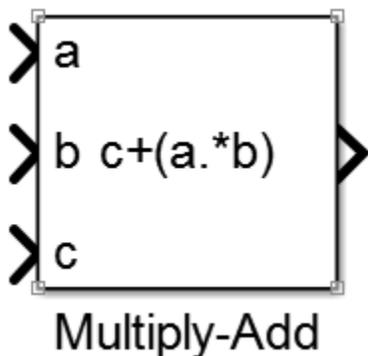
See Also

[Dot Product](#) | [Multiply-Add](#)

Introduced in R2017b

Multiply-Add

Multiply-add combined operation



Library

HDL Coder / HDL Operations

Description

The Multiply-Add block computes the product of the first two inputs, a and b, and adds the result to the third input, c. The inputs can be vectors or scalars.

Operation Precision

The multiplication operation is full precision, regardless of the output type. The **Integer rounding mode**, **Output data type**, and **Saturate on integer overflow** settings apply only to the addition operation.

HDL Code Generation

Use the Multiply-Add block to map a combined multiply-add or a multiply-subtract operation to a DSP unit in your target hardware.

When you generate HDL code for your model, HDL Coder configures the multiply-add operation so that your synthesis tool can map to a DSP unit.

Data Type Support

The Multiply-Add block accepts and outputs signals of any numeric data type that Simulink supports, including fixed-point data types.

For more information, see “Data Types Supported by Simulink”.

Parameters

Function

Specify the function to perform a combined multiply and add or a multiply and subtract operation.

Default: `c+ (a.*b)`

You can set the function to:

- `c+ (a.*b)`
- `c- (a.*b)`
- `(a.*b) -c`

Output data type

Specify the output data type.

Default: `Inherit: Inherit via internal rule`

Set the output data type to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button  to display the Data Type Assistant dialog box, which helps you to set the **Output data type** parameter.

For more information, see “Control Signal Data Types” in *Simulink User's Guide* .

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: Floor

Ceiling

Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

Floor

Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

Nearest

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Rounding” (Fixed-Point Designer).

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off

On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Parameter: SaturateOnIntegerOverflow

Type: character vector

Value: 'off' | 'on'

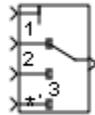
Default: 'off'

See Also

Introduced in R2015b

Multiport Switch

Choose between multiple block inputs



Library

Signal Routing

Description

Note A variant of the Multiport Switch block is the Index Vector block. For information on the Index Vector block, see “Multiport Switch Configured as an Index Vector Block” on page 1-982.

The Multiport Switch block determines which of several inputs to the block passes to the output. The block bases this decision on the value of the first input. The first input is the control input and the remaining inputs are the data inputs. The value of the control input determines which data input passes to the output.

The table summarizes how the block interprets the control input and determines the data input that is passed to the output.

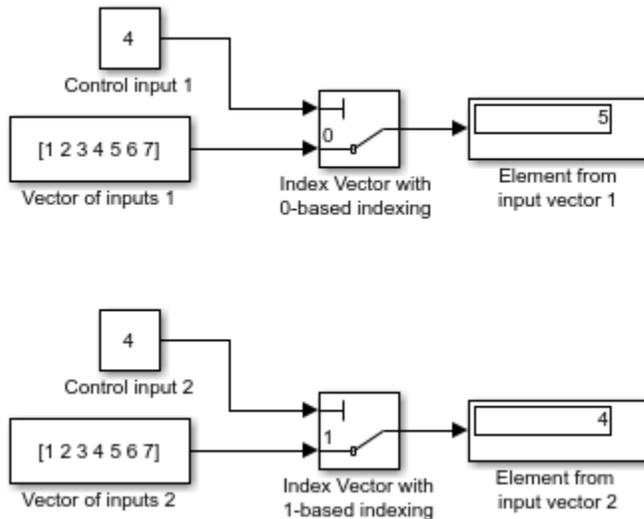
Control Input	Truncation	Setting for Data Port Order	Block Behavior During Simulation	
			Indexing to select data input	Out-of-range condition
Integer value	None	Zero-based contiguous	Zero-based indexing	The control input is less than 0 or greater than the number of data inputs minus one.
		One-based contiguous	One-based indexing	The control input is less than 1 or greater than the number of data inputs.
		Specify indices	Indices you specify	The control input does not correspond to any specified data port index.
Not an integer value	The block truncates the value to an integer by rounding to zero.	Zero-based contiguous	Zero-based indexing	The truncated control input is less than 0 or greater than the number of data inputs minus one.
		One-based contiguous	One-based indexing	The truncated control input is less than 1 or greater than the number of data inputs.
		Specify indices	Indices you specify	The truncated control input does not correspond to any specified data port index.

For information on how the block handles the out-of-range condition, see “How the Block Handles an Out-of-Range Control Input” on page 1-983.

Multiport Switch Configured as an Index Vector Block

An Index Vector is a special configuration of a Multiport Switch block in which you specify one data input and the control input is zero-based. The block output is the element of the input vector whose index matches the control input. For example, if the input vector is [18 15 17 10] and the control input is 3, the element that matches the index of 3 (zero-based) is 10, and that becomes the output value.

This model shows how the Index Vector works with zero-based and one-based indexing.



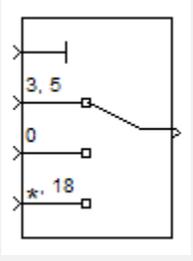
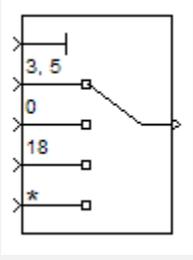
Index Vector is a block in the Simulink **Signal Routing** library. Alternatively, configure a Multiport Switch block to work as an Index Vector block by setting **Number of data ports** to 1 and **Data port order** to Zero-based contiguous.

How the Block Handles an Out-of-Range Control Input

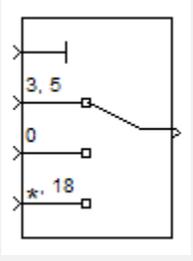
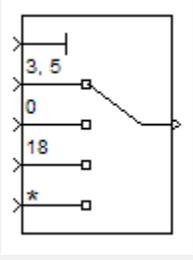
For an input with an integer value less than `intmax('int32')`, the input is out of range when the value does not match any data port indices. For a control input that is not an integer value, the input is out of range when the *truncated* value does not match any data port indices. In both cases, the block behavior depends on your settings for **Data port for default case** and **Diagnostic for default case**.

Note If the control input is larger than `intmax('int32')`, the block wraps the input value to an integer.

The following behavior applies only to simulation for your model.

Data port for default case	Diagnostic for default case		
	None	Warning	Error
<p>Last data port</p> 	<p>Use the last data port and do not report any warning or error.</p>	<p>Use the last data port and report a warning.</p>	<p>Report an error and stop simulation.</p>
<p>Additional data port</p> 	<p>Use the additional data port with a * label and do not report any warning or error.</p>	<p>Use the additional data port with a * label and report a warning.</p>	<p>Report an error and stop simulation.</p>

The following behavior applies to code generation for your model.

Data port for default case	Diagnostic for default case		
	None	Warning	Error
Last data port 	Use the last data port.	Use the last data port.	Use the last data port.
Additional data port 	Use the additional data port with a * label.	Use the additional data port with a * label.	Use the additional data port with a * label.

Use Data Inputs That Have Different Dimensions

If two signals have a different number of dimensions or different dimension lengths, you can use the signals as data inputs to a Multiport Switch block. In the block dialog box, select the parameter **Allow different data input sizes**. In this case, the output of the block is a variable-size signal. If you do not select this parameter, the block generates an error.

For more information about the parameter, see “Allow different data input sizes” on page 1-995. For more information about variable-size signals, see “Variable-Size Signal Basics”.

Rules That Determine the Block Behavior

You specify the number of data inputs with **Number of data ports**.

- If you set **Number of data ports** to 1, the block behaves as an *index selector* or *index vector* and not as a multiport switch. For more details, see “Multiport Switch Configured as an Index Vector Block” on page 1-982.
- If you set **Number of data ports** to an integer greater than 1, the block behaves as a multiport switch. The block output is the data input that corresponds to the value of the control input. If at least one of the data inputs is a vector, the block output is a vector. In this case, the block expands any scalar inputs to vectors.
- If all the data inputs are scalar, the output is a scalar.

Guidelines on Setting Parameters for Enumerated Control Port

When the control port on the Multiport Switch block is of enumerated type, follow these guidelines:

Scenario	What to Do	Rationale
The enumerated type contains a value that represents invalid, out-of-range, or uninitialized values.	<ul style="list-style-type: none"> • Set Data port order to <code>Specify indices</code>. • Set Data port indices to use this value for the last data port. • Set Data port for default case to <code>Last data port</code>. 	This block configuration handles invalid values that the enumerated type explicitly represents.
The enumerated type contains only valid enumerated values. However, a data input port can get invalid values of enumerated type.	<ul style="list-style-type: none"> • Set Data port for default case to <code>Additional data port</code>. 	This block configuration handles invalid values that the enumerated type does not explicitly represent.

Scenario	What to Do	Rationale
The enumerated type contains only valid enumerated values. Data input ports can never get invalid values of enumerated type.	<ul style="list-style-type: none"> Set Data port for default case to <code>Last data port</code>. Set Diagnostic for default case to <code>None</code>. 	This block configuration avoids unnecessary diagnostic action.
The block does not have a data input port for every value of the enumerated type.	<ul style="list-style-type: none"> Set Data port for default case to <code>Additional data port</code>. 	This block configuration handles enumerated values that do not have a data input port, along with invalid values.

Data Type Support

The control signal can be of any data type that Simulink supports, including fixed-point and enumerated types. If the control signal is numeric, it cannot be complex. If the control signal is an enumerated signal, the block uses the value of the underlying integer to select a data port. If the underlying integer does not correspond to a data port, an error occurs.

The data signals can be of any data type that Simulink supports. If any data signal is of an enumerated type, all others must be of the same enumerated type.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

- “Data port order” on page 1-988
- “Number of data ports” on page 1-989
- “Data port indices” on page 1-990
- “Data port for default case” on page 1-991
- “Diagnostic for default case” on page 1-991

- “Sample time” on page 1-992
- “Require all data port inputs to have the same data type” on page 1-992
- “Lock output data type setting against changes by the fixed-point tools” on page 1-992
- “Integer rounding mode” on page 1-993
- “Saturate on integer overflow” on page 1-994
- “Allow different data input sizes” on page 1-995
- “Output minimum” on page 1-996
- “Output maximum” on page 1-996
- “Output data type” on page 1-997
- “Mode” on page 1-998
- “Data type override” on page 1-999
- “Signedness” on page 1-1000
- “Word length” on page 1-1000
- “Scaling” on page 1-1001
- “Fraction length” on page 1-1002
- “Slope” on page 1-1002
- “Bias” on page 1-1002

Data port order

Specify the type of ordering for your data input ports.

Default: One-based contiguous (for Multiport Switch block), Zero-based contiguous (for Index Vector block)

Zero-based contiguous

Block uses zero-based indexing for ordering contiguous data ports.

One-based contiguous

Block uses one-based indexing for ordering contiguous data ports.

Specify indices

Block uses noncontiguous indexing for ordering data ports.

- When the control port is of enumerated type, select `Specify indices`.
- If you select `Zero-based contiguous` or `One-based contiguous`, verify that the control port is not of enumerated type. This configuration is deprecated and produces an error. You can run the Upgrade Advisor on your model to replace each Multiport Switch block of this configuration with a block that explicitly specifies data port indices. See “Model Upgrades”.
- Avoid situations where the block contains unused data ports for simulation or code generation. When the control port is of fixed-point or built-in data type, verify that all data port indices are representable with that type. Otherwise, the following block behavior occurs:

If the block has unused data ports and data port order is...	You get...
<code>Zero-based contiguous</code> or <code>One-based contiguous</code>	A warning
<code>Specify indices</code>	An error

Selecting `Zero-based contiguous` or `One-based contiguous` enables the **Number of data ports** parameter.

Selecting `Specify indices` enables the **Data port indices** parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Number of data ports

Specify the number of data input ports to the block.

Default: 3 (for Multiport Switch block), 1 (for Index Vector block)

The block icon changes to match the number of data input ports you specify.

Selecting `Zero-based contiguous` or `One-based contiguous` for **Data port order** enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Data port indices

Specify an array of indices for your data ports.

Default: {1, 2, 3}

The block icon changes to match the data port indices you specify.

- To specify an array of indices that correspond to all values of an enumerated type, enter `enumeration('type_name')` for this parameter. Do not include braces.

For example, `enumeration('MyColors')` is a valid entry.

- To enter specific values of an enumerated type, use the `type_name.enumerated_name` format. Do not enter the underlying integer value.

For example, `{MyColors.Red, MyColors.Green, MyColors.Blue}` is a valid entry.

- To indicate that more than one value maps to a data port, use brackets.

For example, the following entries are both valid:

- `{MyColors.Red, MyColors.Green, [MyColors.Blue, MyColors.Yellow]}`
- `{[3,5],0,18}`
- If the control port is of fixed-point or built-in data type, the values for **Data port indices** must be representable with that type. Otherwise, an error appears at compile time to alert you to unused data ports.
- If the control port is of enumerated data type, the values for **Data port indices** must be enumerated values of that type.
- If **Data port indices** contains values of enumerated type, the control port must be of that data type.

Selecting `Specify indices for Data port order` enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Data port for default case

Specify whether to use the last data port for out-of-range inputs, or to use an additional port.

Default: Last data port

Last data port

Block uses the last data port for output when the control port value does not match any data port indices.

Additional data port

Block uses an additional data port for output when the control port value does not match any data port indices.

If you set this parameter to Additional data port and **Number of data ports** is 3, the number of input ports on the block is 5. The first input is the control port, the next three inputs are data ports, and the fifth input is the default port for out-of-range inputs.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Diagnostic for default case

Specify the diagnostic action to take when the control port value does not match any data port indices.

Default: Error

None

Simulink software takes no action.

Warning

Simulink software displays a warning and continues the simulation.

Error

Simulink software terminates the simulation and displays an error. In this case, the **Data port for default case** is used only for code generation and not simulation.

For more information, see “How the Block Handles an Out-of-Range Control Input” on page 1-983.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1 . To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Require all data port inputs to have the same data type

Specify allowed data types.

Default: Off

On

Requires all data port inputs to have the same data type.

Off

Allows data port inputs to have different data types.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

 Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: LockScale

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: Floor

Ceiling

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

Floor

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

Nearest

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Parameter: `RndMeth`

Type: character vector

Value: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`

Default: `'Floor'`

For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off

On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Parameter: SaturateOnIntegerOverflow

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Allow different data input sizes

Select this check box to allow input signals with different sizes.

Default: Off

On

Allows input signals with different sizes, and propagate the input signal size to the output signal.

Off

Requires that input signals be the same size.

Parameter: AllowDiffInputSizes

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Output minimum

Specify the minimum value that the block outputs.

Default: []

The default value is [] (unspecified).

Simulink uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

This number must be a finite real double scalar value.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Output maximum

Specify the maximum value the block outputs.

Default: []

The default value is [] (unspecified).

Simulink uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

This number must be a finite real double scalar value.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Output data type

Specify the output data type.

Default: `Inherit: Inherit via internal rule`

`Inherit: Inherit via internal rule`

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit: Inherit via back propagation`

Uses the data type of the driving block.

`double`

Specifies output data type `double`.

`single`

Specifies output data type `single`.

`int8`

Specifies output data type `int8`.

`uint8`

Specifies output data type `uint8`.

`int16`

Specifies output data type `int16`.

`uint16`

Specifies output data type `uint16`.

`int32`

Specifies output data type `int32`.

`uint32`

Specifies output data type `uint32`.

`fixdt(1,16,0)`

Specifies output data type fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`

Specifies output data type fixed point `fixdt(1,16,2^0,0)`.

`<data type expression>`

Uses a data type object, for example, `Simulink.NumericType`.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Specifies inheritance rules for data types. Selecting `Inherit` enables a list of possible values:

- `Inherit` via internal rule (default)
- `Inherit` via back propagation

`Built in`

Specifies built-in data types. Selecting `Built in` enables a list of possible values:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

Fixed point

Specifies fixed-point data types.

Expression

Specifies expressions that evaluate to data types. Selecting `Expression` enables you to enter an expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specifies the fixed-point data as signed.

`Unsigned`

Specifies the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: `16`

Minimum: `0`

Maximum: `32`

Large word sizes represent large values with greater precision than small word sizes.

Selecting **Mode** > Fixed point enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting Slope and bias enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope and bias` enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type”.

Bus Support

The Multiport Switch block is a bus-capable block. The data inputs can be virtual or nonvirtual bus signals subject to the following restrictions:

- All the buses must be equivalent (same hierarchy with identical names and attributes for all elements).
- All signals in a nonvirtual bus input to a Multiport Switch block must have the same sample time. This requirement holds even when the elements of the associated bus object specify inherited sample times.

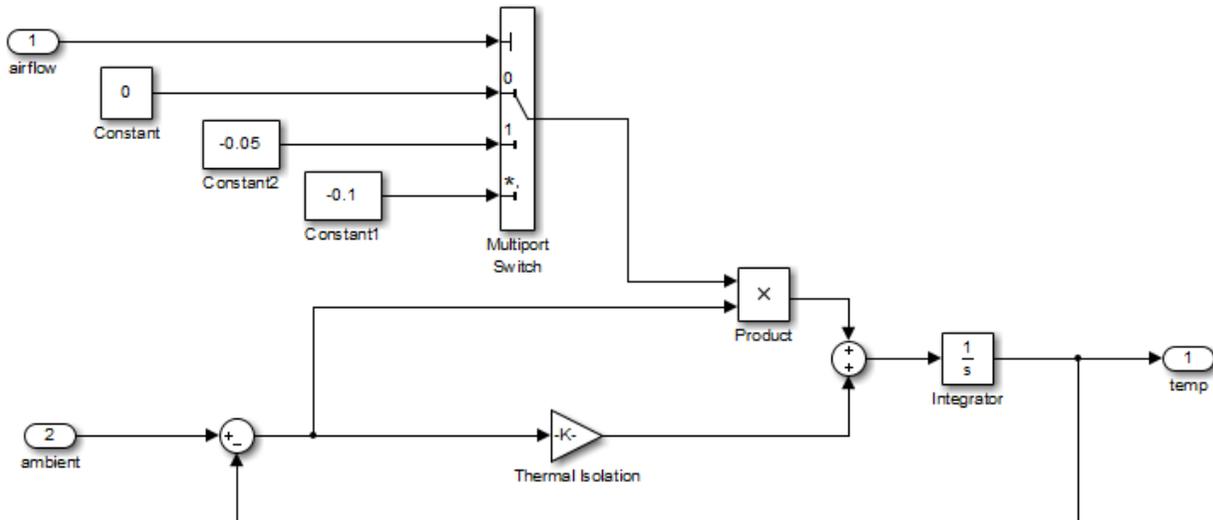
You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “Specify Bus Signal Sample Times” and “Bus-Capable Blocks” for more information.

You can use an array of buses as an input signal to a Multiport Switch block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”. When you use an array of buses with a Multiport Switch block, set **Number of data ports** to a value of 2 or greater.

Examples

Zero-Based Indexing for Data Ports

The `sf_aircontrol` model uses a Multiport Switch block in the `Physical Plant` subsystem. This block uses zero-based indexing for contiguous ordering of three data ports.

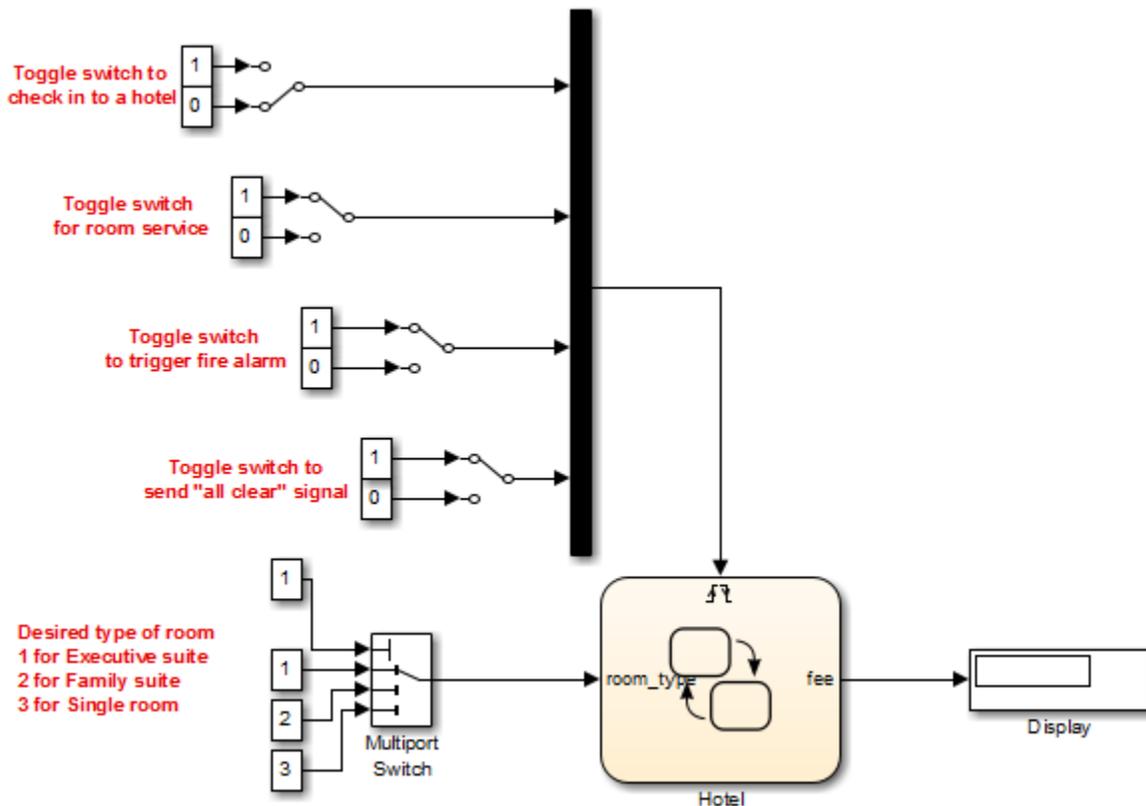


The indices are visible on the data port labels. You do not have to open the block dialog box to determine whether the data ports use zero-based or one-based indexing.

When you set **Data port for default case** to `Last data port`, the last data port includes a * on the label. The comma after the * indicates that the data port index has a value. This port corresponds to the default case, which applies when the control input does not match the data port indices 0, 1, or 2. In this case, the Multiport Switch block outputs a value of `-0.1`.

One-Based Indexing for Data Ports

The `sf_semantics_hotel_checkin` model uses a Multiport Switch block. This block uses one-based indexing for contiguous ordering of three data ports.

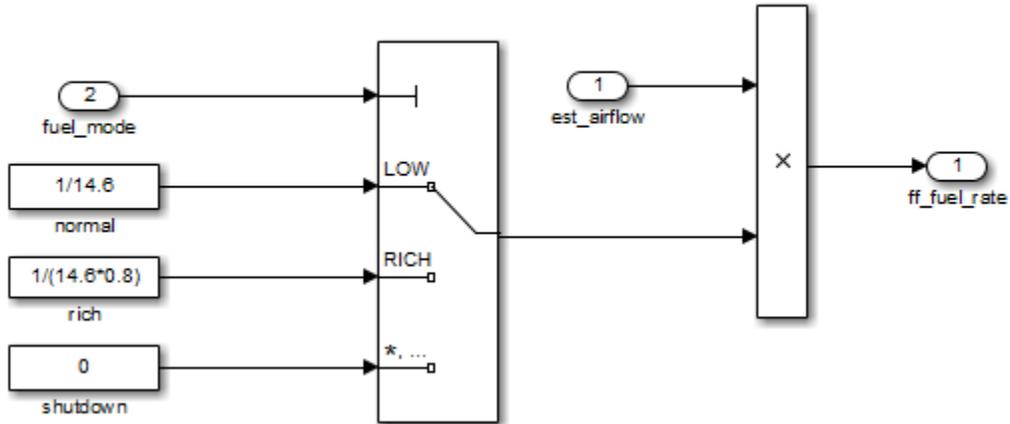


If you increase the size of the block icon, the indices are visible on the data port labels. You do not have to open the block dialog box to determine whether the data ports use zero-based or one-based indexing.

Enumerated Names for Data Port Indices

The `sldemo_fuelsys` model uses a Multiport Switch block in the `fuel_rate_control/fuel_calc/feedforward_fuel_rate` subsystem. This block uses the enumerated type `sld_FuelModes` to specify three data port indices: `LOW`, `RICH`, and `DISABLED`.

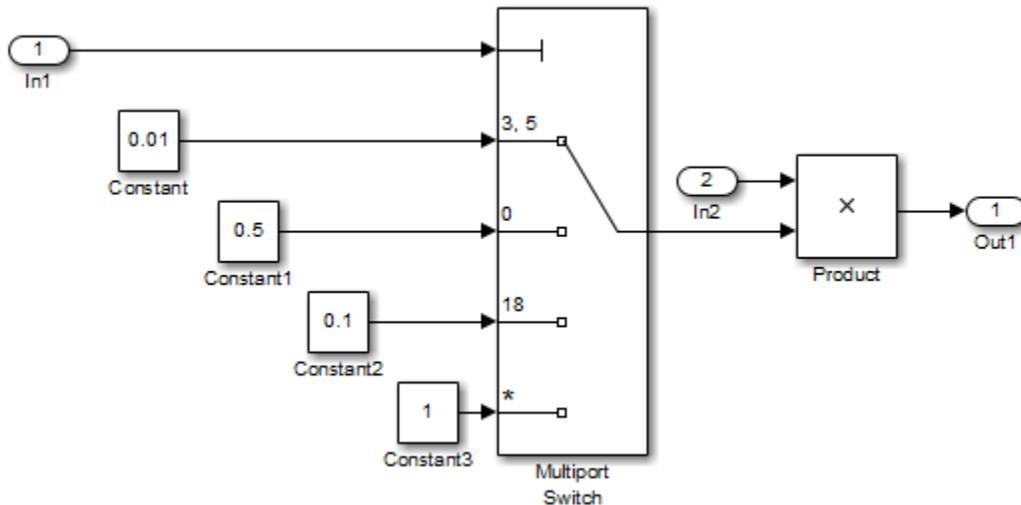
Feedforward Fuel Rate



When you set **Data port for default case** to Last data port, the last data port includes a * on the label. The comma and ellipsis after the * indicate that the data port index has a value. This port corresponds to the default case, which applies when the control input does not match the data port indices LOW, RICH, or DISABLED. In this case, the Multiport Switch block outputs a value of 0.

Noncontiguous Values for Data Port Indices

The following model uses a Multiport Switch block that specifies noncontiguous integer values for data ports.



The values of the indices are visible on the data port labels. You do not have to open the block dialog box to determine which value maps to each data port.

When you set **Data port for default case** to Additional data port, an extra port with a * label appears. This port corresponds to the default case, which applies when the control input does not match the data port indices 3, 5, 0, or 18. In this case, the Multiport Switch block outputs a value of 1.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Switch

Introduced before R2006a

MultiStateImage

Display image reflecting input value

Library: Simulink / Dashboard



Description

The MultiStateImage block displays an image to indicate the value of the input signal. You can use the MultiStateImage block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. You can specify pairs of input values and images to provide the information you want during simulation.

Connection

Dashboard blocks do not use ports to connect to signals. To connect a signal to a Dashboard block, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting signals of interest in your model.
- 3 Mark the option button next to the signal you want to display.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.

- If you turn off logging for a signal connected to a Dashboard block, the model stops sending data from that signal to the block. To view the signal again, reconnect the signal.
- External mode simulation does not support visualization Dashboard blocks.

Parameters

Connection — Select a signal to connect and display

signal connection options

Signal connection options. Populate the **Connection** table by selecting signals of interest in your model. Mark the check box next to the signal you want to display. Click **Apply** to connect the signal.

Scale Mode — Specify how to scale image

'Fill with fixed Aspect Ratio' (default) | Fixed | Fill

Specify how to scale the image.

Fill with fixed Aspect Ratio scales the image to the size of the block while retaining its original aspect ratio.

Fixed displays the image with its fixed true size.

Fill adjusts the image to fill the block.

States

State — Input signal value

1 (default) | scalar

Input signal value that causes the block to display an image. Click the + button to add another state.

Thumbnail — Image to display

image file

Image the block displays when the input signal has the value specified in **State**. The MultiStateImage block can display png, jpg, tif, and bmp image files.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

See Also

Lamp

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2016b

Mux

Combine input signals of same data type and numeric type into virtual vector

Library: Simulink / Commonly Used Blocks
Simulink / Signal Routing



Description

The Mux block combines its inputs into a single vector output. An input can be a scalar or vector signal. All inputs must be of the same data type and numeric type. For information about creating and decomposing vectors, see “Mux Signals”.

The elements of the vector output signal take their order from the top to bottom, or left to right, input port signals. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.

The nonvirtual components of a virtual signal are called *regions*. A virtual signal can contain the same region more than once. For example, if the same nonvirtual signal is connected to two input ports of a Mux block, the block outputs a virtual signal that has two regions. The regions behave as they would if they had originated in two different nonvirtual signals, even though the resulting behavior duplicates information.

Note Simulink provides several techniques for combining signals into a composite signal. For more information, see “Composite Signal Techniques”.

Ports

Input

Port_1 — Accept nonbus vector signal to extract and output signals from real or complex values of any nonbus data type supported by Simulink

Port that accepts the input nonbus vector signal from which to extract and output signals.

Output

Port_1 — Output signals extracted from input vector signal

nonbus signal with real or complex values of any data type supported by Simulink

Output signals extracted from the input vector. The output signal ports are ordered from top to bottom. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.

Parameters

Number of inputs — Inputs

2 (default) | scalar | vector | cell array | comma-separated list of signal names

Specify the number of input signals. You can also specify signal names and sizes. Use one of these formats.

Format	Block Behavior
Scalar	<p>Specifies the number of inputs to the Mux block.</p> <p>When you use this format, the block accepts scalar or vector signals of any size. Simulink assigns each input the name <code>signalN</code>, where N is the input port number.</p>
Vector	<p>The length of the vector specifies the number of inputs. Each element specifies the size of the corresponding input.</p> <p>A positive value specifies that the corresponding port can accept only vectors of that size. For example, <code>[2 3]</code> specifies two input ports of sizes 2 and 3, respectively. If an input signal width does not match the expected width, an error message appears. A value of <code>-1</code> specifies that the corresponding port can accept scalars or vectors of any size.</p>

Format	Block Behavior
Cell array	<p>The length of the cell array specifies the number of inputs. The value of each cell specifies the size of the corresponding input.</p> <p>A scalar value <i>N</i> specifies a vector of size <i>N</i>. A value of <i>-1</i> means that the corresponding port can accept scalar or vector signals of any size.</p>
Signal name list	<p>You can enter a list of signal names separated by commas. Simulink assigns each name to the corresponding port and signal. For example, if you enter <code>position, velocity</code>, the Mux block has two inputs, named <code>position</code> and <code>velocity</code>.</p>

Tip If you specify a scalar for the **Number of inputs** parameter and all of the input ports are connected, as you draw a new signal line close to input side of a Mux block, Simulink adds a port and updates the parameter.

Programmatic Use

Block Parameter:Inputs

Type:scalar, vector, cell array, signal name list

Values:number, vector of port numbers, cell array, or list of signal names

Default:{ '2' }

Display option — Displayed block icon

`bar` (default) | `nonesignal`

By default, the block icon is a solid bar of the block foreground color. To display the icon as a hollow bar containing input signal names, select `signals`. To display the icon as a box containing the block type name, select `none`.

Programmatic Use

Block Parameter:DisplayOption

Type:character vector

Values:'bar' 'signals' 'none' 'bar'

Default:'bar'

Model Examples

Vehicle Suspension Model

Block Characteristics

Data Types	double single Boolean base integer fixed point enumerated
Multidimensional Signals	No
Variable-Size Signals	No

Extended Capabilities

C/C++ Code Generation

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

Actual data type or capability support depends on block implementation.

HDL Code Generation

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

Fixed-Point Conversion

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

Actual data type or capability support depends on block implementation.

See Also

Bus Creator | Bus to Vector | Demux | Vector Concatenate

Topics

“Virtual Signals”

“Composite Signal Techniques”

“Simplify Subsystem Bus Interfaces”

Introduced before R2006a

Out Bus Element

Output signals from a subsystem as a virtual bus

Library: Simulink / Ports & Subsystems
Simulink / Sinks



Description

Note This block has two different names, depending on the library in which it appears. The functionality of both blocks is the same.

- In the Sinks library and the Ports & Subsystems library— Out Bus Element
 - In the Signal Routing library — Bus Element Out
-

To output signals in a virtual bus from a subsystem, use an Out Bus Element block for each signal in the subsystem that you want the bus to contain. This block integrates into one block the functionality of using an Outport block and a Bus Creator block. The Out Bus Element block is of the Outport block type. There are no specifications allowed on an Out Bus Element block, which supports only an inherited workflow. You cannot use the Block Parameters dialog box of an Out Bus Element block to specify bus element attributes, such as data type or dimensions.

To work with buses at subsystem interfaces, consider using In Bus Element and Out Bus Element blocks. This bus element port block combination:

- Reduces signal line complexity and clutter in a block diagram.
- Makes it easy to change the interface incrementally.
- Allows access to a bus element closer to the point of usage.
 - For output, avoid a Goto, From, and Bus Creator block configuration.
 - For input, avoid a duplicate Outport blocks and a Bus Selector, Goto, and From block configuration.

To output multiple signals from a subsystem as a bus signal, create multiple Out Bus Element blocks, one for each signal.

If an Out Bus Element block creates a signal A, then another Out Bus Element block for the same port cannot specify signal A (or a child of signal A) as an element

To add a subbus, in the Block Parameters dialog box, click . To remove blocks associated with selected elements, click .

You can reorder bus elements by dragging and dropping a signal in the list of signals in the Block Parameters dialog box.

You can specify the background color for bus element port blocks, using the Block Parameters dialog box **Set color** option. This action sets the color of blocks associated with selected elements, or to all blocks if you do not select elements.

Ports

The block does not have an output port. Use the Block Parameters dialog box to specify the subsystem output port to which the block sends its input signal.

Input

Port_1 — Input port for bus signal or bus element from within subsystem
signal

The selected input signal is included in a bus signal that the subsystem outputs. The signal can have a real or complex values of any data type that Simulink supports.

Parameters

Port name — Name of subsystem output port
OutBus (default) | text

Specify a name for a subsystem port. That name appears on the Subsystem and Out Bus Element block icons. If you specify a port name, that name cannot already be in use by another block or port. All Out Bus Element blocks that access the same subsystem output port reflect the port name that you specify.

Programmatic Use**Block Parameter:** PortName**Type:** text**Default:** OutBus**Port number — Order in which port appears for subsystem output ports**

1 (default) | integer

Specify the order in which the port appears on the subsystem, with 1 being the top port, 2 the second port down, and so on.

- If you specify a number that exceeds the number of subsystem output ports, new ports are added above the port associated with the Outport Bus Element block.
- If you add an Out Bus Element block that creates another subsystem output port, the port number is the next available number.
- If you delete all Out Bus Element blocks associated with a port, other port numbers are renumbered so that the blocks are in sequence and that no numbers are omitted.

Programmatic Use**Block Parameter:** Port**Value:** integer**Default:** 1

Model Examples

Block Characteristics

Data Types	double single Boolean base integer fixed point enumerated bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes

Extended Capabilities

C/C++ Code Generation

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

Actual data type or capability support depends on block implementation.

Fixed-Point Conversion

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

Actual data type or capability support depends on block implementation.

See Also

Blocks

Bus Creator | In Bus Element | Outport

Topics

“Simplify Subsystem Bus Interfaces”
“Composite Signal Techniques”
“Select a Composite Signal Technique”
“Getting Started with Buses”

Introduced in R2017a

Output

Create output port for subsystem or external output



Library

Ports & Subsystems, Sinks

Description

Output blocks are the links from a system to a destination outside the system.

Simulink software assigns Output block port numbers according to these rules:

- It automatically numbers the Output blocks within a root-level system or subsystem sequentially, starting with 1.
- If you add an Output block, it is assigned the next available number.
- If you delete an Output block, other port numbers are automatically renumbered to ensure that the Output blocks are in sequence and that no numbers are omitted.

Output Blocks in a Subsystem

Output blocks in a subsystem represent outputs from the subsystem. A signal arriving at an Output block in a subsystem flows out of the associated output port on that Subsystem block. The Output block associated with an output port on a Subsystem block is the block whose **Port number** parameter matches the relative position of the output port on the Subsystem block. For example, the Output block whose **Port number** parameter is 1 sends its signal to the block connected to the topmost output port on the Subsystem block.

If you renumber the **Port number** of an Outport block, the block becomes connected to a different output port, although the block continues to send the signal to the same block outside the subsystem.

When you create a subsystem by selecting existing blocks, if more than one Outport block is included in the grouped blocks, Simulink software automatically renumbers the ports on the blocks.

The Outport block name appears in the Subsystem icon as a port label. To suppress display of the label, click the Outport block and select **Format > Hide Name**.

Tip For models that include bus signals composed of many bus elements that feed subsystems, consider using the In Bus Element and Out Bus Element blocks. You can use these bus element port blocks instead of Inport with Bus Selector blocks for inputs, and Outport with Bus Creator blocks for outputs. These bus element port blocks:

- Reduce signal line complexity and clutter in a block diagram.
- Make it easier to change the interface incrementally.
- Allow access to a bus element closer to the point of usage, avoiding the use of a Bus Selector and Goto block configuration.

The Out Bus Element block is of block type Outport. However, there are no specifications allowed on bus element port blocks, which support inherited workflows. You cannot use the Block Parameters dialog box of an Out Bus Element block to specify bus element attributes, such as data type or dimensions.

Initializing Output Blocks in Conditionally Executed Contexts

To set initial conditions for an Outport block in a conditionally executed subsystem, use one of these approaches.

- Inherit initial values from input signals for the subsystem.
- Explicitly specify initial values

For details, see “Conditional Subsystem Initial Output Values”.

Note If the conditional subsystem is driving a Merge block in the same model, you do not need to specify an Initial Condition (IC) for the subsystem’s Outport block. For more information, see “Underspecified initialization detection”.

Top-level Output Block in a Model Hierarchy

Output blocks at the top-level of a model hierarchy have two uses: to supply external outputs to the base MATLAB workspace, which you can do by using either **Configuration Parameters** pane parameters or the `sim` command, and to provide a means for analysis functions to obtain output from the system.

- To supply external outputs to the workspace, use the **Configuration Parameters** > **Data Import/Export** pane (see Exporting Output Data to the MATLAB Workspace) or the `sim` command. For example, if a system has more than one Output block and the save format is array, the following command

```
[t,x,y] = sim(...);
```

writes `y` as a matrix, with each column containing data for a different Output block. The column order matches the order of the port numbers for the Output blocks.

If you specify more than one variable name after the second (state) argument, data from each Output block is written to a different variable. For example, if the system has two Output blocks, to save data from Output block 1 to `speed` and the data from Output block 2 to `dist`, you could specify this command:

```
[t,x,speed,dist] = sim(...);
```

- To provide a means for the `linmod` and `trim` analysis functions to obtain output from the system (see “Linearizing Models”).

Connecting Buses to Root-level Outputs

A root-level Output of a model can accept a virtual bus only if all elements of the bus have the same data type. The Output block automatically unifies the bus to a vector having the same number of elements as the bus, and outputs that vector.

If you want a root-level Output of a model to accept a bus signal that contains mixed types, you must set the Output block **Data type** parameter to use a bus object name for the Bus: `<object name>` or `<data type expression>` option, to define the type of bus that the Output produces. If the bus signal is virtual, it will be converted to nonvirtual, as described in “Bus Conversion”. For more information, see “When to Use Bus Objects”.

Associate Root-Level Output Block with `Simulink.Signal` Object

To associate a root-level Output block with a `Simulink.Signal` object, use the Model Data Editor. See “For Signals”.

Data Type Support

The Output block accepts real or complex signals of any data type that Simulink supports. An Output block can also accept fixed-point and enumerated data types when the block is not a root-level output port. The complexity and data type of the block output are the same as those of its input. The Output block also accepts a bus object as a data type.

Note If you specify a bus object as the data type for this block, do not set the minimum and maximum values for bus data on the block. Simulink ignores these settings. Instead, set the minimum and maximum values for bus elements of the bus object specified as the data type. The values should be finite real double scalar.

For information on the Minimum and Maximum properties of a bus element, see `Simulink.BusElement`.

For more information, see “Data Types Supported by Simulink”.

The elements of a signal array connected to an Output block can be of differing complexity and data types except in the following circumstance: If the output port is in a conditionally executed subsystem and the initial output is specified, all elements of an input array must be of the same complexity and data types.

Typical Simulink data type conversion rules apply to an output port's **Initial output** parameter. If the initial output value is in the range of the block's output data type, Simulink software converts the initial output to the output data type. If the specified initial output is out of the range of the output data type, Simulink software halts the simulation and signals an error.

Parameters

- “Port number” on page 1-1026
- “Specify output when source is unconnected” on page 1-1026
- “Signal name” on page 1-1026
- “Icon display” on page 1-1027
- “Ensure output is virtual” on page 1-1027
- “Source of initial output value” on page 1-1028
- “Output when disabled” on page 1-1029
- “Initial output” on page 1-1029
- “Minimum” on page 1-1030
- “Maximum” on page 1-1031
- “Data type” on page 1-1031
- “Show data type assistant” on page 1-1033
- “Mode” on page 1-1033
- “Data type override” on page 1-1034
- “Signedness” on page 1-1035
- “Word length” on page 1-1035
- “Scaling” on page 1-1036
- “Fraction length” on page 1-1036
- “Slope” on page 1-1037
- “Bias” on page 1-1037
- “Lock output data type setting against changes by the fixed-point tools” on page 1-1038
- “Output as nonvirtual bus in parent model” on page 1-1038
- “Unit (e.g., m, m/s², N*m)” on page 1-1039
- “Port dimensions (-1 for inherited)” on page 1-1040
- “Variable-size signal” on page 1-1040
- “Sample time (-1 for inherited)” on page 1-1041
- “Signal type” on page 1-1041

Port number

Specify the port number of the block.

Default: 1

This parameter controls the order in which the port that corresponds to the block appears on the parent subsystem or model block.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Specify output when source is unconnected

Specify a constant output value to be displayed when source is not connected.

Default: Off

When this option is selected on a non-driven Output block of a Variant Subsystem block, you can specify a constant value as an output of that block.

Specify a constant value as an output for the block.

Select this check box to output a vector of length N if the **Constant value** parameter evaluates to an N-element row or column vector.

Signal name

Specify the name of the corresponding signal data in the generated code. Use this parameter to specify a name for the signal data when you apply a storage class to a root-level Outport block by using **View > Model Data**.

Default: '' (empty character vector)

Specify a name by using text.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Icon display

Specify the information to be displayed on the icon of this input port.

Default: Port number

Signal name

Display the name of the signal connected to this port (or signals if the input is a bus).

Port number

Display port number of this port.

Port number and signal name

Display both the port number and the names of the signals connected to this port.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Ensure output is virtual

This parameter applies to these blocks:

- Conditional subsystem
- Assignment
- Merge
- Model with root Output block

For examples with conditional writes and partial writes, see “Ensure output is virtual”.

Default: Clear

Clear

Use signal buffer on output port. This buffer ensures consistent initialization of the Output block signal.

Select

Simulink tries to remove the signal buffer.

- If the signal buffer is not needed, Simulink remove the buffer.
- If the signal buffer is needed for data consistency and proper execution, Simulink displays an error indicating the buffer could not be removed.

Allow partial writes through Assignment blocks.

Source of initial output value

Select the source of the initial output value of the block.

Default: Dialog

Dialog

The initial output value is specified by the **Initial output** parameter on the dialog.

Input signal

The initial output value is inherited from the input signal. See “Conditional Subsystem Initial Output Values”.

- If you are using classic initialization mode, selecting `Input signal` will cause an error. To inherit the initial output value from the input signal, set this parameter to `Dialog` and specify `[]` (empty matrix) for the **Initial output** value. For more information, see “Conditional Subsystem Initial Output Values”.

This parameter is enabled when the Outport resides in an Conditional Subsystem.

Selecting `Dialog` enables the following parameters:

- **Output when disabled**
- **Initial output**

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Output when disabled

Specify what happens to the block output when the subsystem is disabled.

Default: held

held

Output is held when the subsystem is disabled.

reset

Output is reset to the value given by **Initial output** when the subsystem is disabled.

- When connecting the output of a conditional subsystem to a Merge block, set this parameter to held. Setting it to reset will return an error.
- Selecting `Dialog` in **Source of initial output value** enables this parameter.
- This parameter is enabled when the Output resides in a conditional subsystem with valid enabling and disabling semantics. For example, this parameter is disabled when the Output is placed inside a Triggered Subsystem but is enabled when the Output is placed inside an Enabled Subsystem.
- If an Output is placed inside a function-call subsystem, this parameter is meaningful only if the function-call subsystem is bound to a state in a Stateflow chart. For more information, see “Bind a Function-Call Subsystem to a State” (Stateflow).

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Initial output

For conditionally executed subsystems, specify the block output before the subsystem executes and while it is disabled.

Default: []

Simulink does not allow the initial output of this block to be `inf` or `NaN`. When the input is a virtual bus, an **Initial output** value [] is treated as `double(0)`.

- Specify `[]` (empty matrix) to inherit the initial output value from the input signal. For more information, see “Conditional Subsystem Initial Output Values”.
- For information about specifying an initial condition structure, see “Specify Initial Conditions for Bus Signals”

- Selecting `Dialog` in **Source of initial output value** enables this parameter.
- This parameter is enabled when the Outport resides in an Conditional Subsystem.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Minimum

Specify the minimum value that the block should output.

Default: `[]` (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the `Minimum` property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Maximum

Specify the maximum value that the block should output.

Default: [] (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Data type

Specify the output data type of the external input.

Default: `Inherit: auto`

`Inherit: auto`

A rule that inherits a data type

`double`

Data type is double.

single

Data type is single.

int8

Data type is int8.

uint8

Data type is uint8.

int16

Data type is int16.

uint16

Data type is uint16.

int32

Data type is int32.

uint32

Data type is uint32.

boolean

Data type is boolean.

fixdt(1,16,0)

Data type is fixed point fixdt(1,16,0).

fixdt(1,16,2^0,0)

Data type is fixed point fixdt(1,16,2^0,0).

Enum: <class name>

Data type is enumerated, for example, Enum: BasicColors.

Bus: <object name>

Data type is a bus object.

<data type expression>

The name of a data type object, for example Simulink.NumericType

Do not specify a bus object as the expression.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rule for data types. Selecting `Inherit` enables a second menu/text box to the right.

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`

`Fixed point`

Fixed-point data types.

Enumerated

Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.

Bus object

Bus object. Selecting `Bus` enables a **Bus object** parameter to the right, where you enter the name of a bus object that you want to use to define the structure of the bus. If you need to create or change a bus object, click **Edit** to the right of the **Bus object** field to open the Simulink Bus Editor. For details, see “Create Bus Objects with the Bus Editor”.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Do not specify a bus object as the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data as signed.

`Unsigned`

Specify the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: `16`

Minimum: `0`

Maximum: `32`

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting Slope and bias enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > Binary point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > Slope and bias enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > Slope and bias enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: LockScale

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Output as nonvirtual bus in parent model

Select this parameter if you want the bus emerging in the parent model to be nonvirtual. The bus that is input to the port can be virtual or nonvirtual, regardless of the setting of **Output as nonvirtual bus in parent model**.

Default: Off

On

Select this parameter if you want the bus emerging in the parent model to be nonvirtual.

Off

Clear this parameter if you want the bus emerging in the parent model to be virtual.

- In a nonvirtual bus, all signals must have the same sample time, even if the elements of the associated bus object specify inherited sample times. Any bus operation that would result in a nonvirtual bus that violates this requirement generates an error. For details, see “Connect Multirate Buses to Referenced Models”.
- For a virtual bus, to use a multirate signal, in the root-level Output block, set the **Sample time** parameter to inherited (-1).
- For the top model in a model reference hierarchy, code generation creates a C structure to represent the bus signal output by this block.
- For referenced models, select this option to create a C structure. Otherwise, code generation creates an argument for each leaf element of the bus.

Selecting **Data type** > Bus: <object name> enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Unit (e.g., m, m/s², N*m)

Specify physical unit of the input signal to the block.

Default: inherit

To specify a unit, begin typing in the text box. As you type, the parameter displays potential matching units. For a list of supported units, see Allowed Unit Systems.

To constrain the unit system, click the link to the right of the parameter:

- If a Unit System Configuration block exists in the component, its dialog box opens. Use that dialog box to specify allowed and disallowed unit systems for the component.
- If a Unit System Configuration block does not exist in the component, the model Configuration Parameters dialog box displays. Use that dialog box to specify allowed and disallowed unit systems for the model.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Port dimensions (-1 for inherited)

Specify the dimensions that a signal must have in order to be connected to this Output block.

Default: -1

Valid values are:

-1	A signal of any dimensions can be connected to this port.
N	The signal connected to this port must be a vector of size N.
[R C]	The signal connected to this port must be a matrix having R rows and C columns.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Variable-size signal

Specify the type of signals allowed out of this port.

Default: `Inherit`

`Inherit`

Allow variable-size and fixed-size signals.

`No`

Do not allow variable-size signals.

`Yes`

Allow only variable-size signals.

When the signal at this port is a variable-size signal, the **Port dimensions** parameter specifies the maximum dimensions of the signal.

Parameter: `VarSizeSig`

Type: character vector

Value: `'Inherit' | 'No' | 'Yes'`

Default: 'Inherit'

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

Default: -1

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in “Types of Sample Time”.

See also “Specify Sample Time” in the online documentation for more information.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Signal type

Specify the numeric type of the signal output by this block.

Default: auto

auto

Output the numeric type of the signal that is connected to its input.

real

Output a real-valued signal. The signal connected to this block must be real. If it is not, Simulink software displays an error if you try to update the diagram or simulate the model that contains this block.

complex

Output a complex signal. The signal connected to this block must be complex. If it is not, Simulink software displays an error if you try to update the diagram or simulate the model that contains this block.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from the driving block
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

- Inport
- Out Bus Element
- “Simplify Subsystem Bus Interfaces”

Introduced before R2006a

Permute Dimensions

Rearrange dimensions of multidimensional array dimensions



Library

Math Operations

Description

The block reorders the elements of the input signal so that they are in the order you specify in the **Order** parameter.

Data Type Support

This block accepts signals of any data type that Simulink supports, including fixed-point, enumerated, and nonvirtual bus data types. Output must be the same data type as the input.

You can use an array of buses as an input signal to a Permute Dimensions block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Order

Specify the permutation order to apply to the dimensions of the input signal. This parameter is a vector of elements, where the number of elements in the vector is the number of dimensions of the input signal.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Math Function (`transpose`), `permute` (in the MATLAB reference documentation)

Introduced in R2007a

PID ControllerDiscrete PID Controller

Simulate continuous- or discrete-time PID controllers



Library

Continuous, Discrete

Description

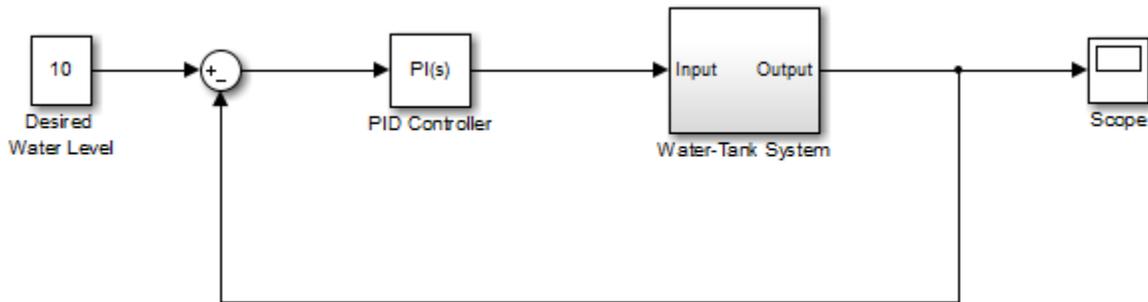
Implement a continuous- or discrete-time controller (PID, PI, PD, P, or I) in your Simulink model. PID controller gains are tunable either manually or automatically. Automatic tuning requires Simulink Control Design™ software.

The PID Controller block output is a weighted sum of the input signal, the integral of the input signal, and the derivative of the input signal. The weights are the proportional, integral, and derivative gain parameters. A first-order pole filters the derivative action.

Configurable options in the PID Controller block include:

- Controller type (PID, PI, PD, P, or I)
- Controller form (Parallel or Ideal)
- Time domain (continuous or discrete)
- Initial conditions and reset trigger
- Output saturation limits and built-in anti-windup mechanism
- Signal tracking for bumpless control transfer and multiloop control

In one common implementation, the PID Controller block operates in the feedforward path of the feedback loop:



The input of the block is typically an error signal, which is the difference between a reference signal and the system output. For a two-input block that permits setpoint weighting, see the PID Controller (2 DOF) block reference page.

You can generate code to implement your controller using any Simulink data type, including fixed-point data types. (Code generation requires Simulink Coder software; fixed-point implementation requires the Fixed-Point Designer product.)

For examples illustrating some applications of the PID Controller block, see the following Simulink examples:

- Anti-Windup Control Using a PID Controller
- Bumpless Control Transfer Between Manual and PID Control

Data Type Support

The PID Controller block accepts real signals of any numeric data type that Simulink software supports, including fixed-point data types. See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Parameters

The following table summarizes the PID Controller block parameters, accessible on the block parameter dialog box.

Task	Parameters
Choose controller form and type.	<ul style="list-style-type: none"> • Controller Form on page 1-1049 in Main tab • Controller on page 1-1052
Choose discrete or continuous time.	<ul style="list-style-type: none"> • Time-domain on page 1-1053 • Sample time on page 1-1055
Choose an integration method (discrete time).	<ul style="list-style-type: none"> • Integrator method on page 1-1053 • Filter method on page 1-1054
Set and tune controller gains.	<ul style="list-style-type: none"> • Controller Parameters Source on page 1-1056 in Main tab • Proportional (P) on page 1-1057 in Main tab • Integral (I) on page 1-1057 in Main tab • Derivative (D) on page 1-1058 in Main tab • Filter coefficient (N) on page 1-1058 in Main tab • Use filtered derivative on page 1-1059 in Main tab
Set integrator and filter initial conditions.	<ul style="list-style-type: none"> • Initial conditions Source on page 1-1060 in Main tab • Integrator Initial condition on page 1-1061 in Main tab • Filter Initial condition on page 1-1061 in Main tab • Initial condition setting on page 1-1062 (command line only) • External reset on page 1-1062 in Main tab • Ignore reset when linearizing on page 1-1063 in Main tab

Task	Parameters
Limit block output.	<ul style="list-style-type: none"> • Limit output on page 1-1064 in PID Advanced tab • Lower saturation limit on page 1-1065 in PID Advanced tab • Upper saturation limit on page 1-1065 in PID Advanced tab • Ignore saturation when linearizing on page 1-1067 in PID Advanced tab
Configure anti-windup mechanism (when you limit block output).	<ul style="list-style-type: none"> • Anti-windup method on page 1-1065 in PID Advanced tab • Back-calculation gain (Kb) on page 1-1067 in PID Advanced tab
Enable signal tracking.	<ul style="list-style-type: none"> • Enable tracking mode on page 1-1068 in PID Advanced tab • Tracking gain (Kt) on page 1-1071 in PID Advanced tab

Task	Parameters
Configure data types.	<ul style="list-style-type: none"> • Parameter data type on page 1-1071 in Data Type Attributes tab • Product output data type on page 1-1073 in Data Type Attributes tab • Summation output data type on page 1-1074 in Data Type Attributes tab • Accumulator data type on page 1-1076 in Data Type Attributes tab • Integrator output data type on page 1-1077 in Data Type Attributes tab • Filter output data type on page 1-1078 in Data Type Attributes tab • Saturation output data type on page 1-1080 in Data Type Attributes tab • Lock output data type setting against changes by the fixed-point tools on page 1-1090 in Data Type Attributes tab • Saturate on integer overflow on page 1-1091 in Data Type Attributes tab • Integer rounding mode on page 1-1092 in Data Type Attributes tab
Configure block for code generation.	<ul style="list-style-type: none"> • State name on page 1-1093 in State Attributes tab • State name must resolve to Simulink signal object on page 1-1093 in State Attributes tab • Code generation storage class on page 1-1094 in State Attributes tab • Code generation storage type qualifier on page 1-1095 in State Attributes tab

Controller form

Select the controller form.

Parallel (Default)

Selects a controller form in which the output is the sum of the proportional, integral, and derivative actions, weighted according to the independent gain parameters **P**, **I**, and **D**. The filter coefficient **N** sets the location of the pole in the derivative filter. For a continuous-time parallel PID controller, the transfer function is:

$$C_{par}(s) = \left[P + I \left(\frac{1}{s} \right) + D \left(\frac{Ns}{s + N} \right) \right]$$

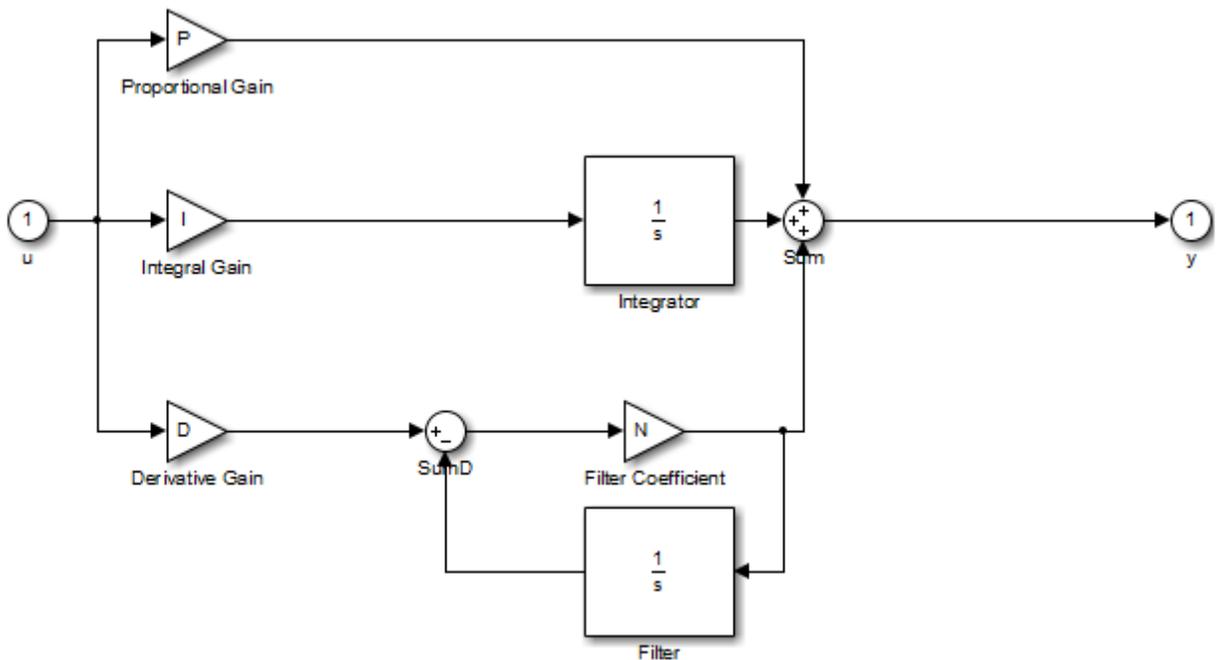
For a discrete-time parallel PID controller, the transfer function takes the form:

$$C_{par}(z) = P + Ia(z) + D \left[\frac{N}{1 + Nb(z)} \right]$$

where the **Integrator method** determines a(z) and the **Filter method** determines b(z) (for sampling time T_s):

	Forward Euler method	Backward Euler method	Trapezoidal method
a(z) (determined by Integrator method)	$\frac{T_s}{z-1}$	$\frac{T_s z}{z-1}$	$\frac{T_s}{2} \frac{z+1}{z-1}$
b(z) (determined by Filter method)	$\frac{T_s}{z-1}$	$\frac{T_s z}{z-1}$	$\frac{T_s}{2} \frac{z+1}{z-1}$

The controller transfer function for the current settings is displayed in the block dialog box.



Parallel PID Controller

Ideal

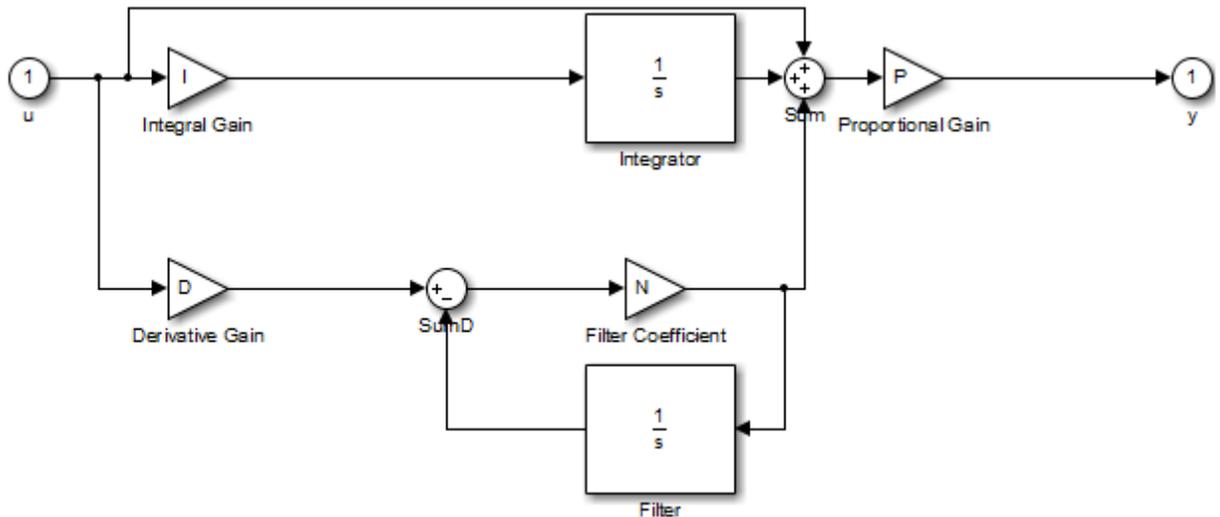
Selects a controller form in which the proportional gain P acts on the sum of all actions. The transfer functions are the same as for the parallel form, except that P multiplies all terms. For a continuous-time ideal PID controller, the transfer function is:

$$C_{id}(s) = P \left[1 + I \left(\frac{1}{s} \right) + D \left(\frac{Ns}{s+N} \right) \right]$$

For a discrete-time ideal PID controller the transfer function is:

$$C_{id}(z) = P \left[1 + Ia(z) + D \frac{N}{1 + Nb(z)} \right]$$

where the **Integrator method** determines $a(z)$ and the **Filter method** determines $b(z)$ as described previously for the parallel controller form.



Ideal PID Controller

Controller

Specify the controller type.

PID (Default)

Implements a controller with proportional, integral, and derivative action.

PI

Implements a controller with proportional and integral action.

PD

Implements a controller with proportional and derivative action.

P

Implements a controller with proportional action.

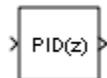
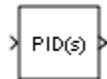
I

Implements a controller with integral action.

The controller transfer function for the current settings is displayed in the block dialog box.

Time-domain

Select continuous or discrete time domain. The appearance of the block changes to reflect your selection.



Continuous-time (Default)

Selects the continuous-time representation.

When the PID Controller block is in a model with synchronous state control (see the State Control block), you cannot select `Continuous-time`.

Discrete-time

Selects the discrete-time representation. Selecting `Discrete-time` also allows you to specify the:

- **Sample time**, which is the discrete interval between samples.
- Discrete integration methods for the integrator and the derivative filter using the **Integrator method** and **Filter method** menus.

Integrator method

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the method used to compute the integrator output. For more information about discrete-time integration methods, see the Discrete-Time Integrator block reference page.

Forward Euler (Default)

Selects the Forward Rectangular (left-hand) approximation.

This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the Forward Euler method can result in instability, even when discretizing a system that is stable in continuous time.

Backward Euler

Selects the Backward Rectangular (right-hand) approximation.

An advantage of the Backward Euler method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result.

Trapezoidal

Selects the Bilinear approximation.

An advantage of the Trapezoidal method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Of all available integration methods, the Trapezoidal method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

Note For integrator methods `BackwardEuler` or `Trapezoidal`, you cannot generate HDL code for the block if either:

- **Limit output** is selected and **Anti-windup method** is set to `back-calculation` or `clamping`.
 - **Enable tracking mode** is selected.
-

Filter method

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the method used to compute the derivative filter output. For more information about discrete-time integration methods, see the [Discrete-Time Integrator block reference page](#).

Forward Euler (Default)

Selects the Forward Rectangular (left-hand) approximation.

This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the Forward Euler method can result in instability, even when discretizing a system that is stable in continuous time.

Backward Euler

Selects the Backward Rectangular (right-hand) approximation.

An advantage of the Backward Euler method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Any filter parameter value $N > 0$ yields a stable result with this method.

Trapezoidal

Selects the Bilinear approximation.

An advantage of the Trapezoidal method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Any filter parameter value $N > 0$ yields a stable result with this method. Of all available filter methods, the Trapezoidal method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

Sample time (-1 for inherited)

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the discrete interval between samples.

Default: 1

By default, the block uses a discrete sample time of 1. To specify a different sample time, enter another discrete value, such as 0.1.

If you specify a value of -1, the PID Controller block inherits the sample time from the upstream block. Do not enter a value of 0; to implement a continuous-time controller, select the **Time-domain** `Continuous-time`.

See “Specify Sample Time” in the online documentation for more information.

Controller Parameters Source

Select the source of the controller gains and filter coefficient. You can provide these parameters explicitly in the block dialog box, or enable external inputs for them on the block. Enabling external inputs for the parameters allows you to compute PID gains and filter coefficients externally to the block and provide them to the block as signal inputs.

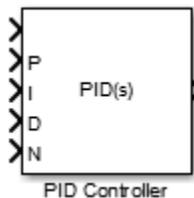
External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control, in which controller gains are determined by logic or other calculation in the Simulink model and passed to the block.

`internal` (Default)

Specify the PID gains and filter coefficient explicitly using the **P**, **I**, **D**, and **N** parameters.

`external`

Specify the PID gains and filter coefficient externally. An additional input port appears under the block input for each parameter that is required for the current controller type:



When you supply gains externally, time variations in the integral and derivative gain values are integrated and differentiated, respectively. This result occurs because of the way the PID gains are implemented within the block. For example, for a continuous-time PID controller with external inputs, the integrator term is implemented as shown in the following illustration.



Within the block, the block's input signal is multiplied by the externally-supplied integrator gain, I , before integration. This implementation yields:

$$y_i = \int uI dt.$$

Thus, the integrator gain is included in the integral. Similarly, in the derivative term of the block, multiplication by the derivative gain precedes the differentiation, which causes the derivative gain D to be differentiated.

Proportional (P)

(Available for PID, PD, PI, and P controllers.) Specify the proportional gain P .

Default: 1

Enter a finite, real gain value into the **Proportional (P)** field. Use either scalar or vector gain values. For a **Parallel PID Controller form**, the proportional action is independent of the integral and derivative actions. For an **Ideal PID Controller form**, the proportional action acts on the integral and derivative actions. See “Controller form” on page 1-1049 for more information about the role of P in the controller transfer function.

When you have Simulink Control Design software installed, you can automatically tune the controller gains. See “Choose a Control Design Approach” (Simulink Control Design).

Integral (I)

(Available for PID, PI, and I controllers.) Specify the integral gain I .

Default: 1

Enter a finite, real gain value into the **Integral (I)** field. Use either scalar or vector gain values.

When you have Simulink Control Design software installed, you can automatically tune the controller gains. See “Choose a Control Design Approach” (Simulink Control Design).

Derivative (D)

(Available for PID and PD controllers.) Specify the derivative gain D .

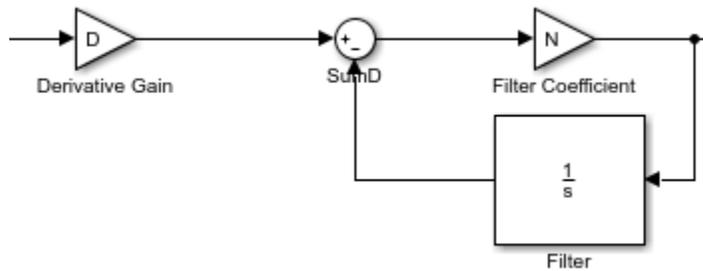
Default: 0

Enter a finite, real gain value into the **Derivative (D)** field. Use either scalar or vector gain values.

When you have Simulink Control Design software installed, you can automatically tune the controller gains. See “Choose a Control Design Approach” (Simulink Control Design).

Filter coefficient (N)

(Available for PID and PD controllers, when **Use filtered derivative** is checked.) Specify the filter coefficient N , which determines the pole location of the filter in the derivative action:



The filter pole falls at $s = -N$ in the Continuous-time **Time-domain**. For Discrete-time, the location of the pole depends on which **Filter method** you select (for sampling time T_s):

- Forward Euler:

$$z_{pole} = 1 - NT_s$$

- Backward Euler:

$$z_{pole} = \frac{1}{1 + NT_s}$$

- Trapezoidal:

$$z_{pole} = \frac{1 - NT_s / 2}{1 + NT_s / 2}$$

Default: 100.

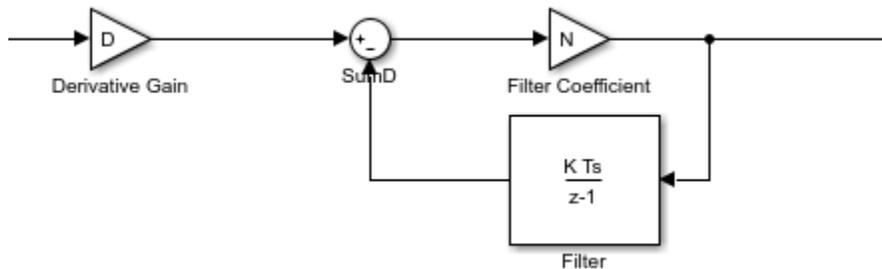
Enter a finite, real gain value into the **Filter Coefficient (N)** field. Use either scalar or vector gain values. Note that the PID controller block does not support $N = \text{inf}$ (ideal unfiltered derivative).

When you have Simulink Control Design software installed, you can automatically tune the controller gains using. See “Choose a Control Design Approach” (Simulink Control Design). Automatic tuning requires $N > 0$.

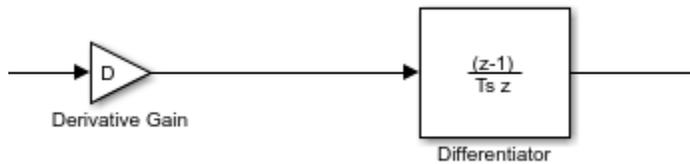
Use Filtered Derivative

Specify whether derivative term is filtered (finite N) or unfiltered. Unfiltered derivative is available only for discrete-time controllers.

Unchecking this option replaces the filtered derivative with a discrete differentiator. For example, if **Filter Method** is Forward Euler, then the filtered derivative term is represented by:



When you uncheck **Use filtered derivative**, the derivative term becomes:



- On (Default)
Use derivative filter (finite N).
- Off
Derivative is unfiltered.

Select Tuning Method

(Requires Simulink Control Design) Select a PID tuning tool. The specified tool opens when you click **Tune**.

Transfer Function Based (PID Tuner App) (Default)

Clicking **Tune** opens the PID Tuner, which lets you interactively tune PID gains while examining relevant system responses to validate performance. By default, PID Tuner works with a linearization of your plant model. For models that cannot be linearized, you can tune PID gains against a plant model estimated from simulated or measured response data. For more information, see “Introduction to Model-Based PID Tuning in Simulink” (Simulink Control Design).

Frequency Response Based

Clicking **Tune** opens the Frequency Response Based PID Tuner, which tunes PID controller gains based on a frequency-response estimation experiment. Frequency Response Based PID Tuner is especially useful for plants that are not linearizable. For more information, see “Design PID Controller from Plant Frequency-Response Data” (Simulink Control Design).

Initial conditions Source

(Only available for controllers with integral or derivative action.) Select the source of the integrator and filter initial conditions. Simulink uses initial conditions to initialize the

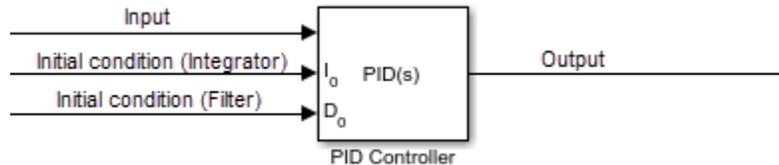
integrator and filter output at the start of a simulation or at a specified trigger event (See “External reset” on page 1-1062). The integrator and filter initial conditions in turn determine the initial block output.

internal (Default)

Specifies the integrator and filter initial conditions explicitly using the **Integrator Initial condition** and **Filter Initial condition** parameters.

external

Specifies the integrator and filter initial conditions externally. An additional input port appears under the block input for each initial condition: I_0 for the integrator and D_0 for the filter:



Integrator Initial condition

(Available only when **Initial conditions Source** is `internal` and the controller includes integral action.) Specify the integrator initial value. Simulink uses the initial condition to initialize the integrator output at the start of a simulation or at a specified trigger event (see “External reset” on page 1-1062). The integrator initial condition, together with the filter initial condition, determines the initial output of the PID controller block.

Default: 0

Simulink does not permit the integrator initial condition to be `inf` or `NaN`.

Filter Initial condition

(Available only when **Initial conditions Source** is `internal`, the controller includes derivative action, and **Use filtered derivative** is checked.) Specify the filter initial value. Simulink uses the initial condition to initialize the filter output at the start of a simulation or at a specified trigger event (see “External reset” on page 1-1062). The filter

initial condition, together with the integrator initial condition, determines the initial output of the PID controller block.

Default: 0

Simulink does not permit the filter initial condition to be `inf` or `NaN`.

Initial condition setting

Specify whether to apply the **Integrator Initial condition** and **Filter Initial condition** parameter to the corresponding block state or output. This parameter can be changed only at the command line using `set_param` to set the `InitialConditionSetting` parameter of the block.

State (most efficient) (Default)

Use this option in all situations except when the block is in a triggered subsystem or a function-call subsystem and simplified initialization mode is enabled.

Output

Use this option when the block is in a triggered subsystem or a function-call subsystem and simplified initialization mode is enabled.

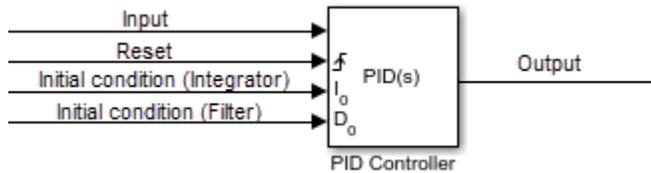
For more information about the **Initial condition setting** parameters, see the Discrete-Time Integrator block.

External reset

Select the trigger event that resets the integrator and filter outputs to the initial conditions you specify in the **Integrator Initial condition** and **Filter Initial condition** fields. Selecting any option other than `none` enables a reset input on the block for the external reset signal, as shown:



Or, if the **Initial conditions Source** is `External`,



The reset signal must be a scalar of type single, double, boolean, or integer. Fixed point data types, except for `ufix1`, are not supported.

Note To be compliant with the Motor Industry Software Reliability Association (MISRA) software standard, your model must use Boolean signals to drive the external reset ports of the PID controller block.

`none` (Default)

Does not reset the integrator and filter outputs to initial conditions.

`rising`

Resets the outputs when the reset signal has a rising edge.

`falling`

Resets the outputs when the reset signal has a falling edge.

`either`

Resets the outputs when the reset signal either rises or falls.

`level`

Resets and holds the outputs to the initial conditions while the reset signal is nonzero.

Ignore reset when linearizing

Force Simulink linearization commands to ignore any reset mechanism that you have chosen with the **External reset** menu. Ignoring reset states allows you to linearize a model around an operating point even if that operating point causes the PID Controller block to reset.

Off (Default)

Simulink linearization commands do not ignore states corresponding to the reset mechanism.

On

Simulink linearization commands ignore states corresponding to the reset mechanism.

Enable zero-crossing detection

Enable zero-crossing detection in continuous-time models upon reset and upon entering or leaving a saturation state.

Zero-crossing detection can accurately locate signal discontinuities without resorting to excessively small time steps that can lead to lengthy simulation times. If you select **Limit output** or activate an **External reset** in your PID Controller block, activating zero-crossing detection can reduce computation time in your simulation. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

On (Default)

Uses zero-crossing detection at any of the following events: reset; entering or leaving an upper saturation state; and entering or leaving a lower saturation state.

Off

Does not use zero-crossing detection.

Enabling zero-crossing detection for the PID Controller block also enables zero-crossing detection for all under-mask blocks that include the zero-crossing detection feature.

Limit output

Limit the block output to values you specify as the **Lower saturation limit** and **Upper saturation limit** parameters.

Activating this option limits the block output internally to the block, obviating the need for a separate Saturation on page 1-1307 block after the controller in your Simulink

model. It also allows you to activate the block's built-in anti-windup mechanism (see “Anti-windup method” on page 1-1065).

Off (Default)

Does not limit the block output, which equals the weighted sum of the proportional, integral, and derivative actions.

On

Limits the block output to the **Lower saturation limit** or the **Upper saturation limit** whenever the weighted sum exceeds those limits. Allows you to select an **Anti-windup method**.

Lower saturation limit

(Available only when you select the **Limit output** check box.) Specify the lower limit for the block output. The block output is held at the **Lower saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions goes below that value.

Default: `-inf`

Upper saturation limit

(Available only when you select the **Limit output** check box.) Specify the upper limit for the block output. The block output is held at the **Upper saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions exceeds that value.

Default: `inf`

Anti-windup method

(Available only when you select the **Limit output** option and the controller includes integral action.) Select an anti-windup mechanism to discharge the integrator when the block is saturated, which occurs when the sum of the block components exceeds the output limits.

When you select the **Limit output** check box and the weighted sum of the controller components exceeds the specified output limits, the block output holds at the specified limit. However, the integrator output can continue to grow (integrator wind-up),

increasing the difference between the block output and the sum of the block components. Without a mechanism to prevent integrator wind-up, two results are possible:

- If the sign of the input signal never changes, the integrator continues to integrate until it overflows. The overflow value is the maximum or minimum value for the data type of the integrator output.
- If the sign of the input signal changes once the weighted sum has grown beyond the output limits, it can take a long time to discharge the integrator and return the weighted sum within the block saturation limit.

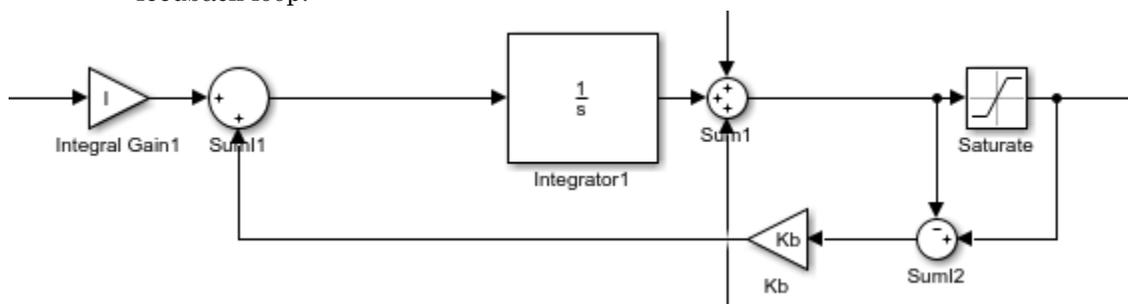
In both cases, controller performance can suffer. To combat the effects of wind-up without an anti-windup mechanism, it may be necessary to detune the controller (for example, by reducing the controller gains), resulting in a sluggish controller. Activating an anti-windup mechanism can improve controller performance.

none (Default)

Does not use an anti-windup mechanism. This setting may cause the block's internal signals to be unbounded even if the output appears to be bounded by the saturation limits. This can result in slow recovery from saturation or unexpected overflows.

back-calculation

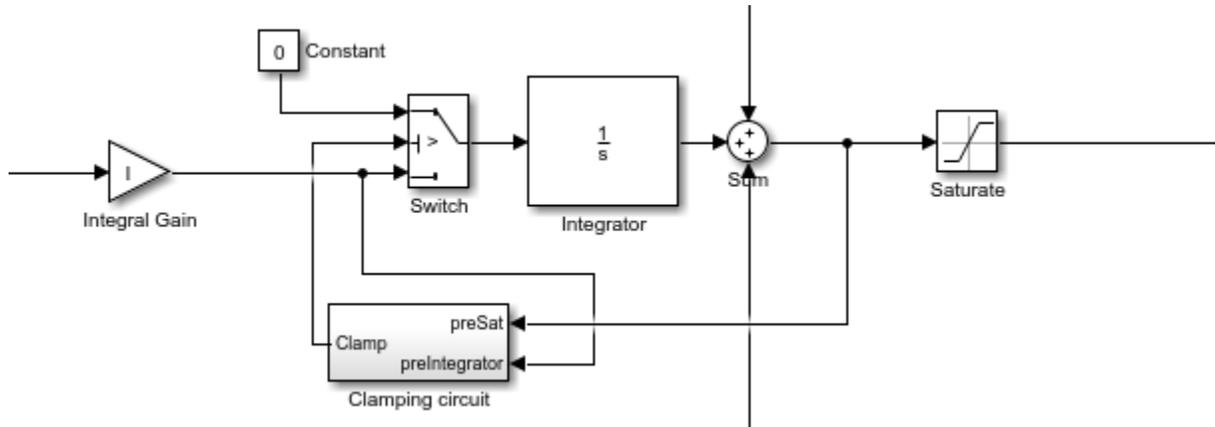
Discharges the integrator when the block output saturates using the integral-gain feedback loop:



You can also specify a value for the **Back-calculation coefficient (Kb)**.

clamping

Stops integration when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Resumes integration when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. The integrator portion of the block is:



The clamping circuit implements the logic necessary to determine whether integration continues.

Back-calculation gain (Kb)

(Available only when the back-calculation **Anti-windup method** is active.) Specify the gain coefficient of the anti-windup feedback loop.

The back-calculation anti-windup method discharges the integrator on block saturation using a feedback loop having gain coefficient Kb.

Default: 1

Ignore saturation when linearizing

Force Simulink linearization commands ignore PID Controller block output limits. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the PID Controller block to exceed the output limits.

On (Default)

Simulink linearization commands ignore states corresponding to saturation.

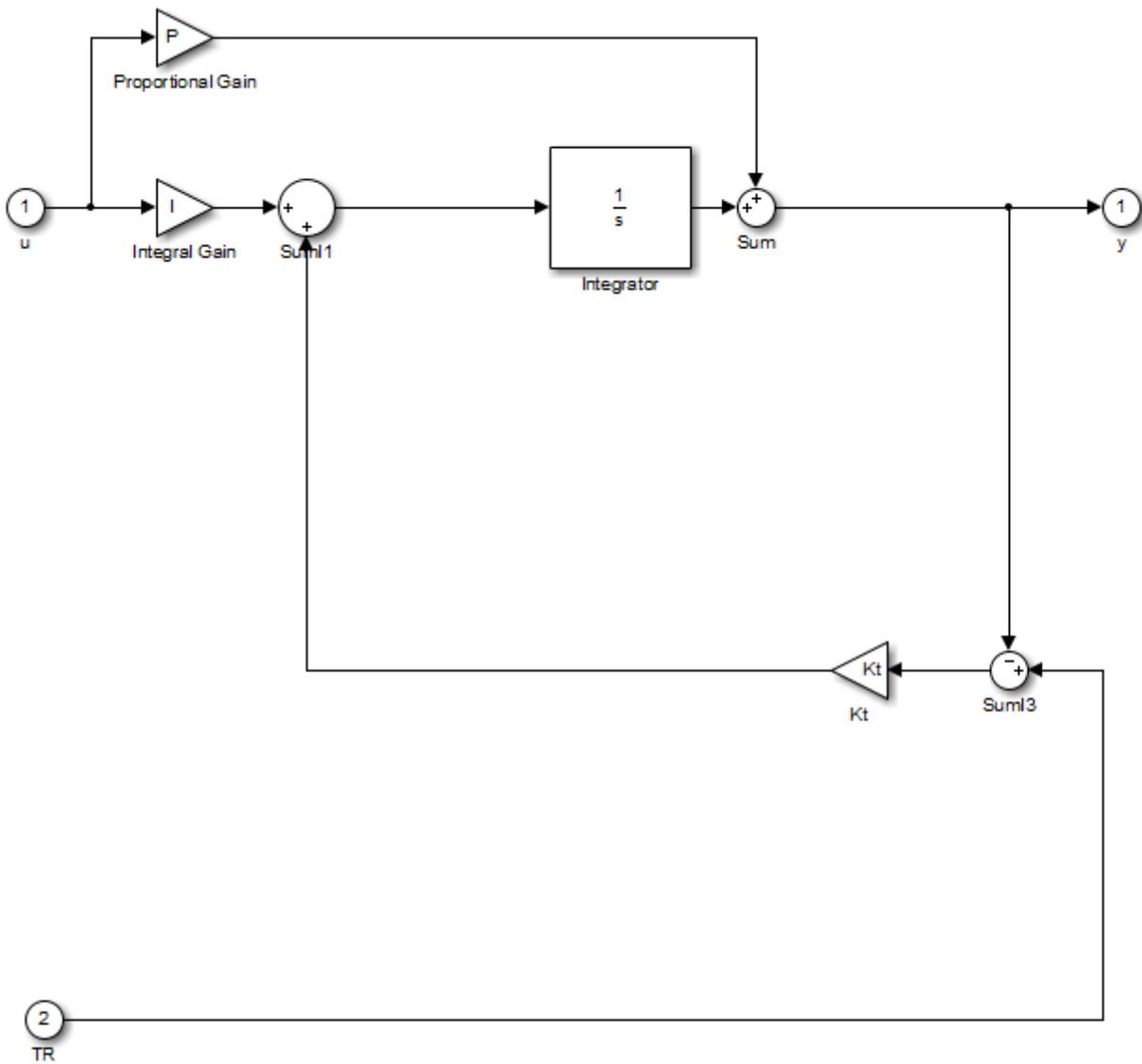
Off

Simulink linearization commands do not ignore states corresponding to saturation.

Enable tracking mode

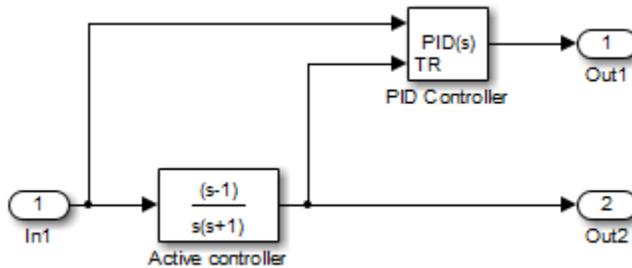
(Available for any controller with integral action.) Activate signal tracking, which lets the output of the PID Controller block follow a tracking signal. Provide the tracking signal to the block at the TR port, which becomes active when you select **Enable tracking mode**.

When signal tracking is active, the difference between the tracked signal and the block output is fed back to the integrator input with a gain K_t . The structure is illustrated for a PI controller:



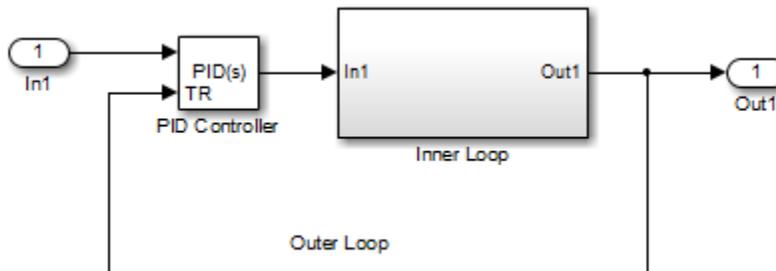
You can also specify the **Tracking coefficient (Kt)**.

Use signal tracking, for example, to achieve bumpless control transfer in systems that switch between two controllers. You can make one controller track the output of the other controller by connecting the TR port to the signal you want to track. For example:

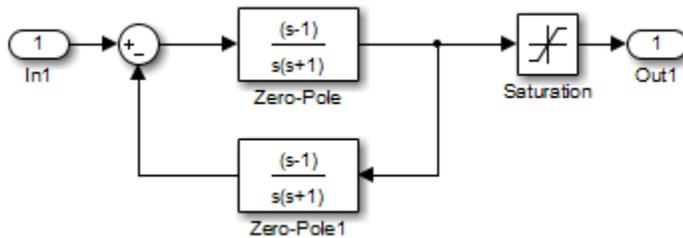


In this example, the outputs Out1 and Out2 can drive a controlled system (not shown) through a switch that transfers control between the “Active controller” block and the PID Controller block. The signal tracking feature of the PID Controller block provides smooth operation upon transfer of control from one controller to another, ensuring that the two controllers have the same output at the time of transfer.

Use signal tracking to prevent block wind-up in multiloop control approaches, as this example illustrates:



The inner-loop subsystem contains the following blocks:



In this example, the inner loop has an effective gain of 1 when it does not saturate. Without signal tracking, the inner loop winds up in saturation. Signal tracking ensures that the PID Controller output does not exceed the saturated output of the inner loop.

Off (Default)

Disables signal tracking and removes TR block input.

On

Enables signal tracking and activates TR input.

Tracking gain (Kt)

(Available only when you select **Enable tracking mode**.) Specify K_t , which is the gain of the signal tracking feedback loop.

Default: 1

Parameter data type

Select the data type of the gain parameters **P**, **I**, **D**, **N**, **Kb**, and **Kt**.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Inherit: Inherit via internal rule (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the

internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit: Same as input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit: Inherit via back propagation`

Use data type of the driving block.

`Inherit: Same as input`

Use data type of input signal.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

<data type expression>

Name of a data type object. For example, `Simulink.NumericType`.

Product output data type

Select the product output data type of the gain parameters **P**, **I**, **D**, **N**, **Kb**, or **Kt**, or the product output data type of the derivative filter numerator and denominator.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Inherit: Inherit via internal rule (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit: Same as input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

Inherit: Inherit via back propagation

Use data type of the driving block.

Inherit: Same as input

Use data type of input signal.

`double`

single
int8
uint8
int16
uint16
int32
uint32
fixdt(1,16)
fixdt(1,16,0)
fixdt(1,16,2^0,0)
<data type expression>

Name of a data type object. For example, `Simulink.NumericType`.

Summation output data type

Select the summation output data type of the sums **Sum**, **Sum D**, **Sum I1**, **SumI2**, **SumI3**, and the summations in the trapezoidal discrete filter numerator and denominator, **SumNum** and **SumDen**. These sums are computed internally within the block, and might not all be present, depending upon the controller configuration. To see where Simulink computes each of these sums, right-click the PID Controller block in your model and select **Look Under Mask**:

- **Sum** is the weighted sum of the proportional, derivative, and integral signals.
- **SumD** is the sum in the derivative filter feedback loop.
- **SumI1** is the sum of the block input signal (weighted by the integral gain I) and **SumI2**. **SumI1** is computed only when **Limit output** and **Anti-windup method back-calculation** are active.
- **SumI2** is the difference between the weighted sum **Sum** and the limited block output. **SumI2** is computed only when **Limit output** and **Anti-windup method back-calculation** are active.

- **SumI3** is the difference between the block output and the signal at the block's tracking input. **SumI3** is computed only when you select the **Enable tracking mode** box.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Inherit: Inherit via internal rule (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of Inherit: Same as first input.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.

Note The accumulator internal rule favors greater numerical accuracy, possibly at the cost of less efficient generated code. To get the same accuracy for the output, set the output data type to Inherit: Same as accumulator.

Inherit: Inherit via back propagation

Use data type of the driving block.

Inherit: Same as first input

Use data type of first input signal.

Inherit: Same as accumulator

Use the same data type as the corresponding accumulator on page 1-1076.

double

single

int8

uint8

int16

uint16

int32

uint32

fixdt(1,16)

fixdt(1,16,0)

fixdt(1,16,2^0,0)

<data type expression>

Name of a data type object. For example, `Simulink.NumericType`.

Accumulator data type

Specify the accumulator data type.

Default: Inherit: Inherit via internal rule

Inherit: Inherit via internal rule

Use internal rule to determine accumulator data type.

Inherit: Same as first input

Use data type of first input signal.

double

Accumulator data type is double.

single

Accumulator data type is single.

int8

Accumulator data type is int8.

uint8

Accumulator data type is uint8.

`int16`

Accumulator data type is `int16`.

`uint16`

Accumulator data type is `uint16`.

`int32`

Accumulator data type is `int32`.

`uint32`

Accumulator data type is `uint32`.

`fixdt(1,16,0)`

Accumulator data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`

Accumulator data type is fixed point `fixdt(1,16,2^0,0)`.

<data type expression>

The name of a data type object, for example `Simulink.NumericType`

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Specify Data Types Using Data Type Assistant”.

Integrator output data type

Select the data type of the integrator output.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Inherit: Inherit via internal rule (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet

your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use `Inherit: Inherit via back propagation`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.

`Inherit: Inherit via back propagation`

Use data type of the driving block.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`

Name of a data type object. For example, `Simulink.NumericType`.

Filter output data type

Select the data type of the filter output.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Inherit: Inherit via internal rule (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use Inherit: Inherit via back propagation.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.

Inherit: Inherit via back propagation

Use data type of the driving block.

`double`

`single`

`int8`

`uint8`

`int16`

`uint16`

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`

Name of a data type object. For example, `Simulink.NumericType`.

Saturation output data type

Select the saturation output data type.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Inherit: Same as input (Default)

Use data type of input signal.

Inherit: Inherit via back propagation

Use data type of the driving block.

double

single

int8

uint8

int16

uint16

int32

uint32

fixdt(1,16)

fixdt(1,16,0)

fixdt(1,16,2^0,0)

<data type expression>

Name of a data type object. For example, `Simulink.NumericType`.

Mode

Select the category of data to specify.

Default: `Inherit`

Inherit

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule (default)`
- `Inherit via back propagation`
- `Same as first input`
- `Same as accumulator`

Built in

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

Fixed point

Fixed-point data types.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via back propagation`
- `Same as input (default)`

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Fixed-point data types.

`Expression`

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Mode

Select the category of accumulator data to specify

Default: `Inherit`

`Inherit`

Specifies inheritance rules for data types. Selecting `Inherit` enables a list of possible values:

- `Inherit via internal rule (default)`
- `Same as first input`

`Built in`

Specifies built-in data types. Selecting `Built in` enables a list of possible values:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Specifies fixed-point data types.

`Expression`

Specifies expressions that evaluate to data types. Selecting `Expression` enables you to enter an expression.

Clicking the **Show data type assistant** button for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data as signed.

`Unsigned`

Specify the fixed-point data as unsigned.

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Signedness

Specify whether you want the fixed-point data to be signed or unsigned.

Default: Signed

Signed

Specify the fixed-point data to be signed.

Unsigned

Specify the fixed-point data to be unsigned.

Selecting **Mode** > Fixed point for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision, Binary point, Integer

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values. This option appears for some blocks.

Integer

Specify integer. This setting has the same result as specifying a binary point location and setting fraction length to 0. This option appears for some blocks.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting Slope and bias enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Binary point

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**

Selecting `Slope` and `bias` enables:

- **Slope**
- **Bias**

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that will hold the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Large word sizes represent large values with greater precision than small word sizes.

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope` and `bias` for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope` and `bias` for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: LockScale

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off

On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Parameter: SaturateOnIntegerOverflow

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: Floor

Ceiling

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

Floor

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

Nearest

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Parameter: RndMeth

Type: character vector

Value: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Default: 'Floor'

For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

State name

Assign unique name to each state. The state names apply only to the selected block.

To assign a name to a single state, enter the name between quotes; for example, 'velocity'.

To assign names to multiple states, enter a comma-delimited list surrounded by braces; for example, {'a', 'b', 'c'}. Each name must be unique. To assign state names with a variable that has been defined in the MATLAB workspace, enter the variable without quotes. The variable can be a character vector, cell, or structure.

Default: ' ' (no name)

State name must resolve to Simulink signal object

Require that state name resolve to Simulink signal object.

Default: Off

On

Require that state name resolve to Simulink signal object.

Off

Do not require that state name resolve to Simulink signal object.

State name enables this parameter. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class**.

Parameter: `StateMustResolveToSignalObject`

Type: character vector

Value: `'off' | 'on'`

Default: `'off'`

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

StorageClass

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than `Simulink`.

State name enables this parameter.

Command-Line Information

Parameter: StateStorageClass

Type: character vector

Value: 'Auto' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'SimulinkGlobal' | 'Custom'

Default: 'Auto'

TypeQualifier

Note **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

- **Default:** ' ' (empty character vector)
- `const`
- `volatile`

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

Parameter Name: RTWStateStorageTypeQualifier

Value Type: character vector

Default: ' ' (empty character vector)

Characteristics

Direct Feedthrough	The following ports support direct feedthrough: <ul style="list-style-type: none">• Reset port• Integrator and filter initial condition port• Input port, for every integration method except Forward Euler
Sample Time	Specified in the Sample time parameter
Scalar Expansion	Supported for gain parameters P , I , and D and for filter coefficient N
States	Inherited from driving block and parameters
Dimensionalized	Yes
Zero-Crossing Detection	Yes (in continuous-time domain)

See Also

Derivative | Discrete Derivative | Discrete-Time Integrator | Gain | Integrator | PID Controller (2 DOF)

Introduced in R2009b

PID Controller (2 DOF) Discrete PID Controller (2 DOF)

Simulate continuous- or discrete-time two-degree-of-freedom PID controllers



Library

Continuous, Discrete

Description

Implement a continuous- or discrete-time two-degree-of-freedom controller (PID, PI, or PD) in your Simulink model. The PID Controller (2DOF) block allows you to implement setpoint weighting in your controller to achieve both smooth setpoint tracking and good disturbance rejection.

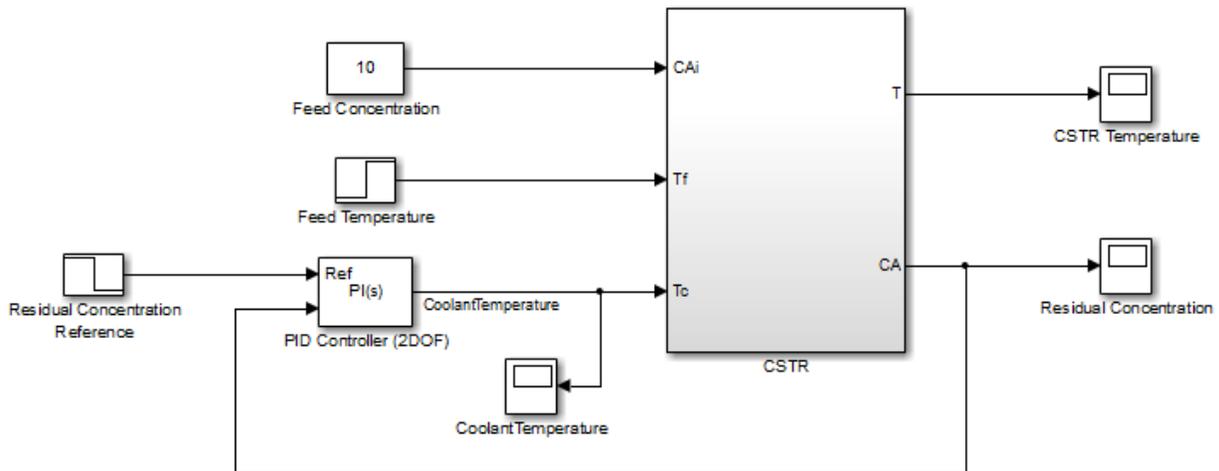
The PID Controller (2DOF) block generates an output signal based on the difference between a reference signal and a measured system output. The block computes a weighted difference signal for each of the proportional, integral, and derivative actions according to the setpoint weights you specify. The block output is the sum of the proportional, integral, and derivative actions on the respective difference signals, where each action is weighted according to the gain parameters. A first-order pole filters the derivative action. Controller gains are tunable either manually or automatically. Automatic tuning requires Simulink Control Design software.

Configurable options in the PID Controller (2DOF) block include:

- Controller type (PID, PI, or PD)
- Controller form (Parallel or Ideal)

- Time domain (continuous or discrete)
- Initial conditions and reset trigger
- Output saturation limits and built-in anti-windup mechanism
- Signal tracking for bumpless control transfer and multiloop control

In one common implementation, the PID Controller (2DOF) block operates in the feedforward path of the feedback loop. The block receives a reference signal at the `Ref` input and a measured system output at the other input. For example:



For a single-input block that accepts an error signal (a difference between a setpoint and a system output), see the PID Controller, block reference page.

You can generate code to implement your controller using any Simulink data type, including fixed-point data types. (Code generation requires Simulink Coder software; fixed-point implementation requires the Fixed-Point Designer product.)

For an example illustrating an application of the PID Controller (2 DOF) block, see the Simulink example Two Degree-of-Freedom PID Control for Setpoint Tracking.

Data Type Support

The PID Controller (2DOF) block accepts real signals of any numeric data type that Simulink software supports, including fixed-point data types. See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Parameters

The following table summarizes the PID Controller (2DOF) block parameters, accessible via the block parameter dialog box.

Task	Parameters
Choose controller form and type.	<ul style="list-style-type: none"> • Controller Form on page 1-1101 in Main tab • Controller on page 1-1105
Choose discrete or continuous time.	<ul style="list-style-type: none"> • Time-domain on page 1-1105 • Sample time on page 1-1108
Choose an integration method (discrete time).	<ul style="list-style-type: none"> • Integrator method on page 1-1106 • Filter method on page 1-1107
Set and tune controller gains.	<ul style="list-style-type: none"> • Controller Parameters Source on page 1-1108 in Main tab • Proportional (P) on page 1-1110 in Main tab • Integral (I) on page 1-1110 in Main tab • Derivative (D) on page 1-1110 in Main tab • Filter coefficient (N) on page 1-1111 in Main tab • Use filtered derivative on page 1-1112 in Main tab • Setpoint weight (b) on page 1-1113 in Main tab • Setpoint weight (c) on page 1-1114 in Main tab

Task	Parameters
Set integrator and filter initial conditions.	<ul style="list-style-type: none"> • Initial conditions Source on page 1-1116 in Main tab • Integrator Initial condition on page 1-1117 in Main tab • Filter Initial condition on page 1-1117 in Main tab • Initial condition setting on page 1-1117 (command line only) • External reset on page 1-1118 in Main tab • Ignore reset when linearizing on page 1-1119 in Main tab
Limit block output.	<ul style="list-style-type: none"> • Limit output on page 1-1120 in PID Advanced tab • Lower saturation limit on page 1-1121 in PID Advanced tab • Upper saturation limit on page 1-1121 in PID Advanced tab • Ignore saturation when linearizing on page 1-1123 in PID Advanced tab
Configure anti-windup mechanism (when you limit block output).	<ul style="list-style-type: none"> • Anti-windup method on page 1-1121 in PID Advanced tab • Back-calculation gain (Kb) on page 1-1123 in PID Advanced tab
Enable signal tracking.	<ul style="list-style-type: none"> • Enable tracking mode on page 1-1124 in PID Advanced tab • Tracking gain (Kt) on page 1-1124 in PID Advanced tab

Task	Parameters
Configure data types.	<ul style="list-style-type: none"> • Parameter data type on page 1-1124 in Data Type Attributes tab • Product output data type on page 1-1126 in Data Type Attributes tab • Summation output data type on page 1-1127 in Data Type Attributes tab • Accumulator data type on page 1-1129 in Data Type Attributes tab • Integrator output data type on page 1-1131 in Data Type Attributes tab • Filter output data type on page 1-1132 in Data Type Attributes tab • Saturation output data type on page 1-1133 in Data Type Attributes tab • Lock output data type setting against changes by the fixed-point tools on page 1-1144 in Data Type Attributes tab • Saturate on integer overflow on page 1-1144 in Data Type Attributes tab • Integer rounding mode on page 1-1145 in Data Type Attributes tab
Configure block for code generation.	<ul style="list-style-type: none"> • State name on page 1-1146 in State Attributes tab • State name must resolve to Simulink signal object on page 1-1147 in State Attributes tab • Code generation storage class on page 1-1147 in State Attributes tab • Code generation storage type qualifier on page 1-1148 in State Attributes tab

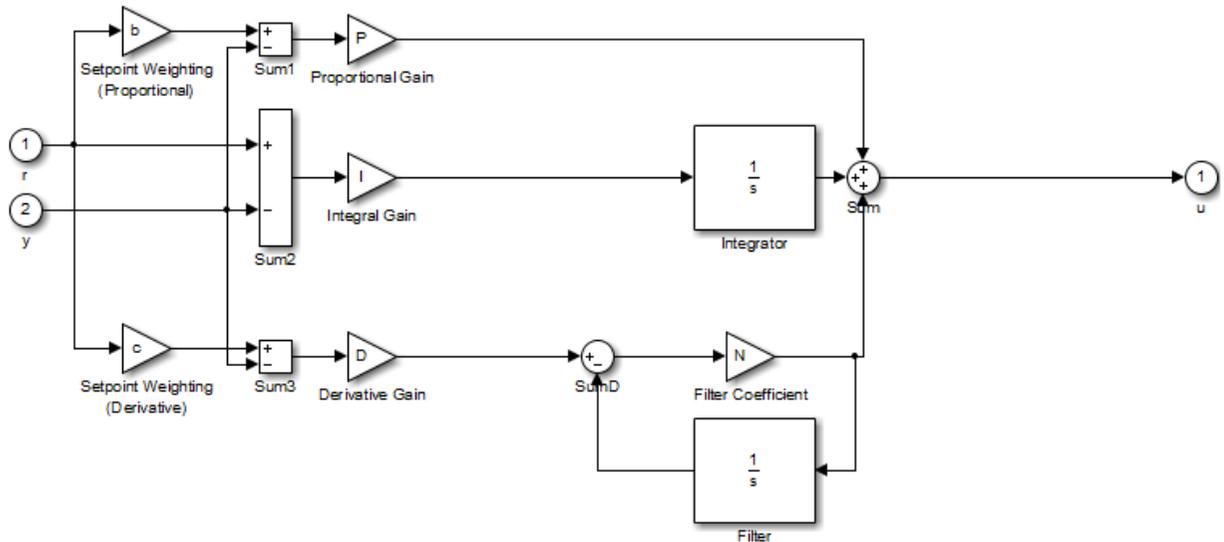
Controller form

Select the controller form.

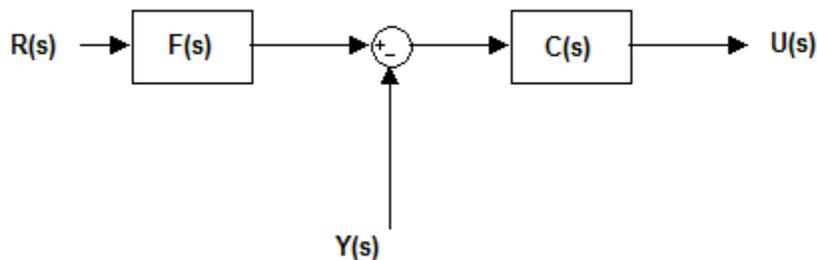
Parallel (Default)

Selects a controller form in which the proportional, integral, and derivative gains **P**, **I**, and **D** operate independently. The filter coefficient **N** sets the location of the pole in the derivative filter.

Parallel two-degree-of-freedom PID controller, where input 1 receives a reference signal and input 2 receives feedback from the measured system output:



The parallel two-degree-of-freedom PID controller can be equivalently modeled by the following block diagram:



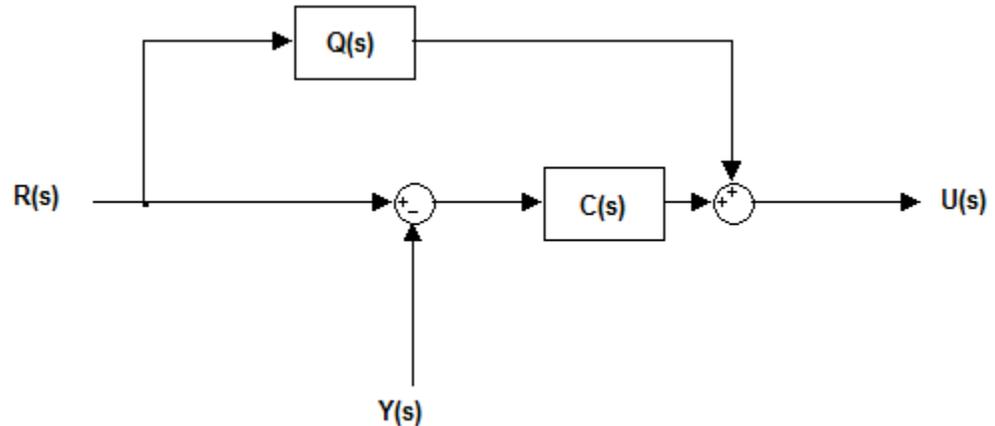
$R(s)$ represents the reference signal and $Y(s)$ represents the feedback from measured system output. In this model, $C(s)$ is a single degree-of-freedom controller, and $F(s)$ acts as a prefilter on the reference signal. For a parallel two-degree-of-freedom PID controller in the Continuous-time **Time-domain**, the transfer functions $F(s)$ and $C(s)$ are:

$$F_{par}(s) = \frac{(bP + cDN)s^2 + (bPN + I)s + IN}{(P + DN)s^2 + (PN + I)s + IN}$$

$$C_{par}(s) = \frac{(P + DN)s^2 + (PN + I)s + IN}{s(s + N)}$$

where b and c are the **Setpoint weight** parameters.

Alternatively, the parallel two-degree-of-freedom PID controller can be modeled by the following block diagram:



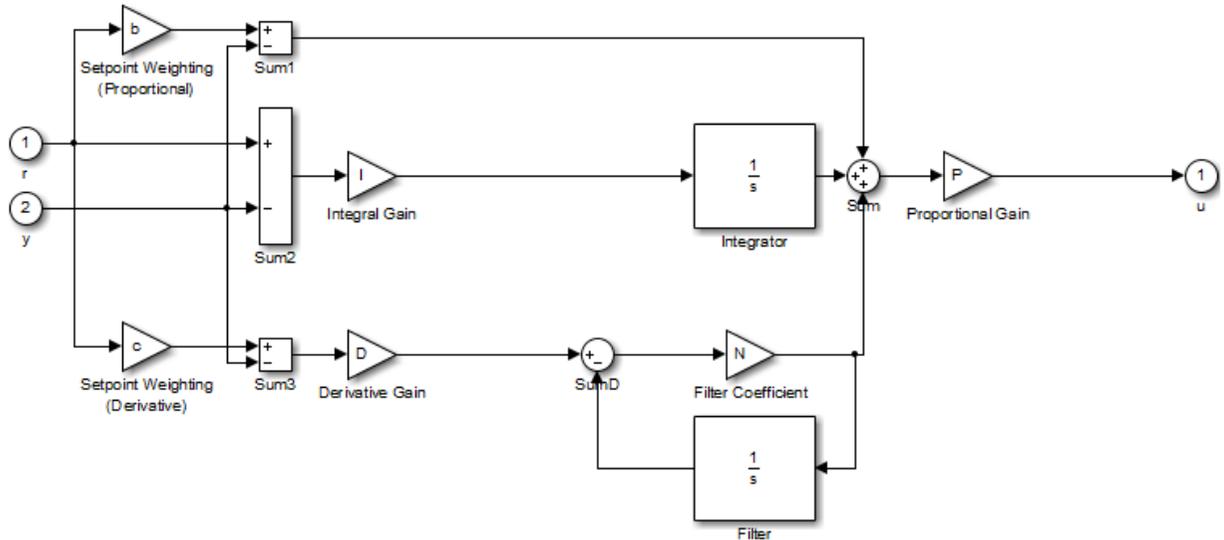
$R(s)$, $Y(s)$, and $C(s)$ are as discussed previously. In this realization, $Q(s)$ acts as feed-forward conditioning on the reference signal $R(s)$. For a parallel PID controller in the Continuous-time **Time-domain**, the transfer function $Q(s)$ is:

$$Q_{par}(s) = \frac{((b-1)P + (c-1)DN)s + (b-1)PN}{s + N}$$

Ideal

Selects a controller form in which the proportional gain P acts on the sum of all actions.

Ideal two-degree-of-freedom PID controller, where input 1 receives a reference signal and input 2 receives feedback from the measured system output:



Similarly to the parallel controller form discussed previously, the ideal two-degree-of-freedom PID controller can be modeled as a single degree-of-freedom controller $C(s)$ with a prefilter $F(s)$. For an ideal two-degree-of-freedom PID controller in the Continuous-time **Time-domain**, the transfer functions $F(s)$ and $C(s)$ are:

$$F_{id}(s) = \frac{(b + cDN)s^2 + (bN + I)s + IN}{(1 + DN)s^2 + (N + I)s + IN}$$

$$C_{id}(s) = P \frac{(1 + DN)s^2 + (N + I)s + IN}{s(s + N)}$$

where b and c are the **Setpoint weight** parameters.

Alternatively, modeling the ideal two-degree-of-freedom PID controller as a one-degree-of-freedom controller $C(s)$ with feed-forward conditioning $Q(s)$ on the reference signal gives, in continuous-time:

$$Q_{id}(s) = P \frac{((b-1) + (c-1)DN)s + (b-1)N}{s + N}$$

The controller transfer function for the current settings is displayed in the block dialog box.

Controller

Specify the controller type.

PID (Default)

Implements a controller with proportional, integral, and derivative action.

PI

Implements a controller with proportional and integral action.

PD

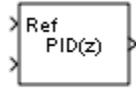
Implements a controller with proportional and derivative action.

The controller transfer function for the current settings is displayed in the block dialog box.

Time-domain

Select continuous or discrete time domain. The appearance of the block changes to reflect your selection.





Continuous-time (Default)

Selects the continuous-time representation.

When the PID Controller (2DOF) block is in a model with synchronous state control (see the State Control block), you cannot select `Continuous-time`.

Discrete-time

Selects the discrete-time representation. Selecting `Discrete-time` also allows you to specify the:

- **Sample time**, which is the discrete interval between samples.
- Discrete integration methods for the integrator and the derivative filter using the **Integrator method** and **Filter method** menus.

Integrator method

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the method used to compute the integrator output. For more information about discrete-time integration methods, see the Discrete-Time Integrator block reference page.

Forward Euler (Default)

Selects the Forward Rectangular (left-hand) approximation.

This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the Forward Euler method can result in instability, even when discretizing a system that is stable in continuous time.

Backward Euler

Selects the Backward Rectangular (right-hand) approximation.

An advantage of the `Backward Euler` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result.

`Trapezoidal`

Selects the Bilinear approximation.

An advantage of the `Trapezoidal` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Of all available integration methods, the `Trapezoidal` method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

Note For integrator methods `BackwardEuler` or `Trapezoidal`, you cannot generate HDL code for the block if either:

- **Limit output** is selected and **Anti-windup method** is set to `back-calculation` or `clamping`.
 - **Enable tracking mode** is selected.
-

Filter method

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the method used to compute the derivative filter output. For more information about discrete-time integration methods, see the `Discrete-Time Integrator` block reference page.

`Forward Euler (Default)`

Selects the Forward Rectangular (left-hand) approximation.

This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the `Forward Euler` method can result in instability, even when discretizing a system that is stable in continuous time.

`Backward Euler`

Selects the Backward Rectangular (right-hand) approximation.

An advantage of the `Backward Euler` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Any filter parameter value $N > 0$ yields a stable result with this method.

Trapezoidal

Selects the Bilinear approximation.

An advantage of the `Trapezoidal` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Any filter parameter value $N > 0$ yields a stable result with this method. Of all available filter methods, the `Trapezoidal` method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

Sample time (-1 for inherited)

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the discrete interval between samples.

Default: 1

By default, the block uses a discrete sample time of 1. To specify a different sample time, enter another discrete value, such as 0.1.

If you specify a value of -1 , the `PID Controller (2DOF)` block inherits the sample time from upstream blocks. Do not enter a value of 0; to implement a continuous-time controller, select the **Time-domain** `Continuous-time`.

See “Specify Sample Time” in the online documentation for more information.

Controller Parameters Source

Select the source of the controller gains, filter coefficient, and setpoint weights. You can provide these parameters explicitly in the block dialog box, or enable external inputs for them on the block. Enabling external inputs for the parameters allows you to compute PID gains and filter coefficients externally to the block and provide them to the block as signal inputs.

External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to

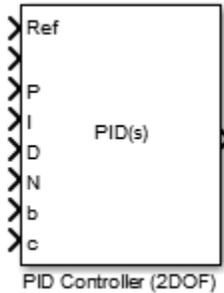
implement gain-scheduled PID control, in which controller gains are determined by logic or other calculation in the Simulink model and passed to the block.

`internal` (Default)

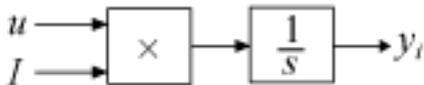
Specify the PID gains and filter coefficient explicitly using the **P**, **I**, **D**, **N**, **b**, and **c** parameters.

`external`

Specify the PID gains and filter coefficient externally. An additional input port appears under the block input for each parameter that is required for the current controller type:



When you supply gains externally, time variations in the integral and derivative gain values are integrated and differentiated, respectively. This result occurs because of the way the PID gains are implemented within the block. For example, for a continuous-time PID controller with external inputs, the integrator term is implemented as shown in the following illustration.



Within the block, the signal to be integrated is multiplied by the externally-supplied integrator gain, I , before integration. This implementation yields:

$$y_i = \int u I dt.$$

Thus, the integrator gain is included in the integral. Similarly, in the derivative term of the block, multiplication by the derivative gain precedes the differentiation, which causes the derivative gain D to be differentiated.

Proportional (P)

Specify the proportional gain P .

Default: 1

Enter a finite, real gain value into the **Proportional (P)** field. Use either scalar or vector gain values. For a **parallel PID Controller form**, the proportional action is independent of the integral and derivative actions. For an **ideal PID Controller form**, the proportional action acts on the integral and derivative actions. See “Controller form” on page 1-1101 for more information about the role of P in the controller transfer function.

When you have Simulink Control Design software installed, you can automatically tune the controller gains. See “Choose a Control Design Approach” (Simulink Control Design).

Integral (I)

(Available for PID and PI controllers.) Specify the integral gain I .

Default: 1

Enter a finite, real gain value into the **Integral (I)** field. Use either scalar or vector gain values.

When you have Simulink Control Design software installed, you can automatically tune the controller gains. See “Choose a Control Design Approach” (Simulink Control Design).

Derivative (D)

(Available for PID and PD controllers.) Specify the derivative gain D .

Default: 0

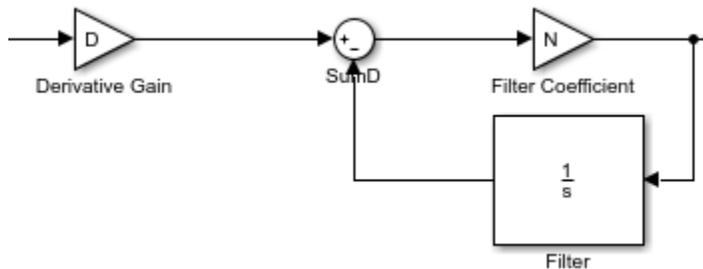
Enter a finite, real gain value into the **Derivative (D)** field. Use either scalar or vector gain values.

When you have Simulink Control Design software installed, you can automatically tune the controller gains using. See “Choose a Control Design Approach” (Simulink Control Design).

Filter coefficient (N)

Specifies the filter coefficient of the controller.

(Available for PID and PD controllers, when **Use filtered derivative** is checked.) Specify the filter coefficient N, which determines the pole location of the filter in the derivative action:



The filter pole falls at $s = -N$ in the Continuous-time **Time-domain**. For Discrete-time, the location of the pole depends on which **Filter method** you select (for sampling time T_s):

- Forward Euler:

$$z_{pole} = 1 - NT_s$$

- Backward Euler:

$$z_{pole} = \frac{1}{1 + NT_s}$$

- Trapezoidal:

$$z_{pole} = \frac{1 - NT_s / 2}{1 + NT_s / 2}$$

Default: 100.

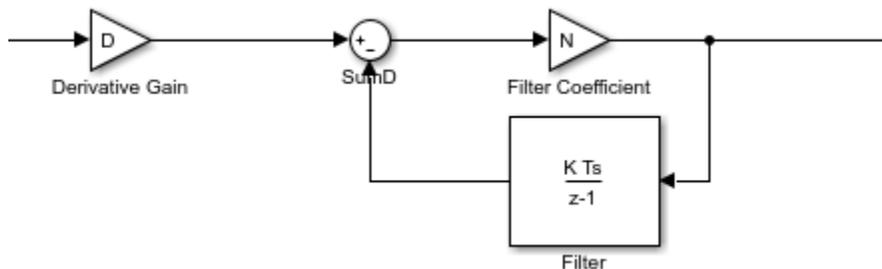
Enter a finite, real gain value into the **Filter Coefficient (N)** field. Use either scalar or vector gain values. Note that the PID controller (2DOF) block does not support $N = \text{inf}$ (ideal unfiltered derivative).

When you have Simulink Control Design software installed, you can automatically tune the controller gains. See “Choose a Control Design Approach” (Simulink Control Design). Automatic tuning requires $N > 0$.

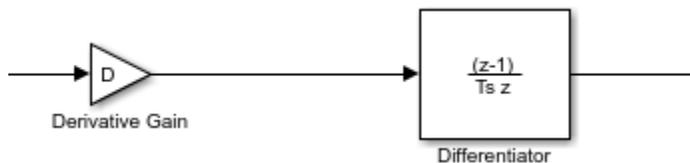
Use Filtered Derivative

Specify whether derivative term is filtered (finite N) or unfiltered. Unfiltered derivative is available only for discrete-time controllers.

Unchecking this option replaces the filtered derivative with a discrete differentiator. For example, if **Filter Method** is Forward Euler, then the filtered derivative term is represented by:



When you uncheck **Use filtered derivative**, the derivative term becomes:



- On (Default)
Use derivative filter (finite N).
- Off
Derivative is unfiltered.

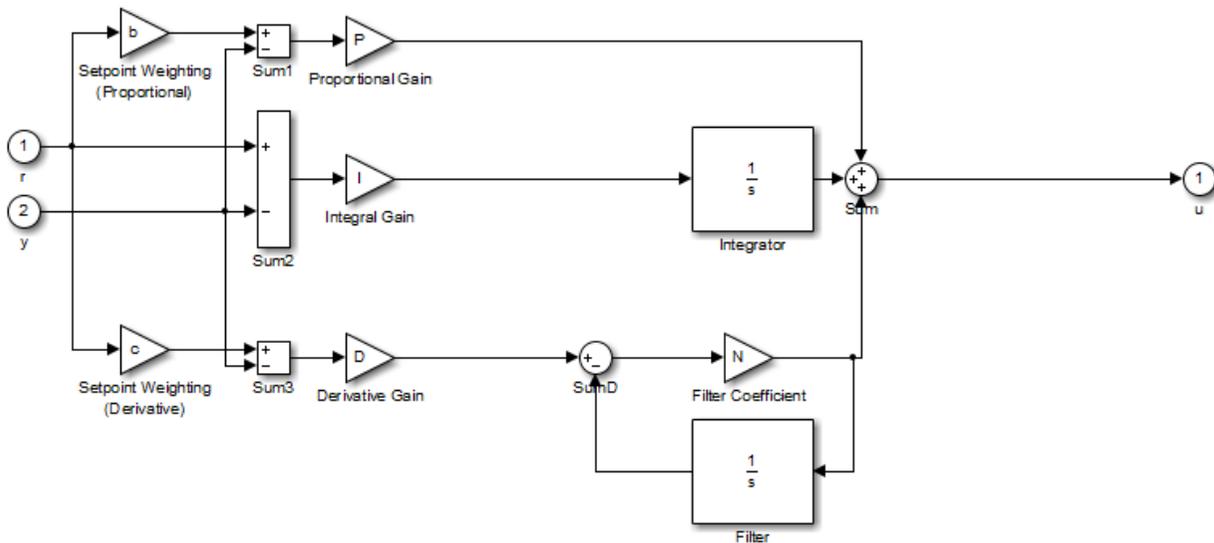
Setpoint weight (b)

Specify the proportional setpoint weight b.

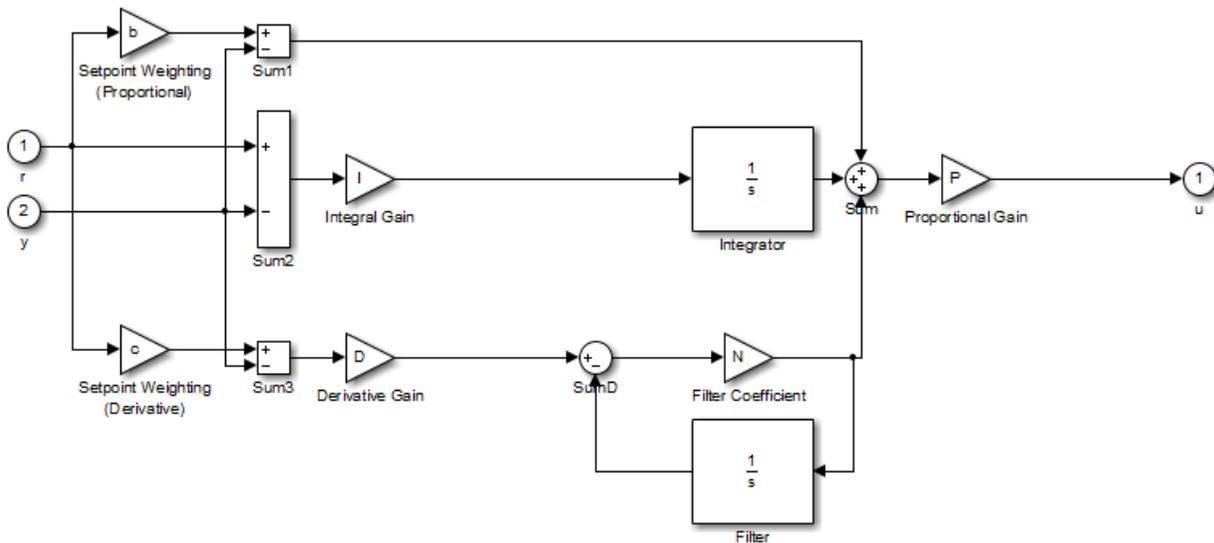
Default: 1

Enter the proportional setpoint weight value into the **Setpoint weight (b)** field. Setting $b = 0$ eliminates the proportional action on the reference signal, which can reduce overshoot in the system response to step changes in the setpoint.

The following diagrams show the role of **Setpoint weight (b)** in PID controllers of Parallel and Ideal form. See “Controller form” on page 1-1101 for a discussion of the corresponding transfer functions.



Parallel Two-Degree-of-Freedom PID Controller



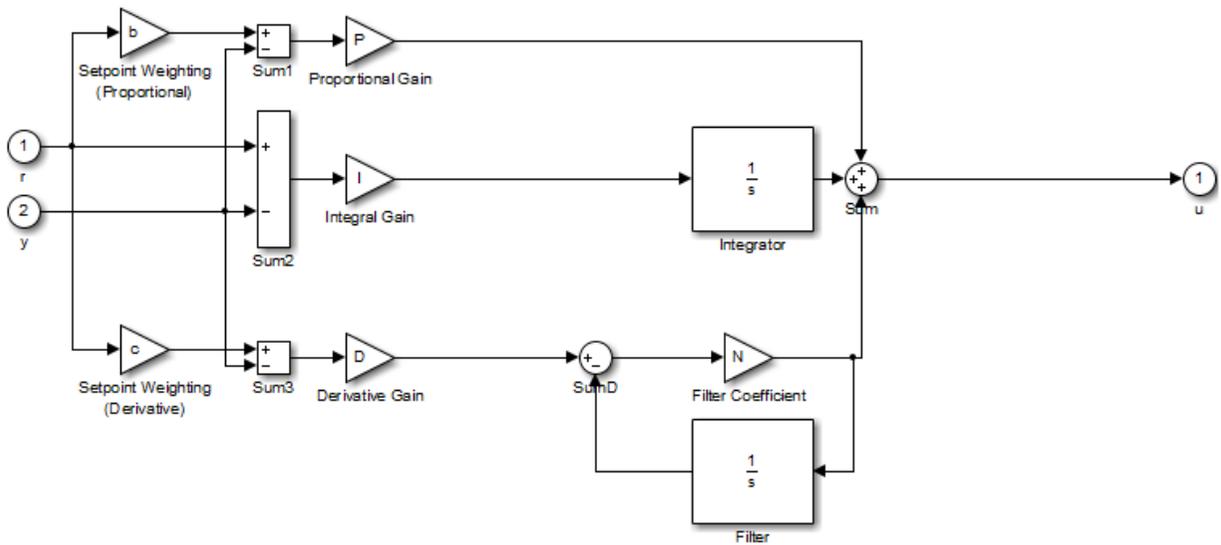
Ideal Two-Degree-of-Freedom PID Controller

Setpoint weight (c)

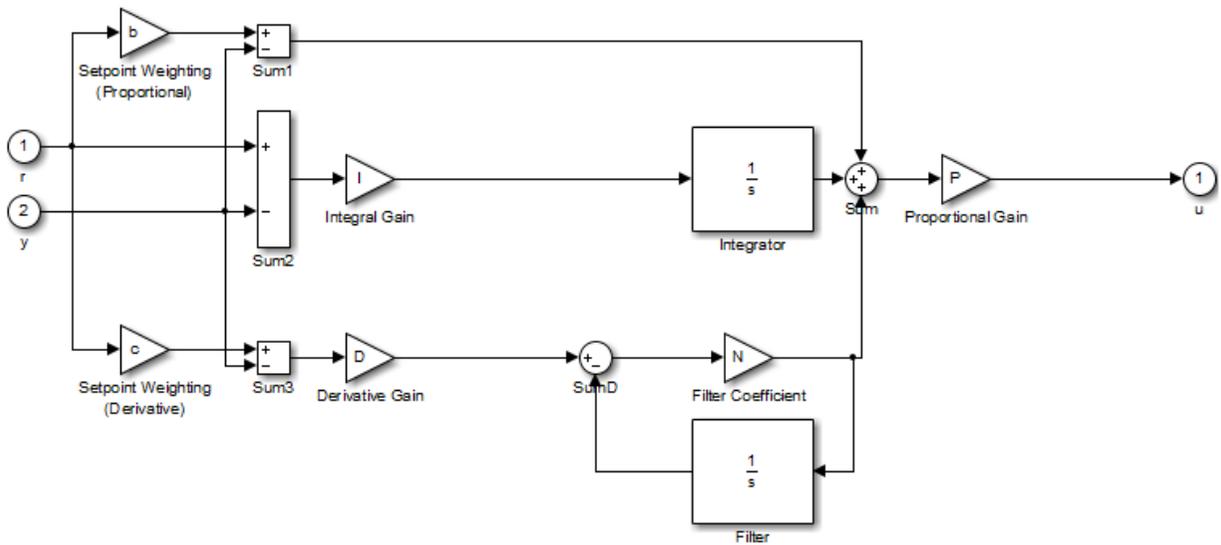
(Available for PID and PD controllers.) Specify the derivative setpoint weight c .

Enter the derivative setpoint weight value into the **Setpoint weight (c)** field. To implement a controller that achieves both effective disturbance rejection and smooth setpoint tracking without excessive transient response, set $c = 0$. Setting $c = 0$ yields a controller with derivative action on the measured system response but not on the reference input.

The following diagrams show the role of **Setpoint weight (c)** in Parallel and Ideal PID controllers. See “Controller form” on page 1-1101 for a discussion of the corresponding transfer functions.



Parallel Two-Degree-of-Freedom PID Controller



Ideal Two-Degree-of-Freedom PID Controller

Select Tuning Method

(Requires Simulink Control Design) Select a PID tuning tool. The specified tool opens when you click **Tune**.

Transfer Function Based (PID Tuner App) (Default)

Clicking **Tune** opens the PID Tuner, which lets you interactively tune PID gains while examining relevant system responses to validate performance. By default, PID Tuner works with a linearization of your plant model. For models that cannot be linearized, you can tune PID gains against a plant model estimated from simulated or measured response data. For more information, see “Introduction to Model-Based PID Tuning in Simulink” (Simulink Control Design).

Frequency Response Based

Clicking **Tune** opens the Frequency Response Based PID Tuner, which tunes PID controller gains based on a frequency-response estimation experiment. Frequency Response Based PID Tuner is especially useful for plants that are not linearizable. For more information, see “Design PID Controller from Plant Frequency-Response Data” (Simulink Control Design).

Initial conditions Source

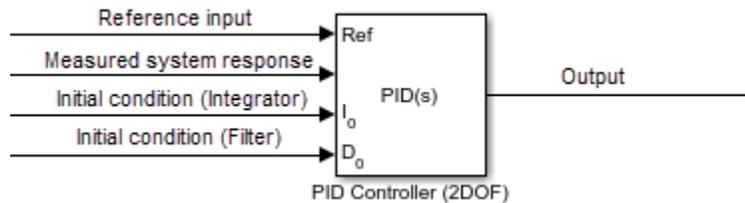
Select the source of the integrator and filter initial conditions. Simulink uses initial conditions to initialize the integrator and filter output at the start of a simulation or at a specified trigger event (see “External reset” on page 1-1118). The integrator and filter initial conditions in turn determine the initial block output.

internal (Default)

Specifies the integrator and filter initial conditions explicitly using the **Integrator Initial condition** and **Filter Initial condition** parameters.

external

Specifies the integrator and filter initial conditions externally. An additional input port appears under the block inputs for each initial condition: I_0 for the integrator and D_0 for the filter:



Integrator Initial condition

(Available only when **Initial conditions Source** is *internal* and the controller includes integral action.) Specify the integrator initial value. Simulink uses the initial condition to initialize the integrator output at the start of a simulation or at a specified trigger event (see “External reset” on page 1-1118). The integrator initial condition, together with the filter initial condition, determines the initial output of the PID Controller (2DOF) block.

Default: 0

Simulink does not permit the integrator initial condition to be *inf* or NaN.

Filter Initial condition

(Available only when **Initial conditions Source** is *internal*, the controller includes derivative action, and **Use filtered derivative** is checked.) Specify the filter initial value. Simulink uses the initial condition to initialize the filter output at the start of a simulation or at a specified trigger event (see “External reset” on page 1-1118). The filter initial condition, together with the integrator initial condition, determines the initial output of the PID Controller (2DOF) block.

Default: 0

Simulink does not permit the filter initial condition to be *inf* or NaN.

Initial condition setting

Specify whether to apply the **Integrator Initial condition** and **Filter Initial condition** parameter to the corresponding block state or output. This parameter can be

changed only at the command line using `set_param` to set the `InitialConditionSetting` parameter of the block.

State (most efficient) (Default)

Use this option in all situations except when the block is in a triggered subsystem or a function-call subsystem and simplified initialization mode is enabled.

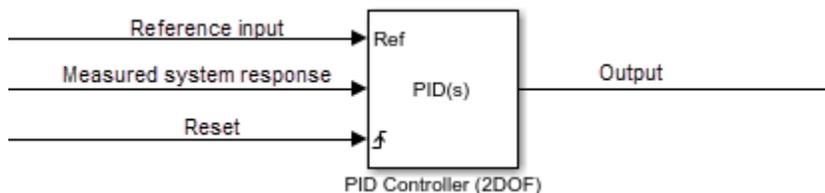
Output

Use this option when the block is in a triggered subsystem or a function-call subsystem and simplified initialization mode is enabled.

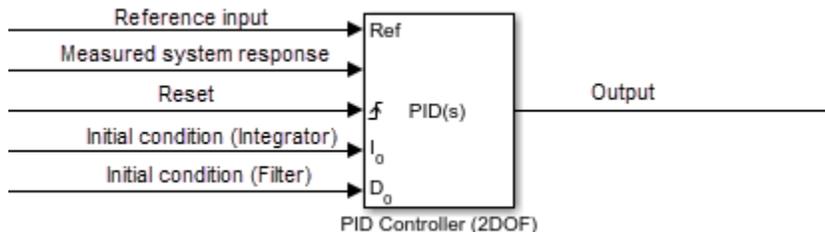
For more information about the **Initial condition setting** parameters, see the Discrete-Time Integrator block.

External reset

Select the trigger event that resets the integrator and filter outputs to the initial conditions you specify in the **Integrator Initial condition** and **Filter Initial condition** fields. Selecting any option other than `none` enables a reset input on the block for the external reset signal, as shown:



Or, if the **Initial conditions Source** is External:



The reset signal must be a scalar of type `single`, `double`, `boolean`, or `integer`. Fixed point data types, except for `ufix1`, are not supported.

Note To be compliant with the Motor Industry Software Reliability Association (MISRA) software standard, your model must use Boolean signals to drive the external reset ports of the PID controller (2DOF) block.

`none` (Default)

Does not reset the integrator and filter outputs to initial conditions.

`rising`

Resets the outputs when the reset signal has a rising edge.

`falling`

Resets the outputs when the reset signal has a falling edge.

`either`

Resets the outputs when the reset signal either rises or falls.

`level`

Resets and holds the outputs to the initial conditions while the reset signal is nonzero.

Ignore reset when linearizing

Force Simulink linearization commands to ignore any reset mechanism that you have chosen with the **External reset** menu. Ignoring reset states allows you to linearize a model around an operating point even if that operating point causes the PID Controller (2DOF) block to reset.

Off (Default)

Simulink linearization commands do not ignore states corresponding to the reset mechanism.

On

Simulink linearization commands ignore states corresponding to the reset mechanism.

Enable zero-crossing detection

Enable zero-crossing detection in continuous-time models upon reset and upon entering or leaving a saturation state.

Zero-crossing detection can accurately locate signal discontinuities without resorting to excessively small time steps that can lead to lengthy simulation times. If you select **Limit output** or activate an **External reset** in your PID Controller (2DOF) block, activating zero-crossing detection can reduce computation time in your simulation. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

On (Default)

Uses zero-crossing detection at any of the following events: reset; entering or leaving an upper saturation state; and entering or leaving a lower saturation state.

Off

Does not use zero-crossing detection.

Enabling zero-crossing detection for the PID Controller (2DOF) block also enables zero-crossing detection for all under-mask blocks that include the zero-crossing detection feature.

Limit output

Limit the block output to values you specify as the **Lower saturation limit** and **Upper saturation limit** parameters.

Activating this option limits the block output internally to the block, obviating the need for a separate Saturation on page 1-1307 block after the controller in your Simulink model. It also allows you to activate the built-in anti-windup mechanism (see “Anti-windup method” on page 1-1121).

Off (Default)

Does not limit the block output, which is the weighted sum of the proportional, integral, and derivative actions.

On

Limits the block output to the **Lower saturation limit** or the **Upper saturation limit** whenever the weighted sum exceeds those limits. Allows you to select an **Anti-windup method**.

Lower saturation limit

(Available only when you select the **Limit Output** box.) Specify the lower limit for the block output. The block output is held at the **Lower saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions goes below that value.

Default: `-inf`

Upper saturation limit

(Available only when you select the **Limit Output** box.) Specify the upper limit for the block output. The block output is held at the **Upper saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions exceeds that value.

Default: `inf`

Anti-windup method

(Available only when you select the **Limit Output** option and the controller includes integral action.) Select an anti-windup mechanism to discharge the integrator when the block is saturated, which occurs when the sum of the block components exceeds the output limits.

When you select the **Limit output** check box and the weighted sum of the controller components exceeds the specified output limits, the block output holds at the specified limit. However, the integrator output can continue to grow (integrator wind-up), increasing the difference between the block output and the sum of the block components. Without a mechanism to prevent integrator wind-up, two results are possible:

- If the sign of the input signal never changes, the integrator continues to integrate until it overflows. The overflow value is the maximum or minimum value for the data type of the integrator output.

- If the sign of the input signal changes once the weighted sum has grown beyond the output limits, it can take a long time to discharge the integrator and return the weighted sum within the block saturation limit.

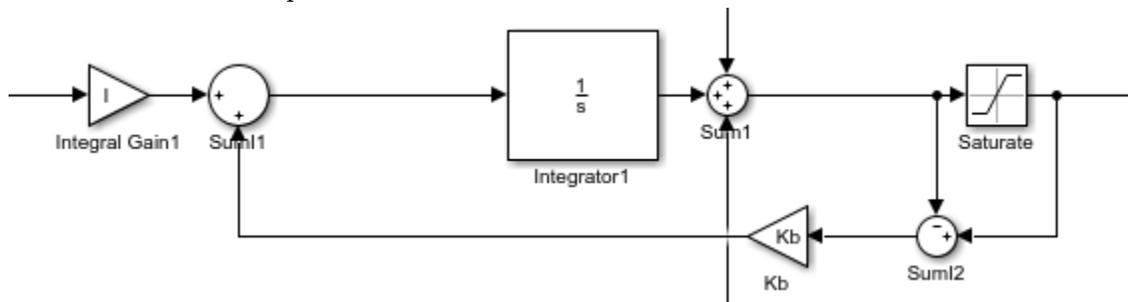
In both cases, controller performance can suffer. To combat the effects of wind-up without an anti-windup mechanism, it may be necessary to detune the controller (for example, by reducing the controller gains), resulting in a sluggish controller. Activating an anti-windup mechanism can improve controller performance.

none (Default)

Does not use an anti-windup mechanism. This setting can cause the block's internal signals to be unbounded even if the output appears to be bounded by the saturation limits. This can result in slow recovery from saturation or unexpected overflows.

back-calculation

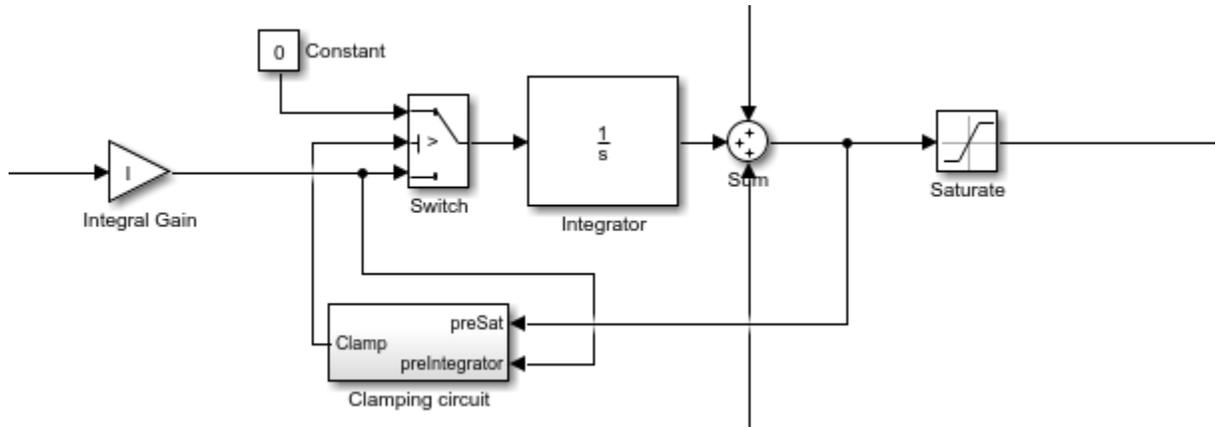
Discharges the integrator when the block output saturates using the integral-gain feedback loop:



You can also specify a value for the **Back-calculation coefficient (Kb)**.

clamping

Stops integration when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Resumes integration when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. The integrator portion of the block is:



The clamping circuit implements the logic necessary to determine whether integration continues.

Back-calculation gain (Kb)

(Available only when the back-calculation **Anti-windup method** is active.) Specify the gain coefficient of the anti-windup feedback loop.

The back-calculation anti-windup method discharges the integrator on block saturation using a feedback loop having gain coefficient Kb.

Default: 1

Ignore saturation when linearizing

Force Simulink linearization commands ignore PID Controller (2DOF) block output limits. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the PID Controller (2DOF) block to exceed the output limits.

On (Default)

Simulink linearization commands ignore states corresponding to saturation.

Off

Simulink linearization commands do not ignore states corresponding to saturation.

Enable tracking mode

(Available for any controller with integral action.) Activate signal tracking, which lets the output of the PID Controller (2DOF) block follow a tracking signal. Provide the tracking signal to the block at the TR port, which becomes active when you select **Enable tracking mode**.

When signal tracking is active, the difference between the tracked signal and the block output is fed back to the integrator input with a gain K_t . You can also specify the value of the **Tracking coefficient (Kt)**.

For information about using tracking mode to implement bumpless control transfer scenarios and multiloop controllers, see “Enable tracking mode” on page 1-1068 in the PID Controller reference page.

Off (Default)

Disables signal tracking and removes TR block input.

On

Enables signal tracking and activates TR input.

Tracking gain (Kt)

(Available only when you select **Enable tracking mode**.) Specify K_t , which is the gain of the signal tracking feedback loop.

Default: 1

Parameter data type

Select the data type of the gain parameters **P**, **I**, **D**, **N**, **Kb**, and **Kt** and the setpoint weighting parameters **b** and **c**.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Inherit: Inherit via internal rule (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of Inherit: Same as input.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use Inherit: Inherit via back propagation and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

Inherit: Inherit via back propagation

Use data type of the driving block.

Inherit: Same as input

Use data type of input signal.

double

single

int8

uint8

int16

uint16

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`

Name of a data type object. For example, `Simulink.NumericType`.

Product output data type

Select the product output data type of the gain parameters **P**, **I**, **D**, **N**, **Kb**, and **Kt**, the setpoint weighting parameters **b** and **c**, or the product output data type of the derivative filter numerator and denominator.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Inherit: Inherit via internal rule (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of **Inherit:** Same as input.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use **Inherit:** Inherit via back propagation and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

Inherit: Inherit via back propagation

Use data type of the driving block.

Inherit: Same as input

Use data type of input signal.

double

single

int8

uint8

int16

uint16

int32

uint32

fixdt(1,16)

fixdt(1,16,0)

fixdt(1,16,2⁰,0)

<data type expression>

Name of a data type object. For example, Simulink.NumericType.

Summation output data type

Select the summation output data type of the sums **Sum**, **Sum D**, **Sum I1**, **SumI2**, **SumI3**, and the summations in the trapezoidal discrete filter numerator and denominator, **SumNum** and **SumDen**. These sums are computed internally within the block, and might not all be present, depending upon the controller configuration. To see where Simulink computes each of these sums, right-click the PID Controller (2 DOF) block in your model and select `Look Under Mask`:

- **Sum** is the weighted sum of the proportional, derivative, and integral signals.
- **Sum1** is the difference between the reference input weighted by **b** and the measured system response.

- **Sum2** is the difference between the reference input weighted by **c** and the measured system response.
- **Sum3** is the difference between the unweighted reference input and the measured system response.
- **SumD** is the sum in the derivative filter feedback loop.
- **SumI1** is the sum of the block input signal (weighted by the integral gain **I**) and **SumI2**. **SumI1** is computed only when **Limit output** and **Anti-windup method back-calculation** are active.
- **SumI2** is the difference between the weighted sum **Sum** and the limited block output. **SumI2** is computed only when **Limit output** and **Anti-windup method back-calculation** are active.
- **SumI3** is the difference between the block output and the signal at the block's tracking input. **SumI3** is computed only when you select the **Enable tracking mode** box.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Inherit: `Inherit via internal rule (Default)`

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit: Same as first input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.

Note The accumulator internal rule favors greater numerical accuracy, possibly at the cost of less efficient generated code. To get the same accuracy for the output, set the output data type to `Inherit: Same as accumulator`.

Inherit: Inherit via back propagation

Use data type of the driving block.

Inherit: Same as first input

Use data type of first input signal.

Inherit: Same as accumulator

Use the same data type as the corresponding accumulator on page 1-1129.

double

single

int8

uint8

int16

uint16

int32

uint32

fixdt(1,16)

fixdt(1,16,0)

fixdt(1,16,2⁰,0)

<data type expression>

Name of a data type object. For example, Simulink.NumericType.

Accumulator data type

Specify the accumulator data type.

Default: Inherit: Inherit via internal rule

Inherit: Inherit via internal rule

Use internal rule to determine accumulator data type.

Inherit: Same as first input

Use data type of first input signal.

double

Accumulator data type is double.

single

Accumulator data type is single.

int8

Accumulator data type is int8.

uint8

Accumulator data type is uint8.

int16

Accumulator data type is int16.

uint16

Accumulator data type is uint16.

int32

Accumulator data type is int32.

uint32

Accumulator data type is uint32.

fixdt(1,16,0)

Accumulator data type is fixed point fixdt(1,16,0).

fixdt(1,16,2^0,0)

Accumulator data type is fixed point fixdt(1,16,2^0,0).

<data type expression>

The name of a data type object, for example `Simulink.NumericType`

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Specify Data Types Using Data Type Assistant”.

Integrator output data type

Select the data type of the integrator output.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Inherit: Inherit via internal rule (Default)

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use Inherit: Inherit via back propagation.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.

Inherit: Inherit via back propagation

Use data type of the driving block.

double

single

int8

uint8

int16

uint16

int32

uint32

```
fixdt(1,16)
```

```
fixdt(1,16,0)
```

```
fixdt(1,16,2^0,0)
```

```
<data type expression>
```

Name of a data type object. For example, `Simulink.NumericType`.

Filter output data type

Select the data type of the filter output.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Inherit: `Inherit via internal rule (Default)`

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use `Inherit: Inherit via back propagation`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.

Inherit: `Inherit via back propagation`

Use data type of the driving block.

```
double
```

```
single
```

```
int8
```

uint8

int16

uint16

int32

uint32

fixdt(1,16)

fixdt(1,16,0)

fixdt(1,16,2^0,0)

<data type expression>

Name of a data type object. For example, `Simulink.NumericType`.

Saturation output data type

Select the saturation output data type.

See “Data Types Supported by Simulink” in the Simulink documentation for more information.

Inherit: Same as input (Default)

Use data type of input signal.

Inherit: Inherit via back propagation

Use data type of the driving block.

double

single

int8

uint8

int16

uint16

`int32`

`uint32`

`fixdt(1,16)`

`fixdt(1,16,0)`

`fixdt(1,16,2^0,0)`

`<data type expression>`

Name of a data type object. For example, `Simulink.NumericType`.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule (default)`
- `Inherit via back propagation`
- `Same as first input`
- `Same as accumulator`

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`

- `uint16`
- `int32`
- `uint32`

Fixed point

Fixed-point data types.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via back propagation`
- `Same as input (default)`

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`

- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

Fixed point

Fixed-point data types.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Mode

Select the category of accumulator data to specify

Default: `Inherit`

`Inherit`

Specifies inheritance rules for data types. Selecting `Inherit` enables a list of possible values:

- `Inherit via internal rule (default)`
- `Same as first input`

Built in

Specifies built-in data types. Selecting `Built in` enables a list of possible values:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

Fixed point

Specifies fixed-point data types.

Expression

Specifies expressions that evaluate to data types. Selecting `Expression` enables you to enter an expression.

Clicking the **Show data type assistant** button for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

Off

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data as signed.

`Unsigned`

Specify the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Signedness

Specify whether you want the fixed-point data to be signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data to be signed.

Unsigned

Specify the fixed-point data to be unsigned.

Selecting **Mode** > Fixed point for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision, Binary point, Integer

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values. This option appears for some blocks.

Integer

Specify integer. This setting has the same result as specifying a binary point location and setting fraction length to 0. This option appears for some blocks.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting Slope and bias enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Binary point

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Selecting **Mode** > Fixed point for the accumulator data type enables this parameter.

Selecting Binary point enables:

- **Fraction length**

Selecting Slope and bias enables:

- **Slope**
- **Bias**

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that will hold the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Large word sizes represent large values with greater precision than small word sizes.

Selecting **Mode** > Fixed point for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > Binary point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > Binary point for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Slope

Specify slope for the fixed-point data type.

Default: 2⁰

Specify any positive real number.

Selecting **Scaling** > Slope and bias enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope` and `bias` for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > Slope and bias for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: LockScale

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

 Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Parameter: SaturateOnIntegerOverflow

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: Floor

Ceiling

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

Floor

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

Nearest

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Parameter: `RndMeth`

Type: character vector

Value: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`

Default: `'Floor'`

For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

State name

Assign unique name to each state. The state names apply only to the selected block.

To assign a name to a single state, enter the name between quotes; for example, 'velocity'.

To assign names to multiple states, enter a comma-delimited list surrounded by braces; for example, {'a', 'b', 'c'}. Each name must be unique. To assign state names with a variable that has been defined in the MATLAB workspace, enter the variable without quotes. The variable can be a character vector, cell, or structure.

Default: ' ' (no name)

State name must resolve to Simulink signal object

Require that state name resolve to Simulink signal object.

Default: Off

On

Require that state name resolve to Simulink signal object.

Off

Do not require that state name resolve to Simulink signal object.

State name enables this parameter. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than None.

Selecting this check box disables **Code generation storage class**.

Parameter: StateMustResolveToSignalObject

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Code generation storage class

Select state storage class for code generation.

Default: Auto

Auto

Auto is the appropriate storage class for states that you do not need to interface to external code.

StorageClass

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than Simulink.

State name enables this parameter.

Command-Line Information

Parameter: StateStorageClass

Type: character vector

Value: 'Auto' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'SimulinkGlobal' | 'Custom'

Default: 'Auto'

TypeQualifier

Note **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

- **Default:** ' ' (empty character vector)
- `const`
- `volatile`

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

Parameter Name: `RTWStateStorageTypeQualifier`

Value Type: character vector

Default: ' ' (empty character vector)

Characteristics

Direct Feedthrough	The following ports support direct feedthrough: <ul style="list-style-type: none"> • Reset port • Integrator and filter initial condition port • Input port, for every integration method except Forward Euler
Sample Time	Specified in the Sample time parameter
Scalar Expansion	Supported for gain parameters P , I , and D and for filter coefficient N , and for setpoint weights b and c
States	Inherited from driving block and parameters
Dimensionalized	Yes
Zero-Crossing Detection	Yes (in continuous-time domain)

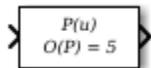
See Also

Derivative | Discrete Derivative | Discrete-Time Integrator | Gain | Integrator | PID Controller

Introduced in R2009b

Polynomial

Perform evaluation of polynomial coefficients on input values



Library

Math Operations

Description

You define a set of polynomial coefficients in the form that the MATLAB `polyval` command accepts. The block evaluates $P(u)$ at each time step for the input u . The inputs and coefficients must be real. The inputs must be vector or scalar.

Data Type Support

The Polynomial block accepts real signals of type `double` or `single`. The **Polynomial coefficients** parameter must be of the same type as the inputs. The output data type is the same as the input data type.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

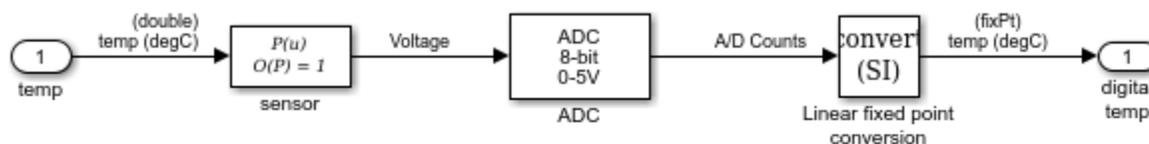
Polynomial coefficients

Specify polynomial coefficients in MATLAB `polyval` form. The first coefficient corresponds to x^N and the remaining coefficients correspond to decreasing orders of x . The last coefficient represents the constant for the polynomial. See `polyval` in the MATLAB documentation for more information.

Examples

The `sldemo_boiler` model shows how to use the Polynomial block.

In the Boiler Plant model/digital thermometer subsystem, the Polynomial block models a first-order polynomial using the coefficients `[0.05 0.75]`:



This subsystem models a digital thermometer composed of a simple temperature sensor and an ADC. The transfer function of the sensor is:

$$V = .05 * T + 0.75$$
 for T in degrees C.

The conversion block inverts the combined transfer function of the sensor and ADC so that the output is an sfix(8) code representing T in degrees C.

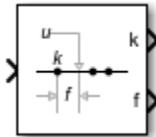
Characteristics

Data Types	Double Single
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Prelookup

Compute index and fraction for Interpolation Using Prelookup block



Library

Lookup Tables

Description

The Prelookup block calculates the index and interval fraction that specify how its input value u relates to the breakpoint dataset. The Prelookup block works best with the Interpolation Using Prelookup block. Feed the resulting index and fraction values into an Interpolation Using Prelookup block to interpolate an n -dimensional table. These two blocks have distributed algorithms. When combined together, they perform the same operation as the integrated algorithm in the n -D Lookup Table block. However, the Prelookup and Interpolation Using Prelookup blocks offer greater flexibility and more efficient simulation and code generation than the n -D Lookup Table block. For more information, see “Efficiency of Performance”.

Supported Block Operations

To use the Prelookup block, you specify a set of breakpoint values directly on the dialog box or feed values into the `bp` input port. Typically, this breakpoint data set corresponds to one dimension of the table data in an Interpolation Using Prelookup block. The Prelookup block generates a pair of outputs for each input value u by calculating:

- The index of the breakpoint set element that is less than or equal to u and forms an interval containing u

- The interval fraction in the range $0 \leq f < 1$, representing the normalized position of u on the breakpoint interval between the index and the next index value for in-range input

For example, if the breakpoint data set is [0 5 10 20 50 100] and the input value u is 55, the index is 4 and the fractional value is 0.1. Labels for the index and interval fraction appear as k and f on the Prelookup block icon. The index value is zero based.

The interval fraction can be negative or greater than 1 for out-of-range input. See the documentation for the **Extrapolation method** block parameter for more information.

Data Type Support

The Prelookup block accepts real signals of numeric data types that Simulink supports, except Boolean. The Prelookup block supports fixed-point data types for signals and breakpoint data. You can also use a bus object as the output data type, which can help to simplify a model.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

- “Main Tab” on page 1-1153
- “Data Types Tab” on page 1-1160

Main Tab

Specification

Specify whether to enter data as explicit breakpoints, as parameters that generate evenly spaced breakpoints, or via a breakpoint object. If you set this parameter to:

- `Explicit values`, the **Source** and **Value** parameters are visible on the dialog box.
- `Even spacing`, the **First point**, **Spacing**, and **Number of points** parameters are visible on the dialog box.

- Breakpoint object, the **Name** parameter is visible on the dialog box.

Source

Specify whether to enter breakpoint data in the dialog box or to inherit the data from an input port. This parameter is available when **Specification** is set to `Explicit` values. If you set **Source** to:

- `Dialog`, enter breakpoint data in the text box under **Value**.
- `Input port`, verify that an upstream signal supplies breakpoint data to the `bp` input port. Each breakpoint data set must be a strictly monotonically increasing vector that contains two or more elements. For this option, your block inherits breakpoint attributes from the `bp` input port.

Value

If you set:

- **Specification** to `Explicit` values and **Source** to `Dialog`, explicitly specify the breakpoint data. Each breakpoint data set must be a strictly monotonically increasing vector that contains two or more elements. For this option, you specify breakpoint attributes on the **Data Types** pane.
- **Source** to `Input port`, verify that an upstream signal supplies table data to the `bp` input port. Each breakpoint data set must be a strictly monotonically increasing vector that contains two or more elements. For this option, you specify breakpoint attributes on the **Data Types** pane.

Click **Edit** to open the Lookup Table Editor (see “Edit Lookup Tables” in the Simulink documentation).

First point

Specify the first point in your evenly spaced breakpoint data. To enable this parameter, set **Specification** to `Even spacing`.

Spacing

Specify the spacing between points in your evenly spaced breakpoint data. To enable this parameter, set **Specification** to `Even spacing`.

Number of points

Specify the number of evenly spaced points in your breakpoint data. To enable this parameter, set **Specification** to `Even spacing`.

Name

If you set **Specification** to `Breakpoint` object, specify the name of an existing Simulink.`Breakpoint` object. An existing breakpoint object already contains breakpoint data.

Index search method

Select `Evenly spaced points`, `Linear search`, or `Binary search`. Each search method has speed advantages in different situations:

- For evenly spaced breakpoint sets (for example, 10, 20, 30, and so on), you achieve optimal speed by selecting `Evenly spaced points` to calculate table indices. This algorithm uses only the first two breakpoints of a set to determine the offset and spacing of the remaining points.
- For unevenly spaced breakpoint sets, follow these guidelines:
 - If input values for `u` do not vary significantly between time steps, selecting `Linear search` with **Begin index search using previous index result** produces the best performance.
 - If input values for `u` jump more than one or two table intervals per time step, selecting `Binary search` produces the best performance.

A suboptimal choice of index search method can lead to slow performance of models that rely heavily on lookup tables.

Note The generated code stores only the first breakpoint, the spacing, and the number of breakpoints when:

- The breakpoint data is not tunable.
 - The index search method is `Evenly spaced points`.
-

Output selection

Specify the signals (`index`, `fraction`, or both) that the block outputs. If you want the block to output the index and interval fraction, you can have it output either individual signals or a bus signal that includes both the index and fraction signals.

- `Index only` outputs just the index, without the fraction. Typical applications for this option include:

- Feeding a Direct Lookup Table (n-D) block, with no interpolation on the interval
- Feeding selection ports of a subtable selection for an Interpolation Using Prelookup block
- Performing nonlinear quantizations
- Index and fraction outputs the index and fraction as individual signals.
- Index and fraction as bus outputs a bus signal that includes the index and fraction signals. Using a bus for these signals:
 - Simplifies the model by tying these two related signals together.
 - Creates a testpoint `DpResult` structure for the AUTOSAR 4.0 library.
 - For the AUTOSAR 4.0 library, avoids the creation of extra copies during code generation when the Prelookup and Interpolation Using Prelookup blocks are in separate models.

Note Selecting Index and fraction as bus displays the **Output** parameter in the **Data Types** pane and sets the **Output** parameter to `Inherit: auto`. Change this default value to specify a user-defined bus object. For details about defining the bus object, see the **Output** parameter description.

Begin index search using previous index result

Select this check box when you want the block to start its search using the index found at the previous time step. For input values of `u` that change slowly with respect to the interval size, enabling this option can improve performance. Otherwise, the linear search and binary search methods can take longer, especially for large breakpoint sets.

Extrapolation method

Specify how to handle out-of-range values for the block input `u`. Options include:

- Clip

Block Input	Block Outputs
Less than the first breakpoint	<ul style="list-style-type: none"> • Index of the first breakpoint (for example, 0) • Interval fraction of 0

Block Input	Block Outputs
Greater than the last breakpoint	<ul style="list-style-type: none"> • Index of the next-to-last breakpoint • Interval fraction of 1

Suppose the range is [1 2 3] and you select this option. If u is 0.5, the index is 0 and the interval fraction is 0. If u is 3.5, the index is 1 and the interval fraction is 1.

- Linear

Block Input	Block Outputs
Less than the first breakpoint	<ul style="list-style-type: none"> • Index of the first breakpoint (for example, 0) • Interval fraction that represents the linear distance from u to the first breakpoint
Greater than the last breakpoint	<ul style="list-style-type: none"> • Index of the next-to-last breakpoint • Interval fraction that represents the linear distance from the next-to-last breakpoint to u

Suppose the range is [1 2 3] and you select this option. If u is 0.5, the index is 0 and the interval fraction is -0.5. If u is 3.5, the index is 1 and the interval fraction is 1.5.

Note The Prelookup block supports linear extrapolation only when all of these conditions apply:

- The input u , breakpoint data, and fraction output use floating-point data types.
 - The index uses a built-in integer data type.
-

Use last breakpoint for input at or above upper limit

Specify how to index input values of u that are greater than or equal to the last breakpoint. The index value is zero-based. When input equals the last breakpoint, block outputs differ as follows:

Check Box	Block Outputs
Selected	<ul style="list-style-type: none">• Index of the last element in the breakpoint data set• Interval fraction of 0
Cleared	<ul style="list-style-type: none">• Index of the next-to-last breakpoint• Interval fraction of 1

This check box is visible only when:

- **Output only the index** is cleared.
- **Extrapolation method** is `Clip`.

However, when **Output only the index** is selected and **Extrapolation method** is `Clip`, the block behaves as if this check box is selected even though it is invisible.

Tip When you select **Use last breakpoint for input at or above upper limit** for a Prelookup block, you must also select **Valid index input may reach last index** for the Interpolation Using Prelookup block to which it connects. This action allows the blocks to use the same indexing convention when accessing the last elements of their breakpoint and table data sets.

Diagnostic for out-of-range input

Specify whether to produce a warning or error when the input `u` is out of range.

Options include:

- `None` — Simulink software takes no action.
- `Warning` — Simulink software displays a warning and continues the simulation.
- `Error` — Simulink software terminates the simulation and displays an error.

Remove protection against out-of-range input in generated code

Specify whether or not to include code that checks for out-of-range breakpoint inputs.

Check Box	Result	When to Use
Selected	Generated code does not include conditional statements to check for out-of-range breakpoint inputs. When the input k or f is out-of-range, it may cause undefined behavior for generated code and simulations using accelerator mode.	For code efficiency
Cleared	Generated code includes conditional statements to check for out-of-range breakpoint inputs.	For safety-critical applications

If your input is not out-of-range, you can select the **Remove protection against out-of-range index in generated code** check box for code efficiency. By default, this check box is cleared. For safety-critical applications, do not select this check box. If you want to select the **Remove protection against out-of-range index in generated code** check box, first check that your model inputs are in range. For example:

- 1 Clear the **Remove protection against out-of-range index in generated code** check box.
- 2 Set the **Diagnostic for out-of-range input** parameter to `Error`.
- 3 Simulate the model in normal mode.
- 4 If there are out-of-range errors, fix them to be in range and run the simulation again.
- 5 When the simulation no longer generates out-of-range input errors, select the **Remove protection against out-of-range index in generated code** check box.

Note When you select the **Remove protection against out-of-range index in generated code** check box and the input `k` or `f` is out-of-range, the behavior is undefined for generated code and simulations using accelerator mode.

Depending on your application, you can run the following Model Advisor checks to verify the usage of this check box:

- **By Product > Embedded Coder > Identify lookup table blocks that generate expensive out-of-range checking code**
- **By Product > Simulink Check > Modeling Standards > DO-178C/DO-331 Checks > Check usage of lookup table blocks**

For more information about the Model Advisor, see “Run Model Checks” in the Simulink documentation.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Data Types Tab

Note The parameters for breakpoint attributes (data type, minimum, and maximum) are not available when you set **Source** to `Input port`. In this case, the block inherits all breakpoint attributes from the `bp` input port.

Breakpoint > Data Type

Specify the breakpoint data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the breakpoint data type.

See “Specify Data Types Using Data Type Assistant” in the *Simulink User’s Guide* for more information.

Tip Specify a breakpoint data type different from the data type of input *u* for these cases:

- Lower memory requirement for storing breakpoint data that uses a smaller type than the input signal *u*
- Sharing of prescaled breakpoint data between two Prelookup blocks with different data types for input *u*
- Sharing of custom storage breakpoint data in the generated code for blocks with different data types for input *u*

Breakpoint > Minimum

Specify the minimum value that the breakpoint data can have. The default value is [] (unspecified).

Breakpoint > Maximum

Specify the maximum value that the breakpoint data can have. The default value is [] (unspecified).

Index > Data Type

Note This parameter displays only when you set **Output selection** on the **Main** tab to `Index and fraction` or `Index only`.

Specify a data type that can index all elements in the breakpoint data set. You can:

- Select a built-in integer data type from the list.
- Specify an integer data type using a fixed-point representation.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the index data type.

See “Specify Data Types Using Data Type Assistant” in the *Simulink User's Guide* for more information.

Fraction > Data Type

Note This parameter displays only when you set **Output selection** on the **Main** tab to `Index and fraction`

Specify the data type of the interval fraction. You can:

- Select a built-in data type from the list.
- Specify data type inheritance through an internal rule.
- Specify a fixed-point data type using the [Slope Bias] or binary-point-only scaling representation.
 - If you use the [Slope Bias] representation, the scaling must be trivial — that is, the slope is 1 and the bias is 0.
 - If you use the binary-point-only representation, the fixed power-of-two exponent must be less than or equal to zero.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the fraction data type.

See “Specify Data Types Using Data Type Assistant” in the *Simulink User's Guide* for more information.

Output

Note This parameter displays only when you set **Output selection** on the **Main** tab to `Index and fraction as bus`.

Output a nonvirtual bus. Use the `Bus: <object name>` template to specify a bus object. Replace `<object name>` with the name of a bus object that contains the index and fraction signals.

- The bus object must contain two elements. The first element corresponds to the index signal and the second to the fraction signal.
- The index and fraction bus element signals cannot be bus signals.
- The data type and the complexity of the bus elements must meet the same constraints that apply to the index and fraction signals if you set **Output selection** to `Index and fraction`.

To create the bus object with the index and fraction bus elements, use MATLAB code similar to this, customizing the bus object name and the names and data types of the bus elements.

```
% Bus object: kfBus
elems(1) = Simulink.BusElement;
elems(1).Name = 'Index';
elems(1).DataType = 'int8';

elems(2) = Simulink.BusElement;
elems(2).Name = 'Fraction';
elems(2).DataType = 'double';

kfBus = Simulink.Bus;
kfBus.Elements = elems;
clear elems;
```

Alternatively, you can use the Bus Editor to create or modify the bus object to use with the Prelookup block.

If you feed the bus output signal from this block to an Interpolation Using Prelookup block, select the **Require index and fraction as bus** check box in that block.

Note Using the Fixed-Point Tool data type override option overrides bus objects with new bus objects that replace fixed-point data types with floating-point data types.

Overridden bus objects used with the Prelookup block can cause an error because the block does not accept floating-point data types for the first element in the bus.

If you encounter this issue, using the **Fix** button redefines the original bus object to protect it from being overridden. For example, suppose you define the first element of the bus object to be an `int32`.

```
myBus.Elements(1).DataType  
  
int32
```

Clicking the **Fix** button redefines the first bus element to be:

```
myBus.Elements(1).DataType = 'fixdt('int32','DataTypeOverride','Off')
```

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

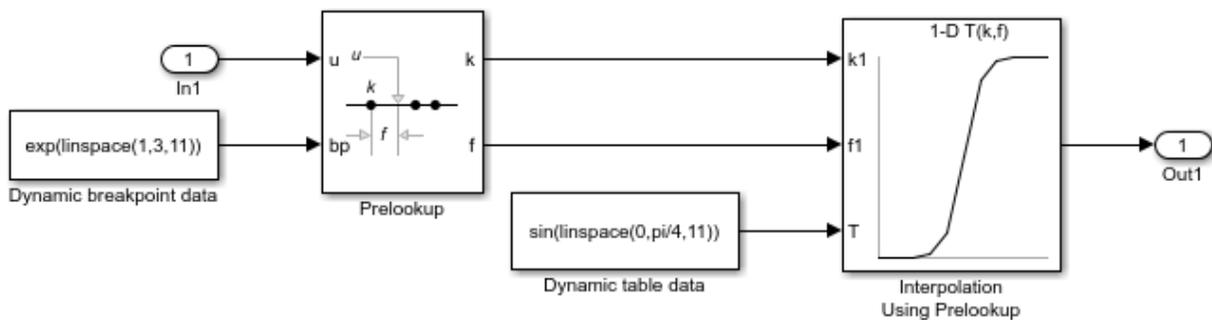
Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer) in the Simulink Fixed Pointo documentation.

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

Examples

Prelookup with External Breakpoint Specification

In the following model, a Constant block feeds the breakpoint dataset to the `bp` input port of the Prelookup block.



The Prelookup block inherits the following breakpoint attributes from the `bp` input port:

Breakpoint Attribute	Value
Minimum	-Inf
Maximum	Inf
Data type	single

Similarly, a Constant block feeds the table data values to the `T` input port of the Interpolation Using Prelookup block, which inherits the following table attributes:

Table Attribute	Value
Minimum	-Inf
Maximum	Inf
Data type	single

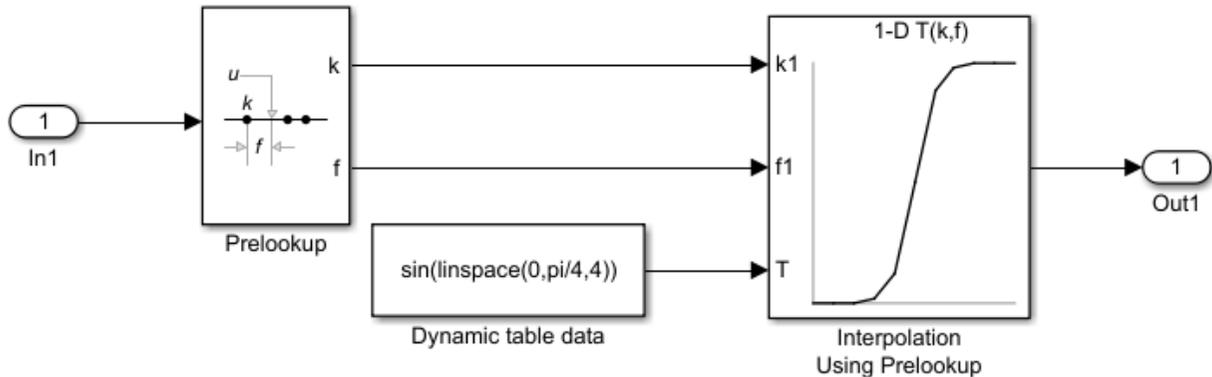
Simulink uses double-precision, floating-point data to perform the computations in this model. However, the model stores the breakpoint and table data as single-precision, floating-point data. Using a lower-precision data type to store breakpoint and table data reduces the memory requirement.

Prelookup with Even Spaced Breakpoint Specification

In this model, you specify evenly spaced breakpoint data to the Prelookup block.

The **Breakpoint Specification** parameter is set to `Even Spacing`. The parameters **First point**, **Spacing**, and **Number of points** are set to 25, 12, and 4 respectively. Specifying these parameters creates four evenly spaced breakpoints: [25, 37, 49, 61].

Specifying even spacing is an alternative way to specify breakpoints that are evenly spaced. You can also set **Breakpoint Specification** to `Explicit` values and set **Value** to `[25:12:61]`.



Simulink uses double-precision, floating-point data to perform the computations in this model. However, the model stores the breakpoints and table data as double.

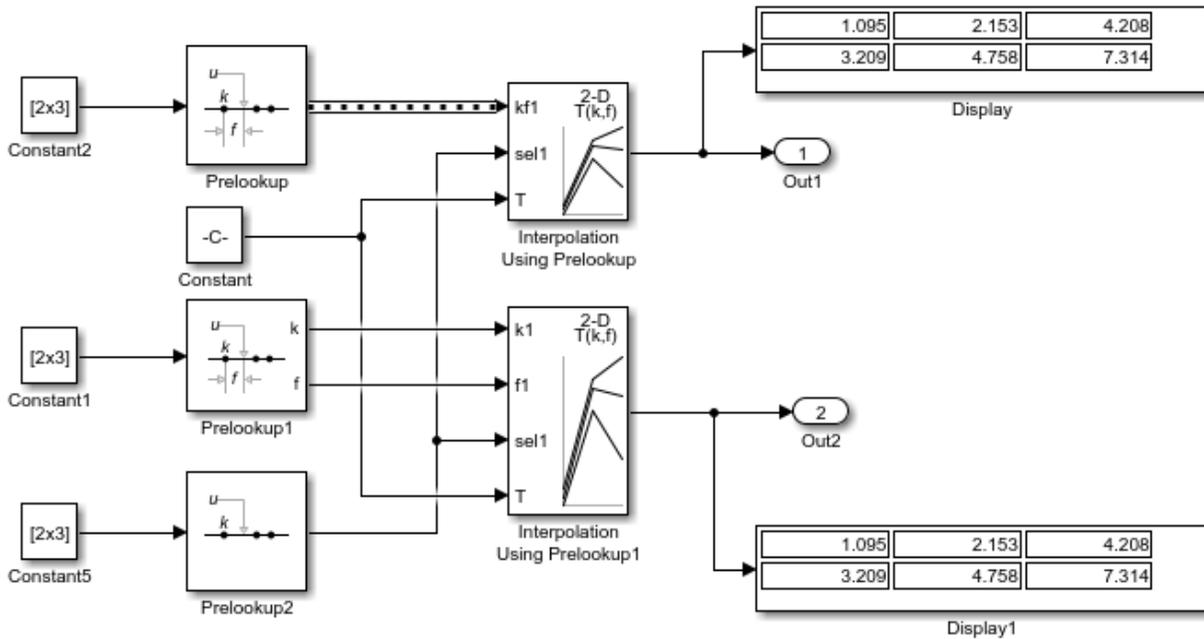
For other examples, see “Prelookup and Interpolation Blocks” in the Simulink documentation.

Use a Bus for Index and Fraction

You can use a bus object to specify the index (k) and fraction (f) in the Prelookup block and then use that bus as input to an Interpolation Using Prelookup block. This example shows how to use a bus signal and how to get the same results without using a bus object.

Open the model and simulate it.

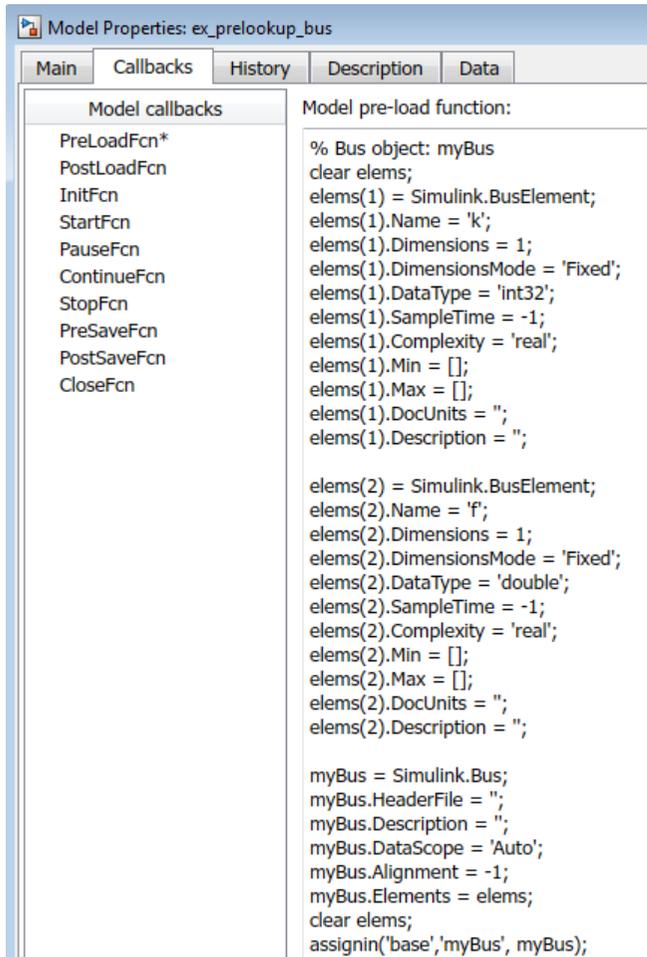
```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
    'examples', 'ex_prelookup_bus.slx')))
sim('ex_prelookup_bus')
```



At the top of the model, open the **Prelookup** block parameters dialog box. In the **Main** tab, note that **Output selection** is set to **Index and fraction as bus**.

In the **Data Types** tab, note that **Output** is set to **Bus: myBus**.

In the Simulink Editor, select **File > Model Properties** and open the **Callbacks** tab. The code defines the bus object **myBus**, which specifies the index as the first bus element and the fraction as the second element.



In the Interpolation Using Prelookup block parameters dialog box, in the **Main** tab, note that **Require index and fraction as bus** check box is selected. That option configures the block to use the bus output from the Prelookup block.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
------------	----------------------------------------------

Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

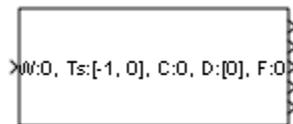
See Also

Interpolation Using Prelookup

Introduced in R2006b

Probe

Output signal attributes, including width, dimensionality, sample time, and complex signal flag



Library

Signal Attributes

Description

The Probe block outputs selected information about the signal on its input. The block can output the input signal's width, dimensionality, sample time, and a flag indicating whether the input is a complex-valued signal. The block has one input port. The number of output ports depends on the information that you select for probing, that is, signal dimensionality, sample time, and/or complex signal flag. Each probed value is output as a separate signal on a separate output port. The block accepts real or complex-valued signals of any built-in data type. It outputs signals of type `double`. During simulation, the block icon displays the probed data.

Data Type Support

The Probe block accepts signals of the following data types:

- Floating point
- Built-in integer

- Fixed point
- Boolean
- Enumerated (input only)
- Bus object

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

You can use an array of buses as an input signal to a Probe block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Parameters

Probe width

Select to output the width, or number of elements, of the probed signal.

Probe sample time

Select to output the sample time of the probed signal. The output is a two-element vector that specifies the period and offset of the sample time, respectively. See “Specify Sample Time” for more information.

Detect complex signal

Select to output 1 if the probed signal is complex; otherwise, 0.

Probe signal dimensions

Select to output the dimensions of the probed signal.

Note The Probe block ignores the **Data type override** setting of the Fixed-Point Tool.

Data type for width

Select the output data type for the width information.

Data type for sample time

Select the output data type for the sample time information.

Data type for signal complexity

Select the output data type for the complexity information.

Data type for signal dimensions

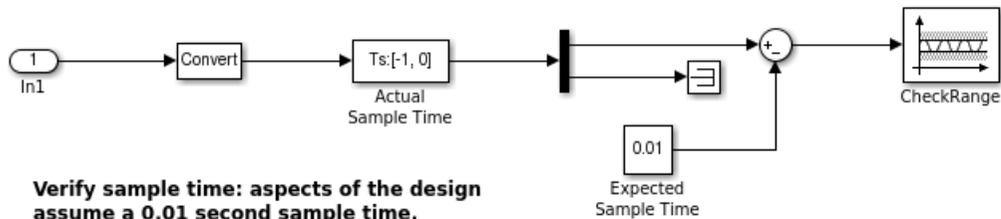
Select the output data type for the dimensions information.

Note For **Data type for width**, **Data type for sample time**, and **Data type for signal dimensions**, the Boolean data type is not supported. Furthermore, if you select Same as input in any of these drop-down lists, and the block's input signal data type is Boolean, when you simulate your model, you see an error.

Examples

The `sldemo_fuelsys` model shows how you can use the Probe block.

In the `fuel_rate_control/validate_sample_time` subsystem, the Probe block determines the sample time of the input signal to verify that it matches the assumed value of the design:



Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No

Code Generation	Yes
-----------------	-----

Introduced before R2006a

Product

Multiply and divide scalars and nonscalars or multiply and invert matrices



Library

Math Operations

Description

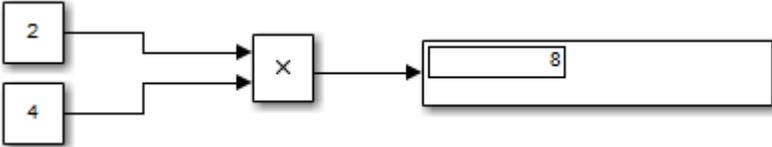
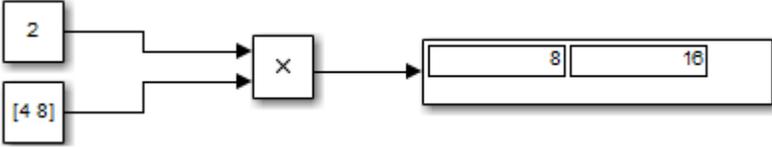
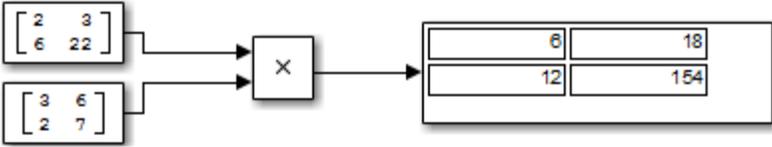
The Divide and Product of Elements blocks are variants of the Product block.

- For information on the Divide block, see [Divide](#).
- For information on the Product of Elements block, see [Product of Elements](#).

The Product block outputs the result of multiplying two inputs: two scalars, a scalar and a nonscalar, or two nonscalars that have the same dimensions. The default parameter values that specify this behavior are:

- **Multiplication:** `Element-wise (.*)`
- **Number of inputs:** 2

The following table shows the output of the Product block for example inputs using default block parameter values.

Inputs and Behavior	Example
<p>Scalar X Scalar</p> <p>Output the product of the two inputs.</p>	
<p>Scalar X Nonscalar</p> <p>Output a nonscalar having the same dimensions as the input nonscalar. Each element of the output nonscalar is the product of the input scalar and the corresponding element of the input nonscalar.</p>	
<p>Nonscalar X Nonscalar</p> <p>Output a nonscalar having the same dimensions as the inputs. Each element of the output is the product of corresponding elements of the inputs.</p>	

The Product block (or the Divide block or Product of Elements block, if appropriately configured) can:

- Numerically multiply and divide any number of scalar, vector, or matrix inputs
- Perform matrix multiplication and division on any number of matrix inputs

The Product block performs scalar or matrix multiplication, depending on the value of the **Multiplication** parameter. The block accepts one or more inputs, depending on the **Number of inputs** parameter. The **Number of inputs** parameter also specifies the operation to perform on each input.

The Product block can input any combination of scalars, vectors, and matrices for which the operation to perform has a mathematically defined result. The block performs the specified operations on the inputs, then outputs the result.

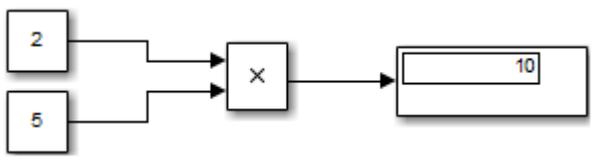
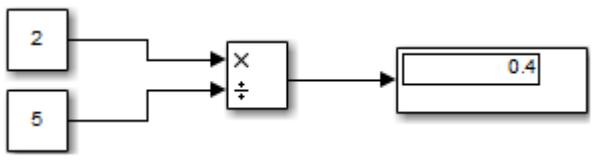
The Product block has two modes: *Element-wise mode*, which processes nonscalar inputs element by element, and *Matrix mode*, which processes nonscalar inputs as matrices. The next two sections describe these two modes.

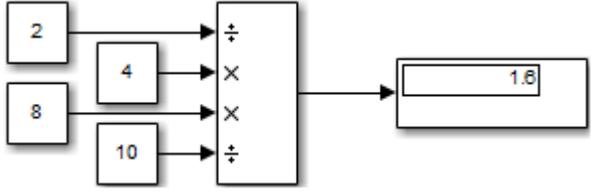
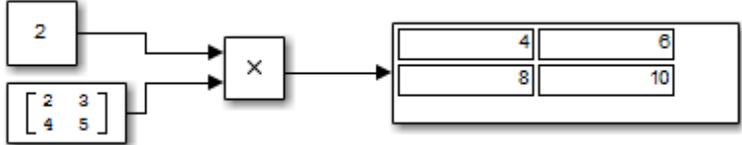
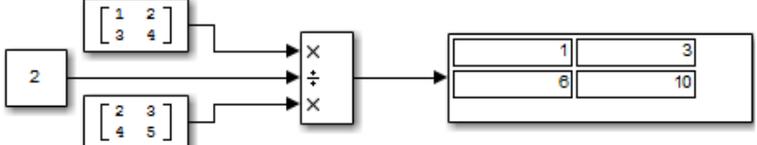
Element-wise Mode

When the value of the **Multiplication** parameter is `Element-wise (.*)`, the Product block is in *Element-wise mode*, in which it operates on the individual numeric elements of any nonscalar inputs. The MATLAB equivalent is the `.*` operator. In element-wise mode, the Product block can perform a variety of multiplication, division, and arithmetic inversion operations.

The value of the **Number of inputs** parameter controls both how many inputs exist and whether each is multiplied or divided to form the output. When the Product block in Element-wise mode has only one input, it is functionally equivalent to a Product of Elements block. When the block has multiple inputs, any nonscalar inputs must have identical dimensions, and the block outputs a nonscalar with those dimensions. To calculate the output, the block first expands any scalar input to a nonscalar that has the same dimensions as the nonscalar inputs.

This table shows the output of the Product block for example inputs, using the indicated values for the **Number of inputs** parameter.

Parameter Values	Examples
Number of inputs: 2	 <p>The diagram shows two input boxes containing the numbers 2 and 5. Arrows from both boxes point to a central block containing a multiplication symbol (×). An arrow from this block points to an output box containing the number 10.</p>
Number of inputs: */	 <p>The diagram shows two input boxes containing the numbers 2 and 5. Arrows from both boxes point to a central block containing a multiplication symbol (×) above a division symbol (÷). An arrow from this block points to an output box containing the number 0.4.</p>

Parameter Values	Examples
Number of inputs: /**/	
Number of inputs: **	
Number of inputs: */*	

Matrix Mode

When the value of the **Multiplication** parameter is `Matrix(*)`, the Product block is in *Matrix mode*, in which it processes nonscalar inputs as matrices. The MATLAB equivalent is the `*` operator. In Matrix mode, the Product block can invert a single square matrix, or multiply and divide any number of matrices that have dimensions for which the result is mathematically defined.

The value of the **Number of inputs** parameter controls both how many inputs exist and whether each input matrix is multiplied or divided to form the output. The syntax of **Number of inputs** is the same as in Element-wise mode. The difference between the modes is in the type of multiplication and division that occur.

Expected Differences Between Simulation and Code Generation

For element-wise operations on complex floating-point inputs, simulation and code generation results might differ in near-overflow cases. Although **complex numbers** is selected and **non-finite numbers** is not selected on the **Code Generation > Interface**

pane of the Configuration Parameters dialog box, the code generator does not emit special case code for intermediate overflows. This method improves the efficiency of embedded operations for the general case that does not include extreme values. If the inputs might include extreme values, please manage these cases explicitly.

The generated code might not produce the exact same pattern of NaN and inf values as simulation when these values are mathematically meaningless. For example, if the simulation output contains a NaN, output from the generated code also contains a NaN, but not necessarily in the same place.

Data Type Support

The Product block accepts real or complex signals of any numeric data type that Simulink supports, including fixed-point data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

The Product block does not support numeric division for complex signals with `boolean` or fixed-point data types. For other types, the block accepts complex signals as divisors only when the input and output signals all specify the same built-in data type. In this case, however, the block ignores its specified rounding mode.

The Product block accepts multidimensional signals when operating in Element-wise mode, but not when operating in Matrix mode. See “Signal Dimensions”, “Element-wise Mode” on page 1-1176, and “Matrix Mode” on page 1-1177 for more information.

Parameters

Number of inputs

Control two properties of the block:

- The number of input ports on the block
- Whether each input is multiplied or divided into the output

Default: 2 for Product block, */ for Divide block, and * for Product of Elements block

- **1 or * or /**

Has one input. In element-wise mode, processes the input as described for the Product of Elements block. In matrix mode, if the parameter value is 1 or *, the block outputs the input value. If the value is /, the input must be a square matrix (including a scalar as a degenerate case) and the block outputs the matrix inverse. See “Element-wise Mode” on page 1-1176 and “Matrix Mode” on page 1-1177 for more information.

- **Integer value > 1**

Has the number of inputs given by the integer value. The inputs are multiplied together in element-wise mode or matrix mode, as specified by the **Multiplication** parameter. See “Element-wise Mode” on page 1-1176 and “Matrix Mode” on page 1-1177 for more information.

- **Unquoted string of two or more * and / characters**

Has the number of inputs given by the length of the character vector. Each input that corresponds to a * character is multiplied into the output. Each input that corresponds to a / character is divided into the output. The operations occur in element-wise mode or matrix mode, as specified by the **Multiplication** parameter. See “Element-wise Mode” on page 1-1176 and “Matrix Mode” on page 1-1177 for more information.

Setting **Number of inputs** to * and selecting `Element-wise(.*)` for **Multiplication** enables the **Multiply over** parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Multiplication

Specify whether the Product block operates in Element-wise mode or Matrix mode.

Default: `Element-wise(.*)`

`Element-wise(.*)`

Operate in Element-wise mode.

`Matrix(*)`

Operate in Matrix mode.

Selecting `Element-wise (.*)` and setting **Number of inputs** to `*` enable the following parameter:

- **Multiply over**

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Multiply over

Affect multiplication on matrix input.

Default: All dimensions

All dimensions

Output a scalar that is product of all elements of the matrix, or the product of their inverses, depending on the value of **Number of inputs**.

Specified dimension

Output a vector, the composition of which depends on the value of the **Dimension** parameter.

- Enable this parameter by selecting `Element-wise (.*)` for **Multiplication** and setting **Number of inputs** to `*` or `1` or `/`.
- Setting this parameter to `Specified dimension` enables the **Dimension** parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Dimension

Affect multiplication on matrix input.

Default: 1

Minimum: 1

Maximum: 2

1

Output a vector that contains an element for each column of the input matrix.

2

Output a vector that contains an element for each row of the input matrix.

Each element of the output vector contains the product of all elements in the corresponding column or row of the input matrix, or the product of the inverses of those elements, depending on the value of **Number of inputs**:

- 1 or *

Multiply the values of the column or row elements

- /

Multiply the inverses of the column or row elements

Enable this parameter by selecting `Specified dimension` for **Multiply over**.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Require all inputs to have the same data type

Require that all inputs have the same data type.

Default: Off

On

Require that all inputs have the same data type.

Off

Do not require that all inputs have the same data type.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Output minimum

Lower value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMin

Type: character vector

Value: '[]'

Default: '[]'

Output maximum

Upper value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMax

Type: character vector

Value: ' [] '

Default: ' [] '

Output data type

Specify the output data type.

Default: Inherit: Inherit via internal rule

Inherit: Inherit via internal rule

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target

hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit: Same as first input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit: Inherit via back propagation`

Use data type of the driving block.

`Inherit: Same as first input`

Use data type of the first input signal.

`double`

Output data type is `double`.

`single`

Output data type is `single`.

`int8`

Output data type is `int8`.

`uint8`

Output data type is `uint8`.

`int16`

Output data type is `int16`.

`uint16`

Output data type is `uint16`.

`int32`

Output data type is `int32`.

```
uint32
```

Output data type is uint32.

```
fixdt(1,16,0)
```

Output data type is fixed point fixdt(1,16,0).

```
fixdt(1,16,2^0,0)
```

Output data type is fixed point fixdt(1,16,2^0,0).

<data type expression>

Use a data type object, for example, `Simulink.NumericType`.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Control Signal Data Types”.

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule (default)`
- `Inherit via back propagation`

- Same as first input

Built in

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

Fixed point

Fixed-point data types.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

Inherit

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

Off

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: Signed

Signed

Specify the fixed-point data as signed.

Unsigned

Specify the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting Slope and bias enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > Binary point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > Slope and bias enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: `LockScale`

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: `Floor`

`Ceiling`

Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

Floor

Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

Nearest

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Rounding” (Fixed-Point Designer).

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off

On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Parameter: SaturateOnIntegerOverflow

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

- Divide
- Dot Product
- Product of Elements

Introduced before R2006a

Product of Elements

Copy or invert one scalar input, or collapse one nonscalar input



Library

Math Operations

Description

The Divide and Product blocks are variants of the Product of Elements block.

- For information on the Divide block, see [Divide](#).
- For information on the Product block, see [Product](#).

The Product of Elements block inputs one scalar, vector, or matrix. You can use the block to:

- Copy a scalar input unchanged
- Invert a scalar input (divide 1 by it)
- Collapse a vector or matrix to a scalar by multiplying together all elements or taking successive inverses of the elements
- Collapse a matrix to a vector by multiplying together the elements of each row or column or taking successive inverses of the elements of each row or column

The Product of Elements block is functionally a Product block that has two preset parameter values:

- **Multiplication:** `Element-wise (.*)`

- **Number of inputs:** *

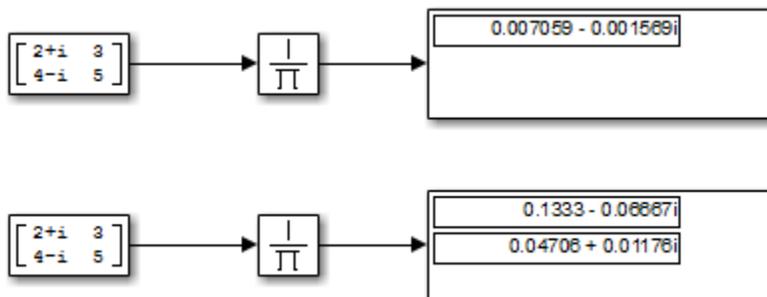
Setting non-default values for either of those parameters can change a Product of Elements block to be functionally equivalent to a Product block or a Divide block. See the documentation of those two blocks for more information.

Algorithm

The Product of Elements block uses the following algorithms to perform element-wise operations on inputs of floating-point, built-in integer, and fixed-point types:

Input	Element-Wise Operation	Algorithm
Real scalar, u	Multiplication	$y = u$
	Division	$y = 1/u$
Real vector or matrix, with elements $u_1, u_2, u_3, \dots, u_N$	Multiplication	$y = u_1 * u_2 * u_3 * \dots * u_N$
	Division	$y = (((1/u_1) / u_2) / u_3) \dots / u_N$
Complex scalar, u	Multiplication	$y = u$
	Division	$y = 1/u$
Complex vector or matrix, with elements $u_1, u_2, u_3, \dots, u_N$	Multiplication	$y = u_1 * u_2 * u_3 * \dots * u_N$
	Division	$y = (((1/u_1) / u_2) / u_3) \dots / u_N$

If the specified dimension for element-wise multiplication or division is a row or column of a matrix, the algorithm applies to that row or column. For example, consider the following model:



The top Product of Elements block collapses the matrix input to a scalar by taking successive inverses of the four elements:

- $y = (((1/2+i)/3)/4-i)/5)$

The bottom Product of Elements block collapses the matrix input to a vector by taking successive inverses along the second dimension:

- $y(1) = ((1/2+i)/3)$
- $y(2) = ((1/4-i)/5)$

Parameters

Number of inputs

Control two properties of the block:

- The number of input ports on the block
- Whether each input is multiplied or divided into the output

Default: *

- **1 or ***
 - Copies a scalar input unchanged
 - Collapses a vector input to a scalar by multiplying all elements together
 - Collapses a matrix input to a scalar or vector by multiplying elements together based on the **Multiply over** parameter

For more information, see “Algorithm” on page 1-1195.

- **/**
 - Outputs the arithmetic inverse of a scalar input
 - Collapses a vector input to a scalar by taking successive inverses of the elements
 - Collapses a matrix input to a scalar or vector by taking successive inverses of elements based on the **Multiply over** parameter

For more information, see “Algorithm” on page 1-1195.

- **Integer value > 1**

Has the number of inputs given by the integer value. The block becomes a product block and the input are multiplied together in element-wise mode or matrix mode, as specified by the **Multiplication** parameter. See “Element-wise Mode” on page 1-1176 and “Matrix Mode” on page 1-1177 in the Product documentation for more information.

- **Unquoted string of two or more * and / characters**

Has the number of inputs given by the length of the character vector. The block becomes a product or divide block and multiplies each input that corresponds to a * character into the output. Each input that corresponds to a / character is divided into the output. The operations occur in Element-wise mode or Matrix mode, as specified by the **Multiplication** parameter. See “Element-wise Mode” on page 1-1176 and “Matrix Mode” on page 1-1177 in the Product block reference page for more information.

Setting **Number of inputs** to * and selecting `Element-wise(.*)` for **Multiplication** enables the following **Multiply over** parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Multiplication

Specify whether the Product block operates in Element-wise mode or Matrix mode.

Default: `Element-wise(.*)`

`Element-wise(.*)`

Operate in Element-wise mode.

`Matrix(*)`

Operate in Matrix mode.

Selecting `Element-wise(.*)` and setting **Number of inputs** to * enable the following parameter:

- **Multiply over**

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Multiply over

Affect multiplication on matrix input.

Default: All dimensions

All dimensions

Output a scalar that is product of all elements of the matrix, or the product of their inverses, depending on the value of **Number of inputs**.

Specified dimension

Output a vector, the composition of which depends on the value of the **Dimension** parameter.

- Enable this parameter by selecting `Element-wise (.*)` for **Multiplication** and setting **Number of inputs** to `*` or `1` or `/`.
- Setting this parameter to `Specified dimension` enables the **Dimension** parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Dimension

Affect multiplication on matrix input.

Default: 1

Minimum: 1

Maximum: 2

1

Output a vector that contains an element for each column of the input matrix.

2

Output a vector that contains an element for each row of the input matrix.

Each element of the output vector contains the product of all elements in the corresponding column or row of the input matrix, or the product of the inverses of those elements, depending on the value of **Number of inputs**:

- 1 or *

Multiply the values of the column or row elements

- /

Multiply the inverses of the column or row elements

Enable this parameter by selecting `Specified dimension` for **Multiply over**.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Require all inputs to have the same data type

Require that all inputs have the same data type.

Default: Off

On

Require that all inputs have the same data type.

Off

Do not require that all inputs have the same data type.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Output minimum

Lower value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMin

Type: character vector

Value: '[]'

Default: '[]'

Output maximum

Upper value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMax

Type: character vector

Value: ' [] '

Default: ' [] '

Output data type

Specify the output data type.

Default: Inherit: Inherit via internal rule

Inherit: Inherit via internal rule

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet

your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit: Same as first input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

`Inherit: Inherit via back propagation`

Use data type of the driving block.

`Inherit: Same as first input`

Use data type of the first input signal.

`double`

Output data type is double.

`single`

Output data type is single.

`int8`

Output data type is int8.

`uint8`

Output data type is uint8.

`int16`

Output data type is int16.

`uint16`

Output data type is uint16.

`int32`

Output data type is int32.

`uint32`

Output data type is uint32.

```
fixdt(1,16,0)
```

Output data type is fixed point `fixdt(1,16,0)`.

```
fixdt(1,16,2^0,0)
```

Output data type is fixed point `fixdt(1,16,2^0,0)`.

<data type expression>

Use a data type object, for example, `Simulink.NumericType`.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Control Signal Data Types”.

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule (default)`
- `Inherit via back propagation`
- `Same as first input`

Built in

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

Fixed point

Fixed-point data types.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

Off

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: Signed

Signed

Specify the fixed-point data as signed.

Unsigned

Specify the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: `Best precision`

`Binary point`

Specify binary point location.

`Slope and bias`

Enter slope and bias.

`Best precision`

Specify best-precision values.

Selecting **Mode** > `Fixed point` enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting `Slope and bias` enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope and bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope and bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off



On

Locks the output data type setting for this block.



Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: LockScale

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: Floor

Ceiling

Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

Floor

Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

Nearest

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Rounding” (Fixed-Point Designer).

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

 Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Parameter: SaturateOnIntegerOverflow

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Examples

This table shows the output of the Product of Elements block for example inputs using default block parameter values, except as shown in the table.

Parameter Values	Examples
Multiplication: Element-wise (.*) Number of inputs: *	
Multiplication: Element-wise (.*) Number of inputs: /	
Multiplication: Element-wise (.*) Number of inputs: *	

Parameter Values	Examples
Multiplication: Element-wise (.*) Number of inputs: * Multiply over: All dimensions	<p>The diagram shows a 2x2 matrix $\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$ on the left, an arrow pointing to a square block containing a product symbol \prod, and another arrow pointing to a rectangular box containing the value 120.</p>
Multiplication: Element-wise (.*) Number of inputs: * Multiply over: Specified dimension Dimension: 1	<p>The diagram shows a 2x2 matrix $\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$ on the left, an arrow pointing to a square block containing a product symbol \prod, and another arrow pointing to a rectangular box divided into two sections, with the value 8 on the left and 15 on the right.</p>
Multiplication: Element-wise (.*) Number of inputs: / Multiply over: Specified dimension Dimension: 2	<p>The diagram shows a 2x2 matrix $\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$ on the left, an arrow pointing to a square block containing a division symbol $\frac{1}{\prod}$, and another arrow pointing to a rectangular box divided into two sections, with the value 0.1667 on the top and 0.05 on the bottom.</p>

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	Yes

Introduced before R2006a

Pulse Generator

Generate square wave pulses at regular intervals

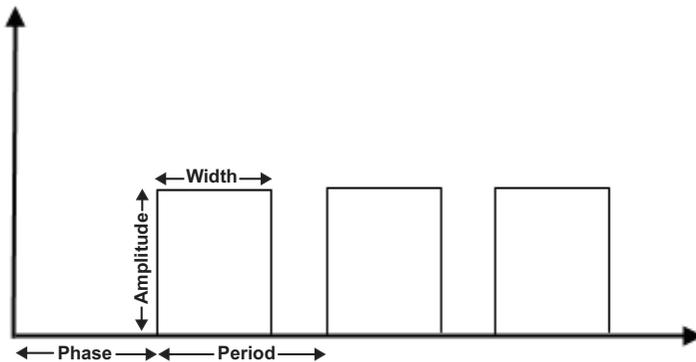


Library

Sources

Description

The Pulse Generator block generates square wave pulses at regular intervals. The block waveform parameters, **Amplitude**, **Pulse Width**, **Period**, and **Phase delay**, determine the shape of the output waveform. The following diagram shows how each parameter affects the waveform.



The Pulse Generator can emit scalar, vector, or matrix signals of any real data type. To cause the block to emit a scalar signal, use scalars to specify the waveform parameters. To cause the block to emit a vector or matrix signal, use vectors or matrices, respectively, to specify the waveform parameters. Each element of the waveform parameters affects

the corresponding element of the output signal. For example, the first element of a vector amplitude parameter determines the amplitude of the first element of a vector output pulse. All the waveform parameters must have the same dimensions after scalar expansion. The data type of the output is the same as the data type of the **Amplitude** parameter.

This block output can be generated in time-based or sample-based modes, determined by the **Pulse type** parameter.

Time-Based Mode

In time-based mode, Simulink computes the block output only at times when the output actually changes. This approach results in fewer computations for the block output over the simulation time period. Activate this mode by setting the **Pulse type** parameter to `Time based`.

The block does not support a time-based configuration that results in a constant output signal. Simulink returns an error if the parameters **Pulse Width** and **Period** satisfy either of these conditions:

$$Period * \frac{PulseWidth}{100} = 0$$

$$Period * \frac{PulseWidth}{100} = Period$$

Depending on the pulse waveform characteristics, the intervals between changes in the block output can vary. For this reason, a time-based Pulse Generator block has a variable sample time. The sample time color of such blocks is brown (see “View Sample Time Information” for more information).

Simulink cannot use a fixed-step solver to compute the output of a time-based pulse generator. If you specify a fixed-step solver for models that contain time-based pulse generators, Simulink computes a fixed sample time for the time-based pulse generators. Then the time-based pulse generators simulate as sample based.

If you use a fixed-step solver and the **Pulse type** is `Time based`, choose the step size such that the period, phase delay, and pulse width (in seconds) are integer multiples of the solver step size. For example, suppose that the period is 4 seconds, the pulse width is 75% (that is, 3 s), and the phase delay is 1 s. In this case, the computed sample time is 1 s. Therefore, choose a fixed-step size of 1 or a number that divides 1 exactly (e.g., 0.25).

You can guarantee this by setting the fixed-step solver step size to `auto` on the **Solver** pane of the Configuration Parameters dialog box.

Sample-Based Mode

In sample-based mode, the block computes its outputs at fixed intervals that you specify. Activate this mode by setting the **Pulse type** parameter to `Sample based`.

An important difference between the time-based and sample-based modes is that in time-based mode, the block output is based on simulation time, and in sample-based mode, the block output depends only on the simulation start, regardless of elapsed simulation time. For more information, see the example “Difference Between Time-Based and Sample-Based Pulse Generation Modes” on page 1-1216.

This block supports reset semantics in sample-based mode. For example, if a Pulse Generator is in a resettable subsystem that hits a reset trigger, the block output resets to its initial condition.

Data Type Support

The Pulse Generator block outputs real signals of any numeric data type that Simulink supports, including fixed-point data types. The data type of the output signal is the same as that of the **Amplitude** parameter.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Pulse type

The pulse type for this block: `Time based` or `Sample based`. The default is `Time based`.

Time (t)

Specifies whether to use simulation time or an external signal as the source of values for the output pulse's time variable. If you specify an external source, the block displays an input port for connecting the source. The output pulse differs as follows:

- `Use simulation time`: The block generates an output pulse where the time variable equals the simulation time.
- `Use external signal`: The block generates an output pulse where the time variable equals the value from the input port, which can differ from the simulation time.

Amplitude

The pulse amplitude. The default is 1.

Period

The pulse period specified in seconds if the pulse type is time-based or as number of sample times if the pulse type is sample-based. The default is 10 seconds.

Pulse Width

The duty cycle specified as the percentage of the pulse period that the signal is on if time-based or as number of sample times if sample-based. The default is 5 percent.

Phase delay

The delay before the pulse is generated specified in seconds if the pulse type is time-based or as number of sample times if the pulse type is sample-based. The default is 0 seconds.

Sample time

The length of the sample time for this block in seconds. This parameter appears only if the block's pulse type is sample-based. See “Specify Sample Time” in the Simulink User's Guide for more information.

Interpret vector parameters as 1-D

If you select this check box and the other parameters are one-row or one-column matrices, after scalar expansion, the block outputs a 1-D signal (vector). Otherwise the output dimensionality is the same as that of the other parameters. See “Determining the Output Dimensions of Source Blocks” in the “Working with Signals” chapter of the Simulink User's Guide.

Examples

The following Simulink examples show how to use the Pulse Generator block:

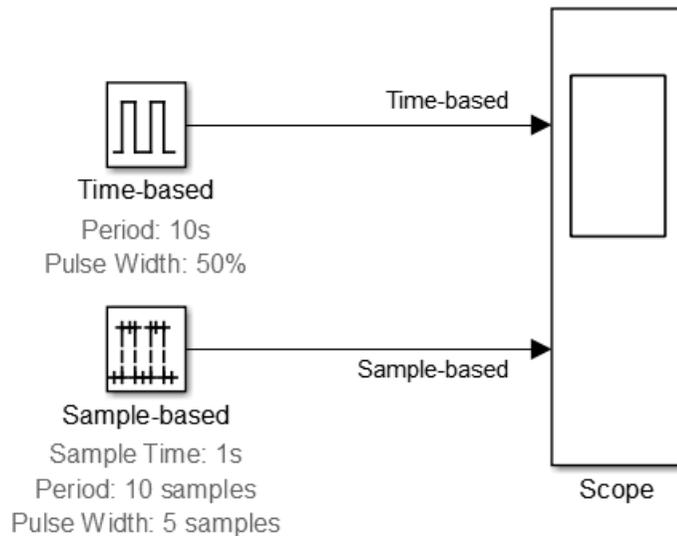
- `sldemo_auto_climatecontrol`

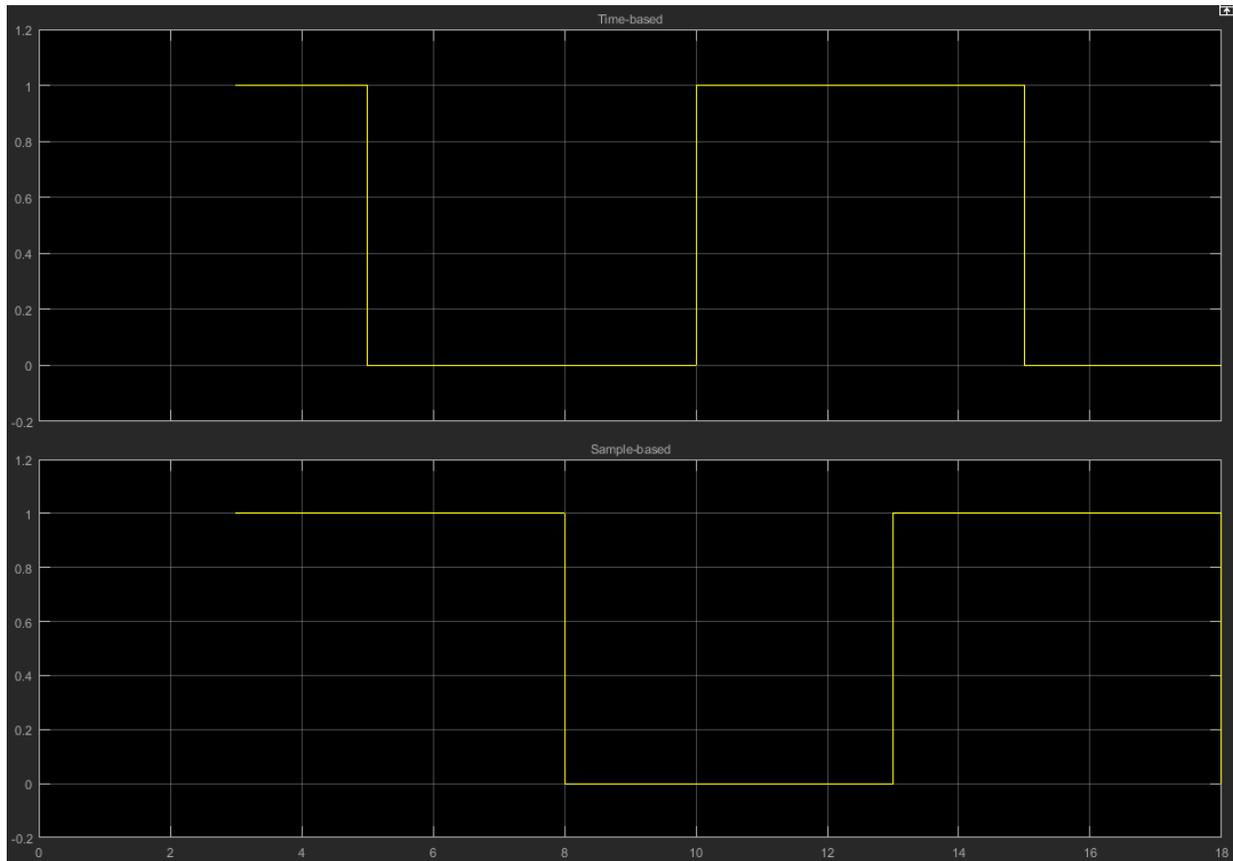
- `sldemo_boiler`

Difference Between Time-Based and Sample-Based Pulse Generation Modes

This example shows the difference in the behavior of the Pulse Generator block in time-based and sample-based modes.

Consider a model with two Pulse Generator blocks. In one block, the **Pulse type** parameter is set to `Time based`. In the other block, it is set to `Sample based`. Both blocks are set up to output a Boolean pulse of 10 seconds: 5 seconds `on` followed by 5 seconds `off`. The simulation runs for 15 seconds from a start time of 3 seconds to a stop time of 18 seconds, specified in the Model Configuration Parameters dialog box. The figure shows the block diagram for this model and the simulation output in the Scope block.



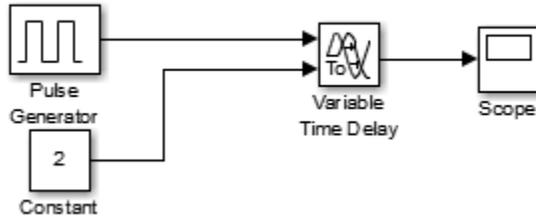


You can see that simulation output starts at 3 seconds, as expected. Notice that the time-based Pulse Generator produces a logical `on` for only 2 seconds, after which its output changes to `off` at $t = 5$ seconds. This is because this block starts computing its output from $t = 0$ seconds, even though it doesn't output it until the simulation starts at $t = 3$ seconds. The time-based block depends on simulation time for its output.

The sample-based block outputs a pulse of 5 seconds `on` followed by 5 seconds `off`. In this case, the block's output does not depend on simulation time, and starts only when the simulation starts.

Tune Phase Delay Parameter During Simulation

You cannot tune the value of the **Phase delay** parameter during simulation. As a workaround, you can add a Constant block and a Variable Time Delay block.



In the Pulse Generator block, set the value of the **Phase delay** parameter to 0. Use the Constant block to specify the delay time (in seconds).

To tune the delay time during simulation, change the value stored in the Constant block.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Push Button

Set value of parameter when button is pressed

Library: Simulink / Dashboard



Description

When you press the Push Button block during a simulation, the value of the connected block parameter changes to a specified value. Use the Push Button block with other Dashboard blocks to create an interactive dashboard to control your model.

Connection

Dashboard block controls connect to variables and block parameters. To connect a Dashboard block to a variable or parameter, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting blocks of interest in your model.
- 3 Mark the option button next to the parameter or variable you want to adjust during simulation.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.

- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined using the variable `engine(1)` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

Parameters

Connection — Select a block parameter to connect

signal connection options

Select the block parameter to control using the **Connection** table. To open the dialog box, double-click the block. Then, populate the **Connection** table by selecting a block in your model. Select the variable parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram with **Ctrl+D**.

Button Text — Text displayed on button

'Button' (default) | character vector

The text displayed on the Push Button block in your model.

On Value — Value assigned to parameter when button is pressed

1 (default) | scalar

The value assigned to the connected block parameter when the button is pressed.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

See Also

Rocker Switch | Slider Switch | Toggle Switch

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2015a

Quantizer

Discretize input at specified interval



Library

Discontinuities

Description

The Quantizer block passes its input signal through a stair-step function so that many neighboring points on the input axis are mapped to one point on the output axis. The effect is to quantize a smooth signal into a stair-step output. The output is computed using the round-to-nearest method, which produces an output that is symmetric about zero.

$$y = q * \text{round}(u/q)$$

where y is the output, u the input, and q the **Quantization interval** parameter.

Data Type Support

The Quantizer block accepts and outputs real or complex signals of type `single` or `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Quantization interval

The interval around which the output is quantized. Permissible output values for the Quantizer block are $n \cdot q$, where n is an integer and q the **Quantization interval**.

The default is 0.5.

Treat as gain when linearizing

Simulink software by default treats the Quantizer block as unity gain when linearizing. This setting corresponds to the large-signal linearization case. If you clear this check box, the linearization routines assume the small-signal case and set the gain to zero.

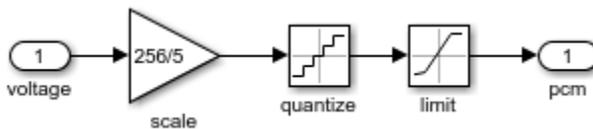
Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Examples

The `sldemo_boiler` model shows how you can use the Quantizer block.

The Quantizer block appears in the Boiler Plant model/digital thermometer/ADC subsystem.



The ADC subsystem digitizes the input analog voltage by:

- Multiplying the analog voltage by 256/5 with the Gain block
- Rounding the value to integer floor with the Quantizer block
- Limiting the output to a maximum of 255 (the largest unsigned 8-bit integer value) with the Saturation block

For more information, see “Explore the Fixed-Point "Bang-Bang Control" Model” (Stateflow) in the Stateflow documentation.

Characteristics

Data Types	Double Single
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Quarter Gauge

Display input value on quadrant scale

Library: Simulink / Dashboard



Description

The Quarter Gauge block displays the connected signal on a quadrant scale during simulation. You can use the Quarter Gauge block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. The Quarter Gauge block provides an indication of the instantaneous value of the connected signal throughout simulation. You can modify the range of the Quarter Gauge block to fit your data. You can also customize the appearance of the Quarter Gauge block to provide more information about your signal. For example, you can color-code in-specification and out-of-specification ranges.

Connection

Dashboard blocks do not use ports to connect to signals. To connect a signal to a Dashboard block, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting signals of interest in your model.
- 3 Mark the option button next to the signal you want to display.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- If you turn off logging for a signal connected to a Dashboard block, the model stops sending data from that signal to the block. To view the signal again, reconnect the signal.
- External mode simulation does not support visualization Dashboard blocks.

Parameters

Connection — Select a signal to connect and display

signal connection options

Signal connection options. Populate the **Connection** table by selecting signals of interest in your model. Mark the check box next to the signal you want to display. Click **Apply** to connect the signal.

Minimum — Minimum tick mark value

0 (default) | scalar

A finite, real, double, scalar value specifying the minimum tick mark value for the scale. The minimum must be less than the value entered for the maximum.

Maximum — Maximum tick mark value

100 (default) | scalar

A finite, real, double, scalar value specifying the maximum tick mark value for the scale. The maximum must be greater than the value entered for the minimum.

Tick Interval — Interval between major tick marks

auto (default) | scalar

A finite, real, positive, integer, scalar value specifying the interval of major tick marks on the scale. When set to `auto`, the block automatically adjusts the tick interval based on the minimum and maximum values.

Scale Colors — Color indications on Gauge scale

colors for scale ranges

Color specifications for ranges on the scale. Press the **+** button to add a color. For each color added, specify the minimum and maximum values of the range where you want to display that color.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

Model Examples

See Also

Gauge | Half Gauge | Linear Gauge

Topics

“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015a

Radio Button

Select parameter value

Library: Simulink / Dashboard



Description

The Radio Button block lets you change the value of the connected parameter during simulation. You can specify a list of values and labels and then select the value for the parameter from that list. Use the Radio Button block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model.

Double-clicking the Radio Button block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

Connection

Dashboard block controls connect to variables and block parameters. To connect a Dashboard block to a variable or parameter, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting blocks of interest in your model.
- 3 Mark the option button next to the parameter or variable you want to adjust during simulation.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined using the variable `engine(1)` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

Parameters

Connection — Select a block parameter to connect

signal connection options

Select the block parameter to control using the **Connection** table. To open the dialog box, double-click the block. Then, populate the **Connection** table by selecting a block in your model. Select the variable parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram with **Ctrl+D**.

States

value — Values for selected option

0/1/2 (default) | scalar

Values assigned to the connected parameter when you select the option with the corresponding **Label**. Click the **+** button to add options.

Label — Option labels

'Label1'/'Label2'/'Label3' (default) | character vector

Label for each option. You can use the **Label** to display the value the connected parameter takes when the switch is positioned at the bottom, or you can enter a text label.

Example: `Gain = 1`

Group Name — Radio Button group name

RadioButtonGroup (default) | character array

Name for the group of values displayed on the Radio Button block. Unlike the **Block Name** and **Label**, the **Group Name** always shows on the Radio Button block.

Example: `Input Amplitude`

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

See Also

Check Box | Combo Box | Rotary Switch

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2017b

Ramp

Generate constantly increasing or decreasing signal



Library

Sources

Description

The Ramp block generates a signal that starts at a specified time and value and changes by a specified rate. The block's **Slope**, **Start time**, and **Initial output** parameters determine the characteristics of the output signal. All must have the same dimensions after scalar expansion.

Data Type Support

The Ramp block outputs signals of type `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Slope

Specify the rate of change of the generated signal. The default is 1.

Start time

Specify the time at which the block begins generating the signal. The default is 0.

Initial output

Specify the initial value of the output signal. The default is 0.

Interpret vector parameters as 1-D

If you select this option and the other parameters are one-row or one-column matrices, after scalar expansion, the block outputs a 1-D signal (vector). Otherwise, the output dimensionality is the same as that of the other parameters. See “Determining the Output Dimensions of Source Blocks” in the Simulink documentation.

Examples

The following Simulink examples show how to use the Ramp block:

- `sldemo_VariableTransportDelay_pipe`

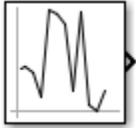
Characteristics

Data Types	Double
Sample Time	Inherited from downstream block
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes
Code Generation	Yes

Introduced before R2006a

Random Number

Generate normally distributed random numbers



Library

Sources

Description

The Random Number block generates normally distributed random numbers. To generate uniformly distributed random numbers, use the Uniform Random Number block.

You can generate a repeatable sequence using any Random Number block with the same nonnegative seed and parameters. The seed resets to the specified value each time a simulation starts. By default, the block produces a sequence that has a mean of 0 and a variance of 1. To generate a vector of random numbers with the same mean and variance, specify the **Seed** parameter as a vector.

Avoid integrating a random signal, because solvers must integrate relatively smooth signals. Instead, use the Band-Limited White Noise block.

The numeric parameters of this block must have the same dimensions after scalar expansion. If you select the **Interpret vector parameters as 1-D** check box and the numeric parameters are row or column vectors after scalar expansion, the block outputs a 1-D signal. If you clear the **Interpret vector parameters as 1-D** check box, the block outputs a signal of the same dimensionality as the parameters.

Data Type Support

The Random Number block outputs a real signal of type `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Mean

Specify the mean of the random numbers. The default is 0.

Variance

Specify the variance of the random numbers. The default is 1.

Seed

Specify the starting seed for the random number generator. The default is 0.

The seed must be 0 or a positive integer. Output is repeatable for a given seed.

Sample time

Specify the time interval between samples. The default is 0.1, which matches the default sample time of the Band-Limited White Noise block. See “Specify Sample Time” in the Simulink documentation for more information.

Interpret vector parameters as 1-D

If you select this check box and the other parameters are row or column vectors after scalar expansion, the block outputs a 1-D signal. Otherwise, the block outputs a signal of the same dimensionality as the other parameters. For more information, see “Determining the Output Dimensions of Source Blocks” in the Simulink documentation.

Characteristics

Data Types	Double
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No

Code Generation	Yes
-----------------	-----

The generator algorithm is identical to the one used in MATLAB Version 4.0 by the `rand` and `randn` functions. For details on the `mcg16807` algorithm, see “Choosing a Random Number Generator” (MATLAB) in the MATLAB documentation.

To use other algorithms supported by MATLAB in a Simulink model, generate a stream of random numbers in MATLAB, and store the output as a `.mat` file. Use this `.mat` file as the random number input for your simulation. For more information, see “Creating and Controlling a Random Number Stream” (MATLAB). To create multiple independent streams using MATLAB, see “Multiple Streams” (MATLAB).

Note Using multiple seeds to generate multiple parallel independent streams for a generator algorithm is not recommended for the `mcg16807` algorithm. Instead, use the method described above.

See Also

Uniform Random Number

Introduced before R2006a

Rate Limiter

Limit rate of change of signal



Library

Discontinuities

Description

The Rate Limiter block limits the first derivative of the signal passing through it. The output changes no faster than the specified limit. The derivative is calculated using this equation:

$$rate = \frac{u(i) - y(i-1)}{t(i) - t(i-1)}$$

$u(i)$ and $t(i)$ are the current block input and time, and $y(i-1)$ and $t(i-1)$ are the output and time at the previous step. The output is determined by comparing $rate$ to the **Rising slew rate** and **Falling slew rate** parameters:

- If $rate$ is greater than the **Rising slew rate** parameter (R), the output is calculated as
$$y(i) = \Delta t \cdot R + y(i-1).$$
- If $rate$ is less than the **Falling slew rate** parameter (F), the output is calculated as
$$y(i) = \Delta t \cdot F + y(i-1).$$
- If $rate$ is between the bounds of R and F , the change in output is equal to the change in input:
$$y(i) = u(i)$$

When the block is running in continuous mode (for example, **Sample time mode** is inherited and **Sample time** of the driving block is zero), the **Initial condition** is ignored. The block output at $t = 0$ is equal to the initial input:

$$y(0) = u(0)$$

When the block is running in discrete mode (for example, **Sample time mode** is inherited and **Sample time** of the driving block is nonzero), the **Initial condition** is preserved:

$$y(-1) = I_c$$

where I_c is the initial condition. The block output at $t = 0$ is calculated as if $rate$ is outside the bounds of R and F . For $t = 0$, $rate$ is calculated as follows:

$$rate = \frac{u(0) - y(-1)}{sampletime}$$

Note You cannot use a Rate Limiter block inside a Triggered Subsystem. Use the Rate Limiter Dynamic block instead.

Data Type Support

The Rate Limiter block accepts and outputs signals of any numeric data type that Simulink supports, except `boolean`. The Rate Limiter block supports fixed-point data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Rising slew rate

Specify the limit of the derivative of an increasing input signal. This parameter is tunable for fixed-point inputs.

Falling slew rate

Specify the limit of the derivative of a decreasing input signal. This parameter is tunable for fixed-point inputs.

Sample time mode

Specify the sample time mode, `continuous` or `inherited` from the driving block.

Initial condition

Set the initial output of the simulation. Simulink software does not allow you to set the initial condition of this block to `inf` or `NaN`.

Treat as gain when linearizing

Linearization commands in Simulink software treat this block as a gain in state space. Select this check box to cause the linearization commands to treat the gain as 1; otherwise, the commands treat the gain as 0.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Continuous or inherited (specified in the Sample time mode parameter)
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Rate Limiter Dynamic

Introduced before R2006a

Rate Limiter Dynamic

Limit rising and falling rates of signal



Library

Discontinuities

Description

The Rate Limiter Dynamic block limits the rising and falling rates of the signal.

- The external signal `up` sets the upper limit on the rising rate of the signal.
- The external signal `lo` sets the lower limit on the falling rate of the signal.

Follow these guidelines when using the Rate Limiter Dynamic block:

- Ensure that the data types of `up` and `lo` are the same as the data type of the input signal `u`.

When the lower limit uses a signed type and the input signal uses an unsigned type, the output signal keeps increasing regardless of the input and the limits.

- Use a fixed-step solver to simulate models that contain this block.

Because the Rate Limiter Dynamic block does not support continuous sample time, simulation with a variable-step solver causes an error.

Data Type Support

The Rate Limiter Dynamic block accepts input signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

The data type of the output signal Y matches the data type of the input signal u .

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

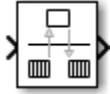
See Also

Rate Limiter

Introduced before R2006a

Rate Transition

Handle transfer of data between blocks operating at different rates



Library

Signal Attributes

Description

Transition Handling Options

The Rate Transition block transfers data from the output of a block operating at one rate to the input of a block operating at a different rate. Use the block parameters to trade data integrity and deterministic transfer for faster response or lower memory requirements. To learn about data integrity and deterministic data transfer, see “Data Transfer Problems” (Simulink Coder) in the Simulink Coder documentation.

Transition Handling Options	Block Parameter Settings
<ul style="list-style-type: none"> • Data integrity • Deterministic data transfer • Maximum latency 	Select: <ul style="list-style-type: none"> • Ensure data integrity during data transfer • Ensure deterministic data transfer

Transition Handling Options	Block Parameter Settings
<ul style="list-style-type: none"> • Data integrity • Nondeterministic data transfer • Minimum latency • Higher memory requirements 	Select: <ul style="list-style-type: none"> • Ensure data integrity during data transfer Clear: <ul style="list-style-type: none"> • Ensure deterministic data transfer
<ul style="list-style-type: none"> • Potential loss of data integrity • Nondeterministic data transfer • Minimum latency • Lower memory requirements 	Clear: <ul style="list-style-type: none"> • Ensure data integrity during data transfer • Ensure deterministic data transfer

Dependencies

The behavior of the Rate Transition block depends on:

- Sample times of the ports to which the block connects (see “Effects of Synchronous Sample Times” on page 1-1243 and “Effects of Asynchronous Sample Times” on page 1-1245)
- Priorities of the tasks for the source and destination sample times (see “Sample time properties” in the Simulink documentation)
- Whether the model specifies a fixed- or variable-step solver (see “Solvers” in the Simulink documentation)

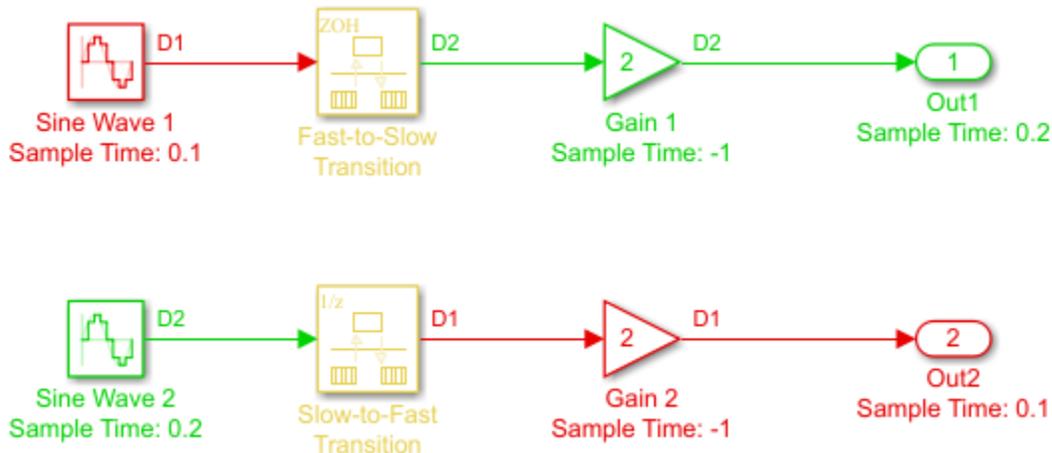
Block Labels

When you update your diagram, a label appears on the Rate Transition block to indicate simulation behavior.

Label	Block Behavior
ZOH	Acts as a zero-order hold
1/z	Acts as a unit delay

Label	Block Behavior
Buf	Copies input to output under semaphore control
Db_buf	Copies input to output using double buffers
Copy	Unprotected copy of input to output
NoOp	Does nothing
Mixed	Expands to multiple blocks with different behaviors

The block behavior label shows the method that ensures safe transfer of data between tasks operating at different rates. You can use the sample-time colors feature (see “View Sample Time Information” in the Simulink documentation) to display the relative rates that the block bridges. Consider, for example, the following model:



Sample-time colors and the block behavior label show that the Rate Transition block at the top of the diagram acts as a zero-order hold in a fast-to-slow transition and the bottom Rate Transition block acts as a unit delay in a slow-to-fast transition.

For more information, see “Handle Rate Transitions” (Simulink Coder) in the Simulink Coder documentation.

Effects of Synchronous Sample Times

The following table summarizes how each label appears if the sample times of the input and output ports ($inTs$ and $outTs$) are periodic, or synchronous.

Block Settings		Block Label		
Rate Transition	Conditions for Rate Transition Block	With Data Integrity and Determinism	With Only Data Integrity	Without Data Integrity or Determinism
(Equal)	$inTsOffset < outTsOffset$	None (error)	Buf	Copy or NoOp (see note that follows the table)
	$inTsOffset = outTsOffset$	Copy or NoOp (see note that follows the table)	Copy or NoOp (see note that follows the table)	
	$inTsOffset > outTsOffset$	None (error)	Db_buf	
(Fast to slow)	$inTs = outTs / N$ $inTsOffset, outTsOffset = 0$	ZOH	Buf	
	$inTs = outTs / N$ $inTsOffset \leq outTsOffset$	None (error)		
	$inTs = outTs / N$ $inTsOffset > outTsOffset$	None (error)	Db_buf	
	$inTs \neq outTs / N$	None (error)		
(Slow to fast)	$inTs = outTs * N$ $inTsOffset, outTsOffset = 0$	$1/z$	Db_buf	
	$inTs = outTs * N$ $inTsOffset \leq outTsOffset$	None (error)		
	$inTs = outTs * N$ $inTsOffset > outTsOffset$	None (error)		
	$inTs \neq outTs * N$	None (error)		

Block Settings		Block Label		
Rate Transition	Conditions for Rate Transition Block	With Data Integrity and Determinism	With Only Data Integrity	Without Data Integrity or Determinism
KEY				
<ul style="list-style-type: none"> • $inTs$, $outTs$: Sample times of input and output ports, respectively • $inTsOffset$, $outTsOffset$: Sample time offsets of input and output ports, respectively • N: Integer value > 1 				

When you select the **Block reduction** parameter on the Configuration Parameters dialog box, `Copy` reduces to `NoOp`. No code generation occurs for a Rate Transition block with a `NoOp` label. To prevent a block from being reduced when block reduction is on, add a test point to the block output (see “Test Points” in the Simulink documentation).

Effects of Asynchronous Sample Times

The following table summarizes how each label appears if the sample time of the input or output port ($inTs$ or $outTs$) is not periodic, or asynchronous.

Block Settings	Block Label		
	With Data Integrity and Determinism	With Only Data Integrity	Without Data Integrity or Determinism
$inTs = outTs$	<code>Copy</code>	<code>Copy</code>	<code>Copy</code>
$inTs \neq outTs$	<code>None (error)</code>	<code>Db_buf</code>	
KEY			
<ul style="list-style-type: none"> • $inTs$, $outTs$: Sample times of input and output ports, respectively 			

Data Type Support

The Rate Transition block accepts most signals that Simulink supports, including fixed-point and enumerated data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

However, do not use the Rate Transition block with frame-based signals. For rate transitions with such signals, use one of these blocks from the DSP System Toolbox instead:

- Buffer
- Unbuffer
- CIC Decimation
- CIC Interpolation
- FIR Decimation
- FIR Interpolation
- Downsample
- Upsample

Parameters

Ensure data integrity during data transfer

Selecting this check box results in generated code that ensures data integrity when the block transfers data. If you select this check box and the transfer is nondeterministic (see **Ensure deterministic data transfer** below), depending on the priority of input rate and output rate, the generated code uses a proper algorithm using single or multiple buffers to protect data integrity during data transfer.

Otherwise, the Rate Transition block is either reduced or generates code using a copy operation to effect the data transfer. This unprotected mode consumes less memory. But the copy operation is also interruptible, which can lead to loss of data integrity during data transfers. Select this check box if you want the generated code to operate with maximum responsiveness (i.e., nondeterministically) and data integrity. For more information, see “Rate Transition Block Options” (Simulink Coder) in the Simulink Coder documentation.

Ensure deterministic data transfer (maximum delay)

Selecting this check box results in generated code that transfers data at the sample rate of the slower block, that is, deterministically. If you do not select this check box, data transfers occur as soon as new data is available from the source block and the receiving block is ready to receive the data. You avoid transfer delays, thus ensuring that the system operates with maximum responsiveness. However, transfers can occur unpredictably, which is undesirable in some applications. For more

information, see “Rate Transition Block Options” (Simulink Coder) in the Simulink Coder documentation.

Initial conditions

This parameter applies only to slow-to-fast transitions. It specifies the initial output of the Rate Transition block at the beginning of a transition when there is no output from the slow block connected to the input of the Rate Transition block. Simulink does not allow the initial output of this block to be `Inf` or `NaN`.

Output port sample time options

Specifies a mode for setting the output port sample time. The options are:

- `Specify` — Allows you to use the **Output port sample time** parameter to specify the output rate to which the Rate Transition block converts its input rate.
- `Inherit` — Specifies that the Rate Transition block inherits an output rate from the block to which the output port is connected.
- `Multiple of input port sample time` — Allows you to use the **Sample time multiple (>0)** parameter to specify the Rate Transition block output rate as a multiple of its input rate.

If you specify `Inherit` and all blocks connected to the output port also inherit sample time, the fastest sample time in the model applies.

Output port sample time

This parameter is visible when you set **Output port sample time options** to `Specify`. Enter a value that specifies the output rate to which the block converts its input rate. The default value (`-1`) specifies that the Rate Transition block inherits the output rate from the block to which the output port is connected. See “Specify Sample Time” in the Simulink documentation for information on how to specify the output rate.

Sample time multiple (>0)

This parameter is visible when you set **Output port sample time options** to `Multiple of input port sample time`. Enter a positive value that specifies the output rate as a multiple of the input port sample time. The default value (`1`) specifies that the output rate is the same as the input rate. A value of `0.5` specifies that the output rate is half of the input rate, while a value of `2` specifies that the output rate is twice the input rate.

Bus Support

The Rate Transition block is a bus-capable block. The input can be a virtual or nonvirtual bus signal, with the restriction that **Initial conditions** must be zero, a nonzero scalar, or a finite numeric structure. For information about specifying an initial condition structure, see “Specify Initial Conditions for Bus Signals”.

All signals in a nonvirtual bus input to a Rate Transition block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “Specify Bus Signal Sample Times” and “Bus-Capable Blocks” in the Simulink documentation for more information.

You can use an array of buses as an input signal to a Rate Transition block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	This block supports discrete-to-discrete transitions
Direct Feedthrough	No, for slow-to-fast transitions for which you select the Ensure data integrity during data transfer check box. Yes, otherwise.
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Real-Imag to Complex

Convert real and/or imaginary inputs to complex signal



Library

Math Operations

Description

The Real-Imag to Complex block converts real and/or imaginary inputs to a complex-valued output signal.

The inputs can both be arrays (vectors or matrices) of equal dimensions, or one input can be an array and the other a scalar. If the block has an array input, the output is a complex array of the same dimensions. The elements of the real input map to the real parts of the corresponding complex output elements. The imaginary input similarly maps to the imaginary parts of the complex output signals. If one input is a scalar, it maps to the corresponding component (real or imaginary) of all the complex output signals.

Data Type Support

The block accepts input signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

Complex fixed-point signals must have trivial slope and zero bias. For more information about support for fixed-point data types, see “Scaling” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Parameters

Input

Specify the kind of input: a real input, an imaginary input, or both.

Real (Imag) part

This parameter appears only when you set **Input** to `Real` or `Imag`. If the input is a real-part signal, this parameter specifies the constant imaginary part of the output signal. If the input is the imaginary part, this parameter specifies the constant real part of the output signal. The title of this parameter changes to reflect its usage.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Complex to Real-Imag

Introduced before R2006a

Relational Operator

Perform specified relational operation on inputs



Library

Logic and Bit Operations

Description

Two-Input Mode

By default, the Relational Operator block compares two inputs using the **Relational operator** parameter that you specify. The first input corresponds to the top input port and the second input to the bottom input port. (See “Port Location After Rotating or Flipping” in the Simulink documentation for a description of the port order for various block orientations.)

You can specify one of the following operations in two-input mode:

Operation	Description
==	TRUE if the first input is equal to the second input
~=	TRUE if the first input is not equal to the second input
<	TRUE if the first input is less than the second input
<=	TRUE if the first input is less than or equal to the second input
>=	TRUE if the first input is greater than or equal to the second input
>	TRUE if the first input is greater than the second input

You can specify inputs as scalars, arrays, or a combination of a scalar and an array.

For...	The output is...
Scalar inputs	A scalar
Array inputs	An array of the same dimensions, where each element is the result of an element-by-element comparison of the input arrays
Mixed scalar and array inputs	An array, where each element is the result of a comparison between the scalar and the corresponding array element

The input with the smaller positive range is converted to the data type of the other input offline using round-to-nearest and saturation. This conversion occurs before the comparison.

You can specify the output data type using the **Output data type** parameter. The output equals 1 for TRUE and 0 for FALSE.

Tip Select an output data type that represents zero exactly. Data types that satisfy this condition include signed and unsigned integers and any floating-point data type.

One-Input Mode

When you select one of the following operations for **Relational operator**, the block switches to one-input mode.

Operation	Description
isInf	TRUE if the input is Inf
isNaN	TRUE if the input is NaN
isFinite	TRUE if the input is finite

For an input that is not floating point, the block produces the following output.

Data Type	Operation	Block Output
<ul style="list-style-type: none"> • Fixed point • Boolean • Built-in integer 	isInf	FALSE
	isNaN	FALSE
	isFinite	TRUE

Rules for Data Type Propagation

The following rules apply for data type propagation when your block has one or more input ports with unspecified data types.

When the block is in...	And...	The block uses...
Two-input mode	Both input ports have unspecified data types	<code>double</code> as the default data type for both inputs
	One input port has an unspecified data type	The data type from the specified input port as the default data type of the other port
One-input mode	The input port has an unspecified data type	<code>double</code> as the default data type for the input

Data Type Support

The Relational Operator block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types. For two-input mode, one input can be real and the other complex when the operator is `==` or `~=`. Complex inputs work only for `==`, `~=`, `isInf`, `isNaN`, and `isFinite`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Relational operator

Specify the operation for comparing two inputs or determining the signal type of one input.

Default: <=

==

TRUE if the first input is equal to the second input

~=

TRUE if the first input is not equal to the second input

<

TRUE if the first input is less than the second input

<=

TRUE if the first input is less than or equal to the second input

>=

TRUE if the first input is greater than or equal to the second input

>

TRUE if the first input is greater than the second input

isInf

TRUE if the input is Inf

isNaN

TRUE if the input is NaN

isFinite

TRUE if the input is finite

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Default: On

On

Enable zero-crossing detection.

Off

Do not enable zero-crossing detection.

Parameter: ZeroCross

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Require all inputs to have the same data type

Require that all inputs have the same data type.

Default: Off

On

Require that all inputs have the same data type.

Off

Do not require that all inputs have the same data type.

This check box is not available when you select `isInf`, `isNaN`, or `isFinite` for **Relational operator**, because the block is in one-input mode.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Output data type

Specify the output data type.

Default: `boolean`

Inherit: Logical (see Configuration Parameters: Optimization)

Uses the **Implement logic signals as Boolean data** configuration parameter (see “Implement logic signals as Boolean data (vs. double)”) to specify the output data type.

Note This option supports models created before the `boolean` option was available. Use one of the other options, preferably `boolean`, for new models.

`boolean`

Specifies output data type is `boolean`.

`fixdt(1,16)`

Specifies output data type is `fixdt(1,16)`.

`<data type expression>`

Uses the name of a data type object, for example, `Simulink.NumericType`.

Tip To enter a built-in data type (`double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`), enclose the expression in single quotes. For example, enter `'double'` instead of `double`.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Mode

Select the category of data to specify.

Default: Built in

Inherit

Specifies inheritance rules for data types. Selecting `Inherit` enables `Logical` (see `Configuration Parameters: Optimization`).

Built in

Specifies built-in data types. Selecting `Built in` enables `boolean`.

Fixed point

Specifies fixed-point data types.

Expression

Specifies expressions that evaluate to data types.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: Inherit

Inherit

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

Off

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For

example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data as signed.

`Unsigned`

Specify the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: `16`

Minimum: `0`

Maximum: `32`

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Integer

Integer

Specify integer. This setting has the same result as specifying a binary point location and setting fraction length to 0.

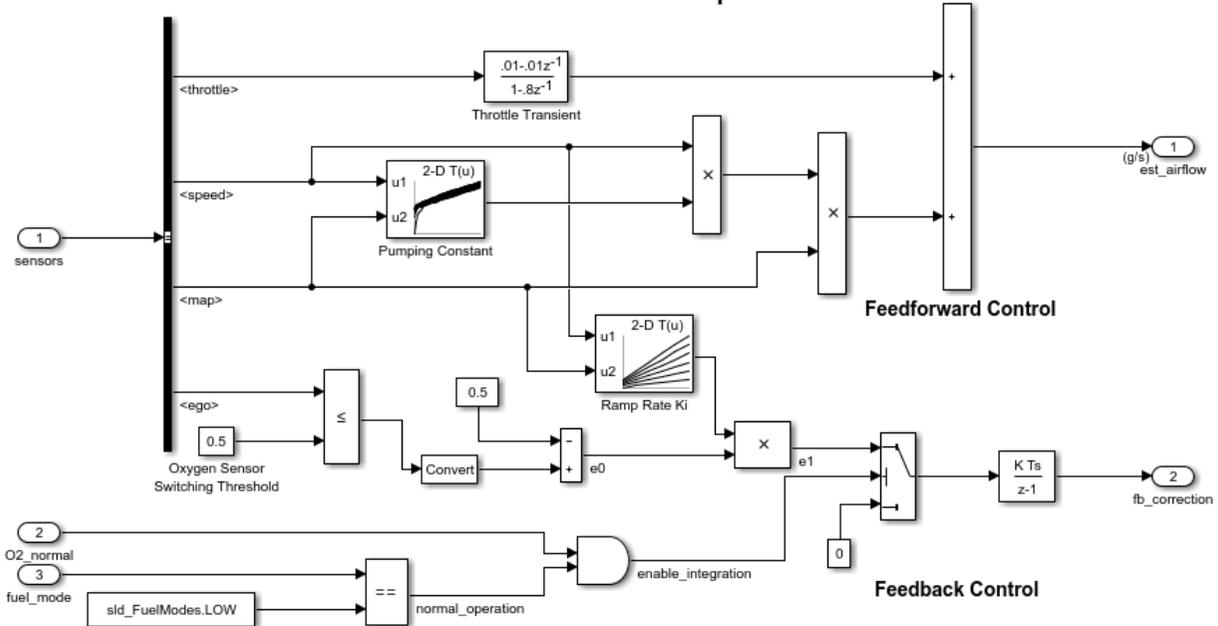
See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type”.

Examples

In the `sldemo_fuelsys` model, the `fuel_rate_control/airflow_calc` subsystem uses two Relational Operator blocks:

Intake Airflow Estimation and Closed-Loop Correction



Both Relational Operator blocks operate in two-input mode.

The block that uses this operator...	Compares...
<=	The value of the oxygen sensor to the threshold value, 0.5
==	The value of the fuel mode to the ideal value, sld_FuelModes.LOW

Characteristics

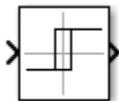
Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes

Variable-Size Signals	Yes
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

Introduced before R2006a

Relay

Switch output between two constants



Library

Discontinuities

Description

The Relay block allows its output to switch between two specified values. When the relay is on, it remains on until the input drops below the value of the **Switch off point** parameter. When the relay is off, it remains off until the input exceeds the value of the **Switch on point** parameter. The block accepts one input and generates one output.

The **Switch on point** value must be greater than or equal to the **Switch off point**. Specifying a **Switch on point** value greater than the **Switch off point** models hysteresis, whereas specifying equal values models a switch with a threshold at that value.

Note When the initial input falls *between* the **Switch off point** and **Switch on point** values, the initial output is the value when the relay is off.

Data Type Support

The Relay block accepts signals of the following data types:

- Floating point
- Built-in integer

- Fixed point
- Boolean
- Enumerated (output only)

If **Output when on** or **Output when off** is an enumerated value, both must be of the same enumerated type as the output.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

The **Main** pane of the Relay block dialog box appears as follows:

Switch on point

The “on” threshold for the relay. The **Switch on point** parameter is converted to the input data type offline using round-to-nearest and saturation.

Switch off point

The “off” threshold for the relay. The **Switch off point** parameter is converted to the input data type offline using round-to-nearest and saturation.

Output when on

The output when the relay is on.

Output when off

The output when the relay is off.

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).
- **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

- **Inherited** — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input `u`. All other input signals must be sample based.

Input Signal <code>u</code>	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Output minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`
- An enumerated data type, for example, `Enum:BasicColors`

In this case, **Output when on** and **Output when off** must be of the same enumerated type.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” in *Simulink User's Guide* for more information.

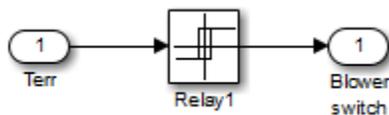
Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Examples

The `sldemo_househeat` model shows how you can use the Relay block.

The Relay block appears in the Thermostat subsystem.



The thermostat allows fluctuations of 5 degrees Fahrenheit above or below the desired room temperature. If air temperature drops below 65 degrees Fahrenheit, the thermostat turns on the heater. The Relay block outputs a value of 1 to turn on the heater and a value of 0 to turn off the heater.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No

Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

Introduced before R2006a

Repeating Sequence

Generate arbitrarily shaped periodic signal



Library

Sources

Description

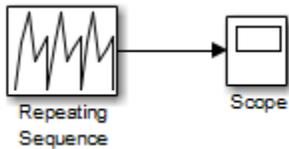
The Repeating Sequence block outputs a periodic scalar signal having a waveform that you specify using the **Time values** and **Output values** parameters. The **Time values** parameter specifies a vector of output times. The **Output values** parameter specifies a vector of signal amplitudes at the corresponding output times. Together, the two parameters specify a sampling of the output waveform at points measured from the beginning of the interval over which the waveform repeats (the period of the signal).

By default, both parameters are `[0 2]`. These default settings specify a sawtooth waveform that repeats every 2 seconds from the start of the simulation and has a maximum amplitude of 2.

Algorithm

The block sets the input period as the difference between the first and last value of the **Time values** parameter. The output at any time t is the output at time $t = t - n \cdot \text{period}$, where n is an integer. The sequence repeats at $t = n \cdot \text{period}$. The block uses linear interpolation to compute the value of the waveform between the output times that you specify.

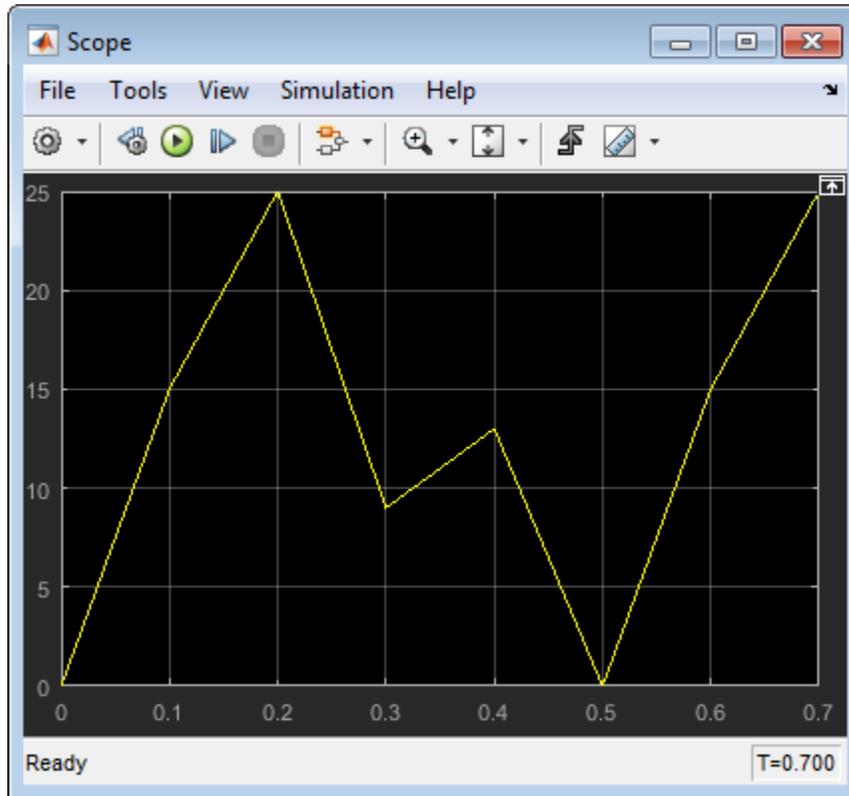
In the following model, the Repeating Sequence block defines **Time values** as [0:0.1:0.5] and **Output values** as [0 15 25 09 13 17]. The stop time of the simulation is 0.7 second.



For the Repeating Sequence block:

- The input period is 0.5.
- The output at any time t is the output at time $t = t - 0.5n$, where $n = 0, 1, 2$, and so on.
- The sequence repeats at $t = 0.5n$, where $n = 0, 1, 2$, and so on.

When you run the model, you get the following results:



At $t = 0.5$, the expected output is equal to the output at $t = 0$, which is 0. Therefore, the last value in the **Output values** vector [0 15 25 09 13 17] does not appear.

Data Type Support

The Repeating Sequence block outputs real signals of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Time values

Specify a vector of strictly monotonically increasing time values. The default is `[0 2]`.

Output values

Specify a vector of output values. Each element corresponds to the time value in the same column. The default is `[0 2]`.

Examples

The following Simulink examples show how to use the Repeating Sequence block:

- `sldemo_fuelsys`
- `sldemo_hydrod`
- `sldemo_VariableTransportDelay`

Characteristics

Data Types	Double
Sample Time	Continuous
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Repeating Sequence Interpolated, Repeating Sequence Stair

Introduced before R2006a

Repeating Sequence Interpolated

Output discrete-time sequence and repeat, interpolating between data points



Library

Sources

Description

The Repeating Sequence Interpolated block outputs a discrete-time sequence and then repeats it. Between data points, the block uses the method you specify for the **Lookup Method** parameter to determine the output.

Data Type Support

The Repeating Sequence Interpolated block outputs signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Vector of output values

Specify the column vector containing output values of the discrete time sequence.

Vector of time values

Specify the column vector containing time values. The time values must be strictly increasing, and the vector must have the same size as the vector of output values.

Lookup Method

Specify the lookup method to determine the output between data points.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to -1 . See “Specify Sample Time” for more information.

Output minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as

SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Repeating Sequence, Repeating Sequence Stair

Introduced before R2006a

Repeating Sequence Stair

Output and repeat discrete time sequence

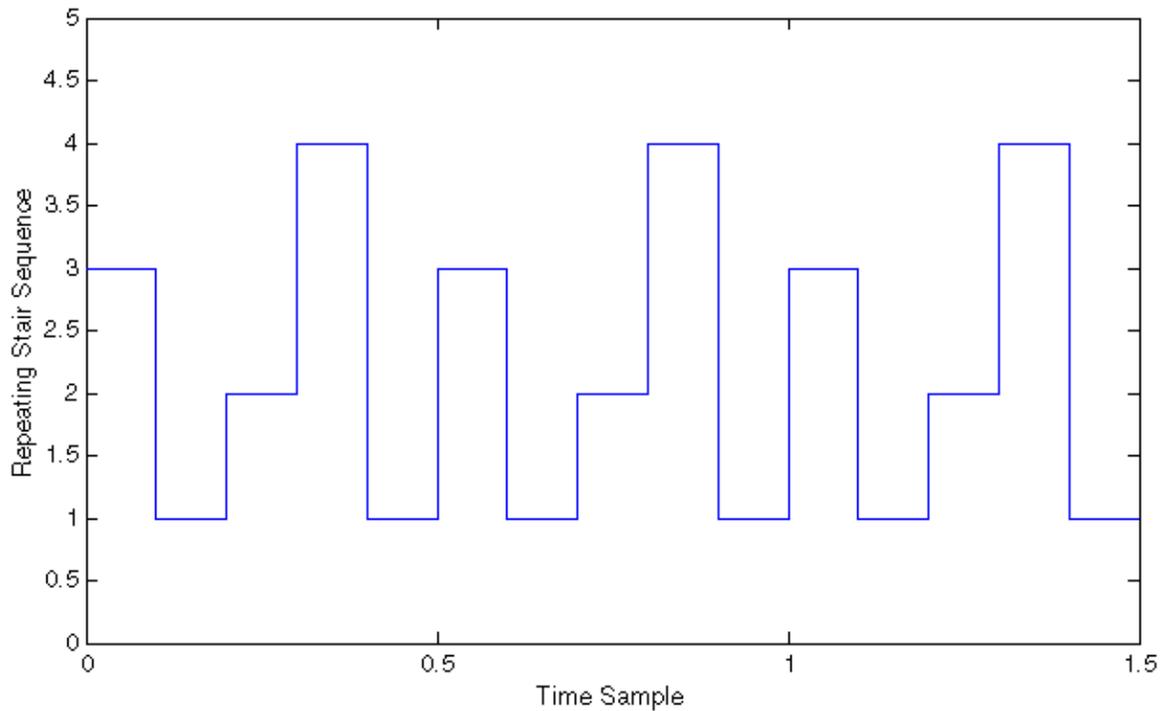


Library

Sources

Description

The Repeating Sequence Stair block outputs and repeats a stair sequence that you specify with the **Vector of output values** parameter. For example, you can specify the vector as `[3 1 2 4 1]'`, which produces the following stair sequence:



Data Type Support

The Repeating Sequence Stair block outputs signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Vector of output values

Specify the vector containing values of the repeating stair sequence.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to -1 . See “Specify Sample Time” for more information.

Output minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” in the *Simulink User's Guide* for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Repeating Sequence, Repeating Sequence Interpolated

Introduced before R2006a

Reset

Add reset port to subsystem

Library: Ports & Subsystems



Description

A Reset block at the root level of a Subsystem block adds a control port to the block. When a reset trigger event occurs on the signal connected to the port, the block states of the subsystem are reset to their initial condition.

Parameters

Reset trigger type — Select the type of trigger event

`level` (default) | `rising` | `falling` | `either` | `level hold`

Select the type of trigger event that resets the subsystem block states.

`level`

Reset the block states when the reset signal is nonzero at the current time step or changes from nonzero at the previous time step to zero at the current time step.

`rising`

Reset the block states when the reset signal rises from a zero to a positive value or from a negative to a positive value.

`falling`

Reset the block states when the reset signal falls from a positive value to zero or from a positive to a negative value.

`either`

Reset the block states when the reset signal changes from a zero to a nonzero value or changes sign.

`level hold`

Reset the block states when the reset signal is nonzero at the current time step.

Programmatic Use

Parameter: ResetTriggerType

Type: character vector

Value: 'level' | 'rising' | 'falling' | 'either' | 'level hold'

Default: 'level'

Propagate sizes of variable-size signals — Select when to propagate a variable-size signal

During execution (default) | Only when enabling

Select when to propagate a variable-size signal.

During execution

Propagate variable-size signals at each time step.

Only when resetting

Propagate variable-size signals when resetting a Subsystem block containing a Reset port block. When you select this option, sample time must be periodic.

Programmatic Use

Block parameter: PropagateVarSize

Type: character vector

Value: 'During execution' | 'Only when resetting'

Default: 'During execution'

Enable zero-crossing detection — Control zero-crossing detection

on (default) | off

Control zero-crossing detection.

on

Detect zero crossings.

off

Do not detect zero crossings.

Programmatic Use

Block parameter: ZeroCross

Type: character vector

Value: 'on' | 'off'

Default: 'on'

See Also

Blocks

Reset | Resettable Subsystem | Subsystem

Topics

“Resettable Subsystems”

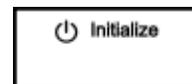
“Simulink Block Support for Variable-Size Signals”

Introduced in R2015a

Reset Function

Executes contents on a model reset event

Library: User-Defined Functions



Description

The Reset Function block is a preconfigured subsystem block created from an Initialize block that executes on a model reset event.

Note The Reset Function block is not in the Simulink Library Browser.

To create a Reset Function block:

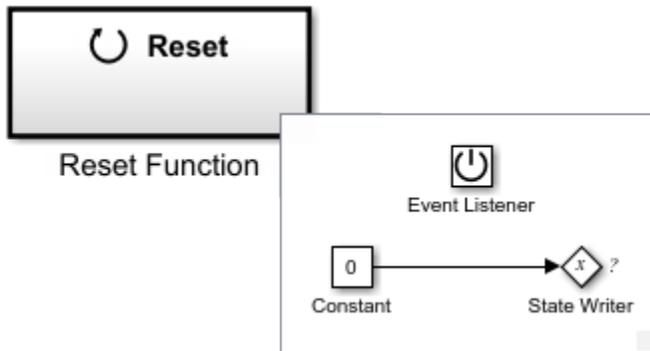
- 1 Add an Initialize block.
- 2 Open the Event Listener block within the Initialize block. From the **Event** drop-down list, select `Reset`.

One parameter appears.

- By default, the **Event name** is set to `Reset`. You can change the name of a reset event by editing this text box.

- 3 Rename the block from `Initialize Function` to `Reset Function`.

The Reset Function block includes an Event Listener block with **Event** set to `Reset`, a Constant block with **Constant value** set to `0`, and a State Writer block.



Replace the Constant block with source blocks that generate the state value for the State Writer block.

For a list of unsupported blocks and features, see “Initialize, Reset, and Terminate Function Limitations”.

A model can potentially have multiple Reset Function blocks with each block having a different **Event name**. Each of these reset events appear in the generated code as a different function.

See Also

Event Listener | Initialize Function | State Reader | State Writer | Terminate Function

Topics

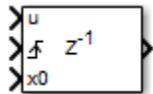
“Create Model to Initialize, Reset, and Terminate State”

“Create Test Harness to Generate Function Calls”

“Generate Code That Responds to Initialize, Reset, and Terminate Events” (Simulink Coder)

Resettable Delay

Delay input signal by variable sample period and reset with external signal



Library

Discrete

Description

The Resettable Delay block is a variant of the Delay block that has the source of the initial condition set to `Input port` and the external reset algorithm set to `Rising`, by default.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	Yes

See Also

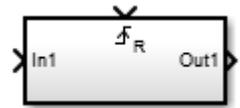
Delay | Tapped Delay | Unit Delay | Variable Integer Delay

Introduced in R2012b

Resettable Subsystem

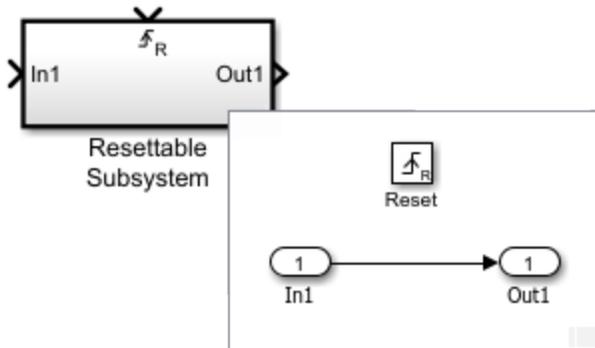
Subsystem whose block states reset with external trigger

Library: Ports & Subsystems



Description

The Resettable Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that resets the block states each time the control signal has a trigger event.



Ports

Input

In — Signal input to a subsystem block

scalar | vector | matrix

Placing an Inport block in a Subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

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

Reset — Control signal input to a subsystem block

`scalar` | `vector` | `matrix`

Placing a Reset block in a Subsystem block adds an external input port to the block and changes the block to a Resettable Subsystem block.

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

Output

out — Signal output from a subsystem

`scalar` | `vector` | `matrix`

Placing an Outport block in a Subsystem block adds an output port from the block. The port label on the Subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

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

See Also

Blocks

Reset | Subsystem

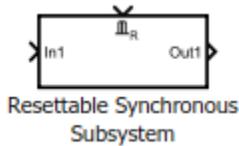
Topics

“Resettable Subsystems”

Introduced in R2015a

Resetable Synchronous Subsystem

Represent resetable subsystem that has synchronous reset and enable behavior



Library

HDL Coder / HDL Subsystems

Description

The Resetable Synchronous Subsystem block uses the `Synchronous` mode of the State Control block with the Resetable Subsystem block. If an **S** symbol appears in the subsystem, then it is synchronous. For more information about the block in HDL Coder, see [Resetable Synchronous Subsystem](#).

For more information about the State Control block, see [State Control](#).

Data Type Support

See [Inport](#) for information on the data types accepted by a subsystem's input ports. See [Outport](#) for information on the data types output by a subsystem's output ports.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Show port labels

Display subsystem port labels on the subsystem block.

Default: `FromPortIcon`

`none`

Does not display port labels on the subsystem block.

`FromPortIcon`

If the corresponding port icon displays a signal name, the parameter displays the signal name on the subsystem block. Otherwise, it displays the port block name.

`FromPortBlockName`

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

`SignalName`

If the signal connected to the subsystem block port is named, this parameter displays the name. Otherwise, it displays the name of the corresponding port block.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Read/Write permissions

Control user access to the contents of the subsystem.

Default: `ReadWrite`

`ReadWrite`

Enables opening and modification of subsystem contents.

`ReadOnly`

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

NoReadOrWrite

Disables the opening or modification of subsystem. If the subsystem resides in a block library, you can create links to the subsystem in a model. You cannot open, modify, change permissions, or create local copies of the subsystem.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Name of error callback function

Enter the name of the function to be called if an error occurs while Simulink software is executing the subsystem.

Default: ' '

Simulink passes two arguments to the function: the subsystem handle and a character vector that specifies the error type. If no function is specified, you get a generic error message.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Permit hierarchical resolution

Specify whether to resolve names of workspace variables referenced by this subsystem.

Default: All

All

Resolve all names of workspace variables used by this subsystem, including those used to specify block parameter values and Simulink data objects (for example, `Simulink.Signal` objects).

ExplicitOnly

Resolve the names of workspace variables used to specify block parameter values, data store memory (where no block exists), signals, and states marked by using the signal resolution icon.

None

Do not resolve any workspace variable names.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Treat as atomic unit

Causes Simulink to treat the subsystem as a unit when determining the execution order of block methods.

Default: Off

On

Cause Simulink to treat the subsystem as a unit when determining the execution order of block methods. For example, when it needs to compute the output of the subsystem, Simulink software invokes the output methods of all the blocks in the subsystem before invoking the output methods of other blocks at the same level as the subsystem block.

Off

Cause Simulink to treat all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem when determining block method execution order. This can cause the execution of block methods in the subsystem to be interleaved with the execution of block methods outside the subsystem.

This parameter enables:

- “Minimize algebraic loop occurrences” on page 1-0
- “Sample time” on page 1-0
- “Function packaging” on page 1-0 (requires a Simulink Coder license)

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Treat as grouped when propagating variant conditions

Causes Simulink software to treat the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks.

Default: On

On

Simulink treats the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks. For example, when Simulink computes the variant condition of the subsystem, it propagates that condition to all blocks in the subsystem.

Off

Simulink treats all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem itself when determining their variant condition.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Function packaging

Specify the code format to be generated for an atomic (nonvirtual) subsystem.

Default: Auto

Auto

Simulink Coder chooses the optimal format based on the type and number of subsystem instances in the model.

Inline

Simulink Coder inlines the subsystem unconditionally.

Nonreusable function

Simulink Coder explicitly generates a separate function in a separate file. In some cases, when you apply this setting, the subsystems generate functions with arguments that depend on the “Function interface” on page 1-0 parameter setting. You can name the generated function and file using parameters “Function

name” on page 1-0 and “File name (no extension)” on page 1-0 . These functions are not reentrant.

Reusable function

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

This option generates a function with arguments that allows subsystem code to be reused in the generated code of a model reference hierarchy. In this case, the subsystem must be in a library.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes
HDL Code Generation	Yes

See Also

Enable | Enabled Synchronous Subsystem | State Control | Synchronous Subsystem

Introduced in R2016b

Reshape

Change dimensionality of signal



Library

Math Operations

Description

The Reshape block changes the dimensionality of the input signal to a dimensionality that you specify, using the block's **Output dimensionality** parameter. For example, you can use the block to change an N-element vector to a 1-by-N or N-by-1 matrix signal, and vice versa.

The **Output dimensionality** parameter lets you select any of the following output options.

Output Dimensionality	Description
1-D array	Converts a multidimensional array to a vector (1-D array) array signal. The output vector consists of the first column of the input matrix followed by the second column, etc. (This option leaves a vector input unchanged.)
Column vector	Converts a vector, matrix, or multidimensional input signal to a column matrix, i.e., an M-by-1 matrix, where M is the number of elements in the input signal. For matrices, the conversion is done in column-major order. For multidimensional arrays, the conversion is done along the first dimension.

Output Dimensionality	Description
Row vector	Converts a vector, matrix, or multidimensional input signal to a row matrix, i.e., a 1-by-N matrix where N is the number of elements in the input signal. For matrices, the conversion is done in column-major order. For multidimensional arrays, the conversion is done along the first dimension.
Customize	Converts the input signal to an output signal whose dimensions you specify, using the Output dimensions parameter. The value of the Output dimensions parameter can be a one- or multi-element vector. A value of [N] outputs a vector of size N. A value of [M N] outputs an M-by-N matrix. The number of elements of the input signal must match the number of elements specified by the Output dimensions parameter. For multidimensional arrays, the conversion is done along the first dimension.
Derive from reference input port	Creates a second input port, Ref, on the block. Derives the dimensions of the output signal from the dimensions of the signal input to the Ref input port. Selecting this option disables the Output dimensions parameter. When you select this parameter, the input signals for both inport ports, U and Ref, must have the same sampling mode (sample-based or frame-based).

Data Type Support

The Reshape block accepts and outputs signals of any data type that Simulink supports, including fixed-point, enumerated, and nonvirtual bus data types.

For more information, see “Data Types Supported by Simulink”.

You can use an array of buses as an input signal to a Reshape block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Parameters

Output dimensionality

Specify the dimensionality of the output signal.

Output dimensions

Specify a custom output dimensionality. This parameter is available only when you set **Output dimensionality** to *Customize*.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Rocker Switch

Toggle parameter between two values

Library: Simulink / Dashboard



Description

The Rocker Switch block toggles the value of the connected block parameter between two values during simulation. For example, you can connect the Rocker Switch block to a Switch block in your model and change its state during simulation. Use the Rocker Switch block with other Dashboard blocks to create an interactive dashboard for your model.

Connection

Dashboard block controls connect to variables and block parameters. To connect a Dashboard block to a variable or parameter, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting blocks of interest in your model.
- 3 Mark the option button next to the parameter or variable you want to adjust during simulation.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined using the variable `engine(1)` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

Parameters

Connection — Select a block parameter to connect

signal connection options

Select the block parameter to control using the **Connection** table. To open the dialog box, double-click the block. Then, populate the **Connection** table by selecting a block in your model. Select the variable parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram with **Ctrl+D**.

States

Label (Top) — Label for top switch position

'On' (default) | character vector

Labels the top switch position. You can use the **Label** to display the value the connected parameter takes when the switch is positioned at the top, or you can enter a text label.

Example: `Gain = 2`

Value (Top) — Value for top switch position

1 (default) | scalar

The value assigned to the connected parameter when the switch is positioned at the top.

Label (Bottom) — Label for bottom switch position

'Off' (default) | character vector

Labels the bottom switch position. You can use the **Label** to display the value the connected parameter takes when the switch is positioned at the bottom, or you can enter a text label.

Example: `Gain = 1`

Value (Bottom) — Value for bottom switch position

0 (default) | scalar

The value assigned to the connected parameter when the switch is positioned at the bottom.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

See Also

Rotary Switch | Slider Switch | Toggle Switch

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

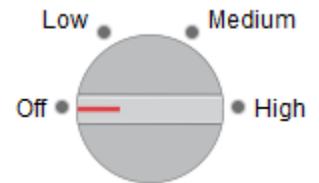
“Decide How to Visualize Simulation Data”

Introduced in R2015a

Rotary Switch

Switch parameter to set values on dial

Library: Simulink / Dashboard



Description

The Rotary Switch changes the value of the connected block parameter to several specified values during simulation. For example, you can connect the Rotary Switch block to the amplitude or frequency of an input signal in your model and change its characteristics during simulation. Use the Rotary Switch block with other Dashboard blocks to create an interactive dashboard to control your model.

Connection

Dashboard block controls connect to variables and block parameters. To connect a Dashboard block to a variable or parameter, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting blocks of interest in your model.
- 3 Mark the option button next to the parameter or variable you want to adjust during simulation.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined using the variable `engine(1)` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

Parameters

Connection — Select a block parameter to connect

signal connection options

Select the block parameter to control using the **Connection** table. To open the dialog box, double-click the block. Then, populate the **Connection** table by selecting a block in your model. Select the variable parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram with **Ctrl+D**.

States

value — Values for switch positions

0/1/2/3 (default) | scalar

The values assigned to the connected parameter when the switch is positioned at the corresponding **Label**. Click the **+** button to add positions.

label — Labels for switch positions

'Off'/'Low'/'Medium'/'High' (default) | character vector

Labels the switch positions. You can use the **Label** to display the value the connected parameter takes when the switch points to the **Label**, or you can enter a descriptive text label. Click the **+** button to add positions.

Example: `Gain = 2`

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

See Also

Knob | Rocker Switch | Slider | Slider Switch | Toggle Switch

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2015a

Rounding Function

Apply rounding function to signal



Library

Math Operations

Description

The Rounding Function block applies a rounding function to the input signal to produce the output signal.

You can select one of the following rounding functions from the **Function** list:

- floor

Rounds each element of the input signal to the nearest integer value towards minus infinity.

- ceil

Rounds each element of the input signal to the nearest integer towards positive infinity.

- round

Rounds each element of the input signal to the nearest integer.

- fix

Rounds each element of the input signal to the nearest integer towards zero.

The name of the selected function appears on the block.

The input signal can be a scalar, vector, or matrix signal having real- or complex-valued elements of type `double`. The output signal has the same dimensions, data type, and numeric type as the input. Each element of the output signal is the result of applying the selected rounding function to the corresponding element of the input signal.

Tip Use the Rounding Function block instead of the Fcn block when you want vector or matrix output, because the Fcn block produces only scalar output.

Also, the Rounding Function block provides two more rounding modes. The Fcn block supports `floor` and `ceil`, but does not support `round` and `fix`.

Data Type Support

The Rounding Function block accepts and outputs real signals of type `double` or `single`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Function

Specify the rounding function.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Characteristics

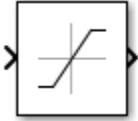
Data Types	Double Single
Sample Time	Inherited from driving block

Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Saturation

Limit range of signal



Library

Discontinuities

Description

The Saturation block imposes upper and lower limits on an input signal.

When the input is...	Where...	The block output is the...
Within the range specified by the Lower limit and Upper limit parameters	$\text{Lower limit} \leq \text{Input value} \leq \text{Upper limit}$	Input value
Less than the Lower limit parameter	$\text{Input value} < \text{Lower limit}$	Lower limit
Greater than the Upper limit parameter	$\text{Input value} > \text{Upper limit}$	Upper limit

When the **Lower limit** and **Upper limit** parameters have the same value, the block output is that value.

Data Type Support

The Saturation block accepts real signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Upper limit

Specify the upper bound on the input signal.

Default: 0.5

Minimum: value from the **Output minimum** parameter

Maximum: value from the **Output maximum** parameter

- When the input signal to the Saturation block is above this value, the output of the block is clipped to this value.
- The **Upper limit** parameter is converted to the output data type offline using round-to-nearest and saturation.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Lower limit

Specify the lower bound on the input signal.

Default: -0.5

Minimum: value from the **Output minimum** parameter

Maximum: value from the **Output maximum** parameter

- When the input signal to the Saturation block is below this value, the output of the block is clipped to this value.
- The **Lower limit** parameter is converted to the output data type offline using round-to-nearest and saturation.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Treat as gain when linearizing

Select this parameter to cause the linearization commands to treat the gain as 1

Default: On

On

Select to cause the linearization commands to treat the gain as 1.

Off

Clear to cause the linearization commands to treat the gain as 0.

Linearization commands in Simulink software treat this block as a gain in state space.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Default: On



On

Enable zero-crossing detection.



Off

Do not enable zero-crossing detection.

Parameter: ZeroCross

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off



On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: LockScale

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: Floor

Ceiling

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

Floor

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

Nearest

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Parameter: `RndMeth`

Type: character vector

Value: `'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'`

Default: `'Floor'`

For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Output minimum

Lower value of the output range that Simulink checks.

Default: `[]` (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMin
Type: character vector
Value: ' [] '
Default: ' [] '

Output maximum

Upper value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMax
Type: character vector
Value: ' [] '
Default: ' [] '

Output data type

Specify the output data type.

Default: Inherit: Same as input

Inherit: Inherit via back propagation

Use data type of the driving block.

Inherit: Same as input

Use data type of input signal.

double

Output data type is double.

single

Output data type is single.

int8

Output data type is int8.

uint8

Output data type is uint8.

int16

Output data type is int16.

uint16

Output data type is uint16.

int32

Output data type is int32.

uint32

Output data type is uint32.

fixdt(1,16,0)

Output data type is fixed point fixdt(1,16,0).

fixdt(1,16,2^0,0)

Output data type is fixed point fixdt(1,16,2^0,0).

<data type expression>

Use a data type object, for example, Simulink.NumericType.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Control Signal Data Types”.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via back propagation`
- `Same as input (default)`

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Fixed-point data types.

`Expression`

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: Signed

Signed

Specify the fixed-point data as signed.

Unsigned

Specify the fixed-point data as unsigned.

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting Slope and bias enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

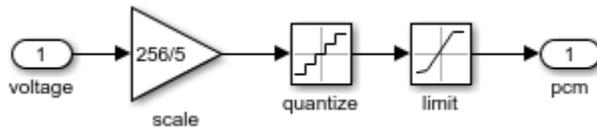
Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Examples

The `sldemo_boiler` model shows how you can use the Saturation block.

The Saturation block appears in the Boiler Plant model/digital thermometer/ADC subsystem.



The ADC subsystem digitizes the input analog voltage by:

- Multiplying the analog voltage by 256/5 with the Gain block
- Rounding the value to integer floor with the Quantizer block
- Limiting the output to a maximum of 255 (the largest unsigned 8-bit integer value) with the Saturation block

For more information, see “Explore the Fixed-Point “Bang-Bang Control” Model” (Stateflow) in the Stateflow documentation.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

See Also

Saturation Dynamic

Introduced before R2006a

Saturation Dynamic

Bound range of input



Library

Discontinuities

Description

The Saturation Dynamic block bounds the range of an input signal to upper and lower saturation values. An input signal outside of these limits saturates to one of the bounds where:

- The input below the lower limit is set to the lower limit.
- The input above the upper limit is set to the upper limit.

The input for the upper limit is the `up` port, and the input for the lower limit is the `lo` port.

Data Type Support

The Saturation Dynamic block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

Tip Although this block accepts Boolean signals, avoid this usage.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Output minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as second input`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” in the *Simulink User's Guide* for more information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer). in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Do not select this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</p>	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code> , which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code> , is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

Saturation

Introduced before R2006a

ScopeTime Scope

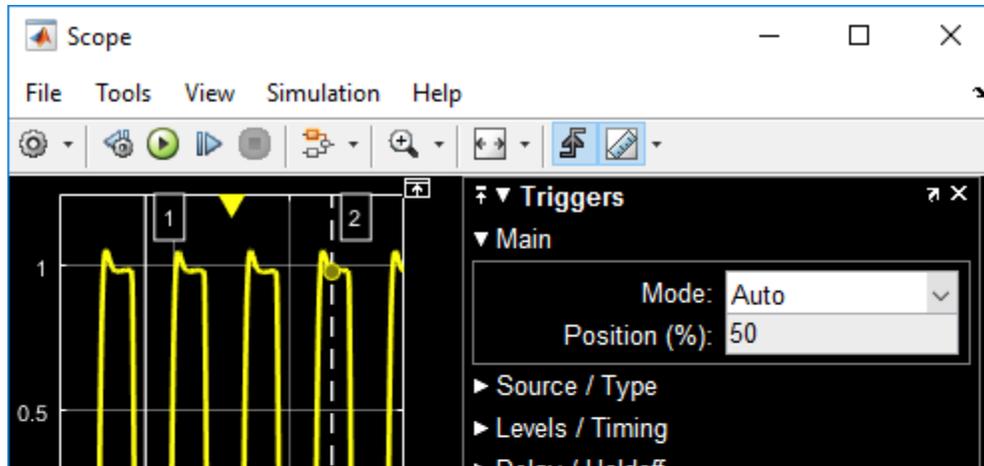
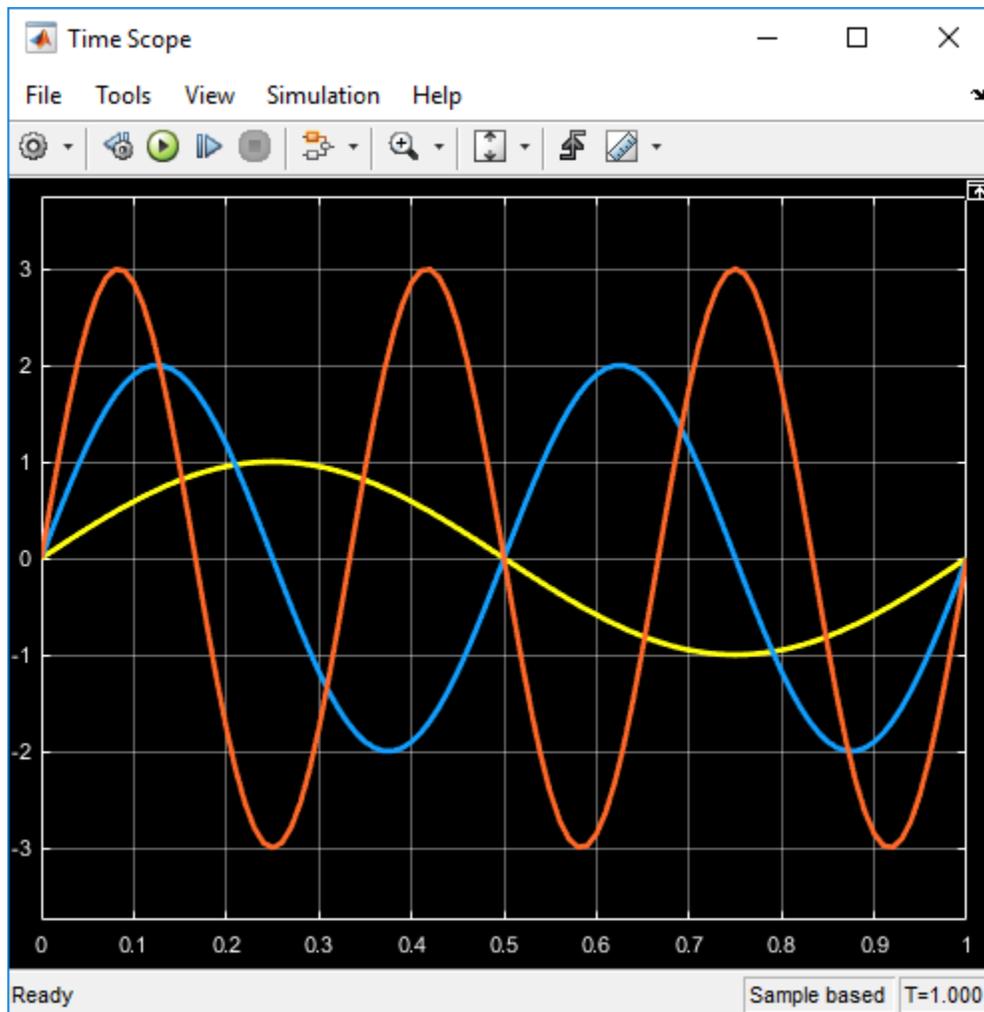
Display signals generated during simulation

Library: Simulink / Sinks



Description

The Simulink Scope block and DSP System Toolbox Time Scope block display time domain signals.



The two blocks have identical functionality, but different default settings. The Time Scope is optimized for discrete time processing. The Scope is optimized for general time-domain simulation. For a side-by-side comparison, see “Simulink Scope Versus DSP System Toolbox Time Scope”.

Oscilloscope features:

- Triggers — Set triggers to sync repeating signals and pause the display when events occur.
- Cursor Measurements — Measure signal values using vertical and horizontal cursors.
- Signal Statistics — Display the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal.
- Peak Finder — Find maxima, showing the x -axis values at which they occur.
- Bilevel Measurements — Measure transitions, overshoots, undershoots, and cycles.

You must have a Simscape™ or DSP System Toolbox license to use the Peak Finder, Bilevel Measurements, and Signal Statistics.

Scope display features:

- Simulation control — Debug models from a Scope window using **Run**, **Step Forward**, and **Step Backward** toolbar buttons.
- Multiple signals — Plot multiple signals on the same y -axis (display) using multiple input ports.
- Multiple y -axes (displays) — Display multiple y -axes. All the y -axes have a common time range on the x -axis.
- Modify parameters — Modify scope parameter values before and during a simulation.
- Axis autoscaling — Autoscale axes during or at the end of a simulation. Margins are drawn at the top and bottom of the axes.
- Display data after simulation — If a Scope is closed at the start of a simulation, scope data is still written to the scope during a simulation. As a result, if you open the Scope after a simulation, the Scope displays simulation results for attached input signals.

For information on controlling a scope programmatically, see “Control Scopes Programmatically”.

Limitations

- Do not use scope blocks in a Library. If you place a scope block inside a library block with a locked link or in a locked library, Simulink displays an error when trying to open the scope window. To display internal data from a library block, add an output port to the library block, and then connect the port to a Scope block in your model.
- A Scope block may plot a single point when connected to a constant signal.

Ports

Input

Port_1 — Signal or signals to visualize

scalar | vector | matrix | array | bus | nonvirtual bus

Connect the signals you want to visualize. You can have up to 96 input ports. Input signals can have these characteristics:

- **Type** - Continuous (sample-based) or discrete (sample-based and frame-based).
- **Data type** - Any data type that Simulink supports. See “Data Types Supported by Simulink”.
- **Dimension** — Scalar, one dimensional (vector), two dimensional (matrix), or multidimensional. Display multiple channels within one signal depending on the dimension. See “Signal Dimensions” and “Determine Output Signal Dimensions”.

Bus Support

You can connect nonvirtual bus and arrays of bus signals to a Scope block. To display the bus signals, use normal or accelerator simulation mode. The Scope block displays each bus element signal in the order the elements appear in the bus, from the top to the bottom. Nested bus elements are flattened.

To log nonvirtual bus signals with a Scope block, set the **Save format** block parameter to Dataset. You can use any **Save format** to log virtual bus signals.

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

Properties

Configuration Properties

The Configuration Properties dialog box controls various properties about the scope displays. From the scope menu, select **View > Configuration Properties**.

Main

Open at simulation start — Specify when scope window opens
off (default for Scope) | on (default for Time Scope)

Select this check box to open the scope window when simulation starts.

Programmatic Use

See `OpenAtSimulationStart`.

Display the full path — Display block path on scope title bar
off (default) | on

Select this check box to display the block path in addition to the block name.

Number of input ports — Number of input ports on scope block
1 (default) | integer

Specify number of input ports on a Scope block, specified as an integer. The maximum number of input ports is 96. This property does not apply to floating scopes and scope viewers.

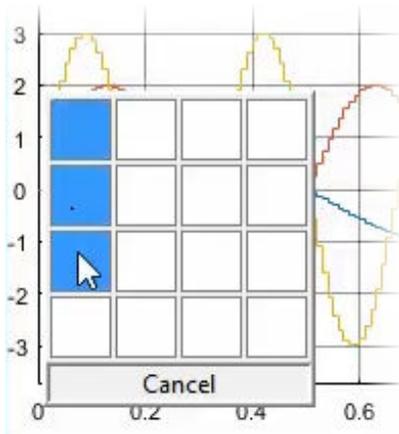
Programmatic Use

See `NumInputPorts`.

Layout — Number and arrangement of displays
1-by-1 display (default) | an arrangement of m-by-n axes

Specify number and arrangement of displays. The maximum layout is 16 rows by 16 columns.

To expand the layout grid beyond 4 by 4, click within the dialog box and drag. Maximum of 16 rows by 16 columns.



If the number of displays is equal to the number of ports, signals from each port appear on separate displays. If the number of displays is less than the number of ports, signals from additional ports appear on the last display. For layouts with multiple columns and rows, ports are mapped down then across.

Programmatic Use

See `LayoutDimensions`.

Sample time — Simulation interval between scope updates

-1 (for inherited) (default) | positive real number

Specify the time interval between updates of the scope display. This property does not apply to floating scopes and scope viewers.

Programmatic Use

See `SampleTime`.

Input processing — Channel or element signal processing

`Elements as channels (sample based) (default for Scope)` | `Columns as channels (frame based) (default for Time Scope)`

- `Elements as channels (sample based)` - Process each element as a unique sample.

- `Columns as channels (frame based)` - Process signal values in a channel as a group of values from multiple time intervals. Frame-based processing is available only with discrete input signals.

Programmatic Use

See `FrameBasedProcessing`.

Maximize axes — Maximize size of plots

`Off` (default for `Scope`) | `Auto` (default for `Time Scope`) | `On`

- `Auto` - If “`Title`” on page 1-0 and “`Y-label`” on page 1-0 properties are not specified, maximize all plots.
- `On` - Maximize all plots. Values in **`Title`** and **`Y-label`** are hidden.
- `Off` - Do not maximize plots.

Programmatic Use

See `MaximizeAxes`.

Time

Time span — Length of x-axis to display

`Auto` (default) | `User defined`

- `Auto` - Difference between the simulation start and stop times.
- The block calculates the beginning and end times of the time range using the “`Time display offset`” on page 1-0 and “`Time span`” on page 1-0 properties. For example, if you set the **`Time display offset`** to 10 and the **`Time span`** to 20, the scope sets the time range from 10 to 30.
- `User defined` - Value less than the total simulation time.

Programmatic Use

See `TimeSpan`.

Time span overrun action — Display data beyond visible x-axis

`Wrap` (default) | `Scroll`

Specify how to display data beyond the visible *x*-axis range.

You can see the effects of this option only when plotting is slow with large models or small step sizes.

- `Wrap` - Draw a full screen of data from left to right, clear the screen, and then restart drawing of data.
- `Scroll` - Move data to the left as new data is drawn on the right. This mode is graphically intensive and can affect run-time performance.

Programmatic Use

See `TimeSpanOverrunAction`.

Time units — *x*-axis units

`None` (default for `Scope`) | `Metric` (default for `Time Scope`) | `Seconds`

- `Metric` - Display time units based on the length of “Time span” on page 1-0 .
- `Seconds` - Display time (seconds).
- `None` - Do not display time units.

Programmatic Use

See `TimeUnits`.

Time display offset — *x*-axis offset

`0` (default) | `scalar` | `vector`

Offset the *x*-axis by a specified time value, specified as a real number or vector of real numbers.

For input signals with multiple channels, you can enter a scalar or vector:

- `Scalar` — Offset all channels of an input signal by the same time value.
- `Vector` — Independently offset the channels.

Programmatic Use

See `TimeDisplayOffset`.

Time-axis labels — Display of *x*-axis labels

`Bottom Displays Only` (default for `Scope`) | `All` (default for `Time Scope`) | `None`

Specify how *x*-axis (time) labels display.

- All - Display *x*-axis labels on all *y*-axes.
- None - Do not display labels. Selecting None also clears the **Show time-axis label** check box.
- Bottom displays only - Display *x*-axis label on the bottom *y*-axis.

Dependencies

The “Active display” on page 1-0 `ActiveDisplay` property determines which display is affected.

To activate this property, set:

- “Show time-axis label” on page 1-0 `ShowTimeAxisLabel` to on.
- “Maximize axes” on page 1-0 `MaximizeAxes` to off.

Programmatic Use

See `TimeAxisLabels`.

Show time-axis label — Display or hide *x*-axis labels

off (default for Scope) | on (default for Time Scope)

Select this check box to show the *x*-axis label for the active display

Dependencies

The “Active display” on page 1-0 `ActiveDisplay` property determines which display is affected.

To activate this property, set “Time-axis labels” on page 1-0 `TimeAxisLabels` to All or Bottom Displays Only.

Programmatic Use

See `ShowTimeAxisLabel`.

Display

Active display — Selected display

1 (default) | positive integer

Selected display. Use this property to control which display is changed when changing style properties and axes-specific properties.

Specify the desired display using a positive integer that corresponds to the column-wise placement index.

Programmatic Use

See “Active display” on page 1-0 .

Title — Display name

`%<SignalLabel>` (default) | character vector | string

Specify a title for display, specified as a character vector or string. The default value `%<SignalLabel>` uses the input signal name for the title.

Dependency

The “Active display” on page 1-0 property determines which display is affected.

Programmatic Use

See Title.

Show legend — Display signal legend

`off` (default) | `on`

Toggle signal legend. The names listed in the legend are the signal names from the model. For signals with multiple channels, a channel index is appended after the signal name. Continuous signals have straight lines before their names, and discrete signals have step-shaped lines.

From the legend, you can control which signals are visible. This control is equivalent to changing the visibility in the **Style** properties. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal, right-click the signal name, which hides all other signals. To show all signals, press **Esc**.

Dependency

The “Active display” on page 1-0 property determines which display is affected.

Programmatic Use

See ShowLegend.

Show grid — Show internal grid lines

`on` (default) | `off`

Select this check box to show grid lines.

Dependency

The “Active display” on page 1-0 property determines which display is affected.

Programmatic Use

See ShowGrid.

Plot signals as magnitude and phase — Split display into magnitude and phase plots

off (default) | on

- On - Display magnitude and phase plots. If the signal is real, plots the absolute value of the signal for the magnitude. The phase is 0 degrees for positive values and 180 degrees for negative values. This feature is useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box returns the absolute value of the signal for the magnitude.
- Off - Display signal plot. If the signal is complex, plots the real and imaginary parts on the same *y*-axis.

Dependency

The “Active display” on page 1-0 property determines which display is affected.

Programmatic Use

See PlotAsMagnitudePhase.

Y-limits (Minimum) — Minimum y-axis value

-10 (default) | real scalar

Specify the minimum value of the *y*-axis as a real number.

Tunable: Yes

Dependency

The “Active display” on page 1-0 property determines which display is affected.

If you select **Plot signals as magnitude and phase**, this property only applies to the magnitude plot. The *y*-axis limits of the phase plot are always [-180 180].

Programmatic Use

See YLimits.

Y-limits (Maximum) — Maximum y-axis value

10 (default) | real scalar

Specify the maximum value of the y-axis as a real number.

Tunable: Yes

Dependency

The “Active display” on page 1-0 `ActiveDisplay` property determines which display is affected.

If you select **Plot signals as magnitude and phase**, this property only applies to the magnitude plot. The y-axis limits of the phase plot are always [-180 180].

Programmatic Use

See YLimits.

Y-label — Y-axis label

none (default for Scope) | 'Amplitude' (default for Time Scope) | character vector

Specify the text to display on the y-axis. To display signal units, add (%<SignalUnits>) to the label. At the beginning of a simulation, Simulink replaces (%SignalUnits) with the units associated with the signals.

Example: For a velocity signal with units of m/s, enter `Velocity (%<SignalUnits>)`.

Dependency

The “Active display” on page 1-0 `ActiveDisplay` property determines which display is affected.

If you select **Plot signals as magnitude and phase**, this property does not apply. The y-axes are labeled `Magnitude` and `Phase`.

Programmatic Use

See YLabel.

Logging

Limit data points to last — Limit buffered data values

off and 5000 (default) | on | positive integer

Specify to limit buffered data values before plotting and saving signals. Data values are from the end of a simulation. To use this property, you must also specify the number of data values by entering a positive integer in the text box.

- On - Save specified number of data values for each signal. If the signal is frame-based, the number of buffered data values is the specified number of data values multiplied by the frame size.

For simulations with **Stop time** set to `inf`, consider selecting **Limit data points to last**.

In some cases, for example where the sample time is small, selecting this parameter can have the effect of plotting signals for less than the entire time range of a simulation. If a scope plots a portion of your signals, consider increasing the number of data values to save.

- Off - Save and plot all data values. Clearing **Limit data points to last** can cause an out-of-memory error for simulations that generate a large amount of data or for systems without enough available memory.

Dependency

To enable this property, select “Log data to workspace” on page 1-0 .

This property limits the data values plotted in the scope and the data values saved to a MATLAB variable specified in “Variable name” on page 1-0 .

Programmatic Use

See `DataLoggingLimitDataPoints` and `DataLoggingMaxPoints`.

Decimation — Reduce amount of scope data to display and save

off, 2 (default) | on | positive integer

- On - Plot and log (save) scope data every Nth data point, where N is the decimation factor entered in the text box. A value of 1 buffers all data values.
- Off - Save all scope data values.

Dependency

To enable this property, select “Log data to workspace” on page 1-0 .

This property limits the data values plotted in the scope and the data values saved to a MATLAB variable specified in “Variable name” on page 1-0 .

Programmatic Use

See `DataLoggingDecimateData` and `DataLoggingDecimation`.

Log data to workspace — Save data to MATLAB workspace

off (default) | on

Select this check box to activate logging and activate the **Variable name**, **Save format**, and **Decimation** properties. This property does not apply to floating scopes and scope viewers.

For an example of saving signals to the MATLAB Workspace using a Scope block, see “Save Simulation Data Using Floating Scope Block”.

Programmatic Use

See `DataLogging`.

Variable name — Name of saved data variable

`ScopeData` (default) | character vector | string

Specify a variable name for saving scope data in the MATLAB workspace, specified as a character vector or string. This property does not apply to floating scopes and scope viewers.

Dependency

To enable this property, select “Log data to workspace” on page 1-0 .

Programmatic Use

See `DataLoggingVariableName`.

Save format — MATLAB variable format

`Dataset` (default) | `Structure With Time` | `Structure` | `Array`

Select variable format for saving data to the MATLAB workspace. This property does not apply to floating scopes and scope viewers.

- **Dataset** - Save data as a dataset object. This format does not support variable-size data, MAT-file logging, or external mode archiving. See `Simulink.SimulationData.Dataset`.
- **Structure With Time** - Save data as a structure with associated time information.
- **Structure** - Save data as a structure.
- **Array** - Save data as an array with associated time information. This format does not support variable-size data.

Dependency

To enable this property, select “Log data to workspace” on page 1-0 .

Programmatic Use

See `DataLoggingSaveFormat`.

Axes Scaling Properties

The **Axes Scaling** dialog controls the axes limits of the scope. To open the Axes Scaling properties, in the scope menu, select **Tools > Axes Scaling > Axes Scaling Properties**.

Axes scaling — Y-axis scaling mode

Manual (default) | Auto | After N Updates

- **Manual** - Manually scale *y*-axis range with **Scale Y-axis Limits** toolbar button.
- **Auto** - Scale *y*-axis range during and after simulation. Selecting this option displays the “Do not allow Y-axis limits to shrink” on page 1-0 check box.

If you want the *y*-axis range to increase and decrease with the maximum value of a signal, set **Axes scaling** to `Auto` and clear the **Do not allow Y-axis limits to shrink** check box.

- **After N Updates** - Scale *y*-axis after the number of time steps specified in the “Number of updates” on page 1-0 text box (10 by default). Scaling occurs only once during each run.

Programmatic Use

See `AxesScaling`.

Do not allow Y-axis limits to shrink — When y-axis limits can change
on (default) | off

Allow y-axis range limits to increase but not decrease during a simulation.

Dependency

To use this property, set “Axes scaling” on page 1-0 to Auto.

Number of updates — Number of updates before scaling
10 (default) | integer

Set this property to delay auto scaling the y-axis.

Dependency

To use this property, set “Axes scaling” on page 1-0 to After N Updates.

Programmatic Use

See AxesScalingNumUpdates.

Scale axes limits at stop — When y-axis limits can change
on (default) | off

- On - Scale axes when simulation stops.
- Off - Scale axes continually.

Dependency

To use this property, set “Axes scaling” on page 1-0 to Auto.

Y-axis Data range (%) — Percent of y-axis to plot on
80 (default) | integer between [1, 100]

Specify percentage of y-axis range for plotting data. For example, if you set this property to 100, plotted data uses the entire y-axis range.

Y-axis Align — Alignment along y-axis
Center (default) | Top | Bottom

Specify where to align plotted data along the y-axis data range when **Y-axis Data range** is set to less than 100 percent.

- **Top** - Align signals with maximum values at top of *y*-axis range.
- **Center** - Center signals between minimum and maximum values.
- **Bottom** - Align signals with minimum values at bottom of *y*-axis range.

Autoscale X-axis limits — Scale *x*-axis range limits

off (default) | on

Scale *x*-axis range to fit all signal values. If **Axes scaling** is set to `Auto`, scales the data currently within the axes, not the entire signal in the data buffer.

X-axis Data range (%) — Percent of *x*-axis to plot on

100 (default) | integer between [1, 100]

Specify the percentage of the *x*-axis range to plot data on. For example, if you set this property to 100, plotted data uses the entire *x*-axis range.

X-axis Align — Alignment along *x*-axis

Center (default) | Top | Bottom

Specify where to align plotted data along the *x*-axis data range when **X-axis Data range** is set to less than 100 percent.

- **Top** - Align signals with maximum values at top of *x*-axis range.
- **Center** - Center signals between minimum and maximum values.
- **Bottom** - Align signals with minimum values at bottom of *x*-axis range.

Style Properties

To open the Style dialog box, from the scope menu, select **View > Style**.

Figure color — Background color for window

black (default) | color

Background color for the scope

Plot type — How to plot signal

Auto (default for Scope) | Line (default for Time Scope) | Stairs | Stem

When you select `Auto`, the plot type is a line graph for continuous signals, a stair-step graph for discrete signals, and a stem graph for Simulink message signals.

Axes colors — Background and axes color for individual displays

black (default) | color

Select the background color for axes (displays) with the first color palette. Select the grid and label color with the second color palette.

Preserve colors for copy to clipboard — Copy scope without changing colors

off (default) | on

Specify whether to use the displayed color of the scope when copying.

When you select **File > Copy to Clipboard**, the software changes the color of the scope to be printer friendly (white background, visible lines). If you want to copy and paste the scope with the colors displayed, select this check box.

Properties for line — Line to change

Channel 1 (default)

Select active line for setting line style properties.

Visible — Line visibility

on (default) | off

Show or hide a signal on the plot.

Dependency

The values of “Active display” on page 1-0 and “Properties for line” on page 1-0 determine which line is affected.

Line — Line style

style | width | color

Select line style, width, and color.

Dependency

The values of “Active display” on page 1-0 and “Properties for line” on page 1-0 determine which line is affected.

Marker — Data point marker style

None (default) | marker shape

Select marker style.

Dependency

The values of “Active display” on page 1-0 and “Properties for line” on page 1-0 determine which line is affected.

Model Examples

See Also

Floating Scope | Scope Viewer

Topics

“Scope Blocks and Scope Viewer Overview”

“Simulate a Model Interactively”

“Step Through a Simulation”

“Common Scope Interactions”

“Floating Scope and Scope Viewer Tasks”

“Scope Triggers Panel”

“Cursor Measurements Panel”

“Scope Signal Statistics Panel”

“Scope Bilevel Measurements Panel”

“Peak Finder Measurements Panel”

“Control Scopes Programmatically”

Scope Block with Apple iOS Devices

“Scope Block with Android Devices” (Simulink Support Package for Android Devices)

Introduced in R2015b

Second-Order Integrator, Second-Order Integrator Limited

Integrate input signal twice



Library

Continuous

Description

The Second-Order Integrator block and the Second-Order Integrator Limited block solve the second-order initial value problem:

$$\begin{aligned}\frac{d^2x}{dt^2} &= u, \\ \left. \frac{dx}{dt} \right|_{t=0} &= dx_o, \\ x|_{t=0} &= x_o,\end{aligned}$$

where u is the input to the system. The block is therefore a dynamic system with two continuous states: x and dx/dt .

Note These two states have a mathematical relationship, namely, that dx/dt is the derivative of x . In order to satisfy this relationship throughout the simulation, Simulink places various constraints on the block parameters and behavior.

The Second-Order Integrator Limited block is identical to the Second-Order Integrator block with the exception that it defaults to limiting the states based on the specified upper and lower limits. For more information, see “Limiting the States” on page 1-1346.

Simulink software can use a number of different numerical integration methods to compute the outputs of the block. Each has advantages for specific applications. Use the **Solver** pane of the Configuration Parameters dialog box to select the technique best suited to your application. (For more information, see “About Solvers”.) The selected solver computes the states of the Second-Order Integrator block at the current time step using the current input value.

Use the block parameter dialog box to:

- Specify whether the source of each state initial condition is internal or external
- Specify a value for the state initial conditions
- Define upper and lower limits on either or both states
- Specify absolute tolerances for each state
- Specify names for both states
- Choose an external reset condition
- Enable zero-crossing detection
- Reinitialize dx/dt when x reaches saturation
- Specify that Simulink disregard the state limits and external reset for linearization operations

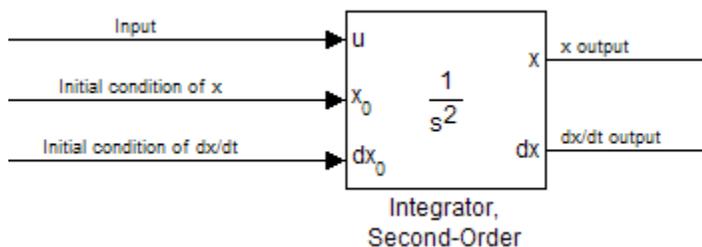
Defining Initial Conditions

You can define the initial conditions of each state individually as a parameter on the block dialog box or input one or both of them from an external signal.

- To define the initial conditions of state x as a block parameter, use the **Initial condition source x** drop-down menu to select `internal` and enter the value in the **Initial condition x** field.
- To provide the initial conditions from an external source for state x , specify the **Initial condition source x** parameter as `external`. An additional input port appears on the block.
- To define the initial conditions of state dx/dt as a block parameter, use the **Initial condition source dx/dt** drop-down menu to select `internal` and enter the value in the **Initial condition dx/dt** field.

- To provide the initial conditions from an external source for state dx/dt , specify **Initial condition source dx/dt** as external. An additional input port appears on the block.

If you choose to use an external source for both state initial conditions, your block appears as follows.



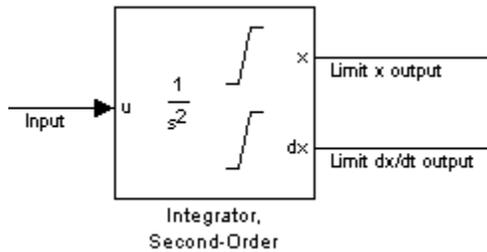
Note

- Simulink does not allow initial condition values of `inf` or `NaN`.
 - If you limit state x or state dx/dt by specifying saturation limits (see “Limiting the States” on page 1-1346) and one or more initial conditions are outside the corresponding limits, then the respective states are initialized to the closest valid value and a set of consistent initial conditions is calculated.
-

Limiting the States

When modeling a second-order system, you may need to limit the block states. For example, the motion of a piston within a cylinder is governed by Newton's Second Law and has constraints on the piston position (x). With the Second-Order Integrator block, you can limit the states x and dx/dt independent of each other. You can even change the limits during simulation; however, you cannot change whether or not the states are limited. An important rule to follow is that an upper limit must be strictly greater than its corresponding lower limit.

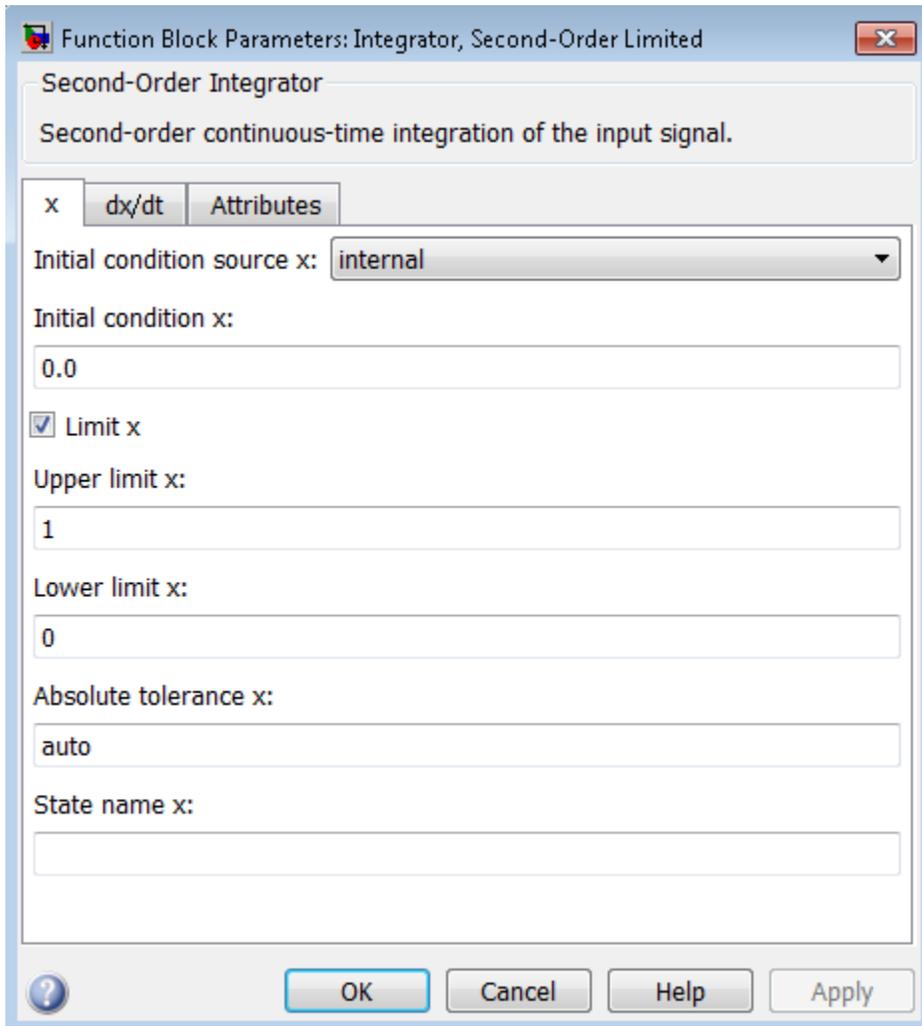
The block appearance changes when you limit one or both states. Shown below is the appearance of the block with both states limited.



For each state, you can use the block parameter dialog box to set appropriate saturation limits.

Limiting x only

If you use the Second-Order Integrator Limited block, both states are limited by default. But you can also manually limit state x on the Second-Order Integrator block by selecting **Limit x** and entering the limits in the appropriate parameter fields.



The block then determines the values of the states as follows:

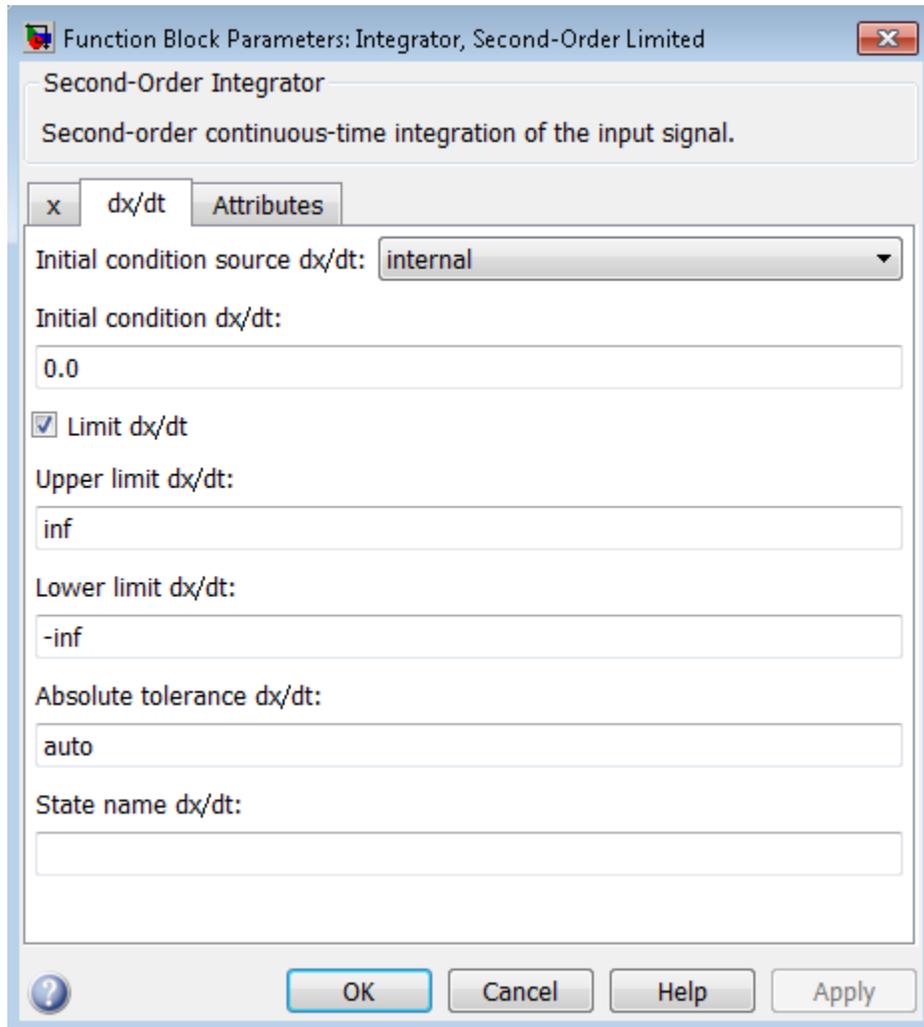
- When x is less than or equal to its lower limit, the value of x is held at its lower limit and dx/dt is set to zero.
- When x is in between its lower and upper limits, both states follow the trajectory given by the second-order ODE.

- When x is greater than or equal to its upper limit, the value of x is held at its upper limit and dx/dt is set to zero.

You can choose to reinitialize dx/dt to a new value at the time when x reaches saturation. See “Reinitializing dx/dt when x reaches saturation” on page 1-1353

Limiting dx/dt only

As with state x , state dx/dt is set as limited by default on the **dx/dt** pane of the Second-Order Integrator Limited parameter dialog box. You can manually set this parameter, **Limit dx/dt** , on the Second-Order Integrator block. In either case, you must enter the appropriate limits for dx/dt .



If you limit only the state dx/dt , then the block determines the values of dx/dt as follows:

- When dx/dt is less than or equal to its lower limit, the value of dx/dt is held at its lower limit.
- When dx/dt is in between its lower and upper limits, both states follow the trajectory given by the second-order ODE.

- When dx/dt is greater than or equal to its upper limit, the value of dx/dt is held at its upper limit.

When state dx/dt is held at its upper or lower limit, the value of x is governed by the first-order initial value problem:

$$\frac{dx}{dt} = L,$$

$$x(t_L) = x_L,$$

where L is the dx/dt limit (upper or lower), t_L is the time when dx/dt reaches this limit, and x_L is the value of state x at that time.

Limiting Both States

When you limit both states, Simulink maintains mathematical consistency of the states by limiting the allowable values of the upper and lower limits for dx/dt . Such limitations are necessary to satisfy the following constraints:

- When x is at its saturation limits, the value of dx/dt must be zero.
- In order for x to leave the upper limit, the value of dx/dt must be strictly negative.
- In order for x to leave its lower limit, the value of dx/dt must be strictly positive.

Consequently, for such cases, the upper limit of dx/dt must be strictly positive and the lower limit of dx/dt must be strictly negative.

When both states are limited, the block determines the states as follows:

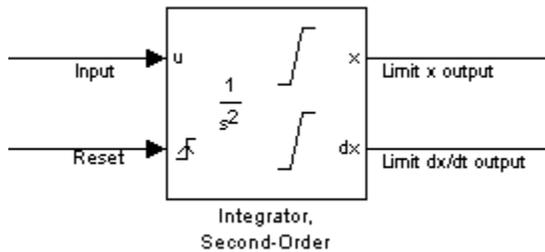
- Whenever x reaches its limits, the resulting behavior is the same as that described in “Limiting x only”.
- Whenever dx/dt reaches one of its limits, the resulting behavior is the same as that described in “Limiting dx/dt only” — including the computation of x using a first-order ODE when dx/dt is held at one of its limits. In such cases, when x reaches one of its limits, it is held at that limit and dx/dt is set to zero.
- Whenever both reach their respective limits simultaneously, the state x behavior overrides dx/dt behavior to maintain consistency of the states.

When you limit both states, you can choose to reinitialize dx/dt at the time when state x reaches saturation. If the reinitialized value is outside specified limits on dx/dt , then

dx/dt is reinitialized to the closest valid value and a consistent set of initial conditions is calculated. See “Reinitializing dx/dt when x reaches saturation” on page 1-1353

Resetting the State

The block can reset its states to the specified initial conditions based on an external signal. To cause the block to reset its states, select one of the **External reset** choices on the **Attributes** pane. A trigger port appears on the block below its input port and indicates the trigger type.



- Select `rising` to reset the states when the reset signal rises from zero to a positive value, from a negative to a positive value, or a negative value to zero.
- Select `falling` to reset the states when the reset signal falls from a positive value to zero, from a positive to a negative value, or from zero to negative.
- Select `either` to reset the states when the reset signal changes from zero to a nonzero value or changes sign.

The reset port has direct feedthrough. If the block output feeds back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results (see “Algebraic Loops”).

Enabling Zero-Crossing Detection

This parameter controls whether zero-crossing detection is enabled for this block. By default, the **Enable zero-crossing detection** parameter is selected on the **Attributes** pane. However, this parameter is only in affect if the **Zero-crossing control**, on the **Solver** pane of the Configuration Parameters dialog, is set to `Use local settings`. For more information, see “Zero-Crossing Detection”.

Reinitializing dx/dt when x reaches saturation

For certain modeling applications, dx/dt must be reinitialized when state x reaches its limits in order to pull x out of saturation immediately. You can achieve this by selecting **Reinitialize dx/dt when x reaches saturation** on the **Attributes** pane.

If this option is on, then at the instant when x reaches saturation, Simulink checks whether the current value of the dx/dt initial condition (parameter or signal) allows the state x to leave saturation immediately. If so, Simulink reinitializes state dx/dt with the value of the initial condition (parameter or signal) at that instant. If not, Simulink ignores this parameter at the current instant and sets dx/dt to zero to make the block states consistent.

This parameter only applies at the time when x actually reaches saturation limit. It does not apply at any future time when x is being held at saturation.

Refer to the sections on limiting the states for more information. For an example, see the `sldemo_bounce` example.

Disregarding State Limits and External Reset for Linearization

For cases where you simplify your model by linearizing it, you can have Simulink disregard the limits of the states and the external reset by selecting **Ignore state limits and the reset for linearization**.

Specifying the Absolute Tolerance for the Block Outputs

By default Simulink software uses the absolute tolerance value specified in the Configuration Parameters dialog box (see “Error Tolerances for Variable-Step Solvers”) to compute the output of the integrator blocks. If this value does not provide sufficient error control, specify a more appropriate value for state x in the **Absolute tolerance x** field and for state dx/dt in the **Absolute tolerance dx/dt** field of the parameter dialog box. Simulink uses the values that you specify to compute the state values of the block.

Specifying the Display of the Output Ports

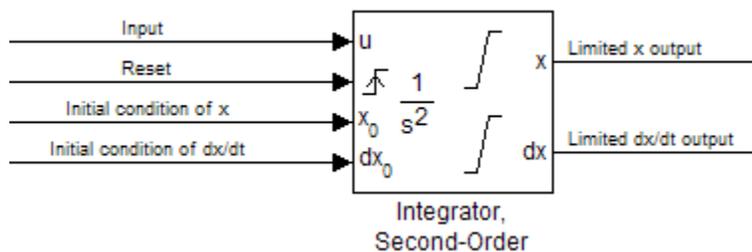
You can control whether or not to display the x or the dx/dt output port using the `ShowOutput` parameter. You can display one output port or both; however, you must select at least one.

Specifying the State Names

You can specify the name of x states and dx/dt states using the `StateNameX` and `StateNameDXDT` parameters. However, you must specify names for either both or neither; you cannot specify names for just x or just dx/dt . Both state names must have identical type and length. Furthermore, the number of names must evenly divide the number of states.

Selecting All Options

When you select all options, the block icon looks like this.



Data Type Support

The Integrator block accepts and outputs signals of type `double` on its data ports. The external reset port accepts signals of type `double` or `Boolean`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

- “Initial condition source x ” on page 1-1355
- “Initial condition x ” on page 1-1356
- “Limit x ” on page 1-1356

- “Upper limit x ” on page 1-1357
- “Lower limit x ” on page 1-1357
- “Wrap x ” on page 1-1358
- “Wrapped upper value x ” on page 1-1359
- “Wrapped lower value x ” on page 1-1359
- “Absolute tolerance x ” on page 1-1359
- “State name x ” on page 1-1360
- “Initial condition source dx/dt ” on page 1-1361
- “Initial condition dx/dt ” on page 1-1361
- “Limit dx/dt ” on page 1-1362
- “Upper limit dx/dt ” on page 1-1363
- “Lower limit dx/dt ” on page 1-1363
- “Absolute tolerance dx/dt ” on page 1-1363
- “State name dx/dt ” on page 1-1364
- “External reset” on page 1-1365
- “Enable zero-crossing detection” on page 1-1365
- “Reinitialize dx/dt when x reaches saturation” on page 1-1366
- “Ignore state limits and the reset for linearization” on page 1-1366
- “Show output” on page 1-1367
- “Characteristics” on page 1-1367

Initial condition source x

Specify the initial condition source for state x .

Default: `internal`

`internal`

Get the initial conditions of state x from the **Initial condition x** parameter.

`external`

Get the initial conditions of state x from an external block.

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Selecting `internal` enables the **Initial condition x** parameter.

Selecting `external` disables the **Initial condition x** parameter..

Parameter: `ICSourceX`

Type: character vector

Value: `'internal' | 'external'`

Default: `'internal'`

Initial condition x

Specify the initial condition of state x .

Default: `0.0`

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Setting **Initial condition source x** to `internal` enables this parameter.

Setting **Initial condition source x** to `external` disables this parameter.

Parameter: `ICX`

Type: scalar or vector

Value: `'0'`

Default: `'0'`

Limit x

Limit state x of the block to a value between the **Lower limit x** and **Upper limit x** parameters.

Default: `Off` for Second-Order Integrator, `On` for Second-Order Integrator Limited

On

Limit state x to a value between the **Lower limit x** and **Upper limit x** parameters.

Off

Do not limit the state x output to a value between the **Lower limit x** and **Upper limit x** parameters.

This parameter enables **Upper limit x** parameter.

This parameter enables **Lower limit x** parameter.

Parameter: LimitX

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Upper limit x

Specify the upper limit of state x .

Default: `inf` for Second-Order Integrator, 1 for Second-Order Integrator Limited

The upper saturation limit for state x must be strictly greater than the lower saturation limit.

Limit x enables this parameter.

Parameter: UpperLimitX

Type: scalar or vector

Value: 'inf'

Default: 'inf'

Lower limit x

Specify the lower limit of state x .

Default: $-\infty$ for Second-Order Integrator, 0 for Second-Order Integrator Limited

The lower saturation limit for state x must be strictly less than the upper saturation limit.

Limit x enables this parameter.

Parameter: LowerLimitX

Type: scalar or vector

Value: $-\infty$

Default: $-\infty$

Wrap x

Enable wrapping of x between the **Wrapped upper value x** and **Wrapped lower value x** parameters. Enabling wrapping of x eliminates the need for zero-crossing detection, reduces solver resets, improves solver performance and accuracy, and increases simulation time span when modeling rotary and cyclic state trajectories.

Default: off

On

Enable wrapping of x between the **Wrapped upper value x** and **Wrapped lower value x** parameters.

If you specify **Wrapped upper value x** as ∞ and **Wrapped lower value x** as $-\infty$, wrapping will never occur.

Off

Do not enable wrapping of x .

This parameter enables **Wrapped upper value x** .

This parameter enables **Wrapped lower value x** .

Parameter: WrapX
Type: character vector
Value: 'off' | 'on'
Default: 'off'

Wrapped upper value x

Specify the upper value for wrapping x .

Default: 'pi'

Wrap x enables this parameter.

Parameter: WrappedUpperValueX
Type: scalar or vector
Value: '2*pi'
Default: 'pi'

Wrapped lower value x

Specify the lower value for wrapping x .

Default: -pi

Wrap x enables this parameter.

Parameter: WrappedLowerValueX
Type: scalar or vector
Value: '0'
Default: '-pi'

Absolute tolerance x

Specify the absolute tolerance for computing state x .

Default: auto

- You can enter `auto`, `-1`, a positive real scalar or vector.
- If you enter `auto` or `-1`, Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute state x .
- If you enter a real scalar value, that value overrides the absolute tolerance in the Configuration Parameters dialog box and is used for computing all x states.
- If you enter a real vector, the dimension of that vector must match the dimension of state x . These values override the absolute tolerance in the Configuration Parameters dialog box.

Parameter: `AbsoluteToleranceX`

Type: character vector, scalar, or vector

Value: `'auto'` | `'-1'` | any positive real scalar or vector

Default: `'auto'`

State name x

Assign a unique name to state x .

Default: `''`

- To assign a name to a single state, enter the name between quotes, for example, `'position'`.
- To assign names to multiple x states, enter a comma-delimited list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- The state names apply only to the selected block.
- If you specify a state name for x , you must also specify a state name for dx/dt .
- State names for x and dx/dt must have identical types and lengths.
- The number of states must be evenly divided by the number of state names. You can specify fewer names than x states, but you cannot specify more names than x states. For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states. However, you must be consistent and apply the same scheme to the state names for dx/dt .
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector or a cell array.

Parameter: StateNameX
Type: character vector
Value: ' ' | user-defined
Default: ' '

Initial condition source dx/dt

Specify the initial condition source for state dx/dt .

Default: `internal`

`internal`

Get the initial conditions of state dx/dt from the **Initial condition dx/dt** parameter.

`external`

Get the initial conditions of state dx/dt from an external block.

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Selecting `internal` enables the **Initial condition dx/dt** parameter.

Selecting `external` disables the **Initial condition dx/dt** parameter.

Parameter: ICSourceDXDT
Type: character vector
Value: 'internal' | 'external'
Default: 'internal'

Initial condition dx/dt

Specify the initial condition of state dx/dt .

Default: 0.0

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

Setting **Initial condition source dx/dt** to `internal` enables this parameter.

Setting **Initial condition source dx/dt** to `external` disables this parameter.

Parameter: ICDXDT

Type: scalar or vector

Value: '0'

Default: '0'

Limit dx/dt

Limit the dx/dt state of the block to a value between the **Lower limit dx/dt** and **Upper limit dx/dt** parameters.

Default: `Off` for Second-Order Integrator, `On` for Second-Order Integrator Limited

On

Limit state dx/dt of the block to a value between the **Lower limit dx/dt** and **Upper limit dx/dt** parameters.

Off

Do not limit state dx/dt of the block to a value between the **Lower limit dx/dt** and **Upper limit dx/dt** parameters.

If you set saturation limits for x , then the interval defined by the **Upper limit dx/dt** and **Lower limit dx/dt** must contain zero.

This parameter enables **Upper limit dx/dt**.

This parameter enables **Lower limit dx/dt**.

Parameter: LimitDXDT

Type: character vector

Value: 'Off' | 'On'

Default: 'Off'

Upper limit dx/dt

Specify the upper limit for state dx/dt .

Default: 'inf'

If you limit x , then this parameter must have a strictly positive value.

Limit dx/dt enables this parameter.

Parameter: UpperLimitDXDT

Type: scalar or vector

Value: 'inf'

Default: 'inf'

Lower limit dx/dt

Specify the lower limit for state dx/dt .

Default: '-inf'

If you limit x , then this parameter must have a strictly negative value.

Limit dx/dt enables this parameter.

Parameter: LowerLimitDXDT

Type: scalar or vector

Value: '-inf'

Default: '-inf'

Absolute tolerance dx/dt

Specify the absolute tolerance for computing state dx/dt .

Default: auto

- You can enter `auto`, `-1`, a positive real scalar or vector.
- If you enter `auto` or `-1`, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute the dx/dt output of the block.
- If you enter a numeric value, that value overrides the absolute tolerance in the Configuration Parameters dialog box.

Parameter: `AbsoluteToleranceDXDT`

Type: character vector, scalar, or vector

Value: `'auto'` | `'-1'` | any positive real scalar or vector

Default: `'auto'`

State name dx/dt

Assign a unique name to state dx/dt .

Default: `''`

- To assign a name to a single state, enter the name between quotes, for example, `'velocity'`.
- To assign names to multiple dx/dt states, enter a comma-delimited list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- The state names apply only to the selected block.
- If you specify a state name for dx/dt , you must also specify a state name for x .
- State names for x and dx/dt must have identical types and lengths.
- The number of states must be evenly divided by the number of state names. You can specify fewer names than dx/dt states, but you cannot specify more names than dx/dt states. For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states. However, you must be consistent and apply the same scheme to the state names for x .
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector or a cell array.

Parameter: `StateNameDXDT`

Type: character vector
Value: ' ' | user-defined
Default: ' '

External reset

Reset the states to their initial conditions when a trigger event occurs in the reset signal.

Default: none

none

Do not reset the state to initial conditions.

rising

Reset the state when the reset signal rises from a zero to a positive value or from a negative to a positive value.

falling

Reset the state when the reset signal falls from a positive value to zero or from a positive to a negative value.

either

Reset the state when the reset signal changes from zero to a nonzero value or changes sign.

Parameter: ExternalReset

Type: character vector

Value: 'none' | 'rising' | 'falling' | 'either'

Default: 'none'

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Default: On

On

Enable zero-crossing detection.

Off

Do not enable zero-crossing detection.

Parameter: ZeroCross

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Reinitialize dx/dt when x reaches saturation

At the instant when state x reaches saturation, reset dx/dt to its current initial conditions.

Default: Off

On

Reset dx/dt to its initial conditions when x becomes saturated.

Off

Do not reset dx/dt to its initial conditions when x becomes saturated.

The dx/dt initial condition must have a value that enables x to leave saturation immediately. Otherwise, Simulink ignores the initial conditions for dx/dt to preserve mathematical consistency of block states.

Parameter: ReinitDXDTwhenXreachesSaturation

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Ignore state limits and the reset for linearization

For linearization purposes, have Simulink ignore the specified state limits and the external reset.

Default: Off

On

Ignore the specified state limits and the external reset.

Off

Apply the specified state limits and the external reset setting.

Parameter: IgnoreStateLimitsAndResetForLinearization

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Show output

Specify the output ports on the block.

Default: both

both

Show both x and dx/dt output ports.

x

Show only the x output port.

dx/dt

Show only the dx/dt output port.

Parameter: ShowOutput

Type: character vector

Value: 'both' | 'x' | 'dxdt'

Default: 'both'

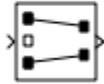
Characteristics

Data Types	Double
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

Introduced in R2010a

Selector

Select input elements from vector, matrix, or multidimensional signal



Library

Signal Routing

Description

The Selector block generates as output selected or reordered elements of an input vector, matrix, or multidimensional signal.

A Selector block accepts vector, matrix, or multidimensional signals as input. The parameter dialog box and the block's appearance change to reflect the number of dimensions of the input.

Based on the value you enter for the **Number of input dimensions** parameter, a table of indexing settings is displayed. Each row of the table corresponds to one of the input dimensions in **Number of input dimensions**. For each dimension, you define the elements of the signal to work with. Specify a vector signal as a 1-D signal and a matrix signal as a 2-D signal. When you configure the Selector block for multidimensional signal operations, the block icon changes.

For example, assume a 6-D signal with a one-based index mode. The table of the Selector block dialog changes to include one row for each dimension. Suppose that you define each dimension with the following entries:

- Dimension 1

- **Index Option**, select `Select all`
- Dimension 2
 - **Index Option**, select `Starting index (dialog)`
 - **Index**, enter `2`
 - **Output Size**, enter `5`
- Dimension 3
 - **Index Option**, select `Index vector (dialog)`
 - **Index**, enter `[1 3 5]`
- Dimension 4
 - **Index Option**, select `Starting index (port)`
 - **Output Size**, enter `8`
- Dimension 5
 - **Index Option**, select `Index vector (port)`
- Dimension 6
 - **Index Option**, select `Starting and ending indices (port)`

The output will be $Y=U(1:end, 2:6, [1\ 3\ 5], Idx4:Idx4+7, Idx5, Idx6(1):Idx6(2))$, where $Idx4$, $Idx5$, and $Idx6$ are the index ports for dimensions 4, 5, and 6.

You can use an array of buses as an input signal to a Selector block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Data Type Support

The data port of the Selector block accepts signals of any signal type and any data type that Simulink supports, including fixed-point, enumerated, and nonvirtual bus data types. The data port accepts mixed-type signals. The index port accepts built-in data types, but not Boolean. The elements of the output have the same type as the corresponding selected input elements.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Note The Selector block is not designed to accept virtual bus input signals. For virtual bus inputs, use a Bus Selector block instead of a Selector block.

Parameters

Number of input dimensions

Enter the number of dimensions of the input signal.

Index mode

Specifies the indexing mode: `One-based` or `Zero-based`. If `One-based` is selected, an index of 1 specifies the first element of the input vector, 2, the second element, and so on. If `Zero-based` is selected, an index of 0 specifies the first element of the input vector, 1, the second element, and so on.

Index Option

Define, by dimension, how the elements of the signal are to be indexed. From the list, select:

- `Select all`

No further configuration is required. All elements are selected.

- `Index vector (dialog)`

Enables the **Index** column. Enter the vector of indices of the elements.

- `Index vector (port)`

No further configuration is required.

- `Starting index (dialog)`

Enables the **Index** and **Output Size** columns. Enter the starting index of the range of elements to select in the **Index** column and the number of elements to select in the **Output Size** column.

- `Starting index (port)`

Enables the **Output Size** column. Enter the number of elements to be selected in the **Output Size** column.

- Starting and ending indices (port)

No further configuration is required.

Using this option results in a variable-size output signal. When you update, the output dimension is set to be the same as the input signal dimension. During execution, the output dimension is updated based on the signal feeding the index.

When logging output signal data, signals not selected are padded with NaN values.

The **Index** and **Output Size** columns appear as needed.

Index

If the **Index Option** is `Index vector (dialog)`, enter the index of each element in which you are interested.

If the **Index Option** is `Starting index vector (dialog)`, enter the starting index of the range of elements to be selected.

Output Size

Enter the width (number of elements from the starting point) of the block output signal.

Input port size

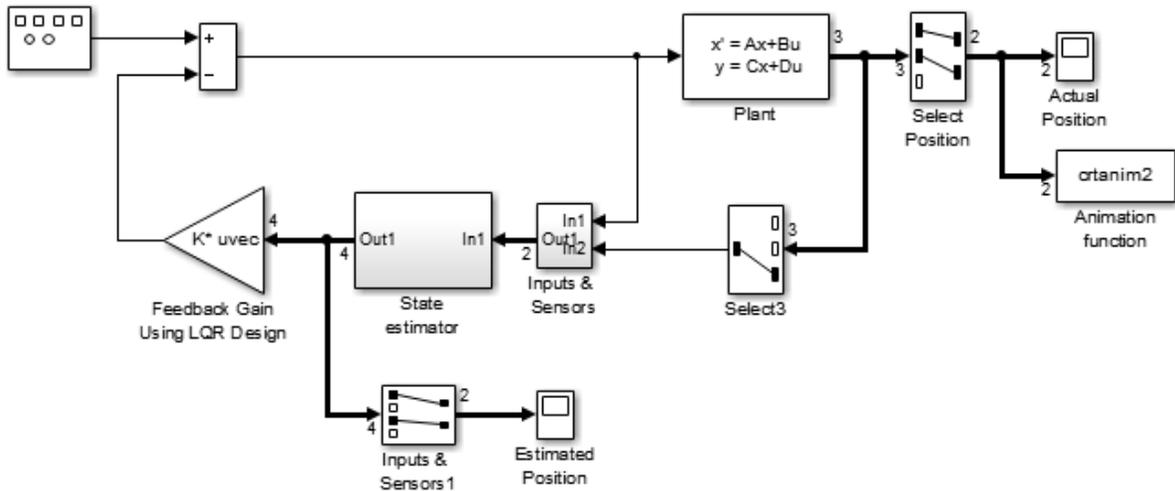
Specify the width of the block input signal (-1 for inherited) — 1-D signals only.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Examples

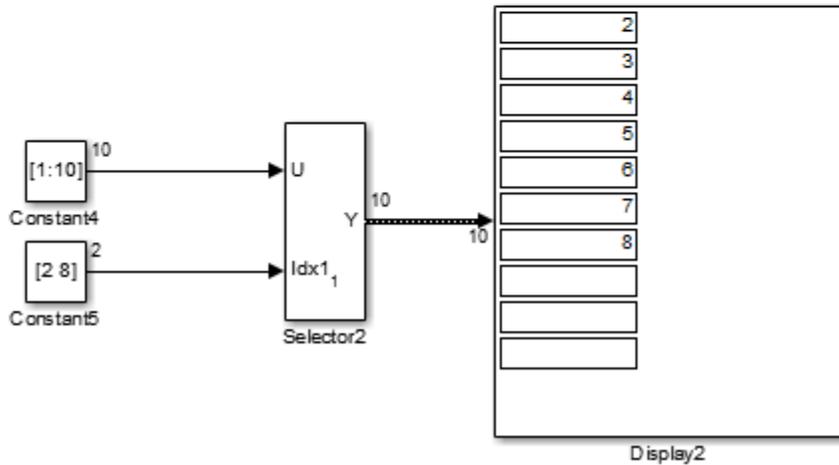
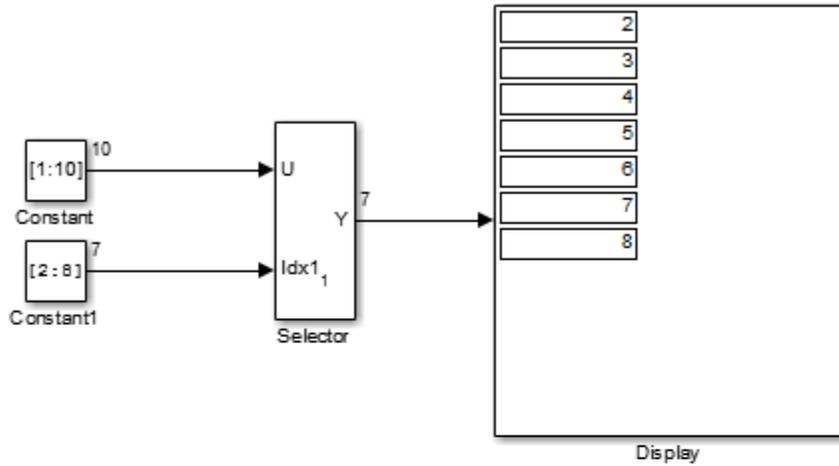
The `sldemo_dbldcart1` model shows the use of three Selector blocks. The following diagram shows what the model looks like after you enable **Display > Signals & Ports > Signal Dimensions** and simulate the model.



All three Selector blocks set the **Index Option** to `Index vector` (dialog), which allows you to specify the indices of the specific signals that you want to select, using the **Index** parameter. The **Input port size** parameter is set to the dimension of the largest input signal.

The following model shows the result of simulating two Selector blocks that have the same kind of input signals, but which have two different **Index Option** settings.

Both Selector blocks select 7 values from the input signal that feeds the `U` port. However, the Selector block outputs a fixed-size signal, whereas the Selector2 block outputs a variable-size signal whose compiled signal dimension is 10 instead of 7.



The Selector block sets **Index Option** to Index vector (port), which uses the input signal from Constant1 as the index vector. The dimension of the input signal is 7, so the Display block shows the 7 values of the Constant1 block. The Selector2 block

sets the **Input port size** parameter to 10, which is the size of the largest input signal to the Selector1 block.

The Selector2 block uses the same configuration as the Selector block, except that the **Index Option** is set to Starting and ending indices (port). The output uses the size of **Input port size** parameter (10), even though the size of the input signal is 7. The Display2 block shows empty boxes for the three extra dimensions.

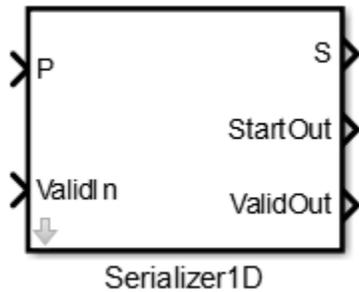
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Serializer1D

Convert vector signal to scalar or smaller vectors



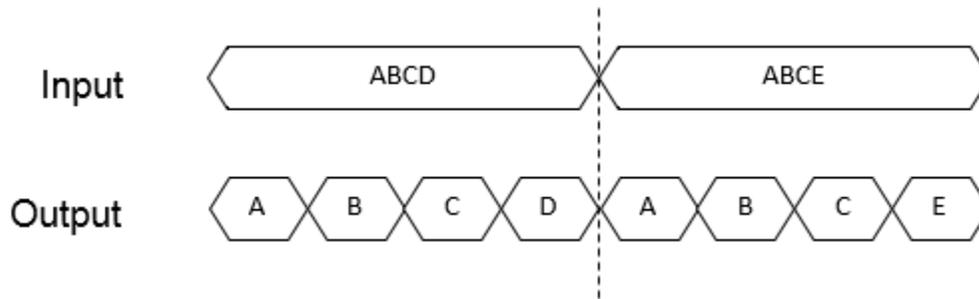
Library

HDL Coder / HDL Operations

Description

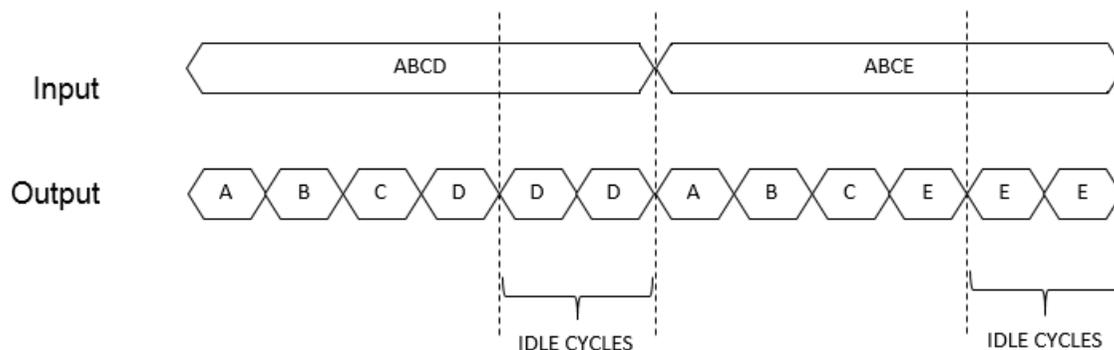
The Serializer1D block converts a slower vector signal into a faster stream of scalar signals or smaller size vector signals based on the **Ratio** and **Idle Cycle** values. To match the faster serialized output, the sample time changes according to this equation:
Output Sample Time = Input Sample Time / (Ratio + Idle Cycles)

Consider this example where the input data is a vector of size 4 and the **Ratio** is set to 4.



The output data serializes each of the vector signals into four scalar signals. The sample time at the output is: $Output\ Sample\ Time = Input\ Sample\ Time / 4$.

To add idle cycles at the end of each output, for **Idle Cycles**, specify an integer greater than zero. Consider this example with **Ratio** set to 4 and **Idle Cycles** set to 2.



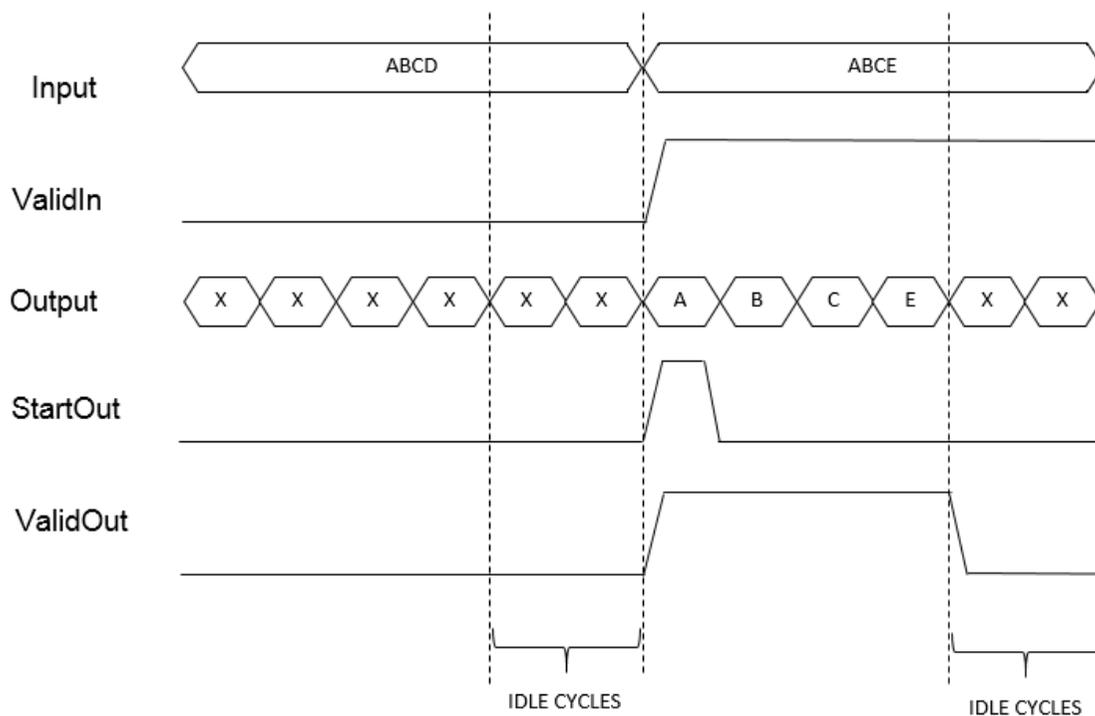
For each slow vector signal, the output has six fast cycles consisting of the four serialized scalar signals and two idle cycles. The sample time at the output is

$$Output\ Sample\ Time = Input\ Sample\ Time / 6$$

The Serializer1D block provides three control signals: **ValidIn**, **ValidOut**, and **StartOut**. You can use **ValidIn** to control **ValidOut** and **StartOut**. The serialized output does not depend on **ValidIn**. To determine whether the output serialized data is

valid, use **ValidIn** and **ValidOut**. If you give a high input to **ValidIn**, and if there are no idle cycles, **ValidOut** gives a high output, which indicates that the output serialized data is valid.

Consider an example that has input data as a vector of size 4, **Ratio** set to 4, **Idle Cycles** set to 2, and uses all three control signals.



For the first input vector, **ABCD**, **ValidIn** is false. **StartOut** and **ValidOut** become false. This means that the output data values are not valid. In the waveform, the data values are represented as **X**, which correspond to *don't care* values.

For the second input vector, **ABCE**, **ValidIn** is true. The output data serializes the vector into four scalar signals. The control signal **StartOut** becomes true at output **A** to indicate the start of deserialization. In the next cycle, the **StartOut** signal becomes false. **ValidOut** is true for all four output signals indicating valid output data for the four

cycles. **ValidOut** becomes false for the idle cycles, and the output data values are *don't care* values.

HDL Code Generation

For simulation results that match the generated HDL code, in the Solver pane of the Configuration Parameters dialog box, clear the checkbox for **Treat each discrete rate as a separate task**. When the checkbox is cleared, single-tasking mode is enabled.

If you simulate this block with **Treat each discrete rate as a separate task** selected, multitasking mode is enabled. The output data can update in the same cycle but in the generated HDL code, the output data is updated one cycle later.

Parameters

Ratio

Serialization factor, specified as a positive scalar. Default is 1.

The ratio is equal to the size of the input vector divided by the size of the output vector. Input vector size must be divisible by the ratio.

Idle Cycles

Number of idle cycles to add at the end of each output. Default is 0.

ValidIn

Activates the **ValidIn** port. Default is `off`.

StartOut

Activates the **StartOut** port. Default is `off`.

ValidOut

Activates the **ValidOut** port. Default is `off`.

Input data port dimensions (-1 for inherited)

Size of the input data signal. Input vector size must be divisible by the ratio. By default, the block inherits size based on the context within the model.

Input sample time (-1 for inherited)

Time interval between sample time hits, or another appropriate sample time such as continuous. By default, the block inherits sample time based on context within the model. For more information, see “Sample Time”.

Input signal type

Input signal type of the block, specified as `auto`, `real`, or `complex`. Default is `auto`.

Ports

P

Input signal to serialize. Bus data types are not supported.

ValidIn

Input control signal. This port is available when you select the **ValidIn** check box.

Data type: Boolean

S

Serialized output signal. Bus data types are not supported.

StartOut

Output control signal that indicates where to start deserialization. You can use this signal as the **StartIn** input to the Deserializer1D block. To use this port, select the **StartOut** check box.

Data type: Boolean

ValidOut

Output control signal that indicates valid output signal. You can use this signal as the **ValidIn** input to the Deserializer1D block. This port is available when you select the **ValidOut** check box.

Data type: Boolean

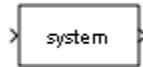
See Also

Deserializer1D

Introduced in R2014b

S-Function

Include S-function in model



Library

User-Defined Functions

Description

The S-Function block provides access to S-functions from a block diagram. The S-function named as the **S-function name** parameter can be a Level-1 MATLAB or a Level-1 or Level-2 C MEX S-function (see “S-Function Basics” for information on how to create S-functions).

Note Use the Level-2 MATLAB S-Function block to include a Level-2 MATLAB S-function in a block diagram.

The S-Function block allows additional parameters to be passed directly to the named S-function. The function parameters can be specified as MATLAB expressions or as variables separated by commas. For example,

```
A, B, C, D, [eye(2,2);zeros(2,2)]
```

Note that although individual parameters can be enclosed in brackets, the list of parameters must not be enclosed in brackets.

The S-Function block displays the name of the specified S-function and the number of input and output ports specified by the S-function. Signals connected to the inputs must have the dimensions specified by the S-function for the inputs.

Data Type Support

Depends on the implementation of the S-Function block.

Parameters

S-function name

The S-function name.

S-function parameters

Additional S-function parameters. See the preceding block description for information on how to specify the parameters.

S-function modules

This parameter applies only if this block represents a C MEX S-function and you intend to use the Simulink Coder software to generate code from the model containing the block. If you use it, when you are ready to generate code, you must force the coder to rebuild the top model as explained in “Control Regeneration of Top Model Code” (Simulink Coder).

For more information on using this parameter, see “Specify Additional Source Files for an S-Function” (Simulink Coder) in the Simulink Coder documentation.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Depends on contents of S-function
Direct Feedthrough	Depends on contents of S-function
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

S-Function Builder

Create S-function from C code that you provide



Library

User-Defined Functions

Description

The S-Function Builder block creates a C MEX S-function from specifications and C source code that you provide. See “Build S-Functions Automatically” for detailed instructions on using the S-Function Builder block to generate an S-function.

Instances of the S-Function Builder block also serve as wrappers for generated S-functions in Simulink models. When simulating a model containing instances of an S-Function Builder block, Simulink software invokes the generated S-function associated with each instance to compute the instance's output at each time step.

Note The S-Function Builder block does not support masking. However, you can mask a Subsystem block that contains an S-Function Builder block. For more information, see “Dynamic Masked Subsystem”.

Data Type Support

The S-Function Builder can accept and output complex, 1-D, or 2-D signals and nonvirtual buses. For each of these cases, the signals must have a data type that Simulink supports.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

See “S-Function Builder Dialog Box” in the online documentation for information on using the S-Function Builder block's parameter dialog box.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

Introduced before R2006a

Shift Arithmetic

Shift bits or binary point of signal

$$\begin{array}{l} Q_y = Q_u \gg 8 \\ V_y = V_u * 2^{-8} \\ E_y = E_u \end{array}$$

Library

Logic and Bit Operations

Description

Supported Shift Operations

The Shift Arithmetic block can shift the bits or the binary point of an input signal, or both.

For example, shifting the binary point on an input of data type `sfixed(8)`, by two places to the right and left, gives these decimal values.

Shift Operation	Binary Value	Decimal Value
No shift (original number)	11001.011	-6.625
Binary point shift right by two places	1100101.1	-26.5
Binary point shift left by two places	110.01011	-1.65625

This block performs arithmetic bit shifts on signed numbers. Therefore, the block recycles the most significant bit for each bit shift. Shifting the bits on an input of data type `sfixed(8)`, by two places to the right and left, gives these decimal values.

Shift Operation	Binary Value	Decimal Value
No shift (original number)	11001.011	-6.625
Bit shift right by two places	11110.010	-1.75

Shift Operation	Binary Value	Decimal Value
Bit shift left by two places	00101.100	5.5

Data Type Support

The block supports input signals of the following data types:

Input Signal	Supported Data Types
u	<ul style="list-style-type: none"> Floating point Built-in integer Fixed point
s	<ul style="list-style-type: none"> Floating point Built-in integer Fixed-point integer

The following rules determine the output data type:

Data Type of Input u	Output Data Type
Floating point	Same as input u
Built-in integer or fixed point	<ul style="list-style-type: none"> Sign of u Word length of u Slope of $u * 2^{(\max(\text{binary points to shift}))}$ Bias of $u * 2^{(\max(\text{binary points to shift} - \text{bits to shift}))}$, for bit shifts where the direction is bidirectional or right Bias of $u * 2^{(\max(\text{binary points to shift} + \text{bits to shift}))}$, for bit shifts where the direction is left

The block parameters support the following data types:

Parameter	Supported Data Types
Bits to shift: Number	<ul style="list-style-type: none"> Built-in integer Fixed-point integer

Parameter	Supported Data Types
Binary points to shift	<ul style="list-style-type: none"> Built-in integer Fixed-point integer

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Bits to shift: Source

Specify whether to enter the bits to shift on the dialog box or to inherit the values from an input port.

Bits to shift: Direction

Specify the direction in which to shift bits: left, right, or bidirectional.

Bits to shift: Number

Specify a scalar, vector, or array of bit shift values. This parameter is available when **Bits to shift: Source** is Dialog.

If the direction is...	Then...
Left or Right	Use positive integers to specify bit shifts.
Bidirectional	Use positive integers for right shifts and negative integers for left shifts.

Binary points to shift

Specify an integer number of places to shift the binary point of the input signal. A positive value indicates a right shift, while a negative value indicates a left shift.

Diagnostic for out-of-range shift value

Specify whether to produce a warning or error during simulation when the block contains an out-of-range shift value. Options include:

- `None` — Simulink software takes no action.
- `Warning` — Simulink software displays a warning and continues the simulation.
- `Error` — Simulink software terminates the simulation and displays an error

For more information, see “Simulation and Accelerator Mode Results for Out-of-Range Bit Shift Values” on page 1-1390.

Check for out-of-range 'Bits to shift' in generated code

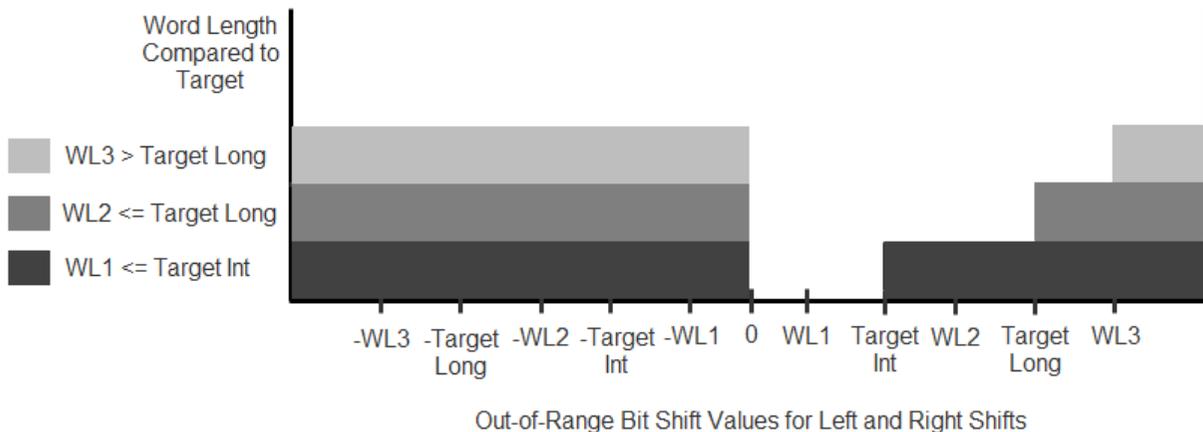
Select this check box to include conditional statements in the generated code that protect against out-of-range bit shift values. This check box is available when **Bits to shift: Source** is `Input port`.

For more information, see “Code Generation for Out-of-Range Bit Shift Values” on page 1-1392.

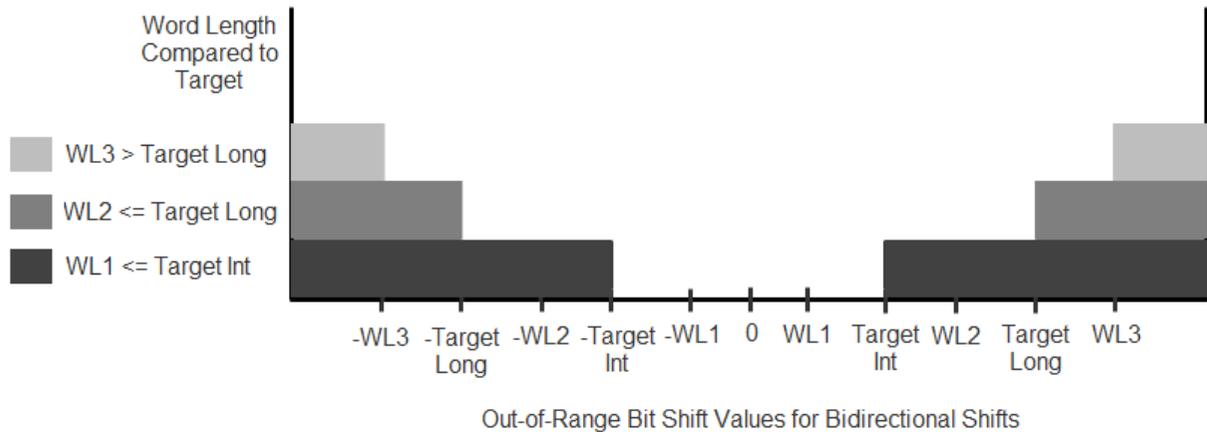
Out-of-Range Bit Shifts

Definition of an Out-of-Range Bit Shift

Suppose that WL is the input word length. The shaded regions in the following diagram show out-of-range bit shift values for left and right shifts.



Similarly, the shaded regions in the following diagram show out-of-range bit shift values for bidirectional shifts.

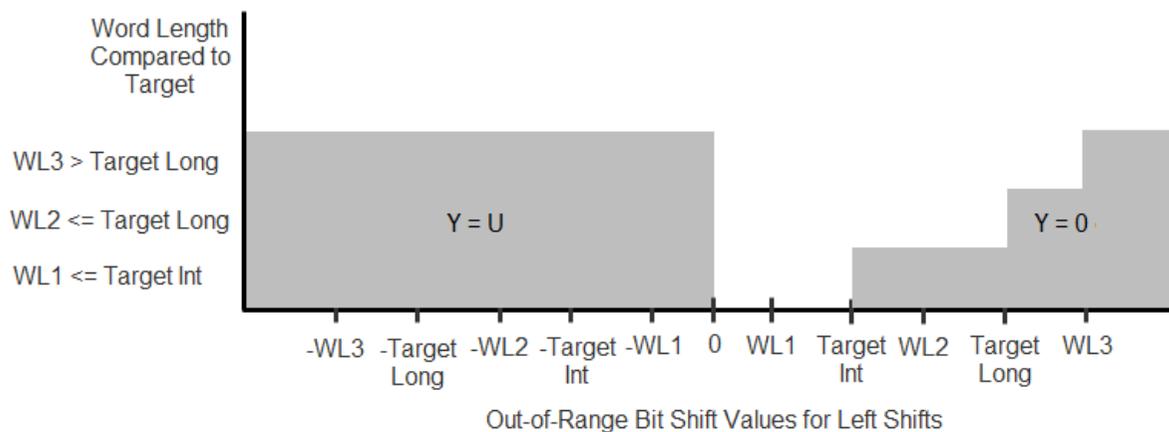


The diagnostic for out-of-range bit shifts responds as follows, depending on the mode of operation:

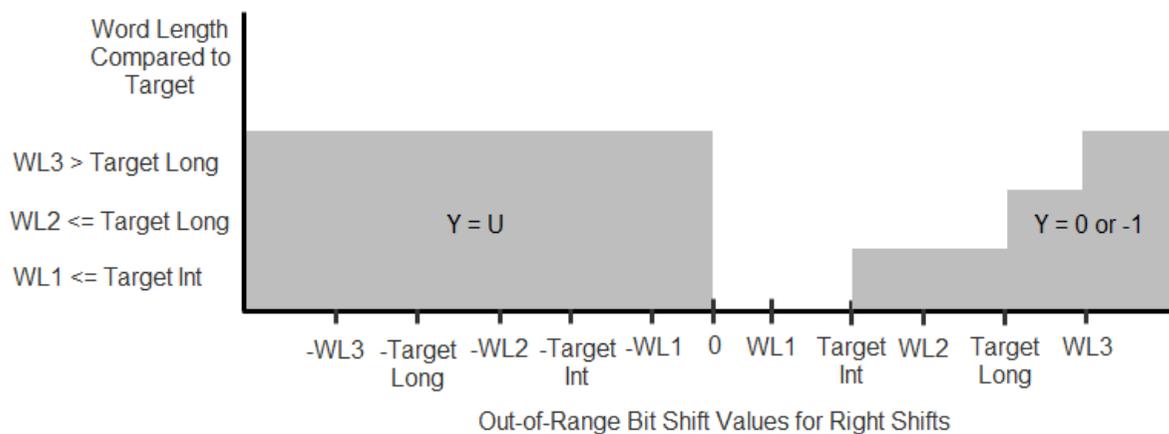
Mode	Diagnostic for out-of-range shift value		
	None	Warning	Error
Simulation	Do not report any warning or error.	Report a warning but continue simulation.	Report an error and stop simulation.
Accelerator modes and code generation	Has no effect.		

Simulation and Accelerator Mode Results for Out-of-Range Bit Shift Values

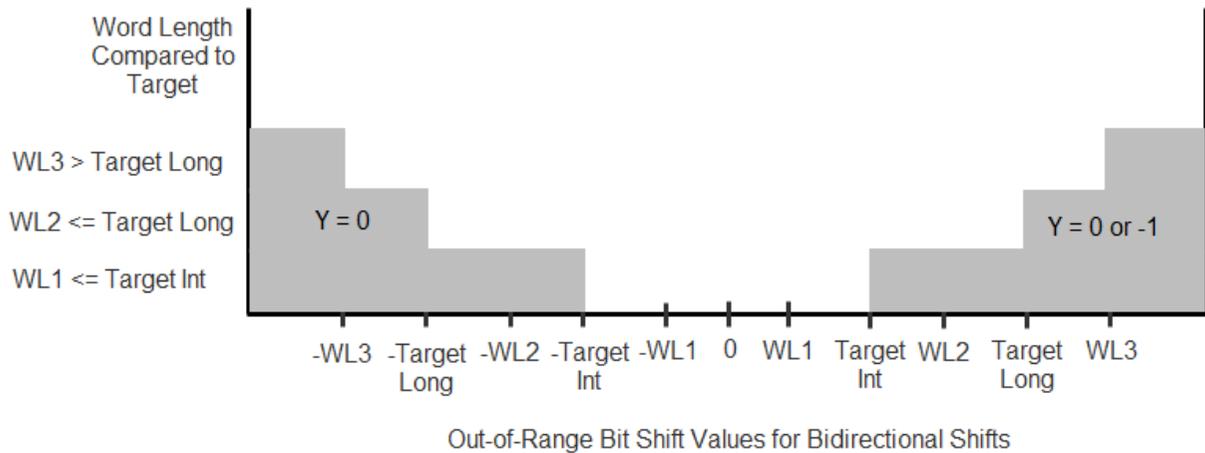
Suppose that U is the input, WL is the input word length, and Y is the output. The output for an out-of-range bit shift value for left shifts is as follows:



Similarly, the output for an out-of-range bit shift value for right shifts is as follows:



For bidirectional shifts, the output for an out-of-range bit shift value is as follows:



Code Generation for Out-of-Range Bit Shift Values

For the generated code, the method for handling out-of-range bit shifts depends on the setting of **Check for out-of-range 'Bits to shift' in generated code**.

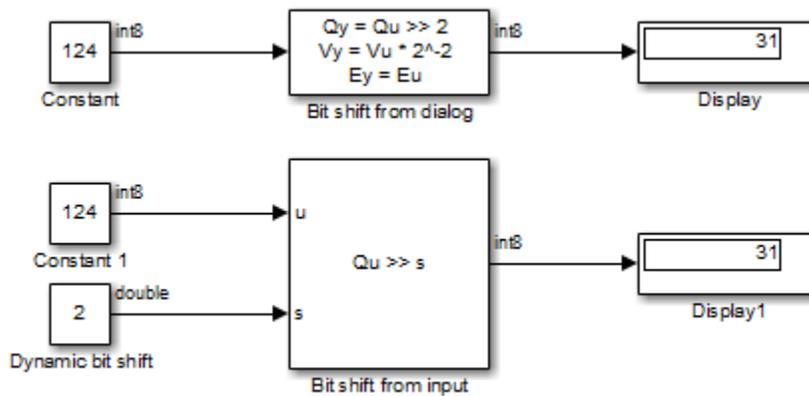
Check Box Setting	Generated Code	Simulation Results Compared to Generated Code
Selected	Includes conditional statements to protect against out-of-range bit shift values.	Simulation and Accelerator mode results match those of code generation.
Cleared	Does not protect against out-of-range bit shift values.	<ul style="list-style-type: none"> For in-range values, simulation and Accelerator mode results match those of code generation. For out-of-range values, the code generation results are compiler specific.

For right shifts on signed negative inputs, most C compilers use an arithmetic shift instead of a logical shift. Generated code for the Shift Arithmetic block depends on this compiler behavior.

Examples

Block Output for Right Bit Shifts

The following model compares the behavior of right bit shifts using the dialog box versus the block input port.



The key block parameter settings of the Constant blocks are:

Block	Parameter	Setting
Constant and Constant1	Constant value	124
	Output data type	int8
Dynamic bit shift	Constant value	2
	Output data type	Inherit: Inherit from 'Constant value'

The key block parameter settings of the Shift Arithmetic blocks are:

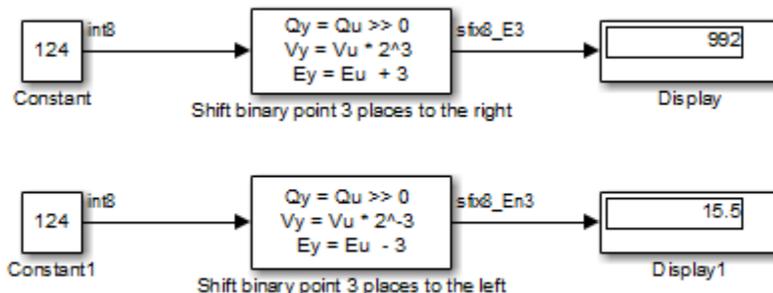
Block	Parameter	Setting
Bit shift from dialog	Bits to shift: Source	Dialog
	Bits to shift: Direction	Right
	Bits to shift: Number	2
Bit shift from input	Bits to shift: Source	Input port
	Bits to shift: Direction	Right

The top Shift Arithmetic block takes an input of 124, which corresponds to 01111100 in binary format. Shifting the number of bits two places to the right produces 00011111 in binary format. Therefore, the block outputs 31.

The bottom Shift Arithmetic block performs the same operation as the top block. However, the bottom block receives the bit shift value through an input port instead of the dialog box. By supplying this value as an input signal, you can change the number of bits to shift during simulation.

Block Output for Binary Point Shifts

The following model shows the effect of binary point shifts.



The key block parameter settings of the Constant blocks are:

Block	Parameter	Setting
Constant and Constant1	Constant value	124
	Output data type	int8

The key block parameter settings of the Shift Arithmetic blocks are:

Block	Parameter	Setting
Shift binary point 3 places to the right	Bits to shift: Source	Dialog
	Bits to shift: Direction	Bidirectional
	Bits to shift: Number	0
	Binary points to shift: Number	3

Block	Parameter	Setting
Shift binary point 3 places to the left	Bits to shift: Source	Dialog
	Bits to shift: Direction	Bidirectional
	Bits to shift: Number	0
	Binary points to shift: Number	-3

The top Shift Arithmetic block takes an input of 124, which corresponds to 01111100 in binary format. Shifting the binary point three places to the right produces 01111100000 in binary format. Therefore, the top block outputs 995.

The bottom Shift Arithmetic block also takes an input of 124. Shifting the binary point three places to the left produces 01111.100 in binary format. Therefore, the bottom block outputs 15.5.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Sign

Indicate sign of input



Library

Math Operations

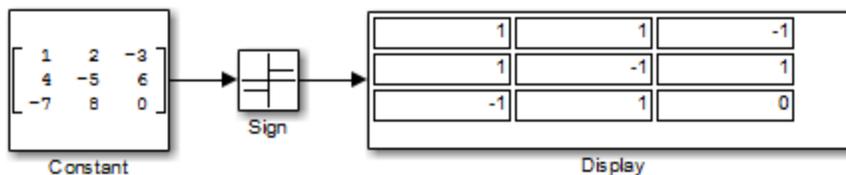
Description

Block Behavior for Real Inputs

For real inputs, the Sign block outputs the sign of the input:

Input	Output
Greater than zero	1
Equal to zero	0
Less than zero	-1

For vector and matrix inputs, the block outputs a vector or matrix where each element is the sign of the corresponding input element, as shown in this example:

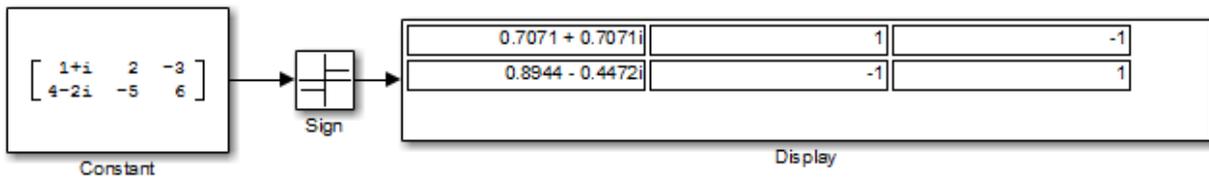


Block Behavior for Complex Inputs

When the input u is a complex scalar, the block output matches the MATLAB result for:

$$\text{sign}(u) = u ./ \text{abs}(u)$$

When an element of a vector or matrix input is complex, the block uses the same formula that applies to scalar input, as shown in this example:



Data Type Support

The Sign block supports real inputs of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The block supports complex inputs only for floating-point data types, double and single. The output uses the same data type as the input.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Examples

The following Simulink examples show how to use the Sign block:

- `sldemo_fuelsys` (in the Engine Gas Dynamics/Throttle & Manifold/Throttle subsystem)
- `sldemo_hardstop`

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

Introduced before R2006a

Signal Builder

Create and generate interchangeable groups of signals whose waveforms are piecewise linear



Library

Sources

Description

The Signal Builder block allows you to create interchangeable groups of piecewise linear signal sources and use them in a model. See “Signal Groups”.

Note Use the `signalbuilder` function to create and access Signal Builder blocks programmatically.

Data Type Support

The Signal Builder block accepts signals only of type `double` and outputs a virtual nonhierarchical bus, scalar, or array of real signals of type `double`. It does not support data stores (see “Data Stores”).

For more information, see “Data Types Supported by Simulink”.

Parameters

The Signal Builder block has the same dialog box as that of a Subsystem block. To display the dialog box, right-click the block and select **Subsystem Parameters**.

Examples

The following examples show how to use the Signal Builder block:

- `sldemo_pid2dof`
- `sf_test_vectors`

Characteristics

Data Types	Double
Sample Time	Specified in the Sample time parameter, accessible by selecting File > Simulation Options in the Signal Builder block <ul style="list-style-type: none">• Zero represents a continuous sample time.• A positive integer represents a discrete sample time.
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes
Code Generation	Yes

See Also

From Spreadsheet | `signalbuilder`

Topics

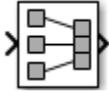
“Signal Groups”

“Converting Harness-Driven Models to Use Harness-Free External Inputs”

Introduced before R2006a

Signal Conversion

Convert signal to new type without altering signal values



Library

Signal Attributes

Description

The Signal Conversion block converts a signal from one type to another. Use the **Output** parameter to select the type of conversion to perform.

Data Type Support

The Signal Conversion block accepts signals of the following data types:

- Scalar
- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated
- Virtual and nonvirtual bus signals

You can use an array of buses as an input signal to a Signal Conversion block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

For more information about data types, see “Data Types Supported by Simulink”.

Parameters

Output

Specify the type of conversion to perform. The type of conversion that you use depends on your modeling goal.

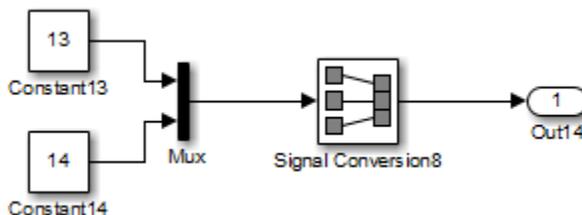
Modeling Goal	Output Option
Reduce generated code for a muxed signal. For an example involving Simulink Coder software, see “Generate Reentrant Code from Subsystems” (Simulink Coder).	Signal copy
Connect a block with a constant sample time to an output port of an enabled subsystem.	Signal copy
Pass a bus signal, or array of buses signal, whose components have different data types to a nonvirtual Inport block in an atomic subsystem that has direct feedthrough. For more information, see “Composite Signals”.	Signal copy
Save memory by converting a nonvirtual bus to a virtual bus.	Virtual bus
Pass a virtual bus signal to a modeling construct that requires a nonvirtual bus, such as a Model block.	Nonvirtual bus
Pass a nonvirtual bus signal from a Bus Selector block.	Nonvirtual bus

- The `Signal copy` option is the default. The type of conversion that the Signal Conversion block performs using the `Signal copy` option depends on the type of input signal.

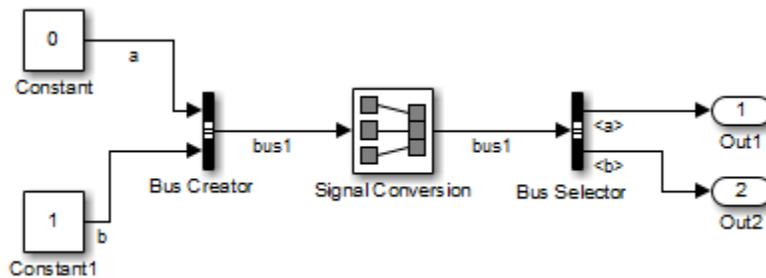
Type of Input Signal	Conversion that the Signal Copy Option Performs
Muxed (nonbus) signal	Converts the muxed signal, whose elements occupy discontinuous areas of memory, to a vector signal, whose elements occupy contiguous areas of memory. The conversion allocates a contiguous area of memory for the elements of the muxed signal and copies the values from the discontinuous areas (represented by the block's input) to the contiguous areas (represented by the block's output) at each time step.
Bus signal	Outputs a contiguous copy of the bus signal that is the input to the Signal Conversion block.

For an array of buses input signal, use the `Signal copy` option.

In the following example, a muxed signal inputs to a Signal Conversion block that has the **Output** parameter set to `Signal copy`. The Signal Conversion block converts the input signal to a vector.

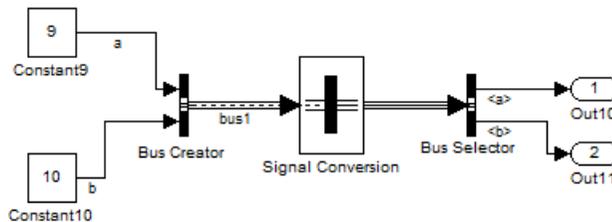


In the following example, a nonvirtual bus signal from a Bus Creator block inputs to a Signal Conversion block that has **Output** set to `Signal copy`. The Signal Conversion block creates another contiguous copy of the input bus signal.



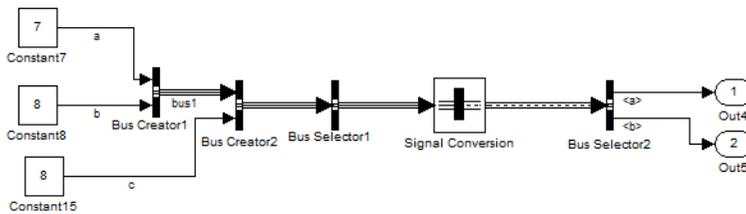
- The `Virtual bus` option converts a nonvirtual bus to a virtual bus.

In the following example, a Bus Creator block inputs to a Signal Conversion block that has **Output** set to `Virtual bus`. The Signal Conversion block converts the nonvirtual bus signal from the Bus Creator block to a virtual bus signal that inputs to the Bus Selector block.



- The `Nonvirtual bus` option converts a virtual bus to a nonvirtual bus.

In the following example, the Signal Conversion block converts a virtual bus signal from the first Bus Selector block to a nonvirtual bus signal that inputs to the second Bus Selector block. The Signal Conversion block has its **Output** parameter set to `Nonvirtual bus`, and specifies a bus object that matches the bus signal hierarchy of the bus that the first Bus Creator block outputs.



Data type

Specify the output data type of the nonvirtual bus that the Signal Conversion block produces.

This option is available only when you set the **Output** parameter to `Nonvirtual bus`.

The default option is `Inherit: auto`, which uses a rule that inherits a data type.

You must specify a `Simulink.Bus` object in the **Data type** parameter for one or both of the following blocks:

- Signal Conversion block
- An upstream Bus Creator block

If you specify a bus object for the Signal Conversion block, but not for its upstream Bus Creator block, then use a bus object that matches the hierarchy of the bus that upstream Bus Creator block outputs.

If you specify a bus object for both the Signal Conversion block and its upstream Bus Creator block, use the same bus object for both blocks.

You can select the button to the right of the Data type parameter to open the Data Type Assistant, which helps you to set the **Data type** parameter.

See “Control Signal Data Types” in *Simulink User's Guide* for more information.

Exclude this block from 'Block reduction' optimization

This option is available only when you set the **Output** parameter to `Signal copy`. If the elements of the input signal occupy contiguous areas of memory, then as an optimization, Simulink software eliminates the block from the compiled model. If you select the **Exclude this block from 'Block reduction' optimization** check box, the optimization occurs the next time you compile the model. For more information, see “Block reduction”.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
------------	---------------------------------------------------------------------

Sample Time	Inherited from driving block
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Bus Creator | Data Type Conversion

Topics

“Buses”

Introduced before R2006a

Signal Editor

Display, create, edit, and switch interchangeable scenarios

Library: Sources



Description

The Signal Editor block displays, creates, and edits interchangeable scenarios. You can also use the block to switch scenarios in and out of models.

The Signal Editor block supports MAT-files that contain one or more scalar `Simulink.SimulationData.Dataset` objects.

Limitations

The Signal Editor block does not support:

- Function-calls
- Array of buses
- Buses while using Rapid Accelerator mode
- `timetable` objects
- Ground signals

Ports

Output Arguments

signal1 — Signals in scenario
multidimensional

One or more signals, which can be:

- A MATLAB `timeseries` object
- A structure of MATLAB `timeseries` objects
- A two-dimensional matrix

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

Parameters

File name — Data set file

`untitled.mat` (default) | character vector

Data set file, specified as character vector, containing one or more scalar `Simulink.SimulationData.Dataset` objects. Do not use a file name from one locale in a different locale.

Dependencies

If `untitled.mat` does not exist in the current folder, these parameters are disabled:

- **Active scenario**
- **Signals**
- **Output a bus signal**
- **Unit**
- **Sample time**
- **Interpolate data**
- **Enable zero-crossing detection**
- **Form output after final data value by**

To create a MAT-file, click . This button starts the Signal Editor user interface, which lets you to create and edit scenario MAT-files.

Active scenario — Active scenario

`Scenario` (default) | character vector

Active scenario, specified as a character vector. The specified MAT-file must exist.

Dependencies

To enable this parameter, ensure that the specified MAT-file exists.

To create and edit scenarios, launch Signal Editor user interface — Start Signal Editor

button click

To start Signal Editor user interface, click .

Signals — Signal to configure

Signal 1 (default) | character vector

Signal to configure, specified as a signal name. This signal is considered the active signal. The MAT-file must exist before you can configure signals.

Dependencies

To enable this parameter, ensure that the specified MAT-file exists.

Output a bus signal — Configure signal as bus

off (default) | on

Configure signal as a bus:

On

Configure signal as a bus.

Off

Do not configure signal as a bus.

The specified MAT-file must exist.

Dependencies

- Selecting **Output a bus signal** check box enables the **Select bus object** parameter.
- To enable this parameter, ensure that the specified MAT-file exists.

Select bus object — Bus object

Bus: BusObject (default) | bus object name

Select the bus object name. To edit the bus object or create a bus object using the Data Type Assistant, click >>. The specified MAT-file must exist.

Dependencies

To enable this parameter, ensure that the specified MAT-file exists.

Mode — Bus object mode

Bus Object (default) | bus object data type

Select the bus object name. If you do not have a bus object, create one by clicking **Edit**, which starts the Bus Editor. For more information, see “Create Bus Objects with the Bus Editor”.

Unit — Physical unit

inherit (default) | supported physical unit

Physical unit of the signal, specified as an allowed unit. To specify a unit, begin typing in the text box. As you type, the parameter displays potential matching units. For more information, see “Unit Specification in Simulink Models”. For a list of supported units, see Allowed Unit Systems.

To constrain the unit system, click the link to the right of the parameter:

- If a Unit System Configuration block exists in the component, its dialog box opens. Use that dialog box to specify allowed and disallowed unit systems for the component.
- If a Unit System Configuration block does not exist in the component, the model Configuration Parameters dialog box displays. Use that dialog box to specify allowed and disallowed unit systems for the model.

The specified MAT-file must exist.

Dependencies

To enable this parameter, ensure that the specified MAT-file exists.

Sample time — Time interval between samples

0 (default) | -1 | sample time in seconds

Time interval between samples, specified in seconds. The specified MAT-file must exist.

Dependencies

To enable this parameter, ensure that the specified MAT-file exists.

Interpolate data — Linearly interpolate data

off (default) | on

Linearly interpolate data at time hits for which no corresponding workspace data exist. The specified MAT-file must exist.

The Signal Editor block linearly interpolates:

On

Linearly interpolate at time hits for which no corresponding workspace data exist, select this option.

Off

The current output equals the output at the most recent time for which data exists.

The Signal Editor block interpolates by using the two corresponding workspace samples:

- For `double` data, linearly interpolates the value by using the two corresponding samples
- For `Boolean` data, uses `false` for the first half of the time between two time values and `true` for the second half
- For a built-in data type other than `double` or `Boolean`:
 - Upcasts the data to `double`
 - Performs linear interpolation (as described for `double` data)
 - Downcasts the interpolated value to the original data type

You cannot use linear interpolation with enumerated (`enum`) data.

The block uses the value of the last known data point as the value of time hits that occur after the last known data point.

To determine the block output after the last time hit for which data is available, combine the settings of these parameters:

- **Interpolate data**
- **Form output after final data value by**

For details, see the **Form output after final data value by** parameter.

Dependencies

To enable this parameter, ensure that the specified MAT-file exists.

Enable zero-crossing detection — Detect zero-crossings

off (default) | on

If you select the **Enable zero-crossing detection** parameter, then when the input array contains multiple entries for the same time hit, Simulink detects a zero crossing. For example, suppose that the input array has this data:

```
time:      0 1 2 2 3
signal:    2 3 4 5 6
```

At time 2, there is a zero crossing from input signal discontinuity. For more information, see “Zero-Crossing Detection”.

For bus signals, Simulink detects zero crossings across all leaf bus elements.

The specified MAT-file must exist.

Dependencies

To enable this parameter, ensure that the specified MAT-file exists.

Form output after final data value by — Block output after the last time hit for which data is available

Setting to zero (default) | Extrapolation | Holding final value

To determine the block output after the last time hit for which workspace data is available, combine the settings of these parameters:

- **Interpolate data**
- **Form output after final data value by**

This table lists the block output, based on the values of the two options.

Setting for Form Output After Final Data Value By	Setting for Interpolate Data	Block Output After Final Data
Extrapolation	On	Extrapolated from final data value
	Off	Error
Setting to zero	On	Zero

Setting for Form Output After Final Data Value By	Setting for Interpolate Data	Block Output After Final Data
	Off	Zero
Holding final value	On	Final value from workspace
	Off	Final value from workspace

For example, the block uses the last two known data points to extrapolate data points that occur after the last known point if you:

- Select **Interpolate data**.
- Set **Form output after final data value by** to `Extrapolation`.

The specified MAT-file must exist.

Dependencies

To enable this parameter, ensure that the specified MAT-file exists.

See Also

Signal Builder | `signalEditor`

Topics

“Create Bus Objects with the Bus Editor”

Introduced in R2017b

Signal Generator

Generate various waveforms



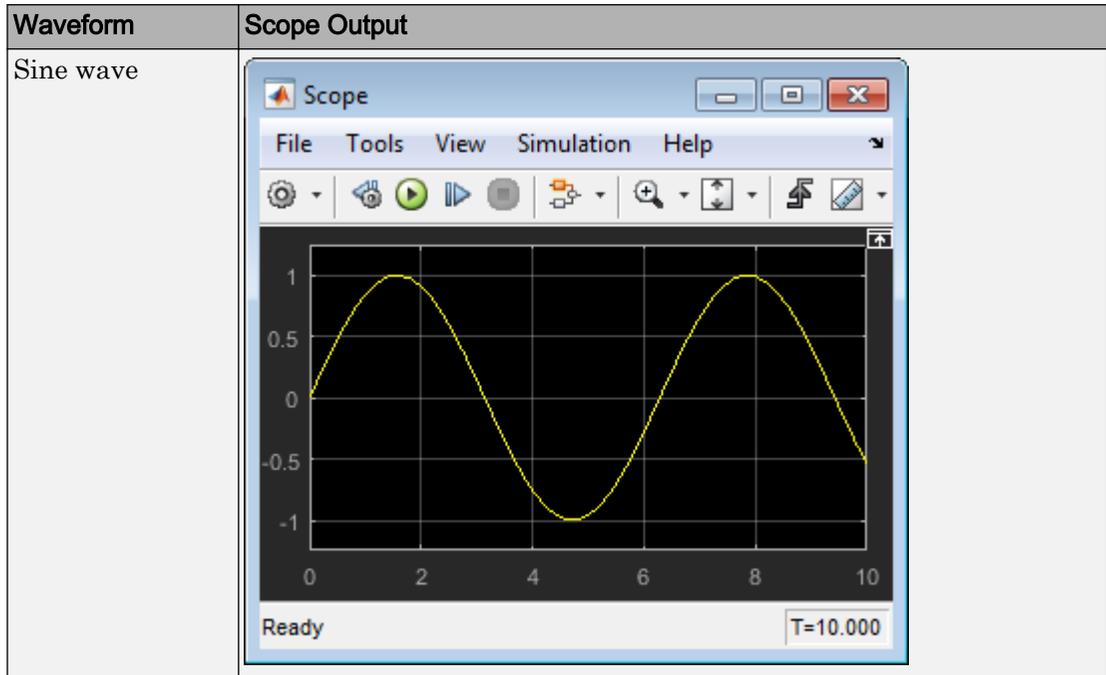
Library

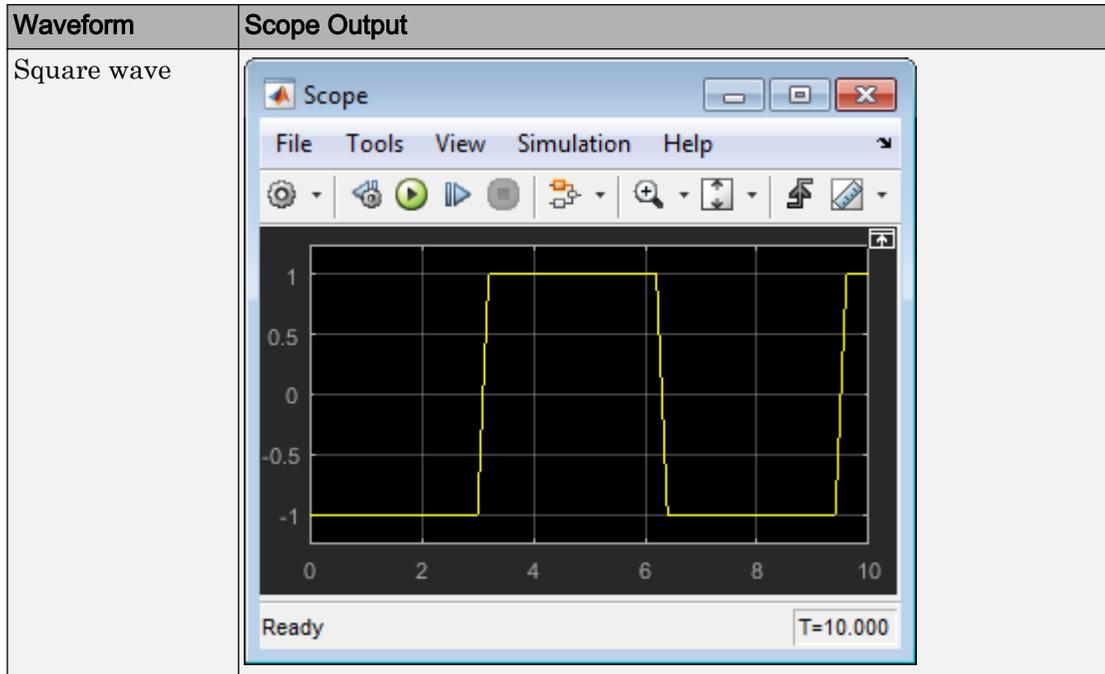
Sources

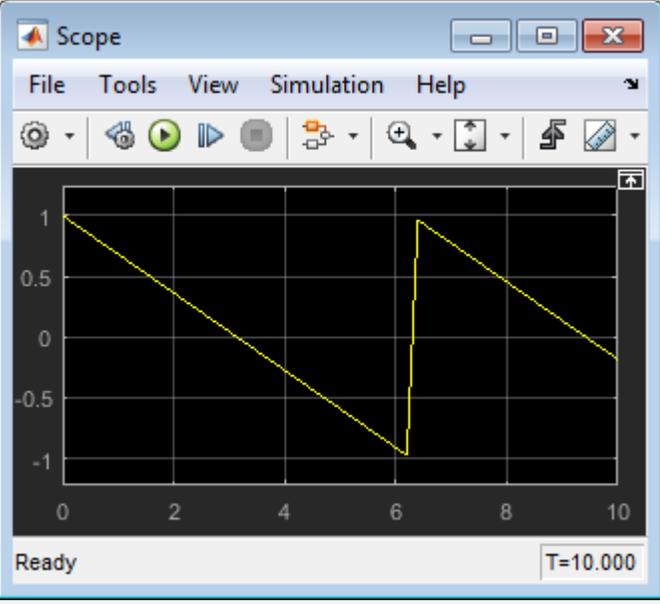
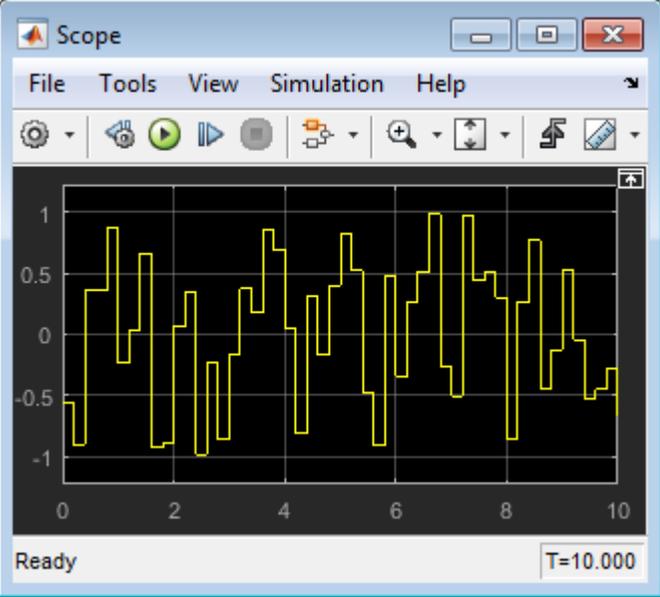
Description

Supported Operations

The Signal Generator block can produce one of four different waveforms: sine wave, square wave, sawtooth wave, and random wave. You can express signal parameters in Hertz (the default) or radians per second. Using default parameter values, you get one of the following waveforms:





Waveform	Scope Output
Sawtooth wave	
Random wave	

A negative **Amplitude** parameter value causes a 180-degree phase shift. You can generate a phase-shifted wave at other than 180 degrees in many ways. For example, you can connect a Clock block signal to a MATLAB Function block and write the equation for the specific wave.

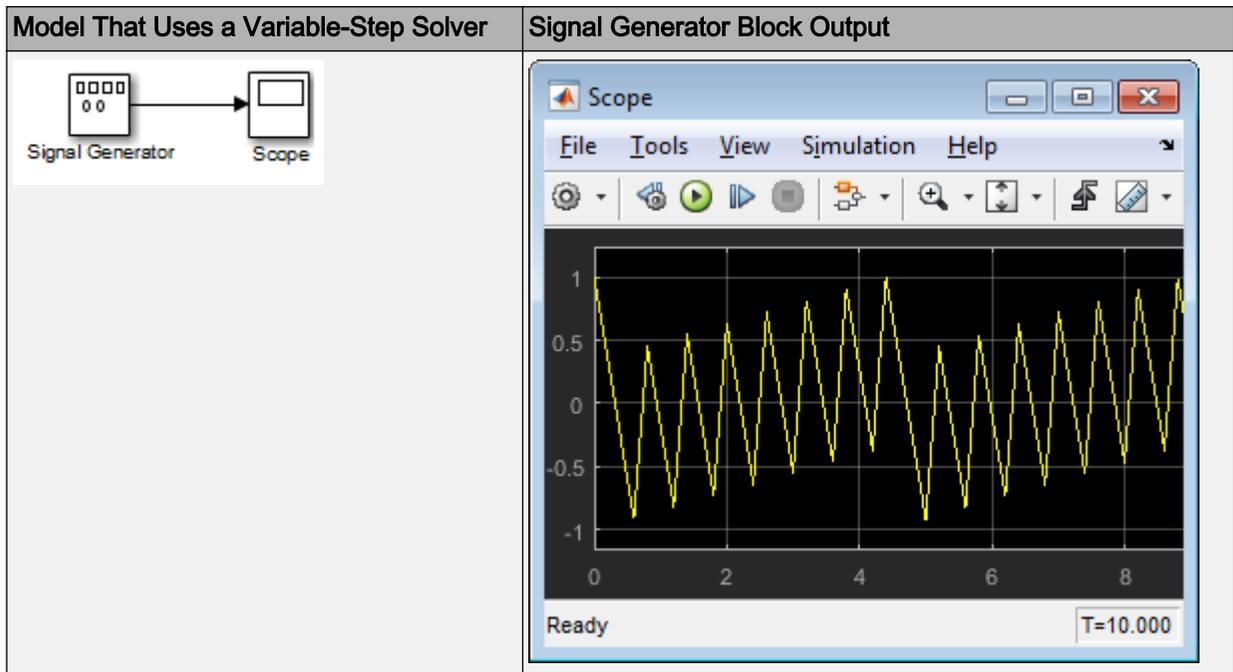
You can vary the output settings of the Signal Generator block while a simulation is in progress. This is useful to determine quickly the response of a system to different types of inputs.

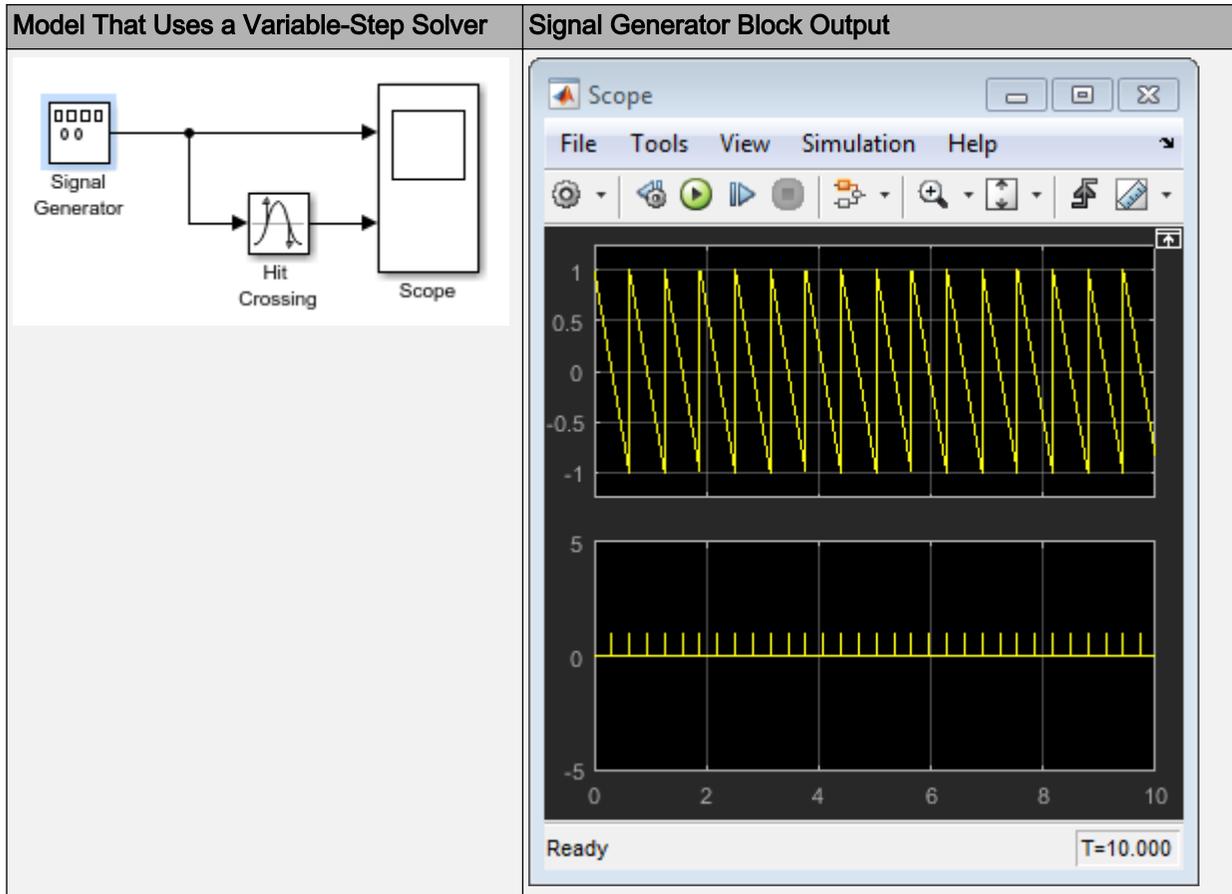
The **Amplitude** and **Frequency** parameters determine the amplitude and frequency of the output signal. The parameters must be of the same dimensions after scalar expansion. If you clear the **Interpret vector parameters as 1-D** check box, the block outputs a signal of the same dimensions as the **Amplitude** and **Frequency** parameters (after scalar expansion). If you select the **Interpret vector parameters as 1-D** check box, the block outputs a vector (1-D) signal if the **Amplitude** and **Frequency** parameters are row or column vectors, that is, single row or column 2-D arrays. Otherwise, the block outputs a signal of the same dimensions as the parameters.

Solver Considerations

If your model uses a fixed-step solver, Simulink uses the same step size for the entire simulation. In this case, the Signal Generator block output provides a uniformly sampled representation of the ideal waveform.

If your model uses a variable-step solver, Simulink might use different step sizes during the simulation. In this case, the Signal Generator block output does not always provide a uniformly sampled representation of the ideal waveform. To ensure that the block output is a uniformly sampled representation, add a Hit Crossing block directly downstream of the Signal Generator block. The following models show the difference in Signal Generator block output with and without the Hit Crossing block.





Data Type Support

The Signal Generator block outputs a scalar or array of real signals of type `double`.

For more information, see “Data Types Supported by Simulink”.

Parameters

Wave form

Specify the wave form: a sine wave, square wave, sawtooth wave, or random wave. The default is a sine wave. This parameter cannot change while a simulation is running.

Time

Specify whether to use simulation time as the source of values for the waveform's time variable or an external signal. If you specify an external time source, the block displays an input port for the time source.

Amplitude

Specify the signal amplitude. The default is 1.

Frequency

Specify the signal frequency. The default is 1.

Units

Specify the signal units as Hertz or rad/sec. The default is rad/sec.

Interpret vector parameters as 1-D

If selected, column or row matrix values for the **Amplitude** and **Frequency** parameters result in a vector output signal (see “Determining the Output Dimensions of Source Blocks”). This option is not available when an external signal specifies time. In this case, if the **Amplitude** and **Frequency** parameters are column or row matrix values, the output is a 1-D vector.

Examples

The following Simulink examples show how to use the Signal Generator block:

- `sldemo_dbldcart1`
- `slexAircraftExample`
- `penddemo`

Characteristics

Data Types	Double
Sample Time	Continuous
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Pulse Generator

Introduced before R2006a

Signal Specification

Specify desired dimensions, sample time, data type, numeric type, and other attributes of signal



Library

Signal Attributes

Description

The Signal Specification block allows you to specify the attributes of the signal connected to its input and output ports. If the specified attributes conflict with the attributes specified by the blocks connected to its ports, Simulink software displays an error when it compiles the model. For example, at the beginning of a simulation, if no conflict exists, Simulink eliminates the Signal Specification block from the compiled model. In other words, the Signal Specification block is a virtual block. It exists only to specify the attributes of a signal and plays no role in the simulation of the model.

You can use the Signal Specification block to ensure that the actual attributes of a signal meet desired attributes. For example, suppose that you and a colleague are working on different parts of the same model. You use Signal Specification blocks to connect your part of the model with your colleague's. If your colleague changes the attributes of a signal without informing you, the attributes entering the corresponding Signal Specification block do not match. When you try to simulate the model, you get an error.

You can also use the Signal Specification block to ensure correct propagation of signal attributes throughout a model. The capability of allowing the Simulink to propagate attributes from block to block is powerful. However, if some blocks have unspecified attributes for the signals they accept or output, the model does not have enough information to propagate attributes correctly. For these cases, the Signal Specification

block is a good way of providing the information Simulink needs. Using the Signal Specification block also helps speed up model compilation when blocks are missing signal attributes.

The Signal Specification block supports signal label propagation.

Data Type Support

The Signal Specification block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types, as well as bus objects. The input data type must match the data type specified by the **Data type** parameter.

Note If you specify a bus object as the data type for this block, do not set the minimum and maximum values for bus data on the block. Simulink ignores these settings. Instead, set the minimum and maximum values for bus elements of the bus object specified as the data type. The values should be finite real double scalar.

For information on the Minimum and Maximum properties of a bus element, see `Simulink.BusElement`.

For more information, see “Data Types Supported by Simulink”.

Parameters

- “Minimum” on page 1-1426
- “Maximum” on page 1-1427
- “Data type” on page 1-1427
- “Show data type assistant” on page 1-1429
- “Mode” on page 1-1429
- “Data type override” on page 1-1431
- “Signedness” on page 1-1431
- “Word length” on page 1-1432

- “Scaling” on page 1-1432
- “Fraction length” on page 1-1433
- “Slope” on page 1-1433
- “Bias” on page 1-1434
- “Require nonvirtual bus” on page 1-1434
- “Lock output data type setting against changes by the fixed-point tools” on page 1-1435
- “Unit (e.g., m, m/s², N*m)” on page 1-1435
- “Dimensions (-1 for inherited)” on page 1-1436
- “Variable-size signal” on page 1-1436
- “Sample time (-1 for inherited)” on page 1-1437
- “Signal type” on page 1-1438

Minimum

Specify the minimum value for the block output.

Default: [] (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see `Simulink.BusElement`.

Simulink uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Parameter: OutMin

Type: character vector

Value: Any valid finite real double scalar value

Default: ' [] '

Maximum

Specify the maximum value for the block output.

Default: [] (unspecified)

This number must be a finite real double scalar value.

Note If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`.

Simulink uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Parameter: OutMax

Type: character vector

Value: Any valid finite real double scalar value

Default: ' [] '

Data type

Specify the output data type.

Default: auto

Inherit: auto

Inherits the data type.

double

Specifies the data type is double.

`single`

Specifies the data type is `single`.

`int8`

Specifies the data type is `int8`.

`uint8`

Specifies the data type is `uint8`.

`int16`

Specifies the data type is `int16`.

`uint16`

Specifies the data type is `uint16`.

`int32`

Specifies the data type is `int32`.

`uint32`

Specifies the data type is `uint32`.

`boolean`

Specifies the data type is `boolean`.

`fixdt(1,16,0)`

Specifies the data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`

Specifies the data type is fixed point `fixdt(1,16,2^0,0)`.

Enum: `<class name>`

Specifies the data type as enumerated.

Bus: `<object name>`

Data type is a bus object.

`<data type expression>`

The name of a data type object, for example `Simulink.NumericType`

Do not specify a bus object as the expression.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Data type** parameters.

Parameter: OutDataTypeStr

Type: character vector

Value: 'Inherit: auto' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'boolean' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>' | 'Bus: <object name>' | <data type expression>

Default: 'Inherit: auto'

“Control Signal Data Types”.

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Mode

Select the category of data to specify.

Default: Inherit

Inherit

Specifies inheritance rules for data types. Selecting `Inherit` enables `auto`.

Built in

Specifies built-in data types. Selecting `Built in` enables a list of possible values:

- `double` (default)
- `single`

- int8
- uint8
- int16
- uint16
- int32
- uint32
- boolean

Fixed point

Specifies fixed-point data types.

Enumerated

Specifies enumerated data types. Selecting `Enumerated` enables you to enter a class name.

Bus

Bus object. Selecting `Bus` enables a **Bus object** parameter to the right, where you enter the name of a bus object that you want to use to define the structure of the bus. If you need to create or change a bus object, click **Edit** to the right of the **Bus object** field to open the Simulink Bus Editor. For details, see “Create Bus Objects with the Bus Editor”.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Do not specify a bus object as the expression.

Clicking the **Show data type assistant** button  enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

“Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data signed or unsigned.

Default: `Signed`

`Signed`

Specifies fixed-point data as signed.

`Unsigned`

Specifies the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

“Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Large word sizes represent large values with greater precision than small word sizes.

Selecting **Mode** > Fixed point enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

“Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting Slope and bias enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > Binary point enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

“Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

“Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

“Specifying a Fixed-Point Data Type”.

Require nonvirtual bus

If you specify a bus object as the data type, use this parameter to specify whether to accept only nonvirtual bus signals.



Default: `off`



Off

Specifies that a signal must come from a *virtual* bus.



On

Specifies that a signal must come from a *nonvirtual* bus.

The following **Data type** values enable this parameter:

- Bus: <object name>
- <data type expression> that specifies a bus object

Parameter: BusOutputAsStruct

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: LockScale

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Unit (e.g., m, m/s², N*m)

Specify physical unit of the input signal to the block.

Default: inherit

To specify a unit, begin typing in the text box. As you type, the parameter displays potential matching units. For a list of supported units, see [Allowed Unit Systems](#).

To constrain the unit system, click the link to the right of the parameter:

- If a Unit System Configuration block exists in the component, its dialog box opens. Use that dialog box to specify allowed and disallowed unit systems for the component.
- If a Unit System Configuration block does not exist in the component, the model Configuration Parameters dialog box displays. Use that dialog box to specify allowed and disallowed unit systems for the model.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Dimensions (-1 for inherited)

Specify the dimensions of the input and output signals.

Default: -1

-1

Specifies that signals inherit dimensions.

n

Specifies vector signal of width n.

[m n]

Specifies matrix signal having m rows and n columns.

Parameter: Dimensions

Type: character vector

Value: '-1' | n | [m n]

Default: '-1'

Variable-size signal

Specify a variable-size signal, fixed-size signal, or both.

Default: Inherit

Inherit

Allows variable-size and fixed-size signals.

No

Does not allow variable-size signals.

Yes

Allows only variable-size signals.

When the signal is a variable-size signal, the **Dimensions** parameter specifies the maximum dimensions of the signal.

If you specify a bus object, the simulation allows variable-size signals only with a disabled bus object.

Parameter: VarSizeSig

Type: character vector

Value: 'Inherit' | 'No' | 'Yes'

Default: 'Inherit'

“Variable-Size Signal Basics”

Sample time (-1 for inherited)

Specify the time interval when simulation updates the block.

Default: -1

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. See “Specify Sample Time” for more information.

Parameter: SampleTime

Type: character vector

Value: Any valid sample time

Default: '-1'

“Specify Sample Time”

Signal type

Specify the numeric type of the input and output signals.

Default: `auto`

`auto`

Accepts either `real` or `complex` as the numeric type.

`real`

Specifies the numeric type as a real number.

`complex`

Specifies the numeric type as a complex number.

Parameter: `SignalType`

Type: character vector

Value: `'auto' | 'real' | 'complex'`

Default: `'auto'`

Bus Support

The Signal Specification block supports virtual and nonvirtual buses. If you specify a bus object as the data type, then set these other block parameters as follows:

Block Parameter	Required Value for a Bus Data Type
Variable-size signal	No

All elements of the bus input to a Signal Specification block must have the same names as specified in the bus object.

All signals in a nonvirtual bus input to a Signal Specification block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “Specify Bus Signal Sample Times” and Bus-Capable Blocks for more information.

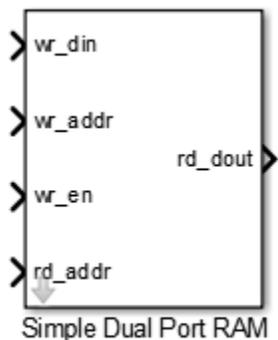
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Specified by the Sample time parameter
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Simple Dual Port RAM

Dual port RAM with single output port



Library

HDL Coder / HDL Operations

Description

The Simple Dual Port RAM block models RAM that supports simultaneous read and write operations, and has a single output port for read data. You can use this block to generate HDL code that maps to RAM in most FPGAs.

The Simple Dual Port RAM is similar to the Dual Port RAM, but the Dual Port RAM has both a write data output port and a read data output port.

Read-During-Write Behavior

During a write operation, if a read operation occurs at the same address, old data appears at the output.

Parameters

Address port width

Address bit width. Minimum bit width is 2, and maximum bit width is 29. The default is 8.

Ports

The block has the following ports:

`wr_din`

Write data input. The data can have any width. It inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

`wr_addr`

Write address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0

`wr_en`

Write enable.

Data type: Boolean

`rd_addr`

Read address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0

`rd_dout`

Output data from read address, `rd_addr`.

See Also

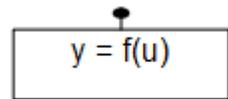
Dual Port RAM | Dual Rate Dual Port RAM | Single Port RAM

Introduced in R2014a

Simulink Function

Function defined with Simulink blocks

Library: User-Defined Functions



Description

The Simulink Function block is a Subsystem block preconfigured as a starting point for graphically defining a function with Simulink blocks. The block provides a text interface to function callers. You can call a Simulink Function block from a Function Caller block, a MATLAB Function block, or a Stateflow Chart.

For a description of the block parameters, see the Subsystem block reference page in the Simulink documentation.

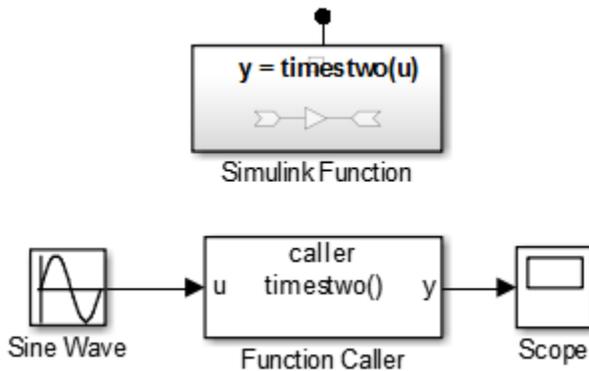
You can visualize Simulink Function calls in the Message Viewer. The viewer shows when calls were made with the argument and the return values. See Message Viewer block reference.

Function Interface

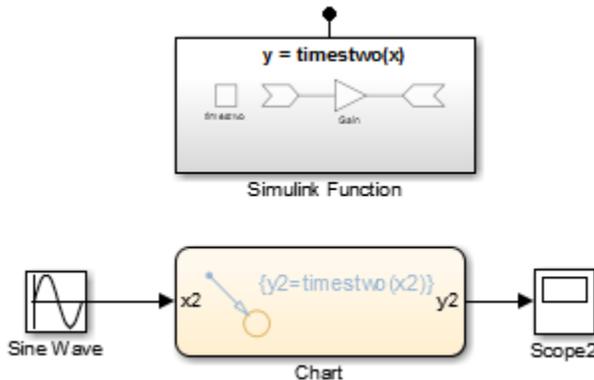
The function interface appears on the face of a Simulink Function block. Editing the block text adds and deletes Argument Inport blocks and Argument Outport blocks from the function definition. Editing also sets the **Function name** parameter in the Trigger block within the Simulink Function block.

For example, entering $y = \text{myfunction}(u)$ on the face of a Simulink Function block adds one Argument Inport block (u) and one Argument Outport block (y) within the subsystem.

When calling a function using a Function Caller block, the parameter **Function prototype** in the Function Caller block must match exactly the function interface you specify on the Simulink Function block. This match includes the name of the function and the names of input and output arguments. For example, the Simulink Function block and the Function Caller block both use the argument names u and y .



When calling a function from a Stateflow transition or state label, you can use different argument names. For example, the Simulink Function block uses x and y arguments while the Stateflow transition uses $x2$ and $y2$ arguments to call the function.



Function-Call Subsystems Versus Simulink Function Blocks

In general, a Function-Call Subsystem block provides better signal traceability with direct signal connections than a Simulink Function block. While a Simulink Function block eliminates the need for routing input and output signal lines through the model hierarchy.

Attribute	Function-Call Subsystem block	Simulink Function block
Method of executing/invoking function	Triggered using a signal line	Called by reference using the function name
Formal input arguments (Argument Inport blocks) and output arguments (Argument Outport blocks)	No	Yes
Local inputs (Inport block) and outputs (Outport block)	Yes	Yes

Ports

Input

In — Signal input to a subsystem block

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the Simulink Function block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment. See “Connect to Local Signals”.

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

ArgIn — Argument input to a subsystem block

scalar | vector | matrix

An Argument Inport block in a subsystem block provides an input port corresponding to an input argument. A port is not displayed on the subsystem block.

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

Output

out — Signal output from a subsystem

scalar | vector | matrix

Placing an Output block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Output block.

Use Output blocks to send signals to the local environment. See “Connect to Local Signals”.

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

ArgOut — Argument output from a subsystem block

scalar | vector | matrix

An Argument Output block in a subsystem block provides an output port corresponding to an out put argument. A port is not displayed on the subsystem block.

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

See Also

Blocks

Argument Inport | Argument Output | Chart | Function Caller | Function-Call Subsystem | Inport | MATLAB Function | Outport | Subsystem | Trigger

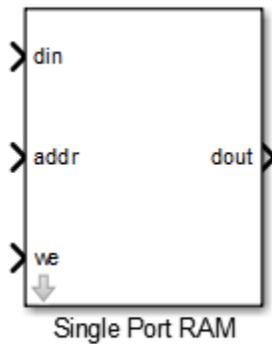
Topics

“Simulink Functions”
“Simulink Functions in Models”
“Argument Specification for Simulink Functions”
“Simulink Functions in Referenced Models”
“Scoping Simulink Functions in Subsystems”
“Diagnostics Using a Client-Server Architecture”

Introduced in R2014b

Single Port RAM

Single port RAM



Library

HDL Coder / HDL Operations

Description

The Single Port RAM block models RAM that supports sequential read and write operations.

If you want to model RAM that supports simultaneous read and write operations, use the Dual Port RAM or Simple Dual Port RAM.

Parameters

Address port width

Address bit width. Minimum bit width is 2, and maximum bit width is 29. The default is 8.

Output data during write

Controls the output data, `dout`, during a write access.

- `New data` (default): During a write, new data appears at the output port, `dout`.
- `Old data`: During a write, old data appears at the output port, `dout`.

Ports

The block has the following ports:

`din`

Data input. The data can have any width. It inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

`addr`

Write address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0

`we`

Write enable.

Data type: Boolean

`dout`

Output data from address, `addr`.

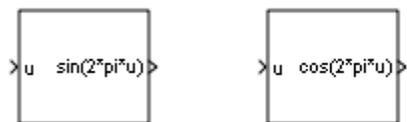
See Also

Dual Port RAM | Dual Rate Dual Port RAM | Simple Dual Port RAM

Introduced in R2014a

Sine, Cosine

Implement fixed-point sine or cosine wave using lookup table approach that exploits quarter wave symmetry



Library

Lookup Tables (Sine block or Cosine block)

Description

The Sine and Cosine block implements a sine and/or cosine wave in fixed point using a lookup table method that exploits quarter wave symmetry.

The Sine and Cosine block can output the following functions of the input signal, depending upon what you select for the **Output formula** parameter:

- $\sin(2\pi u)$
- $\cos(2\pi u)$
- $\exp(i2\pi u)$
- $\sin(2\pi u)$ and $\cos(2\pi u)$

You define the number of lookup table points in the **Number of data points for lookup table** parameter. The block implementation is most efficient when you specify the lookup table data points to be $(2^n) + 1$, where n is an integer.

Tip To obtain meaningful block output, the block input values should fall within the range $[0, 1)$. For input values that fall outside this range, the values are cast to an

unsigned data type, where overflows wrap. For these out-of-range inputs, the block output might not be meaningful.

Use the **Output word length** parameter to specify the word length of the fixed-point output data type. The fraction length of the output is the output word length minus 2.

Data Type Support

The Sine and Cosine block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The output of the block is a fixed-point data type.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Output formula

Select the signal(s) to output.

Number of data points for lookup table

Specify the number of data points to retrieve from the lookup table. The implementation is most efficient when you specify the lookup table data points to be $(2^n) + 1$, where n is an integer.

Output word length

Specify the word length for the fixed-point data type of the output signal. The fraction length of the output is the output word length minus 2.

Note The block uses double-precision floating-point values to construct lookup tables. Therefore, the maximum amount of precision you can achieve in your output is 53

bits. Setting the word length to values greater than 53 bits does not improve the precision of your output.

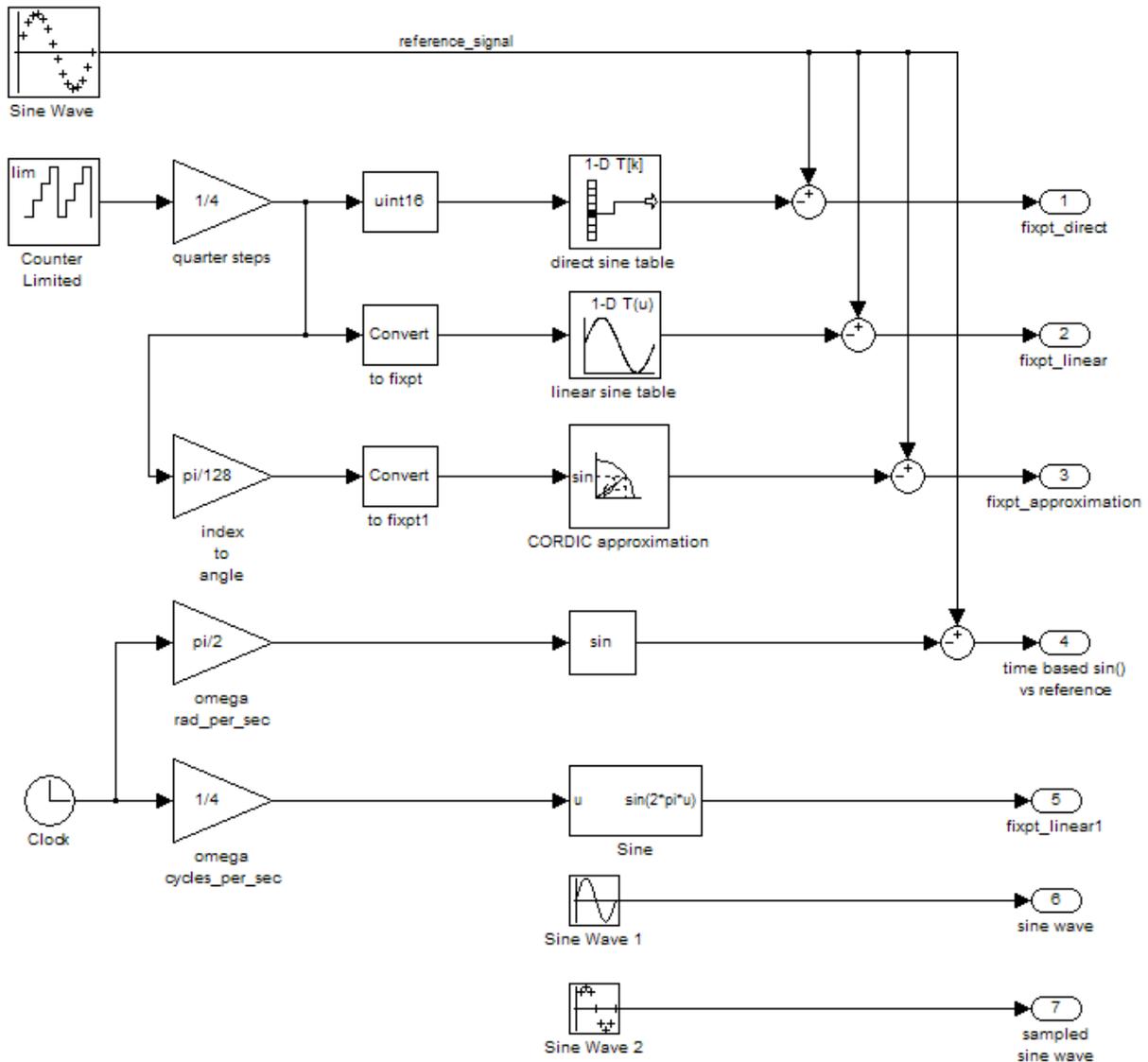
Internal rule priority for lookup table

Specify the internal rule for intermediate calculations. Select *Speed* for faster calculations. If you do, a loss of accuracy might occur, usually up to 2 bits.

Examples

The `sldemo_tonegen_fixpt` model shows how you can use the Sine block to implement a fixed-point sine wave.

Additional blocks to calculate sine and approximate sine using various algorithms.



Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Sine Wave, Trigonometric Function

Introduced before R2006a

Sine HDL Optimized

Implement fixed-point sine wave by using lookup table approach that exploits quarter wave symmetry



Library

HDL Coder / Lookup Tables

Description

The Sine HDL Optimized block implements a fixed-point sine wave by using a lookup table method that exploits quarter-wave symmetry.

You define the number of lookup table points in the **Number of data points** parameter. The block implementation is most efficient for HDL code generation when you specify the lookup table data points to be (2^n) , where n is an integer. For information about the behavior of this block in HDL Coder, see Sine HDL Optimized.

Depending on your selection of the **Output formula** parameter, the blocks can output these functions of the input signal:

- $\sin(2\pi u)$
- $\cos(2\pi u)$
- $\exp(i2\pi u)$
- $\sin(2\pi u)$ and $\cos(2\pi u)$

Use the **Table data type** parameter to specify the word length of the fixed-point output data type. The fraction length of the output is the output word length minus 2.

Data Type Support

The Sine HDL Optimized block accepts signals of these data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The output of the block is a fixed-point data type.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Output formula

Select the signal(s) to output.

Number of data points

Specify the number of data points to retrieve from the lookup table. The implementation is most efficient when you specify the lookup table data points to be (2^n) , where n is an integer.

Table data type

Specify the table data type. You can specify an expression that evaluates to a data type, for example, `fixdt(1,16,0)`.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the table data type.

Show data type assistant

Display the **Data Type Assistant**. In the **Data Type Assistant**, you can select the mode to specify the data type.

Mode

Select the mode of data type specification. If you select `Expression`, enter an expression that evaluates to a data type, for example, `fixdt(1,16,0)`.

If you select `Fixed point`, you can use the options in the **Data Type Assistant** to specify the fixed-point data type. In the `Fixed point` mode, you can choose binary point scaling, and specify the signedness, word length, fraction length, and the data type override setting.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Cosine HDL Optimized | Sine, Cosine | Trigonometric Function

Introduced in R2016b

Sine Wave

Generate sine wave, using simulation time as time source



Library

Sources

Description

The Sine Wave block outputs a sinusoidal waveform. The block can operate in time-based or sample-based mode.

Note This block is the same as the Sine Wave Function block that appears in the Math Operations library. If you select `Use external signal` for the **Time** parameter in the block dialog box, you get the Sine Wave Function block.

Time-Based Mode

The output of the Sine Wave block is determined by:

$$y = \textit{amplitude} \times \sin(\textit{frequency} \times \textit{time} + \textit{phase}) + \textit{bias}.$$

Time-based mode has two submodes: continuous mode or discrete mode. The value of the **Sample time** parameter determines whether the block operates in continuous mode or discrete mode:

- 0 (the default) causes the block to operate in continuous mode.
- >0 causes the block to operate in discrete mode.

See “Specify Sample Time” in the online documentation for more information.

A **Sample time** parameter value of 0 causes the block to operate in continuous mode. When operating in continuous mode, the Sine Wave block can become inaccurate due to loss of precision as time becomes very large.

A **Sample time** parameter value greater than zero causes the block to behave as if it were driving a Zero-Order Hold block whose sample time is set to that value.

Using the Sine Wave block in this way, you can build models with sine wave sources that are purely discrete, rather than models that are hybrid continuous/discrete systems. Hybrid systems are inherently more complex and as a result take more time to simulate.

In discrete mode, this block uses a differential incremental algorithm instead of one based on absolute time. As a result, the block can be useful in models intended to run for an indefinite length of time, such as in vibration or fatigue testing.

The differential incremental algorithm computes the sine based on the value computed at the previous sample time. This method uses the following trigonometric identities:

$$\sin(t + \Delta t) = \sin(t)\cos(\Delta t) + \sin(\Delta t)\cos(t)$$

$$\cos(t + \Delta t) = \cos(t)\cos(\Delta t) - \sin(t)\sin(\Delta t)$$

In matrix form, these identities are:

$$\begin{bmatrix} \sin(t + \Delta t) \\ \cos(t + \Delta t) \end{bmatrix} = \begin{bmatrix} \cos(\Delta t) & \sin(\Delta t) \\ -\sin(\Delta t) & \cos(\Delta t) \end{bmatrix} \begin{bmatrix} \sin(t) \\ \cos(t) \end{bmatrix}$$

Because Δt is constant, the following expression is a constant:

$$\begin{bmatrix} \cos(\Delta t) & \sin(\Delta t) \\ -\sin(\Delta t) & \cos(\Delta t) \end{bmatrix}$$

Therefore, the problem becomes one of a matrix multiplication of the value of $\sin(t)$ by a constant matrix to obtain $\sin(t + \Delta t)$.

Discrete mode reduces but does not eliminate the accumulation of round-off errors, for example, $(4 * \text{eps})$. This accumulation can happen because computation of the block output at each time step depends on the value of the output at the previous time step.

To handle round-off errors when the Sine Wave block operates in time-based discrete mode, use one of the following methods.

Method	Rationale
Insert a Saturation block directly downstream of the Sine Wave block.	By setting saturation limits on the Sine Wave block output, you can remove overshoot due to accumulation of round-off errors.
<p>Set up the Sine Wave block to use the <code>sin()</code> math library function to calculate block output.</p> <ol style="list-style-type: none"> 1 On the Sine Wave block dialog box, set Time to Use external signal so that an input port appears on the block icon. 2 Connect a clock signal to this input port using a Digital Clock block. 3 Set the sample time of the clock signal to the sample time of the Sine Wave block. 	Unlike the block algorithm, the <code>sin()</code> math library function computes block output at each time step <i>independently</i> of output values from other time steps, preventing the accumulation of round-off errors.

Sample-Based Mode

Sample-based mode uses the following formula to compute the output of the Sine Wave block.

$$y = A \sin(2\pi(k + o) / p) + b$$

where

- A is the amplitude of the sine wave.
- p is the number of time samples per sine wave period.
- k is a repeating integer value that ranges from 0 to $p-1$.
- o is the offset (phase shift) of the signal.
- b is the signal bias.

In this mode, Simulink sets k equal to 0 at the first time step and computes the block output, using the preceding formula. At the next time step, Simulink increments k and

recomputes the output of the block. When k reaches p , Simulink resets k to 0 before computing the block output. This process continues until the end of the simulation.

The sample-based method of computing block output at a given time step does not depend on the output of the previous time steps. Therefore, this mode avoids the accumulation of round-off errors. Additionally, sample-based mode supports reset semantics in subsystems that offer it. For example, if a Sine Wave block is in a resettable subsystem that receives a reset trigger, the repeating integer k resets and the block output resets to its initial condition.

Parameter Dimensions

The numeric parameters of this block must have the same dimensions after scalar expansion.

- If **Interpret vector parameters as 1-D** is not selected, the block outputs a signal of the same dimensions and dimensionality as the parameters.
- If **Interpret vector parameters as 1-D** is selected and the numeric parameters are row or column vectors, the block outputs a vector signal. Otherwise, the block outputs a signal of the same dimensionality and dimensions as the parameters.

Data Type Support

The Sine Wave block accepts and outputs real signals of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Sine type

Specify the type of sine wave that this block generates, either time- or sample-based. Some parameters in the dialog box appear depending on whether you select time-based or sample-based.

Time

Specify whether to use simulation time as the source of values for the time variable or an external source. If you specify an external time source, the block displays an input port for the time source.

Amplitude

Specify the amplitude of the signal. The default is 1.

Bias

Specify the constant value added to the sine to produce the output of this block.

Frequency

Specify the frequency, in radians per second. The default is 1. This parameter appears only when you set **Sine type** to time-based.

Samples per period

Specify the number of samples per period. This parameter appears only when you set **Sine type** to sample-based.

Phase

Specify the phase shift, in radians. The default is 0. This parameter appears only when you set **Sine type** to time-based.

You cannot configure this parameter to appear in the generated code as a tunable global variable if you set **Time (t)** to `Use simulation time`. For example, if you set **Default parameter behavior** to `Tunable` or apply a storage class to a `Simulink.Parameter` object, the **Phase** parameter does not appear in the generated code as a tunable global variable.

To generate code so that you can tune the phase during execution, set **Time (t)** to `Use external signal`. You can provide your own time input signal or use a `Digital Clock` block to generate the time signal. For an example, see “Tune Phase Parameter of Sine Wave Block During Code Execution” (Simulink Coder).

Number of offset samples

Specify the offset (discrete phase shift) in number of sample times. This parameter appears only when you set **Sine type** to sample-based.

Sample time

Specify the sample period. The default is 0. If the sine type is sample-based, the sample time must be greater than 0. See “Specify Sample Time” in the online documentation for more information.

Interpret vector parameters as 1-D

If selected, column or row matrix values for numeric parameters result in a vector output signal. Otherwise, the block outputs a signal of the same dimensionality as the parameters. If you do not select this check box, the block always outputs a signal of the same dimensionality as the numeric parameters. See “Determining the Output Dimensions of Source Blocks” in the Simulink documentation. This parameter is not available when an external signal specifies time. In this case, if numeric parameters are column or row matrix values, the output is a 1-D vector.

Examples

The following Simulink examples show how to use the Sine Wave block:

- `sldemo_househeat`
- `sldemo_tonegen_fixpt`
- `sldemo_VariableTransportDelay`
- `sldemo_zeroring`

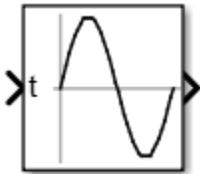
Characteristics

Data Types	Double
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Sine Wave Function

Generate sine wave, using external signal as time source



Library

Math Operations

Description

This block is the same as the Sine Wave block that appears in the Sources library. If you select `Use simulation time` for the **Time** parameter in the block dialog box, you get the Sine Wave block. See the documentation for the Sine Wave block for more information.

Characteristics

Data Types	Double
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

Introduced before R2006a

Slider

Tune parameter value with sliding scale

Library: Simulink / Dashboard



Description

The Slider block tunes the value of the connected block parameter during simulation. For example, you can connect the Slider block to a Gain block in your model and adjust its value during simulation. You can modify the range of the Slider block's scale to fit your data. Use the Slider block with other Dashboard blocks to create an interactive dashboard to control your model.

Connection

Dashboard block controls connect to variables and block parameters. To connect a Dashboard block to a variable or parameter, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting blocks of interest in your model.
- 3 Mark the option button next to the parameter or variable you want to adjust during simulation.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined using the variable `engine(1)` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

Parameters

Connection — Select a block parameter to connect

signal connection options

Select the block parameter to control using the **Connection** table. To open the dialog box, double-click the block. Then, populate the **Connection** table by selecting a block in your model. Select the variable parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram with **Ctrl+D**.

Scale Type — Type of scale

'Linear' (default) | 'Log'

Type of scale displayed on the control. **Linear** specifies a linear scale, and **Log** specifies a logarithmic scale.

Minimum — Minimum tick mark value

0 (default) | scalar

A finite, real, double, scalar value specifying the minimum tick mark value for the scale. The minimum must be less than the value entered for the maximum.

Maximum — Maximum tick mark value

100 (default) | scalar

A finite, real, double, scalar value specifying the maximum tick mark value for the scale. The maximum must be greater than the value entered for the minimum.

Tick Interval — Interval between major tick marks

auto (default) | scalar

A finite, real, positive, integer, scalar value specifying the interval of major tick marks on the scale. When set to `auto`, the block automatically adjusts the tick interval based on the minimum and maximum values.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

See Also

Knob | Rotary Switch

Topics

“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2015b

Slider Gain

Vary scalar gain using slider



Library

Math Operations

Description

Use the Slider Gain block to modify a scalar gain during a simulation using a slider. The block accepts one input and generates one output.

Data Type Support

Data type support for the Slider Gain block is the same as that for the Gain block (see Gain).

Parameters

Low

Specify the lower limit of the slider range. The default is 0.

High

Specify the upper limit of the slider range. The default is 2.

The edit fields indicate the lower limit and the upper limit. You can change the gain in two ways: by manipulating the slider, or by entering a new value in the current value field. You can change the range of gain values by changing the lower and upper limits.

To apply a vector or matrix gain to the block input, consider using the Gain block.

Examples

The following example models show how to use the Slider Gain block:

- `aero_six_dof`

Characteristics

Data Types	Double Single Boolean Base Integer
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Gain

Introduced before R2006a

Spectrum Analyzer

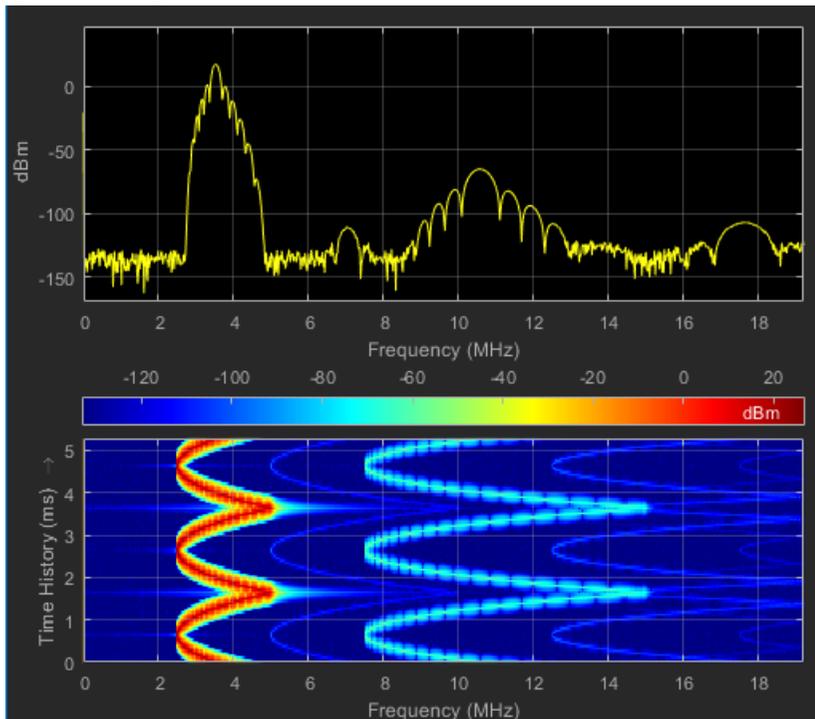
Display frequency spectrum of time-domain signals

Library: DSP System Toolbox / Sinks



Description

The Spectrum Analyzer block, referred to here as the scope, displays the frequency spectra of signals.



The Spectrum Analyzer block accepts input signals that:

- Have discrete sample time
- Are real or complex valued
- Have a fixed number of channels of variable length
- Are floating or fixed point data

You can use the Spectrum Analyzer block in models running in Normal or Accelerator simulation modes. You can also use the Spectrum Analyzer block in models running in Rapid Accelerator or External simulation modes, with some limitations.

You can use the Spectrum Analyzer block inside all subsystems and conditional subsystems. Conditional subsystems include enabled subsystems, triggered subsystems, enabled and triggered subsystems, and function-call subsystems. See “Conditional Subsystems” for more information.

You can configure and display Spectrum Analyzer settings from the command line with `spbscopes.SpectrumAnalyzerConfiguration`.

For information about the Spectrum Analyzer System object, see `dsp.SpectrumAnalyzer`.

Ports

Input

Port_1 — Signals to visualize

scalar | vector | matrix | array

Connect the signals you want to visualize. You can have up to 96 input ports.

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

Parameters

Spectrum Settings

The **Spectrum Settings** pane appears at the right side of the Spectrum Analyzer window. This pane controls how the spectrum is calculated. To show the Spectrum

Settings, in the Spectrum Analyzer menu, select **View > Spectrum Settings** or use the  button in the toolbar.

Main options

Type — Type of spectrum to display

Power (default) | Power density | RMS

Power - Spectrum Analyzer shows the power spectrum.

Power density - Spectrum Analyzer shows the power spectral density. The power spectral density is the magnitude of the spectrum normalized to a bandwidth of 1 hertz.

RMS - Spectrum Analyzer shows the root mean squared spectrum.

Tunable: Yes

Programmatic Use

See SpectrumType.

View — Spectrum view

Spectrum (default) | Spectrogram | Spectrum and spectrogram

Spectrum - Spectrum Analyzer shows the spectrum.

Spectrogram - Spectrum Analyzer shows the spectrogram, which displays frequency content over time. The most recent spectrogram update is at the bottom of the display, and time scrolls from the bottom to the top of the display.

Spectrum and spectrogram - Spectrum Analyzer shows both the spectrum and spectrogram.

Tunable: Yes

Programmatic Use

See ViewType.

Sample rate — Sample rate of the input signal in hertz

Inherited (default) | positive scalar

Select `Inherited` to use the same sample rate as the input signal. To specify a sample rate, delete `Inherited` and enter a sample rate value.

Programmatic Use

See `SampleRate`.

Method — Spectrum estimation method

`Welch` (default) | `Filter Bank`

Select `Welch` or `Filter Bank` as the spectrum estimation method. For more details about the two spectrum estimation algorithms, see “Algorithms” (DSP System Toolbox).

Tunable: No

Programmatic Use

See `Method`.

Full frequency span — Use entire Nyquist frequency interval

`on` (default) | `off`

Select this check box to compute and plot the spectrum over the entire “Nyquist frequency interval” (DSP System Toolbox).

Tunable: Yes

Programmatic Use

See `FrequencySpan`.

Span (Hz) — Frequency span in hertz

`10e3` (default) | real positive scalar

Specify the frequency span in hertz. Use this parameter with the `CF` (Hz) parameter to define the frequency span around a center frequency. This parameter defines the range of values shown on the Frequency axis in the Spectrum Analyzer window.

Tunable: Yes

Dependencies

To use this parameter, you must:

- Clear the “Full frequency span” (DSP System Toolbox) check box.
- Set the **Span (Hz)/Fstart (Hz)** dropdown to `Span (Hz)`.

Programmatic Use

See `FrequencySpan` and `Span`.

CF (Hz) — Center frequency in hertz

0 (default) | scalar

Specify the center frequency, in hertz. Use this parameter with the “Span (Hz)” (DSP System Toolbox) parameter to define the frequency span around a center frequency. This parameter defines the value shown at the middle point of the Frequency axis on the Spectrum Analyzer window.

Tunable: Yes

Dependencies

To use this parameter, you must:

- Clear the “Full frequency span” (DSP System Toolbox) check box.
- Set the **Span (Hz)/Fstart (Hz)** dropdown to “Span (Hz)” (DSP System Toolbox).

Programmatic Use

See `CenterFrequency`.

Fstart (Hz) — Start frequency in hertz

-5e3 (default) | scalar

Specify the start frequency in hertz. Use this parameter with the parameter to define the range of frequency-axis values using start frequency and stop frequency. This parameter defines the value shown at the leftmost side of the Frequency axis on the Spectrum Analyzer window.

Tunable: Yes

Dependencies

To use this parameter, you must:

- Clear the “Full frequency span” (DSP System Toolbox) check box.

- Set the **Span (Hz)/FStart (Hz)** dropdown to `FStart (Hz)`.

Programmatic Use

See `StartFrequency`.

FStop (Hz) — Stop frequency in hertz

`5e3` (default) | scalar

Specify the stop frequency, in hertz. Use this parameter with the “`FStart (Hz)`” (DSP System Toolbox) parameter to define the range of Frequency axis values. This parameter defines the value shown at the rightmost side of the Frequency axis on the Spectrum Analyzer window.

Tunable: Yes

Dependencies

To use this parameter, you must:

- Clear the “Full frequency span” (DSP System Toolbox) check box.
- Set the **Span (Hz)/FStart (Hz)** dropdown to “`FStart (Hz)`” (DSP System Toolbox).

Programmatic Use

See `StopFrequency`.

RBW (Hz) — Resolution bandwidth

`Auto` (default) | positive scalar

The resolution bandwidth in hertz. This parameter defines the smallest positive frequency that can be resolved. By default, this parameter is set to `Auto`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 *RBW* intervals over the specified frequency span.

If you set this parameter to a numeric value, the value must allow at least two *RBW* intervals over the specified frequency span. In other words, the ratio of the overall frequency span to *RBW* must be at least two:

$$\frac{\textit{span}}{\textit{RBW}} > 2$$

Tunable: Yes

Dependency

To use this parameter, set the **RBW (Hz)/Window length/Number of frequency bands** dropdown to `RBW (Hz)`.

Programmatic Use

See `RBW`.

Window length — Length of window in samples

1024 (default) | integer greater than 2

The length of the window, in samples. The window length used to control the frequency resolution and compute the spectral estimates. The window length must be an integer greater than 2.

Dependencies

To use this parameter, set:

- “Method” (DSP System Toolbox) to `Welch`
- Set the **RBW (Hz)/Window length/Number of frequency bands** dropdown to `Window Length`

Programmatic Use

See `WindowLength`.

Number of frequency bands — FFT length

Auto (default) | positive integer

Specify the fast Fourier transform (FFT) length to control the number of frequency bands. If the value is `Auto`, the Spectrum Analyzer uses the entire frame size to estimate the spectrum. If you specify the number of frequency bands, you set the input buffer size.

Dependencies

To use this parameter, set:

- “Method” (DSP System Toolbox) to `Filter Bank`
- Set the **RBW (Hz)/Window length/Number of frequency bands** dropdown to `Number of frequency bands`

Programmatic Use

See `FFTLength`

Taps per band — Number of filter taps

12 (default) | positive integer

Specify the number of filter taps or coefficients for each frequency band. This value corresponds to the number of filter coefficients per polyphase branch. The total number of filter coefficients is equal to `NumTapsPerBand + FFTLength`.

Dependency

To use this parameter, you must set the **RBW (Hz)/Window length/Number of frequency bands** dropdown to “Number of frequency bands” (DSP System Toolbox).

Programmatic Use

See `NumTapsPerBand`.

NFFT — Number of FFT points

Auto (default) | positive integer

Specify the length of the FFT that Spectrum Analyzer uses to compute spectral estimates. Acceptable options are `Auto` or a positive integer.

The **NFFT** value must be greater than or equal to the value of the **Window length** parameter. By default, when **NFFT** is set to `Auto`, the Spectrum Analyzer sets **NFFT** equal to the value of **Window length**. When in RBW mode, the specified RBW value is used to calculate an FFT length that equals the window length.

When this parameter is set to a positive integer, this parameter is equivalent to the `n` parameter of the `fft` function.

Dependencies

To use this parameter, you must set the **RBW (Hz)/Window length/Number of frequency bands** dropdown to “Window length” (DSP System Toolbox).

Programmatic Use

See `FFTLength`.

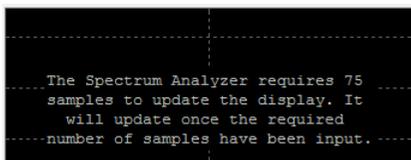
Samples/update — Required number of input samples

positive scalar

This property is read-only.

The number of input samples required to compute one spectral update. You cannot modify this parameter; it is shown in the spectrum analyzer for informational purposes only. This parameter is directly related to **RBW (Hz)/Window length/Number of frequency bands**. For more details, see “Algorithms” (DSP System Toolbox).

If the input does not have enough samples to achieve the resolution bandwidth that you specify, Spectrum Analyzer produces a message on the display.



Spectrogram Settings

Channel — Spectrogram channel

channel name

Select the signal channel for which the spectrogram settings apply.

Dependencies

To use this option:

- Set “View” (DSP System Toolbox) to Spectrogram or Spectrum and spectrogram.
- There must be more than one signal channel input.

Time res. (s) — Time resolution in seconds

Auto (default) | positive number

Time resolution is the amount of data, in seconds, used to compute a spectrogram line. The minimum attainable resolution is the amount of time it takes to compute a single spectral estimate. The tooltip displays the minimum attainable resolution given the current settings.

The time resolution value is determined based on frequency resolution method, the RBW setting, and the time resolution setting.

Method	Frequency Resolution Method	Frequency Resolution Setting	Time Resolution Setting	Resulting Time Resolution in Seconds
Welch or Filter Bank	RBW (Hz)	Auto	Auto	1/RBW
Welch or Filter Bank	RBW (Hz)	Auto	Manually entered	Time Resolution
Welch or Filter Bank	RBW (Hz)	Manually entered	Auto	1/RBW
Welch or Filter Bank	RBW (Hz)	Manually entered	Manually entered	Must be equal to or greater than the minimum attainable time resolution, 1/RBW. Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution. Interpolation is used to obtain time resolution values that are not integer multiples of 1/RBW.
Welch	Window length	—	Auto	1/RBW

Method	Frequency Resolution Method	Frequency Resolution Setting	Time Resolution Setting	Resulting Time Resolution in Seconds
Welch	Window length	—	Manually entered	Must be equal to or greater than the minimum attainable time resolution. Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution. Interpolation is used to obtain time resolution values that are not integer multiples of $1/RBW$.
Filter Bank	Number of frequency bands	—	Auto	$1/RBW$
Filter Bank	Number of frequency bands	—	Manually entered	Must be equal to or greater than the minimum attainable time resolution, $1/RBW$.

Tunable: Yes

Dependency

To use this option, set “View” (DSP System Toolbox) to Spectrogram or Spectrum and spectrogram.

Programmatic Use

See TimeResolution.

Time span — Time span in seconds

Auto (default) | positive scalar

The time span over which the Spectrum Analyzer displays the spectrogram specified in seconds. The time span is the product of the desired number of spectral lines and the time resolution. The tooltip displays the minimum allowable time span, given the current settings. If the time span is set to Auto, 100 spectral lines are used.

Tunable: Yes

Dependency

To use this option, set “View” (DSP System Toolbox) to Spectrogram or Spectrum and spectrogram.

Programmatic Use

See TimeSpan.

Window Options

Overlap (%) — Segment overlap percentage

0 (default) | scalar between 0 and 100

This parameter defines the amount of overlap between the previous and current buffered data segments. The overlap creates a window segment that is used to compute a spectral estimate. The value must be greater than or equal to zero and less than 100.

Tunable: Yes

Programmatic Use

See OverlapPercent.

Window — Windowing method

Hann (default) | Rectangular | Blackman-Harris | Chebyshev | Flat Top | Hamming | Kaiser | custom window function name

The windowing method to apply to the spectrum. Windowing is used to control the effect of sidelobes in spectral estimation. The window you specify affects the window length required to achieve a resolution bandwidth and the required number of samples per update. For more information about windowing, see “Windows” (Signal Processing Toolbox).

Tunable: Yes

Programmatic Use

See Window.

Attenuation — Sidelobe attenuation

60 (default) | scalar greater than or equal to 45

The sidelobe attenuation in decibels (dB). The value must be greater than or equal to 45.

Dependency

This parameter applies only when you set the **Window** parameter to Chebyshev or Kaiser.

Programmatic Use

See SidelobeAttenuation.

NENBW — Normalized effective noise bandwidth

scalar

This property is read-only.

The normalized effective noise bandwidth of the window. You cannot modify this parameter; it is shown for informational purposes only. This parameter is a measure of the noise performance of the window. The value is the width of a rectangular filter that accumulates the same noise power with the same peak power gain.

The rectangular window has the smallest NENBW, with a value of 1. All other windows have a larger NENBW value. For example, the Hann window has an NENBW value of approximately 1.5.

Trace Options

Units — Spectrum units

dBm (default) | dBW | Watts | Vrms | dBV | dBFS

The units of the spectrum. The available values depend on the value of the “Type” (DSP System Toolbox) parameter.

Tunable: Yes

Programmatic Use

See SpectrumUnits.

Full scale — Full scale for dBFS units

Auto (default) | positive real scalar

The full scale used for the dBFS units. By default, the Spectrum Analyzer uses the entire spectrum scale. Specify a positive real scalar for the dBFS full scale.

Tunable: Yes

Dependency

To enable this parameter, set “Units” on page 1-0 to dBFS.

Programmatic Use

See FullScale.

Averages — Number of spectral averages

1 (default) | positive integer

Specify the number of spectral averages as a positive integer. The spectrum analyzer computes the current power spectrum estimate by computing a running average of the last N power spectrum estimates. This parameter defines the number of spectral averages, N .

Dependency

This parameter applies only when the “View” (DSP System Toolbox) parameter is Spectrum or Spectrum and spectrogram.

Programmatic Use

See SpectralAverages.

Reference load — Reference load

1 (default) | positive real scalar

The reference load in ohms that the Spectrum Analyzer uses as a reference to compute power values.

Programmatic Use

See ReferenceLoad.

Scale — Scale of frequency axis

Linear (default) | Logarithmic

Choose a linear or logarithm scale for the frequency axis. When the frequency span contains negative frequency values, you cannot chose the logarithmic option.

Programmatic Use

See FrequencyScale.

Offset — Constant frequency offset

0 (default) | scalar

The constant frequency offset to apply to the entire spectrum, or a vector of frequencies to apply to each spectrum for multiple inputs. The offset parameter is added to the values on the Frequency axis in the Spectrum Analyzer window. This parameter is not used in any spectral computations. You must take the parameter into consideration when you set the **Span (Hz)** and **CF (Hz)** parameters to ensure that the frequency span is within the “Nyquist frequency interval” (DSP System Toolbox).

Programmatic Use

See FrequencyOffset.

Normal trace — Normal trace view

on (default) | off

When this check box is selected, the Spectrum Analyzer calculates and plots the power spectrum or power spectrum density. Spectrum Analyzer performs a smoothing operation by averaging several spectral estimates.

Dependencies

To clear this check box, you must first select either the “Max hold trace” (DSP System Toolbox) or the “Min hold trace” (DSP System Toolbox) parameter. This parameter applies only when “View” (DSP System Toolbox) is Spectrum or Spectrum and spectrogram.

Programmatic Use

See PlotNormalTrace.

Max hold trace — Maximum hold trace view

off (default) | on

Select this check box to enable Spectrum Analyzer to plot the maximum spectral values of all the estimates obtained.

Dependency

This parameter applies only when “View” (DSP System Toolbox) is Spectrum or Spectrum and spectrogram.

Programmatic Use

See PlotMaxHoldTrace.

Min hold trace — Minimum hold trace view

off (default) | on

Select this check box to enable Spectrum Analyzer to plot the minimum spectral values of all the estimates obtained.

Dependency

This parameter applies only when “View” (DSP System Toolbox) is Spectrum or Spectrum and spectrogram.

Programmatic Use

See PlotMinHoldTrace.

Two-sided spectrum — Enable two-sided spectrum view

off (default) | on

Select this check box to enable a two-sided spectrum view. In this view, both negative and positive frequencies are shown. If you clear this check box, Spectrum Analyzer shows a one-sided spectrum with only positive frequencies. Spectrum Analyzer requires that this parameter is selected when the input signal is complex valued.

Programmatic Use

See PlotAsTwoSidedSpectrum.

Configuration Properties

The **Configuration Properties** dialog controls visual aspects of the Spectrum Analyzer. To open the Configuration Properties, in the Spectrum Analyzer menu, select **View >**

Configuration Properties or select the  button in the toolbar dropdown.

Title — Display title

character vector

Specify the display title as a character vector. Enter %<SignalLabel> to use the signal labels in the Simulink model as the axes titles.

Tunable: Yes

Programmatic Use

See Title.

Show legend — Display signal legend

off (default) | on

Show signal legend. The names listed in the legend are the signal names from the model. For signals with multiple channels, a channel index is appended after the signal name. Continuous signals have straight lines before their names and discrete signals have step-shaped lines.

From the legend, you can control which signals are visible. This control is equivalent to changing the visibility in the **Style** parameters. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal, right-click the signal name, which hides all other signals. To show all signals, press **ESC**.

Dependency

To enable this parameter, set “View” (DSP System Toolbox) to `Spectrum` or `Spectrum` and `spectrogram`.

Programmatic Use

See ShowLegend.

Show grid — Show internal grid lines

off (default) | on

Show internal grid lines on the Spectrum Analyzer

Programmatic Use

See ShowGrid.

Y-limits (minimum) — Y-axis minimum

-80 (default) | scalar

Specify the minimum value of the *y*-axis.

Programmatic Use

See YLimits.

Y-limits (maximum) — Y-axis maximum

20 (default) | scalar

Specify the maximum value of the *y*-axis.

Programmatic Use

See YLimits.

Y-label — Y-axis label

character vector

To display signal units, add (%<SignalUnits>) to the label. At the beginning of a simulation, Simulink replaces (%SignalUnits) with the units associated with the signals. For example, if you have a signal for velocity with units of m/s enter

```
Velocity (%<SignalUnits>)
```

Programmatic Use

See YLabel.

Color map — Spectrogram colormap

jet(256) (default) | hot(256) | bone(256) | cool(256) | copper(256) |
gray(256) | 3-column matrix

Select the colormap for the spectrogram, or enter a three-column matrix expression for the colormap. For more information about colormaps, see colormap.

Tunable: Yes

Dependency

To use this parameter, set “View” (DSP System Toolbox) to Spectrogram or Spectrum and spectrogram.

Color-limits (minimum) — Spectrogram minimum

-80 (default) | scalar

Specify the signal power for the minimum color value of the spectrogram.

Tunable: Yes

Dependency

To use this parameter, set “View” (DSP System Toolbox) to Spectrogram or Spectrum and spectrogram.

Programmatic Use

See ColorLimits.

Color-limits (maximum) — Spectrogram maximum

20 (default) | scalar

Specify the signal power for the maximum color value of the spectrogram.

Tunable: Yes

Dependency

To use this parameter, set “View” (DSP System Toolbox) to Spectrogram or Spectrum and spectrogram.

Programmatic Use

See ColorLimits.

Style

The **Style** dialog controls how to Spectrum Analyzer appears. To open the Style properties, in the Spectrum Analyzer menu, select **View > Style** or select the  button in the toolbar dropdown.

Figure color — Window background

gray (default) | color picker

Specify the color that you want to apply to the background of the scope figure.

Plot type — Plot type

Line (default) | Stem

Specify whether to display a Line or Stem plot.

Programmatic Use

See PlotType.

Axes colors — Axes background color

black (default) | color picker

Specify the color that you want to apply to the background of the axes.

Properties for line — Channel for visual property settings

channel names

Specify the channel for which you want to modify the visibility, line properties, and marker properties.

Visible — Channel visibility

on (default) | off

Specify whether the selected channel is visible. If you clear this check box, the line disappears. You can also change signal visibility using the scope legend.

Line — Line style

line, 0.5, yellow (default)

Specify the line style, line width, and line color for the selected channel.

Marker — Data point markers

none (default)

Specify marks for the selected channel to show at its data points. This parameter is similar to the 'Marker' property for plots. You can choose any of the marker symbols from the dropdown.

Axes Scaling

The **Axes Scaling** dialog controls the axes limits of the Spectrum Analyzer. To open the Axes Scaling properties, in the Spectrum Analyzer menu, select **Tools > Axes Scaling > Axes Scaling Properties**.

Axes scaling/Color scaling — Automatic axes scaling

Auto (default) | Manual | After N Updates

Specify when the scope automatically scales the y-axis. If the spectrogram is displayed, specify when the scope automatically scales the color axis. By default, this parameter is set to `Auto`, and the scope does not shrink the y-axis limits when scaling the axes or color. You can select one of the following options:

- `Auto` — The scope scales the axes or color as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** or **Do not allow color limits to shrink**.
- `Manual` — When you select this option, the scope does not automatically scale the axes or color. You can manually scale the axes or color in any of the following ways:
 - Select **Tools > Scaling Properties**.
 - Press one of the **Scale Axis Limits** toolbar buttons.
 - When the scope figure is the active window, press **Ctrl+A**.
- `After N Updates` — Selecting this option causes the scope to scale the axes or color after a specified number of updates. This option is useful, and most efficient, when your frequency signal values quickly reach steady-state after a short period of time. Selecting this option shows the **Number of updates** edit box where you can modify the number of updates to wait before scaling.

Tunable: Yes

Programmatic Use

See `AxesScaling`.

Do not allow Y-axis/color limits to shrink — Axes scaling limits

on (default) | off

When you select this parameter, the y-axis is allowed to grow during axes scaling operations. If the spectrogram is displayed, selecting this parameter allows the color

limits to grow during axis scaling. If you clear this check box, the y-axis or color limits can shrink during axes scaling operations.

Dependency

This parameter appears only when you select `Auto` for the **Axis scaling** or **Color scaling** parameter. When you set the **Axes scaling** or **Color scaling** parameter to `Manual` or `After N Updates`, the y-axis or color limits can shrink.

Number of updates — Number of updates before scaling

10 (default) | positive number

The number of updates after which the axes scale, specified as a positive integer. If the spectrogram is displayed, this parameter specifies the number of updates after which the color axes scales.

Tunable: Yes

Dependency

This parameter appears only when you set “Axes scaling/Color scaling” (DSP System Toolbox) to `After N Updates`.

Scale limits at stop — Scale axes at stop

off (default) | on

Select this check box to scale the axes when the simulation stops. If the spectrogram is displayed, select this check box to scale the color when the simulation stops. The y-axis is always scaled. The x-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

Data range (%) — Percent of axes

100 (default) | number in the range [1,100]

Set the percentage of the axis that the scope uses to display the data when scaling the axes. If the spectrogram is displayed, set the percentage of the power values range within the colormap. Valid values are from 1 through 100. For example, if you set this parameter to 100, the scope scales the axis limits such that your data uses the entire axis range. If you then set this parameter to 30, the scope increases the y-axis or color range such that your data uses only 30% of the axis range.

Tunable: Yes

Align — Alignment along axes

Center (default) | Bottom | Top | Left | Right

Specify where the scope aligns your data along the axis when it scales the axes. If the spectrogram is displayed, specify where the scope aligns your data along the axis when it scales the color. If you are using CCDF Measurements (DSP System Toolbox), the x axis is also configurable.

Tunable: Yes

Model Examples

Algorithms

Welch's Method

When you set the **Method** property to `Welch`, the following algorithms apply. The Spectrum Analyzer uses the `RBW` or the `Window Length` setting in the **Spectrum Settings** pane to determine the data window length. Then, it partitions the input signal into a number of windowed data segments. Finally, Spectrum Analyzer uses the modified periodogram method to compute spectral updates, averaging the windowed periodograms for each segment.

Spectrum Analyzer requires that a minimum number of samples to compute a spectral estimate. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly related to resolution bandwidth, *RBW*, by the following equation, or to the window length, by the equation shown in step 2.

$$N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$$

The normalized effective noise bandwidth, *NENBW*, is a factor that depends on the windowing method. Spectrum Analyzer shows the value of *NENBW* in the **Window Options** pane of the **Spectrum Settings** pane. Overlap percentage, *O_p*, is the value of

the **Overlap %** parameter in the **Window Options** pane of the **Spectrum Settings** pane. F_s is the sample rate of the input signal. Spectrum Analyzer shows sample rate in the **Main Options** pane of the **Spectrum Settings** pane.

- 1 When in **RBW (Hz)** mode, the window length required to compute one spectral update, N_{window} , is directly related to the resolution bandwidth and normalized effective noise bandwidth:

$$N_{window} = \frac{NENBW \times F_s}{RBW}$$

When in **Window Length** mode, the window length is used as specified.

- 2 The number of input samples required to compute one spectral update, $N_{samples}$, is directly related to the window length and the amount of overlap by the following equation.

$$N_{samples} = \left(1 - \frac{O_p}{100}\right) N_{window}$$

When you increase the overlap percentage, fewer new input samples are needed to compute a new spectral update. For example, if the window length is 100, then the number of input samples required to compute one spectral update is given as shown in the following table.

O_p	$N_{samples}$
0%	100
50%	50
80%	20

- 3 The normalized effective noise bandwidth, $NENBW$, is a window parameter determined by the window length, N_{window} , and the type of window used. If $w(n)$ denotes the vector of N_{window} window coefficients, then $NENBW$ is given by the following equation.

$$NENBW = N_{window} \frac{\sum_{n=1}^{N_{window}} w^2(n)}{\left[\sum_{n=1}^{N_{window}} w(n)\right]^2}$$

- 4 When in **RBW (Hz)** mode, you can set the resolution bandwidth using the value of the **RBW (Hz)** parameter on the **Main options** pane of the **Spectrum Settings** pane. You must specify a value to ensure that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two:

$$\frac{span}{RBW} > 2$$

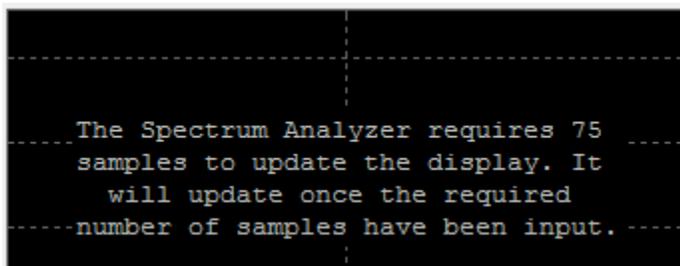
By default, the **RBW (Hz)** parameter on the **Main options** pane is set to `Auto`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified frequency span. When you set **RBW (Hz)** to `Auto`, *RBW* is calculated as:

$$RBW_{auto} = \frac{span}{1024}$$

- 5 When in **Window Length** mode, you specify N_{window} and the resulting *RBW* is:

$$\frac{NENBW * F_s}{N_{window}}$$

Sometimes, the number of input samples provided are not sufficient to achieve the resolution bandwidth that you specify. When this situation occurs, Spectrum Analyzer displays a message:



Spectrum Analyzer removes this message and displays a spectral estimate when enough data has been input.

Note The number of FFT points (N_{fft}) is independent of the window length (N_{window}). You can set them to different values if N_{fft} is greater than or equal to N_{window} .

Filter Bank

When you set the **Method** property to `Filter Bank`, the following algorithms apply. The Spectrum Analyzer uses the **RBW (Hz)** or the **Number of frequency band** property in the **Spectrum Settings** pane to determine the input frame length.

Spectrum Analyzer requires a minimum number of samples to compute a spectral estimate. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly related to resolution bandwidth, *RBW*, by the following equation.

$$N_{samples} = \frac{F_s}{RBW}$$

F_s is the sample rate of the input signal. Spectrum Analyzer shows sample rate in the **Main Options** pane of the **Spectrum Settings** pane.

- 1 When in **RBW (Hz)** mode, you can set the resolution bandwidth using the value of the **RBW (Hz)** parameter on the **Main options** pane of the **Spectrum Settings** pane. You must specify a value to ensure that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two:

$$\frac{span}{RBW} > 2$$

By default, the **RBW** parameter on the **Main options** pane is set to `Auto`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified frequency span. Thus, when you set **RBW**

to `Auto`, it is calculated by the following equation. $RBW_{auto} = \frac{span}{1024}$

- 2 When in **Number of frequency bands** mode, you specify the input frame size. When the number of frequency bands is `Auto`, the resulting RBW is:

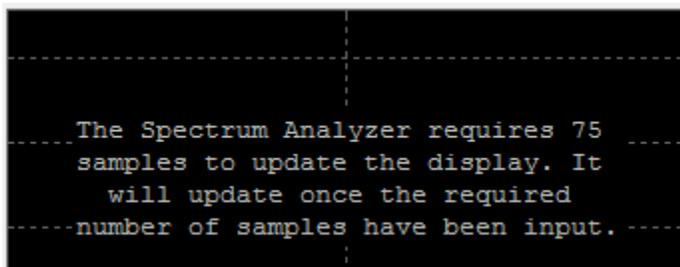
$$RBW = \frac{F_s}{\text{Input Frame Size}}$$

When the number of frequency bands is manually specified, the resulting RBW is:

$$RBW = \frac{F_s}{FFTLengh}$$

For more information about the filter bank algorithm, see “Polyphase Implementation” (DSP System Toolbox).

Sometimes, the number of input samples provided are not sufficient to achieve the resolution bandwidth that you specify. When this situation occurs, Spectrum Analyzer displays a message:



Spectrum Analyzer removes this message and displays a spectral estimate when enough data has been input.

Nyquist frequency interval

When the `PlotAsTwoSidedSpectrum` property is set to `true`, the interval is

$$\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ hertz.}$$

When the `PlotAsTwoSidedSpectrum` property is set to `false`, the interval is

$$\left[0, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ hertz.}$$

Periodogram and Spectrogram

Spectrum Analyzer calculates and plots the power spectrum, power spectrum density, and RMS computed by the modified Periodogram estimator. For more information about the Periodogram method, see `periodogram`.

Power Spectral Density — The power spectral density (PSD) is given by the following equation.

$$PSD(f) = \frac{1}{P} \sum_{p=1}^P \frac{\left| \sum_{n=1}^{N_{FFT}} x^{(p)}[n] e^{-j2\pi f(n-1)T} \right|^2}{F_s \times \sum_{n=1}^{N_{window}} w^2[n]}$$

In this equation, $x[n]$ is the discrete input signal. On every input signal frame, Spectrum Analyzer generates as many overlapping windows as possible, with each window denoted as $x^{(p)}[n]$, and computes their periodograms. Spectrum Analyzer displays a running average of the P most current periodograms.

Power Spectrum — The power spectrum is the product of the power spectral density and the resolution bandwidth, as given by the following equation.

$$P_{spectrum}(f) = PSD(f) \times RBW = PSD(f) \times \frac{F_s \times NENBW}{N_{window}} = \frac{1}{P} \sum_{p=1}^P \frac{\left| \sum_{n=1}^{N_{FFT}} x^{(p)}[n] e^{-j2\pi f(n-1)T} \right|^2}{\left[\sum_{n=1}^{N_{window}} w[n] \right]^2}$$

Occupied BW

The *Occupied BW* is calculated as follows.

- 1 Calculate the total power in the measured frequency range.
- 2 Determine the lower frequency value. Starting at the lowest frequency in the range and moving upward, the power distributed in each frequency is summed until this result is

$$\frac{100 - \text{Occupied BW } \%}{2}$$
 of the total power.
- 3 Determine the upper frequency value. Starting at the highest frequency in the range and moving downward, the power distributed in each frequency is summed until the result reaches

$$\frac{100 - \text{Occupied BW } \%}{2}$$

2

of the total power.

- 4 The bandwidth between the lower and upper power frequency values is the occupied bandwidth.
- 5 The frequency halfway between the lower and upper frequency values is the center frequency.

Distortion Measurements

The *Distortion Measurements* are computed as follows.

- 1 Spectral content is estimated by finding peaks in the spectrum. When the algorithm detects a peak, it records the width of the peak and clears all monotonically decreasing values. That is, the algorithm treats all these values as if they belong to the peak. Using this method, all spectral content centered at DC (0 Hz) is removed from the spectrum and the amount of bandwidth cleared (W_0) is recorded.
- 2 The fundamental power (P_1) is determined from the remaining maximum value of the displayed spectrum. A local estimate (Fe_1) of the fundamental frequency is made by computing the central moment of the power in the vicinity of the peak. The bandwidth of the fundamental power content (W_1) is recorded. Then, the power from the fundamental is removed as in step 1.
- 3 The power and width of the higher-order harmonics (P_2, W_2, P_3, W_3 , etc.) are determined in succession by examining the frequencies closest to the appropriate multiple of the local estimate (Fe_1). Any spectral content that decreases monotonically about the harmonic frequency is removed from the spectrum first before proceeding to the next harmonic.
- 4 Once the DC, fundamental, and harmonic content is removed from the spectrum, the power of the remaining spectrum is examined for its sum ($P_{remaining}$), peak value ($P_{maxspur}$), and median value ($P_{estnoise}$).
- 5 The sum of all the removed bandwidth is computed as $W_{sum} = W_0 + W_1 + W_2 + \dots + W_n$.

The sum of powers of the second and higher-order harmonics are computed as

$$P_{harmonic} = P_2 + P_3 + P_4 + \dots + P_n.$$

- 6 The sum of the noise power is estimated as:

$$P_{noise} = (P_{remaining} \cdot dF + P_{est.noise} \cdot W_{sum}) / RBW$$

Where dF is the absolute difference between frequency bins, and RBW is the resolution bandwidth of the window.

- 7 The metrics for SNR, THD, SINAD, and SFDR are then computed from the estimates.

$$THD = 10 \cdot \log_{10} \left(\frac{P_{harmonic}}{P_1} \right)$$

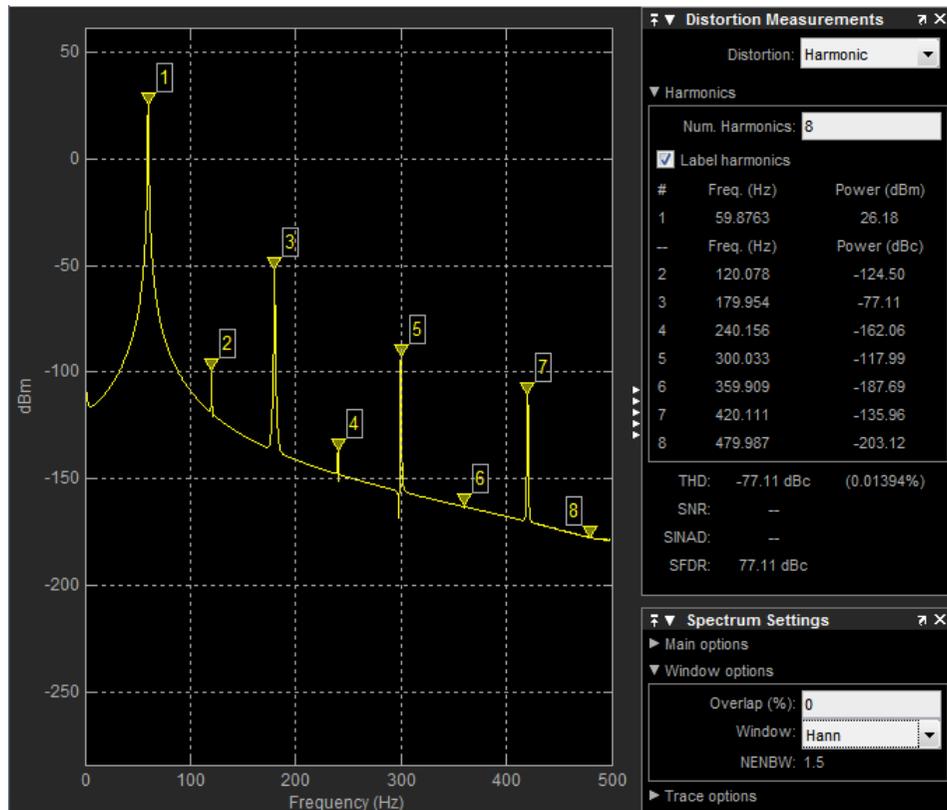
$$SINAD = 10 \cdot \log_{10} \left(\frac{P_1}{P_{harmonic} + P_{noise}} \right)$$

$$SNR = 10 \cdot \log_{10} \left(\frac{P_1}{P_{noise}} \right)$$

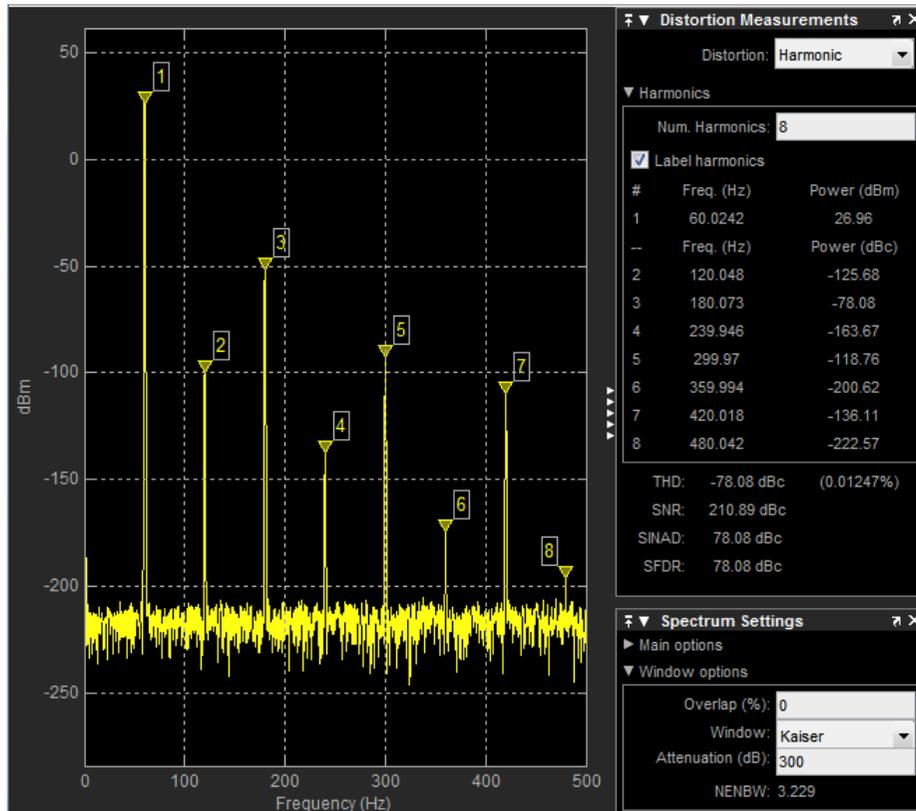
$$SFDR = 10 \cdot \log_{10} \left(\frac{P_1}{\max(P_{maxspur}, \max(P_2, P_3, \dots, P_n))} \right)$$

Harmonic Measurements

- 1 The harmonic distortion measurements use the spectrum trace shown in the display as the input to the measurements. The default Hann window setting of the Spectrum Analyzer may exhibit leakage that can completely mask the noise floor of the measured signal.



The harmonic measurements attempt to correct for leakage by ignoring all frequency content that decreases monotonically away from the maximum of harmonic peaks. If the window leakage covers more than 70% of the frequency bandwidth in your spectrum, you may see a blank reading (–) reported for **SNR** and **SINAD**. If your application can tolerate the increased equivalent noise bandwidth (ENBW), consider using a Kaiser window with a high attenuation (up to 330 dB) to minimize spectral leakage.



- 2 The DC component is ignored.
- 3 After windowing, the width of each harmonic component masks the noise power in the neighborhood of the fundamental frequency and harmonics. To estimate the noise power in each region, Spectrum Analyzer computes the median noise level in the nonharmonic areas of the spectrum. It then extrapolates that value into each region.
- 4 N th order intermodulation products occur at $A * F1 + B * F2$,

where $F1$ and $F2$ are the sinusoid input frequencies and $|A| + |B| = N$. A and B are integer values.

- 5 For intermodulation measurements, the third-order intercept (TOI) point is computed as follows, where P is power in decibels of the measured power referenced to 1 milliwatt (dBm):

- $TOI_{lower} = P_{F1} + (P_{F2} - P_{(2F1-F2)})/2$
- $TOI_{upper} = P_{F2} + (P_{F1} - P_{(2F2-F1)})/2$
- $TOI = + (TOI_{lower} + TOI_{upper})/2$

See Also

Using Objects

Spectrum Analyzer Configuration

System Objects

`dsp.SpectrumAnalyzer`

Functions

`getSpectralMaskStatus` | `getSpectrumData` | `sptool`

Blocks

Array Plot | Time Scope

Topics

“Display Frequency-Domain Data in Spectrum Analyzer” (DSP System Toolbox)

“Spectral Analysis” (DSP System Toolbox)

Introduced in R2014b

Slider Switch

Toggle parameter between two values

Library: Simulink / Dashboard



Description

The Slider Switch block toggles the value of the connected block parameter between two values during simulation. For example, you can connect the Slider Switch block to a Switch block in your model and change its state during simulation. Use the Slider Switch block with other Dashboard blocks to create an interactive dashboard for your model.

Connection

Dashboard block controls connect to variables and block parameters. To connect a Dashboard block to a variable or parameter, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting blocks of interest in your model.
- 3 Mark the option button next to the parameter or variable you want to adjust during simulation.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.
To save connections and properties, save the model file in the SLX format.
- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.

- Dashboard blocks cannot connect to signals inside reference models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined using the variable `engine(1)` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

Parameters

Connection — Select a block parameter to connect

signal connection options

Select the block parameter to control using the **Connection** table. To open the dialog box, double-click the block. Then, populate the **Connection** table by selecting a block in your model. Select the variable parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram with **Ctrl+D**.

States

Label (Left) — Label for left switch position

'On' (default) | character vector

Labels the left switch position. You can use the **Label** to display the value the connected parameter takes when the switch is positioned at the left, or you can enter a text label.

Example: `Gain = 2`

Value (Left) — Value for left switch position

1 (default) | scalar

The value assigned to the connected parameter when the switch is positioned at the left.

Label (Right) — Label for right switch position

'Off' (default) | character vector

Labels the right switch position. You can use the **Label** to display the value the connected parameter takes when the switch is positioned at the right, or you can enter a text label.

Example: `Gain = 1`

Value (Right) — Value for right switch position

0 (default) | scalar

The value assigned to the connected parameter when the switch is positioned at the right.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

Model Examples

See Also

[Rocker Switch](#) | [Rotary Switch](#) | [Toggle Switch](#)

Topics

[“Tune and Visualize Your Model with Dashboard Blocks”](#)

[“Decide How to Visualize Simulation Data”](#)

Introduced in R2015a

Sqrt, Signed Sqrt, Reciprocal Sqrt

Calculate square root, signed square root, or reciprocal of square root



Library

Math Operations

Description

You can select one of the following functions from the **Function** parameter list.

Function	Description	Mathematical Expression	MATLAB Equivalent
<code>sqrt</code>	Square root of the input	$u^{0.5}$	<code>sqrt</code>
<code>signedSqrt</code>	Square root of the absolute value of the input, multiplied by the sign of the input	$\text{sign}(u) * u ^{0.5}$	—
<code>rSqrt</code>	Reciprocal of the square root of the input	$u^{-0.5}$	—

The block output is the result of applying the function to the input. Each function supports:

- Scalar operations
- Element-wise vector and matrix operations

Data Type Support

The block accepts input signals of the following data types:

Function	Input Data Types	Restrictions
sqrt	<ul style="list-style-type: none"> Floating point Built-in integer Fixed point 	None
signedSqrt	<ul style="list-style-type: none"> Floating point Built-in integer Fixed point 	When the input is an integer or fixed-point type, the output must be floating point.
rSqrt	<ul style="list-style-type: none"> Floating point Built-in integer Fixed point 	None

The block accepts real and complex inputs of the following types:

Function	Types of Real Inputs	Types of Complex Inputs
sqrt	Any, except for fixed-point inputs that are negative or have nontrivial slope and nonzero bias	Any, except for fixed-point inputs
signedSqrt		None
rSqrt		None

The block output:

- Uses the data type that you specify for **Output data type**
- Is real or complex, depending on your selection for **Output signal type**

Parameters

Function

Specify the mathematical function. The block icon changes to match the function you select.

Function	Block Icon
sqrt	
signedSqrt	
rSqrt	

Output signal type

Specify the output signal type of the block as auto, real, or complex.

Function	Input Signal Type	Output Signal Type		
		Auto	Real	Complex
sqrt	real	real for nonnegative inputs NaN for negative inputs	real for nonnegative inputs NaN for negative inputs	complex
	complex	complex	error	complex
signedSqrt	real	real	real	complex
	complex	error	error	error
rSqrt	real	real	real	error
	complex	error	error	error

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Note The parameters in the **Algorithm** pane are available only when you set **Function** to rSqrt on the **Main** pane.

Method

Specify the method for computing the reciprocal of a square root.

Method	Data Types Supported	When to Use This Method
Exact	Floating point If you use a fixed-point or built-in integer type, an upcast to a floating-point type occurs.	You do not want an approximation. Note The input or output must be floating point.
Newton-Raphson	Floating-point, fixed-point, and built-in integer types	You want a fast, approximate calculation.

The `Exact` method provides results that are consistent with MATLAB computations.

Note The algorithms for `sqrt` and `signedSqrt` are always of `Exact` type, no matter what selection appears on the block dialog box.

Number of iterations

Specify the number of iterations to perform the Newton-Raphson algorithm. The default value is 3.

This parameter is not available when you select `Exact` for **Method**.

Note If you enter 0, the block output is the initial guess of the Newton-Raphson algorithm.

Intermediate results data type

Specify the data type for intermediate results (available only when you set **Function** to `sqrt` or `rSqrt` on the **Main** pane). You can set the data type to:

- A rule that inherits a data type, for example, `Inherit:Inherit` via internal rule
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object

- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Follow these guidelines on setting an intermediate data type explicitly for the square root function, `sqrt`:

Input and Output Data Types	Intermediate Data Type
Input or output is double.	Use double.
Input or output is single, and any non-single data type is <i>not</i> double.	Use single or double.
Input and output are fixed point.	Use fixed point.

Follow these guidelines on setting an intermediate data type explicitly for the reciprocal square root function, `rSqrt`:

Input and Output Data Types	Intermediate Data Type
Input is double and output is not single.	Use double.
Input is not single and output is double.	Use double.
Input and output are fixed point.	Use fixed point.

Caution Do not set **Intermediate results data type** to `Inherit:Inherit` from output when:

- You select `Newton-Raphson` to compute the reciprocal of a square root.
- The input data type is floating point.
- The output data type is fixed point.

Under these conditions, selecting `Inherit:Inherit` from output yields suboptimal performance and produces an error.

To avoid this error, convert the input signal from a floating-point to fixed-point data type. For example, insert a `Data Type Conversion` block in front of the `Sqrt` block to perform the conversion.

Output data type

Specify the output data type. You can set the data type to:

- A rule that inherits a data type, for example, `Inherit:Inherit` via back propagation

- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” in the *Simulink User's Guide* for more information.

Output minimum

Specify the minimum value that the block can output. The default value is [] (unspecified). Simulink uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value that the block can output. The default value is [] (unspecified). Simulink uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer). in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</p>	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code> , which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code> , is -126.

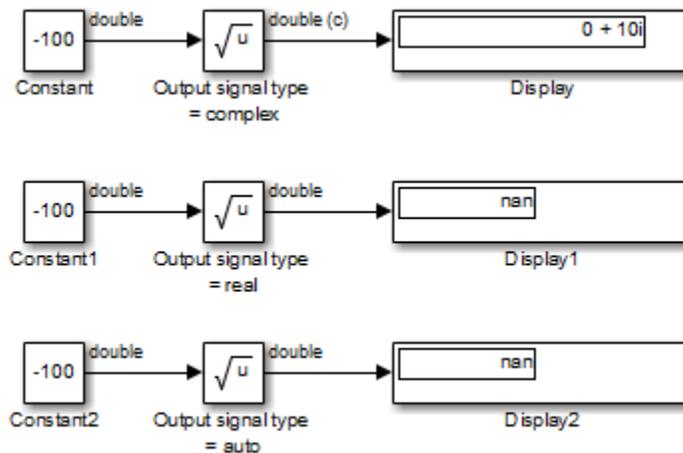
When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect

when overflow is not possible. In this case, the code generator does not produce saturation code.

Examples

sqrt Function

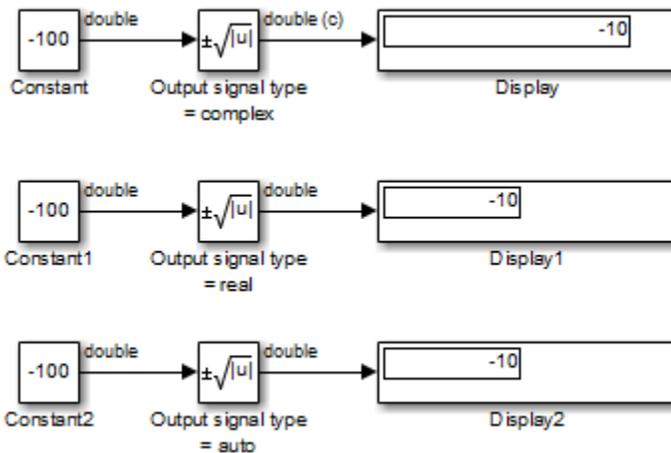
Suppose that you have the following model:



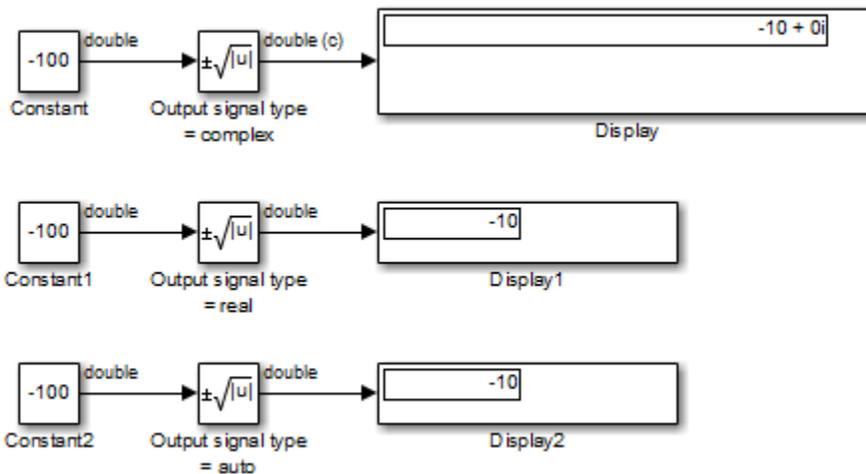
When the input to the Sqrt block is negative and the **Output signal type** is auto or real, the sqrt function outputs NaN. However, setting **Output signal type** to complex produces the correct answer.

signedSqrt Function

Suppose that you have the following model:

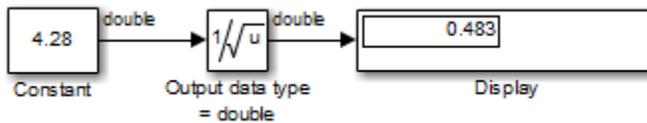


When the input to the Sqrt block is negative, the block output is the same for any **Output signal type** setting. If you change the first Display block format from short to decimal (Stored Integer), you see the value of the imaginary part for the complex output.



rSqrt Function with Floating-Point Inputs

Suppose that you have the following model:



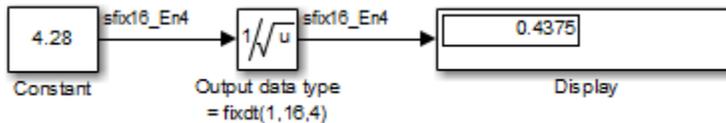
In the Sqrt block dialog box, assume that the following parameter settings apply:

Parameter	Setting
Method	Newton-Raphson
Number of iterations	1
Intermediate results data type	Inherit:Inherit from input

After one iteration of the Newton-Raphson algorithm, the block output is within 0.0004 of the final value (0.4834).

rSqrt Function with Fixed-Point Inputs

Suppose that you have the following model:



In the Sqrt block dialog box, assume that the following parameter settings apply:

Parameter	Setting
Method	Newton-Raphson
Number of iterations	1
Intermediate results data type	Inherit:Inherit from input

After one iteration of the Newton-Raphson algorithm, the block output is within 0.0459 of the final value (0.4834).

Characteristics

Data Types	Double Single Base Integer Fixed-Point
------------	----------------------------------------------

Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Math Function, Trigonometric Function

Introduced in R2010a

Squeeze

Remove singleton dimensions from multidimensional signal



Library

Math Operations

Description

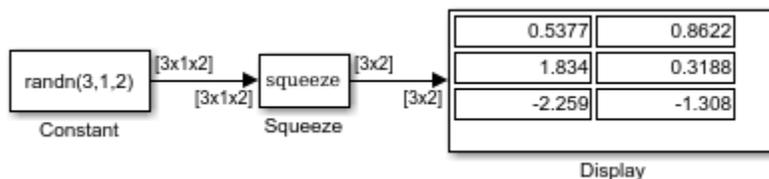
The Squeeze block removes singleton dimensions from its multidimensional input signal. A singleton dimension is any dimension whose size is one. The Squeeze block operates only on signals whose number of dimensions is greater than two. Scalar, one-dimensional (vector), and two-dimensional (matrix) signals pass through the Squeeze block unchanged.

Data Type Support

The Squeeze block accepts input signals of any dimension and of any data type that Simulink supports, including fixed-point and enumerated data types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Examples

In the following model, the Squeeze block converts a multidimensional array of size 3-by-1-by-2 into a 3-by-2 signal:



Because the Constant block supplies a signal with random values to the Squeeze block, the values in the Display block vary from simulation to simulation.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

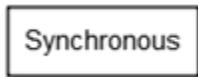
See Also

Reshape

Introduced in R2007b

State Control

Specify synchronous reset and enable behavior for blocks with state



Library

HDL Coder / HDL Subsystems

Description

The State Control block in *Synchronous* mode improves the HDL simulation behavior of blocks with state, or blocks that have reset or enable ports. The simulation behavior in *Classic* mode is the same as when you do not add the State Control block inside the subsystem.

When use the *Synchronous* mode of the block, the Simulink simulation behavior closely matches that of the digital hardware.

If you have HDL Coder installed, you can generate cleaner HDL code with the *Synchronous* mode of the State Control block. For more information, see [State Control](#).

Parameters

State control

Specify whether to use synchronous or classic semantics. The default is *Synchronous*.

Limitations

The following limitations apply to using the State Control block in Simulink. For information about this block in HDL Coder, see State Control in the HDL Coder documentation.

Block-Level Limitations

- For synchronous semantics in S-function blocks, set the method `ssSetStateSemanticsClassicAndSynchronous` to `true`.
- Discrete-Time Integrator blocks with a reset port do not support synchronous semantics.
- All action subsystems connected to If and Switch Case blocks must have the same semantics, either classic or synchronous.
- The following blocks are not allowed in synchronous mode:
 - Continuous time blocks and blocks with continuous rate
 - Simulink blocks with **Input processing** set to `Columns as channels (frame based)`, where this parameter applies.
 - Trigger block
 - From Workspace block
 - The set of unit delay blocks in the **Additional Math & Discrete > Additional Discrete** sublibrary in Simulink, such as the Unit Delay Resettable and Unit Delay External IC blocks

Subsystem-level Limitations

- Conditional subsystems using classic semantics cannot have subsystems with synchronous semantics inside them.
- Conditional subsystems must be single rate when you use the State Control block in synchronous mode.
- Synchronous Enabled Subsystem cannot contain reset subsystems or a reset parameter port. For example, you cannot have a Delay block with an external reset port inside the subsystem.
- These blocks are not supported in synchronous mode:

- For Iterator Subsystem
- While Iterator Subsystem
- Function-Call Subsystem
- Triggered Subsystem

Model-Level Limitations

- Variable-size signals are not supported with synchronous semantics.
- Synchronous semantics do not propagate across model boundaries. If your parent model has synchronous semantics, any referenced model must have synchronous semantics explicitly specified. At the root level of each referenced model, add a State Control block with the **State control** parameter set to *Synchronous*.

See Also

[Enable](#) | [Enabled Subsystem](#) | [Enabled Synchronous Subsystem](#) | [Resettable Synchronous Subsystem](#) | [Synchronous Subsystem](#)

Introduced in R2016a

State Reader

Read a block state

Library: Signal Routing



Description

The State Reader block reads the current state of a supported state owner block.

Add a State Reader block to your model from the Simulink Library Browser.

State Reader blocks can read state from these state owner blocks:

- Discrete State-Space
- Discrete-Time Integrator
- Delay
- Unit Delay
- Discrete Transfer Fcn
- Discrete Filter
- S-Function (with one data type work vector declared as a discrete-state vector)

Ports

Output

out — State value

scalar | vector

State value read from a state owner block.

The dimension of the output is the dimension of the full state vector. Refer to the **Initial conditions** parameter for specific blocks. For example, for a Delay block with a **Delay length** of N , the State Reader block returns a state vector of length $[1 \times N]$.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

State owner block — Show the state owner block

`none` (default)

Show the state owner block whose state this block is reading. To change the state owner block, select a block from the **State Owner Selector Tree**.

Command-Line Information

Parameter: `StateOwnerBlock`

Type: character vector

Value: `'` | `'<model path/block name>'`

Default: `'`

See Also

[Event Listener](#) | [Initialize Function](#) | [Reset Function](#) | [State Writer](#) | [Terminate Function](#)

Topics

“Create Model to Initialize, Reset, and Terminate State”

“Create Test Harness to Generate Function Calls”

State Writer

Write to a block state

Library: Signal Routing



Description

The State Writer block sets the state of a supported state owner block.

Add a State Writer block to your model from the Simulink Library Browser.

State Writer blocks can write state to these state owner blocks:

- Discrete State-Space
- Discrete-Time Integrator
- Delay
- Unit Delay
- Discrete Transfer Fcn
- Discrete Filter
- S-Function (with one data type work vector declared as a discrete-state vector)

Ports

Input

I_n — State value

scalar | vector

State value written to a state owner block.

When writing to a state owner block with an input scalar, the scalar value is expanded to match the dimension of the state. All elements of the state are set to the same value.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

State owner block — Show the state owner block

`none` (default)

Show the state owner block whose state this block is writing. To change the state owner block, select a block from the **State Owner Selector Tree**.

Command-Line Information

Parameter: `StateOwnerBlock`

Type: character vector

Value: `'` | `'<model path/block name>'`

Default: `'`

See Also

[Event Listener](#) | [Initialize Function](#) | [Reset Function](#) | [State Reader](#) | [Terminate Function](#)

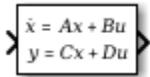
Topics

“Create Model to Initialize, Reset, and Terminate State”

“Create Test Harness to Generate Function Calls”

State-Space

Implement linear state-space system



$$\begin{cases} \dot{x} = Ax + Bu \\ y = Cx + Du \end{cases}$$

Library

Continuous

Description

The State-Space block implements a system whose behavior you define as

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

$$x|_{t=t_0} = x_0,$$

where x is the state vector, u is the input vector, y is the output vector and x_0 is the initial condition of the state vector. The matrix coefficients must have these characteristics:

- A must be an n -by- n matrix, where n is the number of states.
- B must be an n -by- m matrix, where m is the number of inputs.
- C must be an r -by- n matrix, where r is the number of outputs.
- D must be an r -by- m matrix.

	n	m
n	A	B
r	C	D

In general, the block has one input port and one output port. The number of rows in C or D matrix is the same as the width of the output port. The number of columns in the B or D matrix are the same as the width of the input port. If you want to model an autonomous linear system with no inputs, set the B and D matrices to empty. In this

case, the block acts as a source block with no input port and one output port, and implements the following system:

$$\dot{x} = Ax$$

$$y = Cx$$

$$x|_{t=t_0} = x_0.$$

Simulink software converts a matrix containing zeros to a sparse matrix for efficient multiplication.

Data Type Support

A State-Space block accepts and outputs real signals of type double.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

A

Specify the n-by-n matrix coefficient, where n is the number of states.

Default: 1

Parameter: A

Type: matrix

Value: '1'

Default: '1'

B

Specify the n-by-m matrix coefficient, where n is the number of states and m is the number of inputs.

Default: 1

Parameter: B

Type: matrix

Value: '1'

Default: '1'

C

Specify the r-by-n matrix coefficient, where r is the number of outputs and n is the number of states.

Default: 1

Parameter: C

Type: matrix

Value: '1'

Default: '1'

D

Specify the r-by-m matrix coefficient, where r is the number of outputs and m is the number of inputs.

Default: 1

Parameter: D

Type: matrix

Value: '1'

Default: '1'

Initial conditions

Specify the initial state vector.

Default: 0

The initial conditions of this block cannot be `inf` or `NaN`.

Parameter: `X0`

Type: vector

Value: `'0'`

Default: `'0'`

Absolute tolerance

Specify the absolute tolerance for computing block states.

Default: `auto`

- You can enter `auto`, `-1`, a positive real scalar or vector.
- If you enter `auto` or `-1`, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute block states.
- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

Parameter: `AbsoluteTolerance`

Type: character vector, scalar, or vector

Value: `'auto'` | `'-1'` | any positive real scalar or vector

Default: `'auto'`

State Name (e.g., 'position')

Assign a unique name to each state.

Default: `' '`

If this field is blank, no name assignment occurs.

- To assign a name to a single state, enter the name between quotes, for example, `'velocity'`.

- To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.
- The state names apply only to the selected block.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Parameter: ContinuousStateAttributes

Type: character vector

Value: ' ' | user-defined

Default: ' '

Examples

The following Simulink examples show how to use the State-Space block:

- `sldemo_dblcart1`
- `aero_vibrati`

Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	Only if $D \neq 0$
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Discrete State-Space

Transfer Fcn

Introduced before R2006a

Step

Generate step function



Library

Sources

Description

The Step block provides a step between two definable levels at a specified time. If the simulation time is less than the **Step time** parameter value, the block's output is the **Initial value** parameter value. For simulation time greater than or equal to the **Step time**, the output is the **Final value** parameter value.

The numeric block parameters must be of the same dimensions after scalar expansion. If the **Interpret vector parameters as 1-D** option is off, the block outputs a signal of the same dimensions and dimensionality as the parameters. If the **Interpret vector parameters as 1-D** option is on and the numeric parameters are row or column vectors (that is, single row or column 2-D arrays), the block outputs a vector (1-D array) signal. Otherwise, the block outputs a signal of the same dimensionality and dimensions as the parameters.

Data Type Support

The Step block outputs real signals of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Step time

Specify the time, in seconds, when the output jumps from the **Initial value** parameter to the **Final value** parameter. The default is 1 second.

Initial value

Specify the block output until the simulation time reaches the **Step time** parameter. The default is 0.

Final value

Specify the block output when the simulation time reaches and exceeds the **Step time** parameter. The default is 1.

Sample time

Specify the sample rate of step. See “Specify Sample Time” in the online documentation for more information.

Interpret vector parameters as 1-D

If selected, column or row matrix values for the Step block's numeric parameters result in a vector output signal; otherwise, the block outputs a signal of the same dimensionality as the parameters. If this option is not selected, the block always outputs a signal of the same dimensionality as the block's numeric parameters. See “Determining the Output Dimensions of Source Blocks” in the Simulink documentation.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Examples

The following Simulink examples show how to use the Step block:

- `sldemo_doublebounce`
- `sldemo_enginewc`

Characteristics

Data Types	Double
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled.
Code Generation	Yes

See Also

Ramp

Introduced before R2006a

Stop Simulation

Stop simulation when input is nonzero



Library

Sinks

Description

The Stop Simulation block stops the simulation when the input is nonzero. The simulation completes the current time step before terminating. If the block input is a vector, any nonzero vector element causes the simulation to stop.

When you use the Stop Simulation block in a For Iterator subsystem, the stop action occurs after execution of *all* the iterations in the subsystem during a time step. The stop action does not interrupt execution until the start of the next time step.

You cannot use the Stop Simulation block to pause the simulation. To create a block that pauses the simulation, see “Pause Simulation Using Assertion Blocks” in the Simulink documentation.

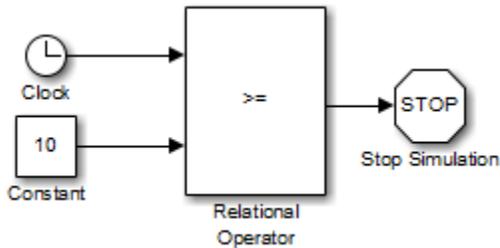
Data Type Support

The Stop Simulation block accepts real signals of type `double` or `Boolean`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Examples

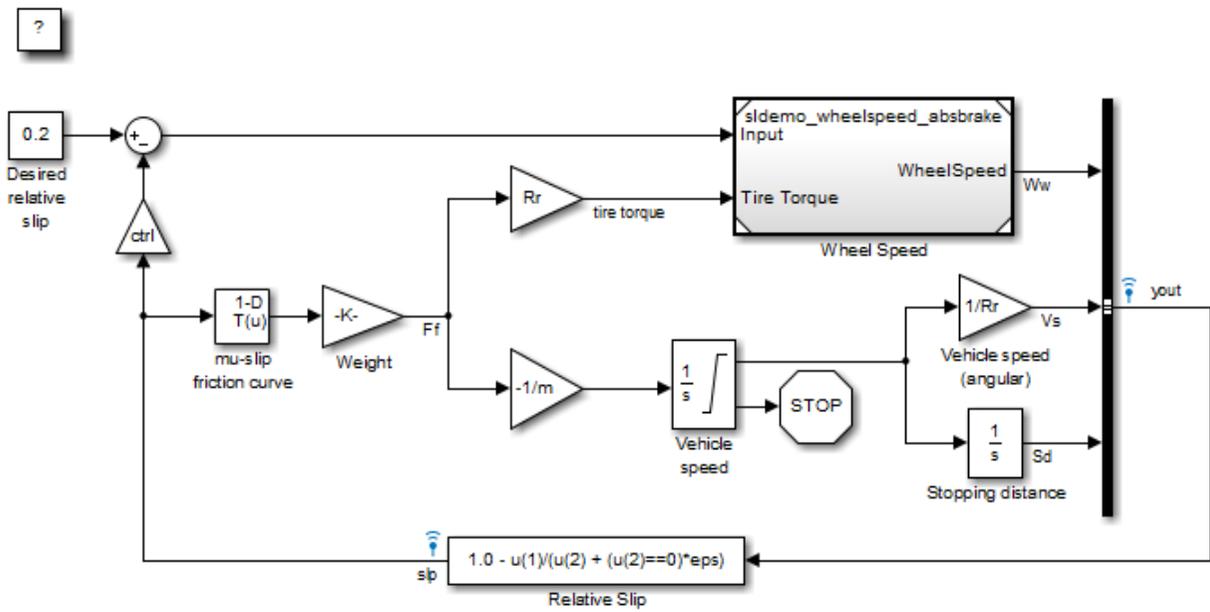
Usage with the Relational Operator Block

You can use the Stop Simulation block with the Relational Operator block to control when a simulation stops. For example, the following model stops simulation when the simulation time reaches 10.



Usage with the Integrator Block

You can use the Stop Simulation block with the Integrator block to control when a simulation stops. For example, the `sldemo_absbrake` model stops simulation when the saturation port of the Integrator block outputs a value of 1 or -1 .



Characteristics

Data Types	Double Boolean
Sample Time	Inherited from driving block
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

Group blocks to create model hierarchy

Library: Ports & Subsystems



Description

A Subsystem block contains a subset of blocks within a model or system. The Subsystem block can represent a virtual subsystem or a nonvirtual subsystem.

- Nonvirtual subsystem – Control when the contents of the subsystem are evaluated as a single unit (atomic execution). Create conditionally executed subsystems that run only when an event occurs on a triggering, function-call, action, or enabling input (see “Conditionally Executed Subsystems”).
- Virtual subsystems – Subsystem is neither conditionally nor atomically executed. Virtual subsystems do not have checksums. To determine if a subsystem is virtual, use the `get_param` function for the Boolean block parameter `IsSubsystemVirtual`.

An atomic subsystem is a Subsystem block with the block parameter **Treat as atomic unit** selected.

A codereuse subsystem is a Subsystem block with the parameter **Treat as atomic unit** selected and the parameter **Function packaging** set to `Reusable function`, specifying the function code generation format for the subsystem.

To create a subsystem, do one of the following:

- Copy a Subsystem block from the Ports & Subsystems library into your model. Then add blocks to the subsystem by opening the Subsystem block and copying blocks into it.
- Select all blocks and lines that make up the subsystem, and select **Diagram > Subsystem & Model Reference > Create Subsystem from Selection**. Simulink replaces the blocks with a Subsystem block, along with the necessary Inport and Outport blocks to reflect signals entering and leaving the subsystem.

The number of input ports drawn on the Subsystem block icon corresponds to the number of Inport blocks in the subsystem. Similarly, the number of output ports drawn on the block corresponds to the number of Outport blocks in the subsystem.

The Subsystem block supports signal label propagation through subsystem Inport and Outport blocks.

Ports

Input

in — Signal input to a subsystem block

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

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

Output

out — Signal output from a subsystem

scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

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

Parameters

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

Main

Show port labels — Select how to display port labels

`FromPortIcon` (default) | `FromPortBlockName` | `SignalName`

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

`none`

Do not display port labels.

`FromPortIcon`

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

`FromPortBlockName`

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

`SignalName`

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

Programmatic Use

Parameter: `ShowPortLabels`

Type: character vector

Value: `'FromPortIcon'` | `'FromPortBlockName'` | `'SignalName'`

Default: `'FromPortIcon'`

Read/Write permissions — Select access to contents of subsystem

`ReadWrite` (default) | `ReadOnly` | `NoReadOrWrite`

Control user access to the contents of the subsystem.

`ReadWrite`

Enable opening and modification of subsystem contents.

`ReadOnly`

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

NoReadOrWrite

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

Programmatic Use

Parameter: Permissions

Type: character vector

Value: 'ReadWrite' | 'ReadOnly' | 'NoReadOrWrite'

Default: 'ReadWrite'

Name of error callback function — Specify name of function

' ' (default) | function name

Enter name of a function to be called if an error occurs while Simulink is executing the subsystem.

Simulink passes two arguments to the function: the handle of the subsystem and a character vector that specifies the error type. If no function is specified, Simulink displays a generic error message if executing the subsystem causes an error.

Programmatic Use

Parameter: ErrorFcn

Type: character vector

Value: '' | '<function name>'

Default: ''

Permit hierarchical resolution — Select how to resolve workspace variable names

All (default) | ExplicitOnly | None

Select whether to resolve names of workspace variables referenced by this subsystem.

See “Symbol Resolution” and “Symbol Resolution Process” in the Simulink User's Guide for more information.

All

Resolve all names of workspace variables used by this subsystem, including those used to specify block parameter values and Simulink data objects (for example, Simulink.Signal objects).

ExplicitOnly

Resolve only names of workspace variables used to specify block parameter values, data store memory (where no block exists), signals, and states marked as “must resolve”.

None

Do not resolve any workspace variable names.

Programmatic Use

Parameter: PermitHierarchicalResolution

Type: character vector

Value: 'All' | 'ExplicitOnly' | 'None'

Default: 'All'

Treat as atomic unit — Control execution of a subsystem as one unit

off (default) | on

Causes Simulink to treat the subsystem as a unit when determining the execution order of block methods.

off

Treat all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem when determining block method execution order. This can cause execution of methods of blocks in the subsystem to be interleaved with execution of methods of blocks outside the subsystem.

on

Treat the subsystem as a unit when determining the execution order of block methods. For example, when it needs to compute the output of the subsystem, Simulink invokes the output methods of all the blocks in the subsystem before invoking the output methods of other blocks at the same level as the subsystem block.

Dependency

Selecting this parameter, enables the **Minimize algebraic loop occurrences**, **Sample time**, and **Function packaging** parameters. Using **Function packaging** requires a Simulink Coder license.

Programmatic Use**Parameter:** TreatAsAtomicUnit**Type:** character vector**Value:** 'off' | 'on'**Default:** 'off'**Minimize algebraic loop occurrences — Control elimination of algebraic loops**

off (default) | on

Try to eliminate any artificial algebraic loops that include the atomic subsystem

See also “Eliminate Artificial Algebraic Loops Caused by Atomic Subsystems”.

 off

Do not try to eliminate any artificial algebraic loops that include the atomic subsystem.

 on

Try to eliminate any artificial algebraic loops that include the atomic subsystem.

Dependency

To enable this parameter, select the **Treat as atomic unit** parameter.

Programmatic Use**Parameter:** MinAlgLoopOccurrences**Type:** character vector**Value:** 'off' | 'on'**Default:** 'off'**Sample time — Specify time interval**

-1 (default) | [Ts 0]

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

- If the blocks in the subsystem can run at different rates, specify the subsystem's 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 subsystem 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 subsystem must run 5 times a second. To ensure this, specify the sample time of the subsystem as `0.2`. In this example, if any of the blocks in the subsystem 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. Use this sample time if the blocks in the subsystem can run at different rates.

`[Ts 0]`

Specify periodic sample time.

Dependency

To enable this parameter, select the **Treat as atomic unit** parameter.

Programmatic Use

Parameter: `SystemSampleTime`

Type: character vector

Value: `'-1'` | `'[Ts 0]'`

Default: `'-1'`

Propagate execution context across subsystem boundary — Control execution across block boundary

`off` (default) | `on`

Enable execution context propagation across the boundary of this subsystem.

`off`

Do not enable execution context propagation across this subsystem's boundary.

`on`

Enable execution context propagation across this subsystem's boundary.

Dependency

Enable this parameter by adding an Enable port or Trigger port block to the Subsystem block.

Programmatic Use**Parameter:** PropExecContextOutsideSubsystem**Type:** character vector**Value:** 'off' | 'on'**Default:** 'off'**Variant control — Specify variant control (condition) expression**

Variant (default) | logical expression

Specify variant control (condition) expression that executes a variant Simulink Function block when the expression evaluates to `true`.

See also `Simulink.Variant`

Variant

Default name for a logical (Boolean) expression.

logical expression

A logical (Boolean) expression or a `Simulink.Variant` object representing a logical expression.

The function is activated when the expression evaluates to `true`.

If you want to generate code for your model, define the variables in the expression as `Simulink.Parameter` objects.

Dependency

Enable this parameter by adding a Subsystem block inside a Variant Subsystem block.

Programmatic Use**Block parameter:** VariantControl**Type:** character vector**Value:** 'Variant' | '<logical expression>'**Default:** 'Variant'**Treat as grouped when propagating variant conditions — Control treating subsystem as unit**

on (default) | off

Causes Simulink to treat the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks.

on

Simulink treats the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks. For example, when Simulink computes the variant condition of the subsystem, it propagates that condition to all the blocks in the subsystem.

off

Simulink treats all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem itself when determining their variant condition.

Programmatic Use

Parameter: `TreatAsGroupedWhenPropagatingVariantConditions`

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Code Generation

Function packaging — Select code format

`Auto` (default) | `Inline` | `Nonreusable function` | `Reusable function`

Select the code format to be generated for an atomic (nonvirtual) subsystem.

`Auto`

Simulink Coder chooses the optimal format for you based on the type and number of instances of the subsystem that exist in the model.

`Inline`

Simulink Coder inlines the subsystem unconditionally.

`Nonreusable function`

Simulink Coder explicitly generates a separate function in a separate file. Subsystems with this setting generate functions that might have arguments depending on the “Function interface” on page 1-0 parameter setting. You can name the generated function and file using parameters “Function name” on page 1-0 and “File name (no extension)” on page 1-0. These functions are not reentrant.

`Reusable function`

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

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

Tips

- When you want multiple instances of a subsystem to be represented as one reusable function, you can designate each one of them as `Auto` or as `Reusable function`. It is best to use one or the other, as 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 control of the function or file name for the subsystem code.
- The `Reusable function` and `Auto` options both try to determine if multiple instances of a subsystem exist and if the code can be reused. The difference between the options' behavior is that when reuse is not possible:
 - `Auto` yields inlined code, or if circumstances prohibit inlining, separate functions for each subsystem instance.
 - `Reusable function` yields a separate function with arguments for each subsystem instance in the model.
- If you select `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 a Simulink Coder license.
- To enable this parameter, select **Treat as atomic unit**.
- 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 a license for Embedded Coder and an ERT-based system target file)
 - Memory section for execution functions (requires a license for 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: character vector

Value: `'Auto' | 'Inline' | 'Nonreusable function' | 'Reusable function'`

Default: `'Auto'`

See also

- “Create a Subsystem”
- “Code Generation of Subsystems” (Simulink Coder)
- “Generate Code and Executables for Individual Subsystem” (Simulink Coder)
- “Inline Subsystem Code” (Simulink Coder)
- “Generate Subsystem Code as Separate Function and Files” (Simulink Coder)
- “Generate Reusable Code from Library Subsystems Shared Across Models” (Simulink Coder)
- “Generate Reusable Code from Library Subsystems Shared Across Models” (Simulink Coder)

Function name options — Select how to name generated function

`Auto (default) | Use subsystem name | User specified`

Select how Simulink Coder names the function it generates for the subsystem.

If you have an Embedded Coder license, you can control function names with options on the Configuration Parameter **Code Generation > Symbols** pane.

`Auto`

Assign a unique function name using the default naming convention, `model_subsystem()`, where `model` is the name of the model and `subsystem` is the name of the subsystem (or that of an identical one when code is being reused).

If you select `Reusable function` for the **Function packaging** parameter and there are multiple instances of the reusable subsystem in a model reference hierarchy, in order to generate reusable code for the subsystem, **Function name options** must be set to `Auto`.

Use subsystem name

Use the subsystem name as the function name. By default, the function name uses the naming convention *model_subsystem*.

Note When a subsystem is in a library block and the subsystem parameter “Function packaging” on page 1-0 is set to `Reusable function`, if you set the `Use subsystem name` option, the code generator uses the name of the library block for the subsystem's function name and file name.

User specified

Enable the **Function name** field. Enter any legal C or C++ function name, which must be unique.

Dependency

- This parameter requires a Simulink Coder license.
- Setting Code generation function packaging to `Nonreusable function` or `Reusable function` enables this parameter.
- Setting this parameter to `User specified` enables the Code generation function name parameter.

Programmatic Use

Parameter: `RTWFcnNameOpts`

Type: character vector

Value: `'Auto'` | `'Use subsystem name'` | `'User specified'`

Default: `'Auto'`

See also

For more information, see “Code Generation of Subsystems” (Simulink Coder).

Function name — Specify function name

`'` (default) | function name

Specify a unique, valid C or C++ function name for subsystem code.

Use this parameter if you want to give the function a specific name instead of allowing the Simulink Coder code generator to assign its own autogenerated name or use the

subsystem name. For more information, see “Code Generation of Subsystems” (Simulink Coder).

Dependency

- This parameter requires a Simulink Coder license.
- To enable this parameter, set the **Function name options** parameter to `User specified`.

Programmatic Use

Parameter: `RTWFcnName`

Type: character vector

Value: `''` | `'<function name>'`

Default: `''`

File name options — Specify how to name generated file

`Auto` (default) | `Use subsystem name` | `Use function name` | `User specified`

Select how Simulink Coder names the separate file for the function it generates for the subsystem.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

`Auto`

Depending on the configuration of the subsystem and how many instances are in the model, `Auto` yields different results:

- If the code generator does *not* generate a separate file for the subsystem, the subsystem code is generated within the code module generated from the subsystem's parent system. If the subsystem's parent is the model itself, the subsystem code is generated within `model.c` or `model.cpp`.
- If you select `Reusable function` for the **Function packaging** parameter and your generated code is under source control, consider specifying a **File name options** value other than `Auto`. This prevents the generated file name from changing due to unrelated model modifications, which is problematic for using source control to manage configurations.
- If you select `Reusable function` for the **Function packaging** parameter and there are multiple instances of the reusable subsystem in a model reference hierarchy, in order to generate reusable code for the subsystem, **File name options** must be set to `Auto`.

Use subsystem name

The code generator generates a separate file, using the subsystem (or library block) name as the file name.

Note When **File name options** is set to Use subsystem name, the subsystem file name is mangled if the model contains Model blocks, or if a model reference target is being generated for the model. In these situations, the file name for the subsystem consists of the subsystem name prefixed by the model name.

Use function name

The code generator uses the function name specified by **Function name options**) as the file name.

User specified

This option enables the **File name (no extension)** text entry field. The code generator uses the name you enter as the file name. Enter any file name, but do not include the .c or .cpp (or any other) extension. This file name need not be unique.

Note While a subsystem source file name need not be unique, you must avoid giving nonunique names that result in cyclic dependencies (for example, `sys_a.h` includes `sys_b.h`, `sys_b.h` includes `sys_c.h`, and `sys_c.h` includes `sys_a.h`).

Dependency

- This parameter requires a Simulink Coder license.
- To enable this parameter, set **Function packaging** to Nonreusable function or Reusable function.
- Setting this parameter to User specified enables the **File name (no extension)** parameter.

Programmatic Use

Parameter: RTWFileNameOpts

Type: character vector

Value: 'Auto' | 'Use subsystem name' | 'Use function name' | 'User specified'

Default: 'Auto'

File name (no extension) — Specify file name

' ' (default) | file name

The filename that you specify does not have to be unique. However, avoid giving non-unique names that result in cyclic dependencies (for example, `sys_a.h` includes `sys_b.h`, `sys_b.h` includes `sys_c.h`, and `sys_c.h` includes `sys_a.h`).

For more information, see “Code Generation of Subsystems” (Simulink Coder).

Dependency

- This parameter requires a Simulink Coder license.
- To enable this parameter, set **File name options** to `User specified`.

Programmatic Use

Parameter: `RTWFileName`

Type: character vector

Value: ' ' | '<file name>'

Default: ' '

Function with separate data — Control code generation for subsystem

off (default) | on

Generate subsystem function code in which the internal data for an atomic subsystem is separated from its parent model and is owned by the subsystem.

off

Do not generate subsystem function code in which the internal data for an atomic subsystem is separated from its parent model and is owned by the subsystem.

on

Generate subsystem function code in which the internal data for an atomic subsystem is separated from its parent model and is owned by the subsystem. The subsystem data structure is declared independently from the parent model data structures. A subsystem with separate data has its own block I/O and `DWork` data structure. As a result, the generated code for the subsystem is easier to trace and test. The data separation also tends to reduce the maximum size of global data structures throughout the model, because they are split into multiple data structures.

Dependency

- This parameter requires a license for Embedded Coder and an ERT-based system target file.
- To enable this parameter, set **Function packaging** to `Nonreusable function`.
- Selecting this parameter enables these parameters:
 - **Memory section for constants**
 - **Memory section for internal data**
 - **Memory section for parameters**

Programmatic Use

Parameter: `FunctionWithSeparateData`

Type: character vector

Value: `'off' | 'on'`

Default: `'off'`

See also

- See the Subsystem block reference page for more information.
- For details on how to generate modular function code for an atomic subsystem, see “Generate Modular Function Code” (Embedded Coder) in the Embedded Coder documentation.
- For details on how to apply memory sections to atomic subsystems, see “Override Default Placement by Using Atomic Subsystems” (Embedded Coder) in the Embedded Coder documentation.

Function interface — Select to use arguments with generate function

`void_void` (default) | `Allow arguments`

Select to use arguments with generated function.

`void_void`

Generate a function without arguments and pass data as global variables. For example:

```
void subsystem_function(void)
```

`Allow arguments`

Generate a function that uses arguments instead of passing data as global variables. This specification reduces global RAM. It might reduce code size and improve

execution speed, and allow the code generator to apply additional optimizations. For example:

```
void subsystem_function(real_T rtu_In1, real_T rtu_In2,  
                       real_T *rty_Out1)
```

Dependency

- This parameter requires an Embedded Coder license and an ERT-based system target file.
- To enable this parameter, set **Function packaging** to `Nonreusable function`.

Programmatic Use

Parameter: `FunctionInterfaceSpec`

Type: character vector

Value: `'void_void'` | `'Allow arguments'`

Default: `'void_void'`

See also

- “Reduce Global Variables in Nonreusable Subsystem Functions” (Embedded Coder)
- “Generate Modular Function Code” (Embedded Coder) in the Embedded Coder documentation

Memory section for initialize/terminate functions — Select how to apply memory sections

`Inherit from model (default)` | `Default` | `The memory section of interest`

Select how Embedded Coder applies memory sections to the subsystem initialization and termination functions.

`Inherit from model`

Apply the root model's memory sections to the subsystem's function code

`Default`

Do not apply memory sections to the subsystem's system code, overriding any model-level specification

`The memory section of interest`

Apply one of the model's memory sections to the subsystem

Tips

- The possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See “Control Data and Function Placement in Memory by Inserting Pragmas” (Embedded Coder) and “Model Configuration Parameters: Code Generation Memory Sections” (Embedded Coder) in the Embedded Coder documentation.
- If you have not configured the model with a package, `Inherit from model` is the only value that appears. Otherwise, the list includes `Default` and all memory sections the model's package contains.
- These options can be useful for overriding the model's memory section settings for the given subsystem.

Dependency

- This parameter requires a license for Embedded Coder software and an ERT-based system target file.
- To enable this parameter, set **Function packaging** to `Nonreusable function` or `Reusable function`.

Programmatic Use**Parameter:** `RTWMemSecFuncInitTerm`**Type:** character vector**Value:** `'Inherit from model' | 'Default' | 'The memory section of interest'`**Default:** `'Inherit from model'`**See also**

- See the Subsystem block reference page for more information.
- For details on how to apply memory sections to atomic subsystems, see “Override Default Placement by Using Atomic Subsystems” (Embedded Coder) in the Embedded Coder documentation.

Memory section for execution functions — Select how to apply memory sections`Inherit from model (default) | Default | The memory section of interest`

Select how Embedded Coder applies memory sections to the subsystem's execution functions.

Inherit from model

Apply the root model's memory sections to the subsystem's function code

Default

Do not apply memory sections to the subsystem system code, overriding any model-level specification

The memory section of interest

Apply one of the model's memory sections to the subsystem

Tips

- The possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See “Control Data and Function Placement in Memory by Inserting Pragmas” (Embedded Coder) and “Model Configuration Parameters: Code Generation Memory Sections” (Embedded Coder) in the Embedded Coder documentation.
- If you have not configured the model with a package, `Inherit from model` is the only value that appears. Otherwise, the list includes `Default` and all memory sections the model's package contains.
- These options can be useful for overriding the model's memory section settings for the given subsystem.

Dependency

- This parameter requires a license for Embedded Coder software and an ERT-based system target file.
- To enable this parameter, set **Function packaging** to `Nonreusable function` or `Reusable function`.

Programmatic Use

Parameter: `RTWMemSecFuncExecute`

Type: character vector

Value: `'Inherit from model' | 'Default' | 'The memory section of interest'`

Default: `'Inherit from model'`

See also

- See the Subsystem block reference page for more information.

- For details on how to apply memory sections to atomic subsystems, see “Override Default Placement by Using Atomic Subsystems” (Embedded Coder) in the Embedded Coder documentation.

Memory section for constants — Select how to apply memory sections

`Inherit from model` (default) | `Default` | `The memory section of interest`

Select how Embedded Coder applies memory sections to the subsystem constants.

`Inherit from model`

Apply the root model's memory sections to the subsystem's data

`Default`

Not apply memory sections to the subsystem's data, overriding any model-level specification

`The memory section of interest`

Apply one of the model's memory sections to the subsystem

Tips

- The memory section that you specify applies to the corresponding global data structures in the generated code. For basic information about the global data structures generated for atomic subsystems, see “Default Data Structures in the Generated Code” (Simulink Coder).
- Can be useful for overriding the model's memory section settings for the given subsystem.
- The possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See “Control Data and Function Placement in Memory by Inserting Pragmas” (Embedded Coder) in the Embedded Coder documentation.
- If you have not configured the model with a package, `Inherit from model` is the only value that appears. Otherwise, the list includes `Default` and all memory sections the model's package contains.

Dependency

- This parameter requires a license for Embedded Coder and an ERT-based system target file.
- To enable this parameter, set **Function packaging** to `Nonreusable function` and select the **Function with separate data** parameter

Programmatic Use

Parameter: RTWMemSecDataConstants

Type: character vector

Value: 'Inherit from model' | 'Default' | 'The memory section of interest'

Default: 'Inherit from model'

See also

- See the Subsystem block reference page for more information.
- For details on how to apply memory sections to atomic subsystems, see “Override Default Placement by Using Atomic Subsystems” (Embedded Coder) in the Embedded Coder documentation.

Memory section for internal data — Select how to apply memory sections

Inherit from model (default) | Default | The memory section of interest

Select how Embedded Coder applies memory sections to the subsystem internal data.

Inherit from model

Apply the root model's memory sections to the subsystem's data

Default

Not apply memory sections to the subsystem's data, overriding any model-level specification

The memory section of interest

Apply one of the model's memory sections to the subsystem

Tips

- The memory section that you specify applies to the corresponding global data structures in the generated code. For basic information about the global data structures generated for atomic subsystems, see “Default Data Structures in the Generated Code” (Simulink Coder).
- Can be useful for overriding the model's memory section settings for the given subsystem.
- The possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See “Control Data and Function Placement in Memory by Inserting Pragmas” (Embedded Coder) in the Embedded Coder documentation.

- If you have not configured the model with a package, `Inherit from model` is the only value that appears. Otherwise, the list includes `Default` and all memory sections the model's package contains.

Dependency

- This parameter requires a license for Embedded Coder and an ERT-based system target file.
- To enable this parameter, set **Function packaging** to `Nonreusable function` and select the **Function with separate data** parameter.

Programmatic Use**Parameter:** `RTWMemSecDataInternal`**Type:** character vector**Value:** `'Inherit from model' | 'Default' | 'The memory section of interest'`**Default:** `'Inherit from model'`**See also**

- See the Subsystem block reference page for more information.
- For details on how to apply memory sections to atomic subsystems, see “Override Default Placement by Using Atomic Subsystems” (Embedded Coder) in the Embedded Coder documentation.

Memory section for parameters — Select how to apply memory sections`Inherit from model (default) | Default | The memory section of interest`

Select how Embedded Coder applies memory sections to the subsystem parameters.

`Inherit from model`

Apply the root model's memory sections to the subsystem's function code

`Default`

Not apply memory sections to the subsystem's system code, overriding any model-level specification

`The memory section of interest`

Apply one of the model's memory sections to the subsystem

Tips

- The memory section that you specify applies to the corresponding global data structure in the generated code. For basic information about the global data structures generated for atomic subsystems, see “Default Data Structures in the Generated Code” (Simulink Coder).
- Can be useful for overriding the model's memory section settings for the given subsystem.
- The possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See “Control Data and Function Placement in Memory by Inserting Pragmas” (Embedded Coder) in the Embedded Coder documentation.
- If you have not configured the model with a package, `Inherit from model` is the only value that appears. Otherwise, the list includes `Default` and all memory sections the model's package contains.

Dependency

- This parameter requires a license for Embedded Coder and an ERT-based system target file.
- To enable this parameter, set **Function packaging** to `Nonreusable function` and select the **Function with separate data** parameter.

Programmatic Use

Parameter: `RTWMemSecDataParameters`

Type: character vector

Value: `'Inherit from model' | 'Default' | 'The memory section of interest'`

Default: `'Inherit from model'`

See also

- See the Subsystem block reference page for more information.
- For details on how to apply memory sections to atomic subsystems, see “Override Default Placement by Using Atomic Subsystems” (Embedded Coder) in the Embedded Coder documentation.

See Also

Enabled Subsystem | Enabled and Triggered Subsystem | Function-Call Subsystem | Triggered Subsystem

Topics

“Create a Subsystem”

“Using Function-Call Subsystems”

“Export-Function Models”

Introduced in R2007a

Sum, Add, Subtract, Sum of Elements

Add or subtract inputs



Library

Math Operations

Description

The Sum block performs addition or subtraction on its inputs. This block can add or subtract scalar, vector, or matrix inputs. It can also collapse the elements of a signal.

You specify the operations of the block with the **List of signs** parameter. Plus (+), minus (-), and spacer () characters indicate the operations to be performed on the inputs:

- If there are two or more inputs, then the number of + and - characters must equal the number of inputs. For example, “+-+” requires three inputs and configures the block to subtract the second (middle) input from the first (top) input, and then add the third (bottom) input.
- All nonscalar inputs must have the same dimensions. Scalar inputs will be expanded to have the same dimensions as the other inputs.
- A spacer character creates extra space between ports on the block's icon.
- For a round Sum block, the first input port is the port closest to the 12 o'clock position going in a counterclockwise direction around the block. Similarly, other input ports appear in counterclockwise order around the block.
- If only addition of all inputs is required, then a numeric parameter value equal to the number of inputs can be supplied instead of “+” characters.

- If only one input port is required, a single “+” or “-” collapses the element via the specified operation.

The Sum block first converts the input data type(s) to its accumulator data type, then performs the specified operations. The block converts the result to its output data type using the specified rounding and overflow modes.

Calculation of Block Output

Output calculation for the Sum block depends on the number of block inputs and the sign of input ports:

If the Sum block has...	And...	The formula for output calculation is...	Where...
One input port	The input port sign is +	$y = e[0] + e[1] + e[2] \dots + e[m]$	$e[i]$ is the i^{th} element of input u
	The input port sign is -	$y = 0.0 - e[0] - e[1] - e[2] \dots - e[m]$	
Two or more input ports	All input port signs are -	$y = 0.0 - u[0] - u[1] - u[2] \dots - u[n]$	$u[i]$ is the input to the i^{th} input port
	The k^{th} input port is the first port where the sign is +	$y = u[k] - u[0] - u[1] - u[2] - u[k-1] (+/-) u[k+1] \dots (+/-) u[n]$	

Data Type Support

The Sum block accepts real or complex signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The inputs can be of different data types, unless you select the **Require all inputs to have the same data type** parameter. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Show data type assistant

Display the **Data Type Assistant**.

The **Data Type Assistant** helps you set the **Output data type** parameter.

For more information, see “Control Signal Data Types”.

Icon shape

Designate the icon shape of the block.

Default: round

rectangular

Designate the icon shape of the block as rectangular.

round

Designate the icon shape of the block as round.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

List of signs

Enter plus (+) and minus (-) characters.

Default: |++

- Addition is the default operation, so if you only want to add the inputs, enter the number of input ports.

- For a single vector input, “+” or “-” will collapse the vector using the specified operation.
- Enter as many plus (+) and minus (-) characters as there are inputs.

You can manipulate the positions of the input ports on the block by inserting spacers (|) between the signs in the **List of signs** parameter. For example, “++|--” creates an extra space between the second and third input ports.

Entering only one element enables the **Sum over** parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Sum over

Select dimension over which to perform the sum over operation.

Default: All dimensions

All dimensions

Sum all input elements, yielding a scalar.

Specified dimension

Display the **Dimension** parameter, where you specify the dimension over which to perform the operation.

Selecting Specified dimension enables the **Dimension** parameter.

List of signs (when it has only one element) enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Dimension

Specify the dimension over which to perform the operation.

Default: 1

The block follows the same summation rules as the MATLAB `sum` function.

Suppose that you have a 2-by-3 matrix U .

- Setting **Dimension** to 1 results in the output Y being computed as:

$$Y = \sum_{i=1}^2 U(i, j)$$

- Setting **Dimension** to 2 results in the output Y being computed as:

$$Y = \sum_{j=1}^3 U(i, j)$$

If the specified dimension is greater than the dimension of the input, an error message appears.

Setting **Sum over** to `Specified dimension` enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Require all inputs to have the same data type

Require that all inputs have the same data type.

Default: Off

On

Require that all inputs have the same data type.

Off

Do not require that all inputs have the same data type.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Lock data type settings against changes by the fixed-point tools

Select to lock data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks all data type settings for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change data type settings for this block.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: Floor

Ceiling

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

Floor

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

Nearest

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Parameter: `RndMeth`

Type: character vector

Value: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`

Default: `'Floor'`

For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off

On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Parameter: SaturateOnIntegerOverflow

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Accumulator data type

Specify the accumulator data type.

Default: Inherit: Inherit via internal rule

Inherit: Inherit via internal rule

Use internal rule to determine accumulator data type.

Inherit: Same as first input

Use data type of first input signal.

double

Accumulator data type is double.

single

Accumulator data type is single.

int8

Accumulator data type is int8.

uint8

Accumulator data type is uint8.

int16

Accumulator data type is int16.

uint16

Accumulator data type is uint16.

int32

Accumulator data type is int32.

uint32

Accumulator data type is uint32.

fixdt(1,16,0)

Accumulator data type is fixed point fixdt(1,16,0).

fixdt(1,16,2^0,0)

Accumulator data type is fixed point fixdt(1,16,2^0,0).

<data type expression>

The name of a data type object, for example `Simulink.NumericType`

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Specify Data Types Using Data Type Assistant”.

Mode

Select the category of accumulator data to specify

Default: `Inherit`

`Inherit`

Specifies inheritance rules for data types. Selecting `Inherit` enables a list of possible values:

- `Inherit` via internal rule (default)
- Same as first input

`Built in`

Specifies built-in data types. Selecting `Built in` enables a list of possible values:

- `double` (default)
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Specifies fixed-point data types.

`Expression`

Specifies expressions that evaluate to data types. Selecting `Expression` enables you to enter an expression.

Clicking the **Show data type assistant** button for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data to be signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data to be signed.

`Unsigned`

Specify the fixed-point data to be unsigned.

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Word length

Specify the bit size of the word that will hold the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Large word sizes represent large values with greater precision than small word sizes.

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: `Binary point`

`Binary point`

Specify binary point location.

`Slope and bias`

Enter slope and bias.

Selecting **Mode** > `Fixed point` for the accumulator data type enables this parameter.

Selecting `Binary point` enables:

- **Fraction length**

Selecting `Slope` and `bias` enables:

- **Slope**
- **Bias**

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope` and `bias` for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope` and `bias` for the accumulator data type enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Output minimum

Lower value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMin

Type: character vector

Value: ' [] '

Default: ' [] '

Output maximum

Upper value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMax

Type: character vector

Value: ' [] '

Default: ' [] '

Output data type

Specify the output data type.

Default: Inherit: Inherit via internal rule

Inherit: Inherit via internal rule

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of `Inherit: Same as first input`.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

Note The accumulator internal rule favors greater numerical accuracy, possibly at the cost of less efficient generated code. To get the same accuracy for the output, set the output data type to `Inherit: Inherit same as accumulator`.

Inherit: Inherit via back propagation

Use data type of the driving block.

Inherit: Same as first input

Use data type of first input signal.

Inherit: Same as accumulator

Output data type is the same as accumulator data type.

double

Output data type is double.

single

Output data type is single.

int8

Output data type is int8.

uint8

Output data type is uint8.

int16

Output data type is int16.

uint16

Output data type is uint16.

int32

Output data type is int32.

uint32

Output data type is uint32.

fixdt(1,16,0)

Output data type is fixed point fixdt(1,16,0).

fixdt(1,16,2^0,0)

Output data type is fixed point fixdt(1,16,2^0,0).

<data type expression>

Use a data type object, for example, Simulink.NumericType.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

For more information, see “Control Signal Data Types”.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables a second menu/text box to the right. Select one of the following choices:

- `Inherit via internal rule (default)`
- `Inherit via back propagation`
- `Same as first input`
- `Same as accumulator`

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

`Fixed point`

Fixed-point data types.

`Expression`

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: Signed

Signed

Specify the fixed-point data as signed.

Unsigned

Specify the fixed-point data as unsigned.

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Binary point

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting Slope and bias enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope and bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope and bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Examples

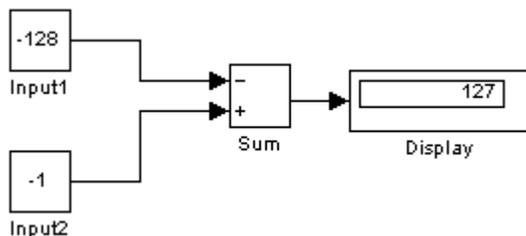
How the Sum Block Reorders Inputs

If you use $-$ on the first input port, the Sum block reorders the inputs so that, if possible, the first input uses a $+$ operation. For example, in the expression $\text{output} = -a-b+c$, the Sum block reorders the inputs so that $\text{output} = c-a-b$. To initialize the accumulator, the Sum block uses the first $+$ input port.

The block avoids performing a unary minus operation on the first operand a because doing so can change the value of a for fixed-point data types. In that case, the output value differs from the result of accumulating the values for a , b , and c .

Tip To explicitly specify a unary minus operation for $\text{output} = -a-b+c$, you can use the Unary Minus block in the Math Operations library.

Suppose that you have the following model:



The following block parameters apply:

- Both Constant blocks, Input1 and Input 2, use `int8` for the **Output data type**.
- The Sum block uses `int8` for both **Accumulator data type** and **Output data type**.
- The Sum block has **Saturate on integer overflow** turned on.

The Sum block reorders the inputs so that the following operations occur and you get the ideal result of 127.

Step	Block Operation
1	Reorders inputs from $(-Input1 + Input2)$ to $(Input2 - Input1)$.
2	Initializes the accumulator by using the first + input port: $Accumulator = int8(-1) = -1$
3	Continues to accumulate values: $Accumulator = Accumulator - int8(-128) = 127$
4	Calculates the block output: $Output = int8(127) = 127$

If the Sum block does *not* reorder the inputs, the following operations occur instead and you get the nonideal result of 126.

Step	Block Operation
1	Initializes the accumulator by using the first input port: $Accumulator = int8(-(-128)) = 127$ Because saturation is on, the initial value of the accumulator saturates at 127 and does not wrap.
2	Continues to accumulate values: $Accumulator = Accumulator + int8(-1) = 126$
3	Calculates the block output: $Output = int8(126) = 126$

Characteristics

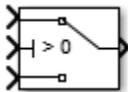
Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes

Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Switch

Switch output between first input and third input based on value of second input



Library

Signal Routing

Description

Types of Block Inputs

The Switch block passes through the first input or the third input based on the value of the second input. The first and third inputs are called *data inputs*. The second input is called the *control input*. Specify the condition under which the block passes the first input by using the **Criteria for passing first input** and **Threshold** parameters.

To immediately back propagate a known output data type to the first and third input ports, set the **Output data type** parameter to `Inherit: Inherit via internal rule` and select the **Require all data port inputs to have the same data type** check box.

Limitations on Data Inputs

The sizes of the two data inputs can be different if you select **Allow different data input sizes** on the block dialog box. However, this block does not support variable-size input signals. Therefore, the size of each input cannot change during simulation.

If the data inputs to the Switch block are buses, the element names of both buses must be the same. Using the same element names ensures that the output bus has the same element names no matter which input bus the block selects. To ensure that your model meets this requirement, use a bus object to define the buses and set the **Element name**

mismatch diagnostic to **error**. See “Connectivity Diagnostics Overview” for more information.

Block Icon Appearance

The block icon helps you identify **Criteria for passing first input** and **Threshold** without having to open the block dialog box.

For information about port order for various block orientations, see “Port Location After Rotating or Flipping” in the Simulink documentation.

Block Behavior for Boolean Control Input

When the control input is a Boolean signal, use one of these combinations of criteria and threshold value:

- $u2 \geq \text{Threshold}$, where the threshold value equals 1
- $u2 > \text{Threshold}$, where the threshold value equals 0
- $u2 \sim 0$

Otherwise, the Switch block ignores the threshold and uses the Boolean input for signal routing. For a control input of 1, the block passes the first input, and for a control input of 0, the block passes the third input. In this case, the block icon changes *after compile time* and uses **T** and **F** to label the first and third inputs, respectively.

Data Type Support

The control input can be of any data type that Simulink supports, including fixed-point and enumerated types. The control input cannot be complex. If the control input is enumerated, the **Threshold** parameter must be a value of the same enumerated type.

The data inputs can be of any data type that Simulink supports. If either data input is of an enumerated type, the other must be of the same enumerated type.

When the output is of enumerated type, both data inputs should use the same enumerated type as the output.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

- “Criteria for passing first input” on page 1-1588
- “Threshold” on page 1-1589
- “Enable zero-crossing detection” on page 1-1590
- “Sample time” on page 1-1590
- “Require all data port inputs to have the same data type” on page 1-1591
- “Lock output data type setting against changes by the fixed-point tools” on page 1-1591
- “Integer rounding mode” on page 1-1592
- “Saturate on integer overflow” on page 1-1593
- “Allow different data input sizes” on page 1-1594
- “Output minimum” on page 1-1594
- “Output maximum” on page 1-1595
- “Output data type” on page 1-1595
- “Mode” on page 1-1597
- “Data type override” on page 1-1598
- “Signedness” on page 1-1599
- “Word length” on page 1-1599
- “Scaling” on page 1-1600
- “Fraction length” on page 1-1601
- “Slope” on page 1-1601
- “Bias” on page 1-1601

Criteria for passing first input

Select the condition under which the block passes the first input. If the control input meets the condition set in the **Criteria for passing first input** parameter, the block passes the first input. Otherwise, the block passes the third input.

Default: `u2 > Threshold`

`u2 >= Threshold`

Checks whether the control input is greater than or equal to the threshold value.

`u2 > Threshold`

Checks whether the control input is greater than the threshold value.

`u2 ~=0`

Checks whether the control input is nonzero.

Note The Switch block does not support `u2 ~=0` mode for enumerated data types.

When the control input is a Boolean signal, use one of these combinations of condition and threshold value:

- `u2 >= Threshold`, where the threshold value equals 1
- `u2 > Threshold`, where the threshold value equals 0
- `u2 ~=0`

Otherwise, the Switch block ignores threshold values and uses the Boolean value for signal routing. For a value of 1, the block passes the first input, and for a value of 0, the block passes the third input. A warning message that describes this behavior also appears in the MATLAB Command Window.

Selecting `u2 ~=0` disables the **Threshold** parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Threshold

Assign the switch threshold that determines which input the block passes to the output.

Default: 0

Minimum: value from the **Output minimum** parameter

Maximum: value from the **Output maximum** parameter

To specify a nonscalar threshold, use brackets. For example, the following entries are valid:

- [1 4 8 12]
- [MyColors.Red, MyColors.Blue]

Setting **Criteria for passing first input** to `u2 ~=0` disables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Default: On

On

Enable zero-crossing detection.

Off

Do not enable zero-crossing detection.

Parameter: ZeroCross

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Require all data port inputs to have the same data type

Require all data inputs to have the same data type.

Default: Off

On

Requires all data inputs to have the same data type.

Off

Does not require all data inputs to have the same data type.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: LockScale

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: `Floor`

`Ceiling`

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

`Zero`

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Parameter: `RndMeth`

Type: character vector

Value: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`

Default: `'Floor'`

For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off

On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Parameter: SaturateOnIntegerOverflow

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Allow different data input sizes

Select this check box to allow input signals with different sizes.

Default: Off

On

Allows input signals with different sizes, and propagates the input signal size to the output signal. If the two data inputs are variable-size signals, the maximum size of the signals can be equal or different.

Off

Inputs signals must be the same size.

Parameter: AllowDiffInputSizes

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Output minimum

Specify the minimum value that the block should output.

Default: []

The default value is [] (unspecified).

Simulink uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL

or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

This number must be a finite real double scalar value.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Output maximum

Specify the maximum value that the block should output.

Default: []

The default value is [] (unspecified).

Simulink uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

This number must be a finite real double scalar value.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Output data type

Specify the output data type.

Default: Inherit: Inherit via internal rule

Inherit: Inherit via internal rule

Uses the following rules to determine the output data type.

Data Type of First Input Port	Output Data Type
Has a larger positive range than the third input port	Inherited from the first input port
Has the same positive range as the third input port	Inherited from the third input port
Has a smaller positive range than the third input port	

Inherit: Inherit via back propagation

Uses data type of the driving block.

double

Specifies output data type is double.

single

Specifies output data type is single.

int8

Specifies output data type is int8.

uint8

Specifies output data type is uint8.

int16

Specifies output data type is int16.

uint16

Specifies output data type is uint16.

int32

Specifies output data type is int32.

uint32

Specifies output data type is uint32.

fixdt(1,16,0)

Specifies output data type is fixed point `fixdt(1,16,0)`.

fixdt(1,16,2^0,0)

Specifies output data type is fixed point `fixdt(1,16,2^0,0)`.

Enum: <class name>

Uses an enumerated data type, for example, Enum: BasicColors.

<data type expression>

Uses a data type object, for example, Simulink.NumericType.

When the output is of enumerated type, both data inputs should use the same enumerated type as the output.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Control Signal Data Types” for more information.

Mode

Select the category of data to specify.

Default: Inherit

Inherit

Specifies inheritance rules for data types. Selecting Inherit enables a list of possible values:

- Inherit via internal rule (default)
- Inherit via back propagation

Built in

Specifies built-in data types. Selecting Built in enables a list of possible values:

- double (default)
- single
- int8
- uint8
- int16
- uint16

- `int32`
- `uint32`

Fixed point

Specifies fixed-point data types.

Enumerated

Specifies enumerated data types. Selecting `Enumerated` enables you to enter a class name.

Expression

Specifies expressions that evaluate to data types. Selecting `Expression` enables you to enter an expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

Data type override

Specify data type override mode for this signal.

Default: `Inherit`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For

example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specifies the fixed-point data as signed.

`Unsigned`

Specifies the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: `16`

Minimum: `0`

Maximum: `32`

Large word sizes represent large values with greater precision than small word sizes.

Selecting **Mode** > `Fixed point` enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Best precision

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Best precision

Specify best-precision values.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Selecting Slope and bias enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

For more information, see “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > Binary point enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > Slope and bias enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > Slope and bias enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type” for more information.

Bus Support

The Switch block is a bus-capable block. The data inputs can be virtual or nonvirtual bus signals subject to the following restrictions:

- All the buses must be equivalent (same hierarchy with identical names and attributes for all elements).
- All signals in a nonvirtual bus input to a Switch block must have the same sample time. The requirement holds even if the elements of the associated bus object specify inherited sample times.

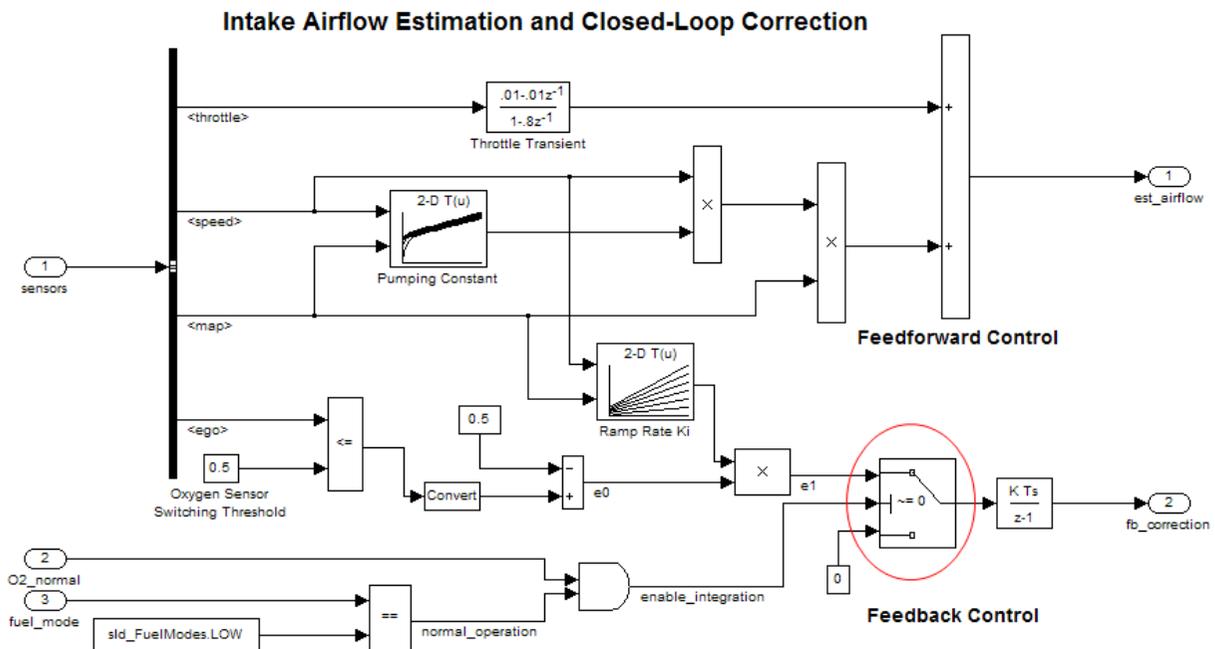
You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “Specify Bus Signal Sample Times” and Bus-Capable Blocks for more information.

You can use an array of buses as an input signal to a Switch block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”. When using an array of buses with a Switch block, set the **Threshold** parameter to a scalar value.

Examples

Use of Boolean Input for the Control Port

In the `sldemo_fuelsys` model, the `fuel_rate_control/airflow_calc` subsystem uses a Switch block:

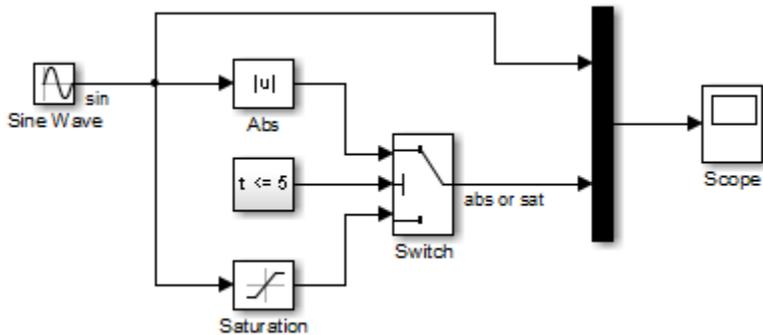


The value of the control port on the Switch block determines whether or not feedback correction occurs. The control port value depends on the output of the Logical Operator block.

When the Logical Operator block output is...	The control port of the Switch block is...	And feedback control...
TRUE	1	Occurs
FALSE	0	Does not occur

Use of Simulation Time for the Control Port

The `sldemo_zerowing` model uses a Switch block:



The value of the control port on the Switch block determines when the output changes from the first input to the third input.

When simulation time is...	The Switch block output is...
Less than or equal to 5	The first input from the Abs block
Greater than 5	The third input from the Saturation block

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

See Also

Multiport Switch

Introduced before R2006a

Switch Case

Implement C-like `switch` control flow statement



Library

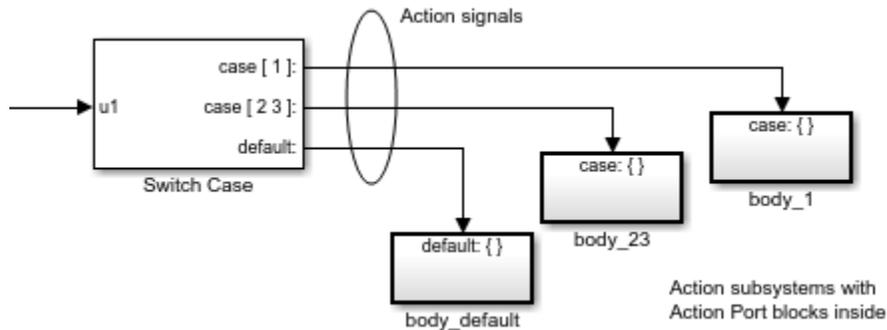
Ports & Subsystems

Description

A Switch Case block receives a single input. Each output port is attached to a Switch Case Action Subsystem. Data outputs are action signals to Switch Case Action subsystems, which you create with Action Port blocks and subsystems.

The Switch Case block uses its input value to select a case condition that determines which subsystem to execute. The cases are evaluated top down starting with the first case. If a case value (in brackets) corresponds to the value of the input, its Switch Case Action Subsystem is executed.

If a `default` case exists, it executes if none of the other cases executes. Providing a `default` case is optional, even if the other case conditions do not exhaust every possible value. The following diagram shows a completed Simulink `switch` control flow statement:



Cases for the Switch Case block contain an implied break after their Switch Case Action subsystems are executed. Thus there is no fall-through behavior for the Simulink switch control flow statement as found in standard C switch statements. The following pseudocode represents generated code for the preceding switch control example:

```
switch (u1) {
    case [u1=1]:
        body_1;
        break;
    case [u1=2 or u1=3]:
        body_23;
        break;
    default:
        body_default;
}
```

To construct the Simulink switch control flow statement shown in the above example:

- 1 Place a Switch Case block in the current system and attach the input port labeled u1 to the source of the data you are evaluating.
- 2 Open the Switch Case block dialog box and update parameters:
 - a Populate the **Case conditions** field with the individual cases.
 - b To show a default case, select the **Show default case** check box.
- 3 Create a Switch Case Action Subsystem for each case port you added to the Switch Case block.

These consist of subsystems with Action Port blocks inside them. When you place the Action Port block inside a subsystem, the subsystem becomes an atomic subsystem with an input port labeled `Action`.

- 4 Connect each case output port and the default output port of the Switch Case block to the Action port of an Action subsystem.

Each connected subsystem becomes a case body. This is indicated by the change in label for the Switch Case Action Subsystem block and the Action Port block inside of it to the name `case{}`.

During simulation of a `switch` control flow statement, the Action signals from the Switch Case block to each Switch Case Action Subsystem turn from solid to dashed.

- 5 In each Switch Case Action Subsystem, enter the Simulink logic appropriate to the case it handles.

Control Algorithm Execution Using Enumerated Signal

This example shows how to use a signal of an enumerated data type to control the execution of a block algorithm. For basic information about using enumerated data types in models, see “Use Enumerated Data in Simulink Models”.

When you use enumerated data in a Switch Case block, follow these best practices:

- Use the same enumerated type for the input `u1` and all of the case condition values.
- Use a different underlying integer for each of the enumerated values that you specify in the **Case conditions** box.

Copy the enumerated type definition `ex_SwitchCase_MyColors` into a script file in your current folder.

```
classdef ex_SwitchCase_MyColors < Simulink.IntEnumType
    enumeration
        Red(0)
        Yellow(1)
        Blue(2)
        Mauve(3)
    end
end
```

Alternatively, you can use the function `Simulink.defineIntEnumType` to define the type.

```
Simulink.defineIntEnumType('ex_SwitchCase_MyColors',...
    {'Red','Yellow','Blue','Mauve'},[0;1;2;3])
```

- 1 Open the example model `ex_enum_switch_case`.
 - 2 Open the Enumerated Constant block dialog box. The constant output value is `ex_SwitchCase_MyColors.Blue`.
 - 3 Open the Switch Case block dialog box. The **Case conditions** box is set to a cell array containing three of the four possible enumeration members. The block has four outputs corresponding to the three specified enumeration members and a default case.
 - 4 Open the Switch Case Action Subsystem blocks. The subsystems each contain a Constant block that uses a unique constant value.
-
- 1 In the Simulink Editor, set the simulation stop time to `Inf`.
 - 2 Simulate the model. The Display block shows the value 5, which corresponds to the case `ex_SwitchCase_MyColors.Blue`.
 - 3 In the Enumerated Constant block dialog box, set **Value** to `ex_SwitchCase_MyColors.Red` and click **Apply**. The Display block shows 19.
 - 4 Set **Value** to `ex_SwitchCase_MyColors.Mauve` and click **Apply**. The Display block shows 3, which corresponds to the default case.

Data Type Support

The input to the port labeled `u1` of a Switch Case block can be:

- A scalar value having a built-in data type that Simulink supports. The block does not support Boolean or fixed-point data types and truncates the numeric inputs to 32-bit signed integers.
- A scalar value of any enumerated data type, as described in “Control Algorithm Execution Using Enumerated Signal” on page 1-1608.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Case conditions

Specify the case conditions using MATLAB cell notation. For example, entering `{1, [7, 9, 4]}` specifies that output port `case[1]` is run when the input value is 1, and output port `case[7 9 4]` is run when the input value is 7, 9, or 4.

You can use colon notation to specify a range of integer case conditions. For example, entering `{[1:5]}` specifies that output port `case[1 2 3 4 5]` is run when the input value is 1, 2, 3, 4, or 5.

Depending on block size, cases with long lists of conditions are displayed in shortened form in the Switch Case block, using a terminating ellipsis (...).

You can use the `enumeration` function to specify a case condition that includes a case for every value in an enumerated type.

Show default case

If you select this check box, the default output port appears as the last case on the Switch Case block, allowing you to specify a default case. This case executes when the input value does not match any of the case values specified in the **Case conditions** field. With **Show default case** selected, a default output port always appears, even if the preceding cases exhaust all possibilities for the input value.

Enable zero-crossing detection

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Characteristics

Data Types	Double Single Base Integer Enumerated
Sample Time	Inherited from driving block

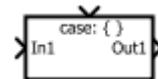
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	Yes, if enabled
Code Generation	Yes

Introduced before R2006a

Switch Case Action Subsystem

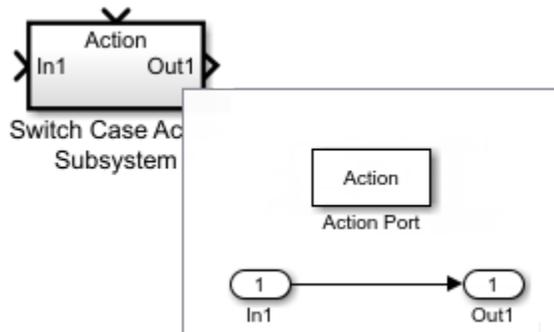
Subsystem whose execution is triggered by Switch Case block

Library: Ports & Subsystems



Description

The Switch Case Action Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem whose execution is triggered by an Switch Case block.



All blocks in a Switch Case Action Subsystem block must run at the same rate as the driving Switch Case block. You can achieve this requirement by setting each block's sample time parameter to be either inherited (-1) or the same value as the Switch Case block sample time.

Ports

Input

In — Signal input to a subsystem block

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

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

Action — Control signal input to a subsystem block

`scalar` | `vector` | `matrix`

Placing an Action Port block in a subsystem block adds an external input port to the block and changes the block to a Switch Case Action Subsystem block.

Dot-dash lines from a Switch Case block to an Switch Case Action Subsystem block represent *action* signals.

Data Types: `action`

Output

out — Signal output from a subsystem

`scalar` | `vector` | `matrix`

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

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

See Also

Blocks

Action Port | If | If Action Subsystem | Subsystem | Switch Case

Topics

“Use Control Flow Logic”

“Action Subsystem”

Introduced before R2006a

Synchronous Subsystem

Represent subsystem that has synchronous reset and enable behavior



Library

HDL Coder / HDL Subsystems

Description

A Synchronous Subsystem is a subsystem that uses the `Synchronous` mode of the State Control block. If an **S** symbol appears in the subsystem, then it is synchronous.

To create a Synchronous Subsystem, add the block to your Simulink model from the HDL Subsystems block library. You can also add a State Control block with **State control** set to `Synchronous` inside a subsystem. For more information about the State Control block, see State Control.

Data Type Support

See Inport for information on the data types accepted by a subsystem's input ports. See Outport for information on the data types output by a subsystem's output ports.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Show port labels

Cause Simulink software to display labels for the subsystem's ports on the subsystem's icon.

Default: `FromPortIcon`

`none`

Does not display port labels on the subsystem block.

`FromPortIcon`

If the corresponding port icon displays a signal name, display the signal name on the subsystem block. Otherwise, display the port block's name.

`FromPortBlockName`

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

`SignalName`

If a name exists, display the name of the signal connected to the port on the subsystem block; otherwise, the name of the corresponding port block.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Read/Write permissions

Control user access to the contents of the subsystem.

Default: `ReadWrite`

`ReadWrite`

Enables opening and modification of subsystem contents.

`ReadOnly`

Enables opening but not modification of the subsystem. If the subsystem resides in a block library, you can create and open links to the subsystem and can make and

modify local copies of the subsystem but cannot change the permissions or modify the contents of the original library instance.

`NoReadOrWrite`

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

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Name of error callback function

Enter name of a function to be called if an error occurs while Simulink software is executing the subsystem.

Default: ' '

Simulink software passes two arguments to the function: the handle of the subsystem and a character vector that specifies the error type. If no function is specified, Simulink software displays a generic error message if executing the subsystem causes an error.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Permit hierarchical resolution

Specify whether to resolve names of workspace variables referenced by this subsystem.

Default: All

All

Resolve all names of workspace variables used by this subsystem, including those used to specify block parameter values and Simulink data objects (for example, `Simulink.Signal` objects).

`ExplicitOnly`

Resolve only names of workspace variables used to specify block parameter values, data store memory (where no block exists), signals, and states marked as “must resolve”.

None

Do not resolve any workspace variable names.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Treat as atomic unit

Causes Simulink software to treat the subsystem as a unit when determining the execution order of block methods.

Default: Off

On

Cause Simulink software to treat the subsystem as a unit when determining the execution order of block methods. For example, when it needs to compute the output of the subsystem, Simulink software invokes the output methods of all the blocks in the subsystem before invoking the output methods of other blocks at the same level as the subsystem block.

Off

Cause Simulink software to treat all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem when determining block method execution order. This can cause execution of methods of blocks in the subsystem to be interleaved with execution of methods of blocks outside the subsystem.

This parameter enables:

- “Minimize algebraic loop occurrences” on page 1-0 .
- “Sample time” on page 1-0
- “Function packaging” on page 1-0 (requires a Simulink Coder license)

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Treat as grouped when propagating variant conditions

Causes Simulink software to treat the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks.

Default: On

On

Simulink treats the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks. For example, when Simulink computes the variant condition of the subsystem, it propagates that condition to all the blocks in the subsystem.

Off

Simulink treats all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem itself when determining their variant condition.

“Treat as grouped when propagating variant conditions” on page 1-0 enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Function packaging

Specify the code format to be generated for an atomic (nonvirtual) subsystem.

Default: Auto

Auto

Simulink Coder software chooses the optimal format for you based on the type and number of instances of the subsystem that exist in the model.

Inline

Simulink Coder software inlines the subsystem unconditionally.

Nonreusable function

Simulink Coder software explicitly generates a separate function in a separate file. Subsystems with this setting generate functions that might have arguments

depending on the “Function interface” on page 1-0 parameter setting. You can name the generated function and file using parameters “Function name” on page 1-0 and “File name (no extension)” on page 1-0 . These functions are not reentrant.

Reusable function

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

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

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes
HDL Code Generation	Yes

See Also

Enabled Synchronous Subsystem | Resettable Synchronous Subsystem | State Control

Introduced in R2016a

Tapped Delay

Delay scalar signal multiple sample periods and output all delayed versions



Library

Discrete

Description

The Tapped Delay block delays an input by the specified number of sample periods and outputs all the delayed versions. Use this block to discretize a signal in time or resample a signal at a different rate.

The block accepts one scalar input and generates an output vector that contains each delay. Specify the order of the delays in the output vector with the **Order output vector starting with** parameter:

- **Oldest** orders the output vector starting with the oldest delay version and ending with the newest delay version.
- **Newest** orders the output vector starting with the newest delay version and ending with the oldest delay version.

Specify the output vector for the first sampling period with the **Initial condition** parameter. Careful selection of this parameter can minimize unwanted output behavior.

Specify the time between samples with the **Sample time** parameter. Specify the number of delays with the **Number of delays** parameter. A value of -1 instructs the block to inherit the number of delays by back propagation. Each delay is equivalent to the z^{-1} discrete-time operator, which the Unit Delay block represents.

Data Type Support

The Tapped Delay block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Initial condition

Specify the initial output of the simulation. The **Initial condition** parameter is converted from a double to the input data type offline using round-to-nearest and saturation. Simulink software does not allow you to set the initial condition of this block to `inf` or `NaN`.

Sample time

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” in the online Simulink documentation for more information.

Number of delays

Specify the number of discrete-time operators.

Order output vector starting with

Specify whether to output the oldest delay version first, or the newest delay version first.

Include current input in output vector

Select to include the current input in the output vector.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Specified in the Sample time parameter
Direct Feedthrough	Yes, when Include current input in output vector check box is selected. No, otherwise.
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

See Also

Delay | Resettable Delay | Unit Delay | Variable Integer Delay

Introduced before R2006a

Terminate Function

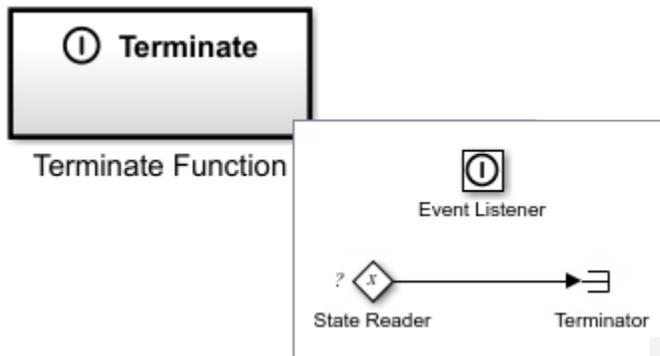
Execute contents on a model terminate event

Library: User-Defined Functions



Description

The Terminate Function block is a pre-configured subsystem block that executes on a model terminate event. By default, the Terminate Function block includes an Event Listener block with **Event** set to `Terminate`, a Terminator block, and a State Reader block.



Replace the Terminator block with blocks to save the state value from the State Reader block.

For a list of unsupported blocks and features, see “Initialize, Reset, and Terminate Function Limitations”.

The input and output ports of a component containing Initialize Function and Terminate Function blocks must connect to input and output port blocks.

The code generated from this block is part of the `model_terminate` function that is called once at the end of model execution.

See Also

[Event Listener](#) | [Initialize Function](#) | [Reset Function](#) | [State Reader](#) | [State Writer](#)

Topics

“Create Model to Initialize, Reset, and Terminate State”

“Create Test Harness to Generate Function Calls”

“Generate Code That Responds to Initialize, Reset, and Terminate Events” (Simulink Coder)

Terminator

Terminate unconnected output port



Library

Sinks

Description

Use the Terminator block to cap blocks whose output ports do not connect to other blocks. If you run a simulation with blocks having unconnected output ports, Simulink issues warning messages. Using Terminator blocks to cap those blocks helps prevent warning messages.

Data Type Support

The Terminator block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Examples

The following Simulink examples show how to use the Terminator block:

- `sldemo_bounce`
- `sldemo_fuelsys`
- `aero_six_dof`

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from driving block
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Timed-Based Linearization

Generate linear models in base workspace at specific times



Library

Model-Wide Utilities

Description

This block calls `linmod` or `dlinmod` to create a linear model for the system when the simulation clock reaches the time specified by the **Linearization time** parameter. No trimming is performed. The linear model is stored in the base workspace as a structure, along with information about the operating point at which the snapshot was taken. Multiple snapshots are appended to form an array of structures.

The block sets the following model parameters to the indicated values:

- `BufferReuse` = 'off'
- `RTWInlineParameters` = 'on'
- `BlockReductionOpt` = 'off'

The name of the structure used to save the snapshots is the name of the model appended by `_Timed_Based_Linearization`, for example, `vdp_Timed_Based_Linearization`. The structure has the following fields:

Field	Description
a	The A matrix of the linearization
b	The B matrix of the linearization

Field	Description
c	The C matrix of the linearization
d	The D matrix of the linearization
StateName	Names of the model's states
OutputName	Names of the model's output ports
InputName	Names of the model's input ports
OperPoint	A structure that specifies the operating point of the linearization. The structure specifies the operating point time (OperPoint.t). The states (OperPoint.x) and inputs (OperPoint.u) fields are not used.
Ts	The sample time of the linearization for a discrete linearization

Use the Trigger-Based Linearization block if you need to generate linear models conditionally.

You can use state and simulation time logging to extract the model states and inputs at operating points. For example, suppose that you want to get the states of the f14 example model at linearization times of 2 seconds and 5 seconds.

- 1 Open the model and drag an instance of this block from the Model-Wide Utilities library and drop the instance into the model.
- 2 Open the block's parameter dialog box and set the **Linearization time** to 2 and 5.
- 3 Open the model's **Model Configuration Parameters** dialog box.
- 4 Select the **Data Import/Export** pane.
- 5 Check **States** and **Time** on the **Save to Workspace** control panel
- 6 Select OK to confirm the selections and close the dialog box.
- 7 Simulate the model.

At the end of the simulation, the following variables appear in the MATLAB workspace: f14_Timed_Based_Linearization, tout, and xout.

- 8 Get the indices to the operating point times by entering the following at the MATLAB command line:

```
ind1 = find(f14_Timed_Based_Linearization(1).OperPoint.t==tout);
ind2 = find(f14_Timed_Based_Linearization(1).OperPoint.t==tout);
```

- 9 Get the state vectors at the operating points.

```
x1 = xout(ind1,:);  
x2 = xout(ind2,:);
```

Data Type Support

Not applicable.

Parameters

Linearization time

Time at which you want the block to generate a linear model. Enter a vector of times if you want the block to generate linear models at more than one time step.

Sample time (of linearized model)

Specify a sample time to create discrete-time linearizations of the model (see “Discrete-Time System Linearization” on page 2-47).

Characteristics

Data Types	Not applicable
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	No

See Also

Trigger-Based Linearization

Introduced in R2010a

To File

Write data to file



Library

Sinks

Description

The To File block inputs a signal and writes the signal data into a MAT-file. Use the To File block to log signal data.

The To File block icon shows the name of the output file.

The block writes to the output file incrementally, with minimal memory overhead during simulation. If the output file exists when the simulation starts, the block overwrites the file. The file automatically closes when simulation is complete or paused. If simulation terminates abnormally, the To File block saves the data it has logged up until the point of the abnormal termination.

Tip If MATLAB encounters memory issues when you log a large number of signals in a long simulation that has many time steps, consider logging to persistent storage. When you log to persistent storage, the Dataset format logging data is stored in a MAT-file. Compared to logging to persistent storage, connecting a To File block to signals:

- Is a per-signal approach that can clutter a model with several To File blocks attached to individual signals.
- Creates a separate MAT-file for each To File block, compared to the one MAT-file that logging to persistent storage uses.

For details, see “Log Data to Persistent Storage”.

Specifying the Format for Writing Data

Use the **Save format** parameter to specify the format for writing data:

- `Timeseries` (default)
- `Array`

Use the `Array` format only for vector, double, noncomplex signals. To save bus data, use the `Timeseries` format.

For the `Timeseries` format, the `To File` block:

- Writes data in a MATLAB `timeseries` object
- Supports writing multidimensional, real or complex output values
- Supports writing output values that have any built-in data type, including `Boolean`, enumerated (`enum`), and fixed-point data with a word length of up to 32 bits
- For bus input signals, creates a MATLAB structure that matches the bus hierarchy. Each leaf of the structure is a MATLAB `timeseries` object.

For the `Array` format, the `To File` block:

- Writes data into a matrix containing two or more rows. The matrix has the following form:

$$\begin{bmatrix} t_1 & t_2 & \dots & t_{final} \\ u1_1 & u1_2 & \dots & u1_{final} \\ \dots & & & \\ un_1 & un_2 & \dots & un_{final} \end{bmatrix}$$

Simulink writes one column to the matrix for each data sample. The first element of the column contains the time stamp. The remainder of the column contains data for the corresponding output values.

- Supports writing data that is one-dimensional, double, and noncomplex.

The following table shows how simulation mode support depends on the **Save format** value.

Simulation Mode	Timeseries	Array
Normal	Supported.	Supported.
Accelerator	Supported.	Supported.
Rapid Accelerator	Supported.	Supported.
Software-in-the-Loop (SIL)	Not supported.	Supported if MAT-file logging is enabled.
Processor-in-the-Loop (PIL)	Not supported.	Supported if MAT-file logging is available and enabled.
External	Not supported.	Supported if MAT-file logging is enabled.
RSim target	Supported.	Supported if MAT-file logging is enabled.

Controlling When Data Is Written to the File

The To File block **Decimation** and **Sample Time** parameters control when data is written to the file.

The To File block does not log data outside of the intervals specified by the **Model Configuration Parameters > Data Import/Export > Logging intervals** parameter. The block stores the logged data in the file associated with the block instead of storing the data in the variable that you specify for the **Single simulation output** parameter.

Saving Data for Use by a From File Block

The From File block can use data written by a To File block in any format (**Timeseries** or **Array**) without any modifications to the data or other special provisions.

Saving Data for Use by a From Workspace Block

The From Workspace block can read data that is in the **Array** format and is the transposition of the data written by the To File block. To provide the required format, use MATLAB commands to load and transpose the data from the MAT-file.

Simulation Stepper Interaction with To File Block

If you pause using the Simulation Stepper, the To File block captures the simulation data up to the point of the pause. When you step back, the To File data file no longer contains any simulation data past the new reduced time of the last output.

Limitations of To File blocks in a Referenced Model

When a To File block is in a referenced model, that model must be a single-instance model. Only one instance of such a model can exist in a model hierarchy. See “Specify Reusability of Referenced Models” for more information.

Compressing MAT-File Data

To avoid the overhead of compressing data in real time, the To File block writes an uncompressed Version 7.3 MAT-file. To compress the data within the MAT-file, load and save the file in MATLAB. The resaved file is smaller than the original MAT-file that the To File block created, because the **Save** command compresses the data in the MAT-file.

Saving Bus Data

The To File block supports virtual and nonvirtual bus input.

To save bus data, set the **Save format** parameter to `Timeseries`.

If the input signal is a bus, then the To File block creates a MATLAB structure that matches the bus hierarchy. Each leaf of the structure is a MATLAB `timeseries` object.

Pausing a Simulation

After pausing a simulation, do not alter any file that a To File block logs into. For example, do not save such a file with the MATLAB `save` command. Altering the file can cause an error when you resume the simulation. If you want to alter the file after pausing, copy the file and work with the copy of the file.

Generating Code

To generate code for a To File block, on the **Code Generation > Interface** pane, you must select the configuration parameter “MAT-file logging” (Simulink Coder).

Data Type Support

The To File block accepts real or complex signal data of any data type that Simulink supports, with the exception that the word length for fixed-point data must be 32 bits or less.

The To File block accepts bus data.

Parameters

File name

The path or file name of the MAT-file in which to store the output. On UNIX systems, the pathname can start with a tilde (~) character signifying your home folder. The default file name is `untitled.mat`. If you specify a file name without path information, Simulink stores the file in the MATLAB working folder. (To determine the working folder, type `pwd` at the MATLAB command line.) If the file already exists, Simulink overwrites it.

Variable name

The name of the matrix contained in the named file. The default name is `ans`.

Save format

The data format that the To File block uses for writing data:

- `Timeseries` (default)
- `Array`

Decimation

The decimation factor, n , where n specifies writing data at every n th time that the block executes. The default decimation is 1, which writes data at every time step.

Sample time

Specifies the sample period and offset at which to collect points. This parameter is useful when you are using a variable-step solver where the interval between time steps might not be constant. The default is -1, which inherits the sample time from the driving block. See “Specify Sample Time” for more information.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	Yes
Variable-Size Signals	No
Code Generation	Yes

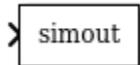
See Also

“Save Runtime Data from Simulation”, “Convert Logged Data to Dataset Format”, From File, From Workspace, To Workspace

Introduced before R2006a

To Workspace

Write data to workspace



Library

Sinks

Description

The To Workspace block inputs a signal and writes the signal data to a workspace. During the simulation, the block writes data to an internal buffer. When the simulation is completed or paused, that data is written to the workspace. Data is not available until the simulation is stopped or paused.

- For menu-based simulation, data is written in the MATLAB base workspace.
- A `sim` command in a MATLAB function sends data logged with the To Workspace block to the workspace of the calling function, not to the MATLAB (base) workspace. To send the logged data to the base workspace, use an `assignin` command in the function. For example:

```
function myfunc
    a = sim('mTest','SimulationMode','normal');
    b = a.get('simout')
    assignin('base','b',b);
end
```

The block icon shows the name of the variable to which the data is written. To specify the name of the workspace variable to which the To Workspace block writes the data, use the **Variable name** parameter.

To specify the data format of the variable, use the **Save format** parameter. You can specify to save the data in one of the following formats:

- A MATLAB `timeseries` object (or structure of MATLAB `timeseries` objects for bus data)
- An array
- Structure
- Structure with time

From one of these formats, you can convert the data to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see “Convert Logged Data to Dataset Format”. You can also use signal logging with a variable-size signal exception.

Saving Data for Use by a From Workspace Block

To use a `From Workspace` block to read into Simulink the sample-based data that was saved by a `To Workspace` block in a previous simulation, in the `To Workspace` block, specify time information using the `Timeseries` or `Structure with time` format.

Controlling the Amount of Data Saved

If you specify intervals with the **Model Configuration Parameters > Data Import/Export > Logging intervals** parameter, the block does not log data outside of the intervals. For example, the block logs no data if the intervals are empty (`[]`).

For variable-step solvers, to control the amount of data available to the `To Workspace` block, use the **Model Configuration Parameters > Data Import/Export > Additional parameters > Output options** parameter. For example, to have Simulink write data at identical time points over multiple simulations, select the `Produce specified output only` option.

Then use `To Workspace` block parameters to control when and how much of this data the block writes:

- Use the **Limit data points to last** parameter to specify how many sample points to save. If the simulation generates more data points than the specified maximum, the simulation saves only the most recently generated samples. To capture all the data, set this value to `inf`.

- Use the **Decimation** parameter to have the To Workspace block write data at every n th sample, where n is the decimation factor. The default decimation, 1, writes data at every time hit.
- Use the **Sample time** parameter to specify a sampling interval at which to collect points. This parameter is useful when you are using a variable-step solver where the interval between time hits might not be the same. The default value of -1 causes the block to inherit the sample time from the driving block when determining the points to write. See “Specify Sample Time” in the online documentation for more information.

For example, suppose you have a simulation where the start time is 0, the **Limit data points to last** is 100, the **Decimation** is 1, and the **Sample time** is 0.5. The To Workspace block collects a maximum of 100 points, at time values of 0, 0.5, 1.0, 1.5, ..., seconds. Specifying a **Decimation** value of 1 directs the block to write data at each step.

In a similar example, the **Limit data points to last** is 100 and the **Sample time** is 0.5, but the **Decimation** is 5. In this example, the block collects up to 100 points, at time values of 0, 2.5, 5.0, 7.5, ..., seconds. Specifying a **Decimation** value of 5 directs the block to write data at every fifth sample. The sample time ensures that data is written at these points.

In another example, all parameters are as defined in the first example except that the **Limit data points to last** is 3. In this case, only the last three sample points collected are written to the workspace. If the simulation stop time is 100, data corresponds to times 99.0, 99.5, and 100.0 seconds (three points).

MAT-File Logging

When you enable the **MAT-file logging** parameter in **Configuration Parameters**, To Workspace logs its data to a MAT-file. For information about this parameter, in the Simulink Coder documentation, see “MAT-file logging” (Simulink Coder).

Frame-Based Signals

By default, the To Workspace block treats input signals as sample-based.

To have the To Workspace block treat input signals as frame-based, set:

- 1 **Save format** to either **Array** or **Structure**

2 **Save 2-D signals as** to 2-D array (concatenate along first dimension)

Data Type Support

The To Workspace block can save to the MATLAB workspace real or complex inputs of any data type that Simulink supports, including fixed-point and enumerated data types, as well as bus objects.

For more information, see “Data Types Supported by Simulink”.

Parameters

Variable name

Specify the name of the variable for the saved the data.

Limit data points to last

Specify the maximum number of input samples to save. The default is `inf`.

Decimation

Specify the decimation factor. The default is 1.

Save format

Specify one of these formats for saving simulation output to the workspace:

- `Timeseries` (Default)

Save non-bus signals as a MATLAB `timeseries` object and bus signals as a structure of MATLAB `timeseries` objects.

- `Array`

Save the input as an N-dimensional array where N is one more than the number of dimensions of the input signal. For example, if the input signal is a vector, the

resulting workspace array is two-dimensional. If the input signal is a matrix, then the array is three-dimensional.

How Simulink stores samples in the array depends on whether the input signal is a scalar, vector, or matrix. If the input is a scalar or a vector, each input sample is output as a row of the array. For example, suppose that the name of the output array is `simout`. Then, `simout(1, :)` corresponds to the first sample, `simout(2, :)` corresponds to the second sample, and so on. If the input signal is a matrix, time corresponds to the third dimension. For example, suppose again that `simout` is the name of the resulting workspace array. Then, `simout(:, :, 1)` is the value of the input signal at the first sample point; `simout(:, :, 2)` is the value of the input signal at the second sample point; and so on.

If you select `Array`, the **Save 2-D signals** as parameter appears.

To treat input signals as frame-based, set **Save format** to either `Array` or `Structure` and set the **Save 2-D signals** parameter to `2-D array (concatenate along first dimension)`.

- `Structure`

This format consists of a structure with three fields:

- `time` — This field is empty for this format.
- `signals` — A structure with three fields: `values`, `dimensions`, and `label`. The `values` field contains the array of signal values. The `dimensions` field specifies the dimensions of the corresponding signals. The `label` field contains the label of the input line.
- `blockName` — Name of the To Workspace block

If you select `Structure`, the **Save 2-D signals** as parameter appears.

To treat input signals as frame-based, set **Save format** to either `Structure` or `Array` and set the **Save 2-D signals** parameter to `2-D array (concatenate along first dimension)`.

- `Structure With Time`

This format is the same as `Structure`, except that the `time` field contains a vector of simulation time hits.

To read To Workspace block output directly with a From Workspace block, use either the `Timeseries` or `Structure` with `Time` format. For details, see “Comparison of Signal Loading Techniques”.

`Structure` with `Time` format does not support frame-based signals. Use `Array` or `Structure` format instead.

The following table shows how simulation mode support depends on the **Save format** value.

Simulation Mode	Timeseries	Array, Structure, or Structure With Time
Normal	Supported.	Supported.
Accelerator	Supported.	Supported only in top model, not referenced models.
Rapid Accelerator	Not supported.	Supported only in top model, not referenced models.
Software-in-the-Loop (SIL)	Not supported.	If MAT-file logging is enabled, supported only in top model, not referenced models.
Processor-in-the-Loop (PIL)	Not supported.	If MAT-file logging is available and enabled, supported only in top model, not referenced models.
External	Not supported.	Supported only in top model, not referenced models.
Simulink Coder Targets	Not supported.	If MAT-file logging is enabled, supported only in top model, not referenced models.

Save 2-D signal as

If you set **Save format** to `Array` or `Structure`, the **Save 2-D signals as** parameter appears.

Specify one of these formats for saving 2-D signals to the workspace:

- 3-D array (concatenate along third dimension) (Default)

This setting is well-suited for sample-based signals. Data is concatenated along the third dimension. For example, 2-by-4 matrix input for 10 samples is stored as a 2x4x10 array.

- 2-D array (concatenate along first dimension)

This setting is well-suited for frame-based signals. The data is concatenated along the first dimension. For example, 2-by-4 matrix input for 10 samples is stored as a 20x4 array

- Inherit from input (this choice will be removed – see release notes)

This setting is for backward compatibility. To configure this block to treat input signals as frame-based in future releases, set this parameter to 2-D array (concatenate along first dimension). To configure this block to treat input signals as sample-based in future releases, set this parameter to 3-D array (concatenate along third dimension).

When the **Save format** is set to Array or Structure, the dimensions of the output depend on the input dimensions and the setting of the **Save 2-D signals as** parameter. The following table summarizes the output dimensions under various conditions. In the table, K represents the value of the **Limit data points to last** parameter.

Input Signal Dimensions	Save 2-D Signals as ...	Signal To Workspace Output Dimension
M -by- N matrix	2-D array (concatenate along first dimension)	<p>K-by-N matrix.</p> <p>If you set the Limit data points to last parameter to <code>inf</code>, K represents the total number of samples acquired in each column by the end of simulation. This is equivalent to multiplying the input frame size (M) by the total number of M-by-N inputs acquired by the block.</p>

Input Signal Dimensions	Save 2-D Signals as ...	Signal To Workspace Output Dimension
M -by- N matrix	3-D array (concatenate along third dimension)	M -by- N -by- K array. If you set the Limit data points to last parameter to <code>inf</code> , K represents the total number of M -by- N inputs acquired by the end of the simulation.
Length- N unoriented vector	Any setting	K -by- N matrix
N -dimensional array where $N > 2$	Any setting	Array with $N+1$ dimensions, where the size of the last dimension is equal to K . If you set the Limit data points to last parameter to <code>inf</code> , K represents the total number of M -by- N inputs acquired by the end of simulation

Log fixed-point data as a fi object

By default, the To Workspace block logs fixed-point data to the MATLAB workspace as a Fixed-Point Designer `fi` object. If you clear this parameter, fixed-point data is logged to the workspace as `double`.

Sample time

Specify the sample period and offset at which to collect data. This parameter is useful when you are using a variable-step solver where the interval between time hits might not be constant. The default is `-1`, which inherits the sample time from the driving block. See “Specify Sample Time” for more information.

Examples

The `sldemo_varsize_basic` example shows how to use the To Workspace block.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	No

See Also

“Export Simulation Data”, “Convert Logged Data to Dataset Format”, From File, From Workspace, To File

Introduced before R2006a

Toggle Switch

Toggle parameter between two values
Library: Simulink / Dashboard

On



Off

Description

The Toggle Switch block toggles the value of the connected block parameter between two values during simulation. For example, you can connect the Toggle Switch block to a Switch block in your model and change its state during simulation. Use the Toggle Switch block with other Dashboard blocks to create an interactive dashboard for your model.

Connection

Dashboard block controls connect to variables and block parameters. To connect a Dashboard block to a variable or parameter, use the **Connection** table in the block's parameters.

- 1 Double-click the block.
- 2 Populate the **Connection** table by selecting blocks of interest in your model.
- 3 Mark the option button next to the parameter or variable you want to adjust during simulation.
- 4 Click **Apply**.

Limitations

- You cannot save the block connections or properties in model files that use the MDL format.

To save connections and properties, save the model file in the SLX format.

- Dashboard blocks can only connect to real scalar signals.
- Dashboard blocks cannot connect to blocks that are commented out.
- Dashboard blocks cannot connect to signals inside reference models.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined using the variable `engine(1)` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

Parameters

Connection — Select a block parameter to connect

signal connection options

Select the block parameter to control using the **Connection** table. To open the dialog box, double-click the block. Then, populate the **Connection** table by selecting a block in your model. Select the variable parameter you want to control, and click **Apply**.

Note To see workspace variables in the connection table, update the model diagram with **Ctrl+D**.

States

Label (Top) — Label for top switch position

'On' (default) | character vector

Labels the top switch position. You can use the **Label** to display the value the connected parameter takes when the switch is positioned at the top, or you can enter a text label.

Example: `Gain = 2`

Value (Top) — Value for top switch position

1 (default) | scalar

The value assigned to the connected parameter when the switch is positioned at the top.

Label (Bottom) — Label for bottom switch position

'Off' (default) | character vector

Labels the bottom switch position. You can use the **Label** to display the value the connected parameter takes when the switch is positioned at the bottom, or you can enter a text label.

Example: `Gain = 1`

Value (Bottom) — Value for bottom switch position

0 (default) | scalar

The value assigned to the connected parameter when the switch is positioned at the bottom.

Label — Block label position

'Top' (default) | 'Bottom' | 'Hide'

Position of the block label. When the block is connected to a signal, the label is the name of the connected signal. When the block is not connected, the label is the instructional text.

Model Examples

See Also

Rocker Switch | Rotary Switch | Slider Switch

Topics

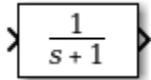
“Tune and Visualize Your Model with Dashboard Blocks”

“Decide How to Visualize Simulation Data”

Introduced in R2015a

Transfer Fcn

Model linear system by transfer function



Library

Continuous

Description

The Transfer Fcn block models a linear system by a transfer function of the Laplace-domain variable s . The block can model single-input single-output (SISO) and single-input multiple output (SIMO) systems.

Conditions for Using This Block

The Transfer Fcn block assumes the following conditions:

- The transfer function has the form

$$H(s) = \frac{y(s)}{u(s)} = \frac{num(s)}{den(s)} = \frac{num(1)s^{nn-1} + num(2)s^{nn-2} + \dots + num(nn)}{den(1)s^{nd-1} + den(2)s^{nd-2} + \dots + den(nd)},$$

where u and y are the system input and outputs, respectively, nn and nd are the number of numerator and denominator coefficients, respectively. $num(s)$ and $den(s)$ contain the coefficients of the numerator and denominator in descending powers of s .

- The order of the denominator must be greater than or equal to the order of the numerator.
- For a multiple-output system, all transfer functions have the same denominator and all numerators have the same order.

Modeling a Single-Output System

For a single-output system, the input and output of the block are scalar time-domain signals. To model this system:

- 1 Enter a vector for the numerator coefficients of the transfer function in the **Numerator coefficients** field.
- 2 Enter a vector for the denominator coefficients of the transfer function in the **Denominator coefficients** field.

Modeling a Multiple-Output System

For a multiple-output system, the block input is a scalar and the output is a vector, where each element is an output of the system. To model this system:

- 1 Enter a matrix in the **Numerator coefficients** field.

Each *row* of this matrix contains the numerator coefficients of a transfer function that determines one of the block outputs.

- 2 Enter a vector of the denominator coefficients common to all transfer functions of the system in the **Denominator coefficients** field.

Specifying Initial Conditions

A transfer function describes the relationship between input and output in Laplace (frequency) domain. Specifically, it is defined as the Laplace transform of the response (output) of a system with zero initial conditions to an impulse input.

Operations like multiplication and division of transfer functions rely on zero initial state. For example, you can decompose a single complicated transfer function into a series of simpler transfer functions. Apply them sequentially to get a response equivalent to that of the original transfer function. This will not be correct if one of the transfer functions assumes a non-zero initial state. Furthermore, a transfer function has infinitely many time domain realizations, most of whose states do not have any physical meaning.

For these reasons, Simulink presets the initial conditions of the Transfer Fcn block to zero. To specify initial conditions for a given transfer function, convert the transfer function to its controllable, canonical state-space realization using `tf2ss`. Then, use the State-Space block. The `tf2ss` utility provides the A, B, C, and D matrices for the system.

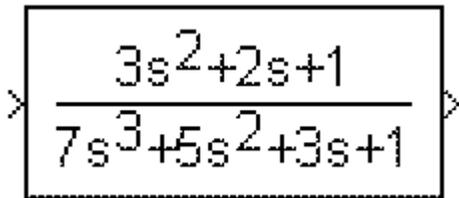
For more information, type `help tf2ss` or see the Control System Toolbox™ documentation.

Transfer Function Display on the Block

The Transfer Fcn block displays the transfer function depending on how you specify the numerator and denominator parameters.

- If you specify each parameter as an expression or a vector, the block shows the transfer function with the specified coefficients and powers of s . If you specify a variable in parentheses, the block evaluates the variable.

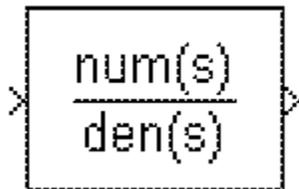
For example, if you specify **Numerator coefficients** as `[3, 2, 1]` and **Denominator coefficients** as `(den)`, where `den` is `[7, 5, 3, 1]`, the block looks like this:



$$\frac{3s^2 + 2s + 1}{7s^3 + 5s^2 + 3s + 1}$$

- If you specify each parameter as a variable, the block shows the variable name followed by (s) .

For example, if you specify **Numerator coefficients** as `num` and **Denominator coefficients** as `den`, the block looks like this:



$$\frac{\text{num}(s)}{\text{den}(s)}$$

Data Type Support

The Transfer Fcn block accepts and outputs signals of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Numerator coefficients

Define the row vector of numerator coefficients.

Default: [1]

- For a single-output system, enter a vector for the numerator coefficients of the transfer function.
- For a multiple-output system, enter a matrix. Each row of this matrix contains the numerator coefficients of a transfer function that determines one of the block outputs.

Parameter: Numerator

Type: vector or matrix

Value: ' [1] '

Default: ' [1] '

See the Transfer Fcn block reference page for more information.

Denominator coefficients

Define the row vector of denominator coefficients.

Default: [1 1]

- For a single-output system, enter a vector for the denominator coefficients of the transfer function.
- For a multiple-output system, enter a vector containing the denominator coefficients common to all transfer functions of the system.

Parameter: Denominator

Type: vector

Value: ' [1 1] '

Default: '[1 1]'

Absolute tolerance

Specify the absolute tolerance for computing block states.

Default: `auto`

- You can enter `auto`, `-1`, a positive real scalar or vector.
- If you enter `auto` or `-1`, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute block states.
- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

Parameter: `AbsoluteTolerance`

Type: character vector, scalar, or vector

Value: `'auto'` | `'-1'` | any positive real scalar or vector

Default: `'auto'`

State Name (e.g., 'position')

Assign a unique name to each state.

Default: ' '

If this field is blank, no name assignment occurs.

- To assign a name to a single state, enter the name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- The state names apply only to the selected block.

- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Parameter: ContinuousStateAttributes

Type: character vector

Value: ' ' | user-defined

Default: ' '

Examples

The following Simulink examples show how to use the Transfer Fcn block:

- `slexAircraftExample`
- `sldemo_absbrake`
- `penddemo`

Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	Only if the lengths of the Numerator coefficients and Denominator coefficients parameters are equal
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

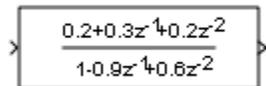
See Also

Discrete Transfer Fcn

Introduced before R2006a

Transfer Fcn Direct Form II

Implement Direct Form II realization of transfer function


$$\frac{0.2 + 0.3z^{-1} + 0.2z^{-2}}{1 - 0.9z^{-1} + 0.6z^{-2}}$$

Library

Additional Math & Discrete / Additional Discrete

Description

The Transfer Fcn Direct Form II block implements a Direct Form II realization of the transfer function that the **Numerator coefficients** and **Denominator coefficients** **excluding lead** parameters specify. The block supports only single input-single output (SISO) transfer functions.

The block automatically selects the data types and scalings of the output, the coefficients, and any temporary variables.

Data Type Support

The Transfer Fcn Direct Form II block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Numerator coefficients

Specify the numerator coefficients.

Denominator coefficients excluding lead

Specify the denominator coefficients, excluding the leading coefficient, which must be 1.0.

Initial condition

Set the initial condition.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

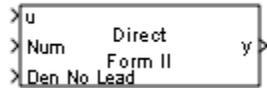
See Also

Transfer Fcn Direct Form II Time Varying

Introduced before R2006a

Transfer Fcn Direct Form II Time Varying

Implement time varying Direct Form II realization of transfer function



Library

Additional Math & Discrete / Additional Discrete

Description

The Transfer Fcn Direct Form II Time Varying block implements a Direct Form II realization of the specified transfer function. The block supports only single input-single output (SISO) transfer functions.

The input signal labeled `Den No Lead` contains the denominator coefficients of the transfer function. The full denominator has a leading coefficient of one, but it is excluded from the input signal. For example, a denominator of $[1 -1.7 \ 0.72]$ is represented by a signal with the value $[-1.7 \ 0.72]$. The input signal labeled `Num` contains the numerator coefficients. The data types of the numerator and denominator coefficients can be different, but the length of the numerator vector and the full denominator vector must be the same. Pad the numerator vector with zeros, if needed.

The block automatically selects the data types and scalings of the output, the coefficients, and any temporary variables.

Data Type Support

The Transfer Fcn Direct Form II Time Varying block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Initial condition

Set the initial condition.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

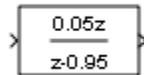
See Also

Transfer Fcn Direct Form II

Introduced before R2006a

Transfer Fcn First Order

Implement discrete-time first order transfer function



Library

Discrete

Description

The Transfer Fcn First Order block implements a discrete-time first order transfer function of the input. The transfer function has a unity DC gain.

Data Type Support

The Transfer Fcn First Order block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Pole (in Z plane)

Set the pole.

Initial condition for previous output

Set the initial condition for the previous output.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer). in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

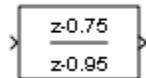
Characteristics

Data Types	Double Single Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Transfer Fcn Lead or Lag

Implement discrete-time lead or lag compensator



Library

Discrete

Description

The Transfer Fcn Lead or Lag block implements a discrete-time lead or lag compensator of the input. The instantaneous gain of the compensator is one, and the DC gain is equal to $(1-z)/(1-p)$, where z is the zero and p is the pole of the compensator.

The block implements a lead compensator when $0 < z < p < 1$, and implements a lag compensator when $0 < p < z < 1$.

Data Type Support

The Transfer Fcn Lead or Lag block accepts signals of any numeric data type that Simulink supports, including fixed-point data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Pole of compensator (in Z plane)

Set the pole.

Zero of compensator (in Z plane)

Set the zero.

Initial condition for previous output

Set the initial condition for the previous output.

Initial condition for previous input

Set the initial condition for the previous input.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

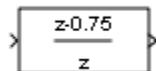
Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Transfer Fcn Real Zero

Implement discrete-time transfer function that has real zero and no pole



Library

Discrete

Description

The Transfer Fcn Real Zero block implements a discrete-time transfer function that has a real zero and effectively no pole.

Data Type Support

The Transfer Fcn Real Zero block accepts signals of any numeric data type that Simulink supports, including fixed-point data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Zero (in Z plane)

Set the zero.

Initial condition for previous input

Set the initial condition for the previous input.

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- `Elements as channels (sample based)` — Treat each element of the input as a separate channel (sample-based processing).
- `Columns as channels (frame based)` — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

- `Inherited` — Inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input `u`. All other input signals must be sample based.

Input Signal <code>u</code>	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate to max or min when overflows occur

Select to have overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Transport Delay

Delay input by given amount of time



Library

Continuous

Description

The Transport Delay block delays the input by a specified amount of time. You can use this block to simulate a time delay. The input to this block should be a continuous signal.

At the start of simulation, the block outputs the **Initial output** parameter until the simulation time exceeds the **Time delay** parameter. Then, the block begins generating the delayed input. During simulation, the block stores input points and simulation times in a buffer. You specify this size with the **Initial buffer size** parameter.

When you want output at a time that does not correspond to times of the stored input values, the block interpolates linearly between points. When the delay is smaller than the step size, the block extrapolates from the last output point, which can produce inaccurate results. Because the block does not have direct feedthrough, it cannot use the current input to calculate an output value. For example, consider a fixed-step simulation with a step size of 1 and the current time at $t = 5$. If the delay is 0.5, the block must generate a point at $t = 4.5$. Because the most recent stored time value is at $t = 4$, the block performs forward extrapolation.

The Transport Delay block does not interpolate discrete signals. Instead, the block returns the discrete value at the required time.

This block differs from the Unit Delay block, which delays and holds the output on sample hits only.

Tip Avoid using `linmod` to linearize a model that contains a Transport Delay block. For more information, see “Linearizing Models” in the Simulink documentation.

Data Type Support

The Transport Delay block accepts and outputs real signals of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Time delay

Specify the amount of simulation time to delay the input signal before propagation to the output.

Default: 1

This value must be nonnegative.

Parameter: `DelayTime`

Type: scalar or vector

Value: '1'

Default: '1'

Initial output

Specify the output that the block generates until the simulation time first exceeds the time delay input.

Default: Run-to-run tunable parameter

A Run-to-run tunable parameter cannot be changed during a simulation’s run time. However, changing it before a simulation begins will not cause Accelerator or Rapid Accelerator to regenerate code.

Also, the initial output of this block cannot be `inf` or `NaN`.

Parameter: `InitialOutput`

Type: scalar or vector

Value: '0'

Default: '0'

Initial buffer size

Define the initial memory allocation for the number of input points to store.

Default: 1024

- If the number of input points exceeds the initial buffer size, the block allocates additional memory.
- After simulation ends, a message shows the total buffer size needed.
- Because allocating memory slows down simulation, choose this value carefully if simulation speed is an issue.
- For long time delays, this block can use a large amount of memory, particularly for dimensionalized input.

Parameter: `BufferSize`

Type: scalar

Value: '1024'

Default: '1024'

Use fixed buffer size

Specify use of a fixed-size buffer to save input data from previous time steps.

Default: Off

On

The block uses a fixed-size buffer.

Off

The block does not use a fixed-size buffer.

The **Initial buffer size** parameter specifies the size of the buffer. If the buffer is full, new data replaces data already in the buffer. Simulink software uses linear extrapolation to estimate output values that are not in the buffer.

Note If you have a Simulink Coder license, ERT or GRT code generation uses a fixed-size buffer even if you do not select this check box.

- If the input data is linear, selecting this check box can save memory.
- If the input data is nonlinear, do not select this check box. Doing so can yield inaccurate results.

Parameter: FixedBuffer

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Direct feedthrough of input during linearization

Cause the block to output its input during linearization and trim, which sets the block mode to direct feedthrough.

Default: Off

On

Enables direct feedthrough of input.

Off

Disables direct feedthrough of input.

- Selecting this check box can cause a change in the ordering of states in the model when you use the functions `linmod`, `dlinmod`, or `trim`. To extract this new state ordering:

- 1 Compile the model using the following command, where `model` is the name of the Simulink model.

```
[sizes, x0, x_str] = model([], [], [], 'lincompile');
```

- 2 Terminate the compilation with the following command.

```
model([], [], [], 'term');
```

- The output argument `x_str`, which is a cell array of the states in the Simulink model, contains the new state ordering. When you pass a vector of states as input to the `linmod`, `dlinmod`, or `trim` functions, the state vector must use this new state ordering.

Parameter: `TransDelayFeedthrough`

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Pade order (for linearization)

Set the order of the Pade approximation for linearization routines.

Default: 0

- The default value is 0, which results in a unity gain with no dynamic states.
- Setting the order to a positive integer `n` adds `n` states to your model, but results in a more accurate linear model of the transport delay.

Parameter: `PadeOrder`

Type: character vector

Value: '0'

Default: '0'

Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	No

Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Variable Time Delay

Introduced before R2006a

Trigger

Add trigger port to subsystem or model

Library: Ports & Subsystems



Description

The Trigger block allows an external signal to control the execution of a subsystem or a model. To enable this functionality, add this block to a Subsystem block or at the root level of a model that is referenced in a Model block.

Then, configure the Trigger block to execute a subsystem or model:

- Once at each time step, when the value of the trigger signal changes in a way that you specify.
- Multiple times during a time step, when the trigger signal is a function-call from a Stateflow chart, Function-Call Generator block or S-function block.

Ports

Output

Trigger signal — External trigger signal for a subsystem or model

scalar

Trigger signal attached externally to the outside of an Subsystem block or a Model block that is passed to the inside of the block. To enable this port, select **Show output port**.

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

Parameters

Main

Trigger type — Select the type of event

`rising` (default) | `falling` | `either` | `function-call`

Select the type of event that triggers execution of the subsystem or model.

`rising`

Trigger execution of subsystem or model when the trigger signal rises from a negative or zero value to a positive value. If the initial value is negative, a rising signal to zero triggers execution.

`falling`

Trigger execution of subsystem or model when the trigger signal falls from a positive or a zero value to a negative value. If the initial value is positive, a falling signal to zero triggers execution.

`either`

Trigger execution of subsystem or model when the trigger signal is either rising or falling.

`function-call`

Trigger execution of subsystem or model when the trigger signal is a function-call event from a Stateflow chart, Function-Call Generator block, or an S-function block.

Programmatic Use

Block parameter: `TriggerType`

Type: character vector

Value: `'rising'` | `'falling'` | `'either'` | `'function-call'`

Default: `'rising'`

Treat as Simulink function — Create Simulink Function block

`off` (default) | `on`

Create a Simulink Function block by configuring a Subsystem block that is callable with arguments from a function caller.

`off`

Remove configuration.

on

Configure a Subsystem block as a Simulink Function block. The Trigger block must reside within the subsystem.

You can edit the function prototype that displays on the block face to specify input and output arguments for the block.

Dependency

To display and enable this parameter, select `function-call` from the **Trigger type** list.

Programmatic Use

Block parameter: `IsSimulinkFunction`

Type: character vector

Value: `'off' | 'on'`

Default: `'off'`

Function name — Specify function name for Simulink Function block

`f` (default) | function name

Specify the function name for a Simulink Function block. Alternatively, you can specify the name by editing the function prototype on the face of the block.

`f`

Default name for a Simulink Function block.

function name

Function name that displays on the face of a Simulink Function block.

Dependency

To display and enable this parameter, select `function-call` from the **Trigger type** list and select the **Treat as a Simulink Function** check box.

Programmatic Use

Block parameter: `FunctionName`

Type: character vector

Value: `'f' | '<function name>'`

Default: `'f'`

Enable variant condition — Controls activating the variant control (condition)

on (default) | off

Control activating the variant control (condition) defined with the **Variant Control** parameter.

off

Deactivate variant control of subsystem.

on

Activate variant control of subsystem. Selecting this parameter:

- Enables the **Variant control** parameter.
- Displays a variant badge on the face of the block indicating variant conditions are enabled.

Dependency

To display and enable this parameter, select `function-call` from the **Trigger type** list and select the **Treat as a Simulink Function** check box..

Programmatic Use

Block parameter: Variant

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Variant control — Specify variant control (condition) expression

(inherit) (default) | <logical expression>

Specify variant control (condition) expression that executes a variant Simulink Function block when the expression evaluates to true.

(inherit)

Default value for variant control. Inherits the variant condition from the corresponding Function Caller blocks in the model. When **Variant Control** is set as (inherit) the value for **Generate preprocessor conditionals** is inherited automatically from the Function Caller block in the model.

logical expression

A logical (Boolean) expression or a `Simulink.Variant` object representing a logical expression.

The function is activated when the expression evaluates to `true`.

If you want to generate code for your model, define the variables in the expression as `Simulink.Parameter` objects.

Dependency

To display and enable this parameter, select `function-call` from the **Trigger type** list, select the **Treat as a Simulink Function** check box and then select the **Enable variant condition** check box.

Programmatic Use

Block parameter: `VariantControl`

Type: character vector

Value: `'(inherit)'` | `<logical expression>` | `Simulink.Variant` object

Default: `'(inherit)'`

Generate preprocessor conditionals — Control enclosing variant choices

`off` (default) | `on`

Control enclosing variant choices within C preprocessor conditional statements.

`off`

Do not enclose variant choices within C preprocessor conditional statements.

`on`

When generating code for an ERT target, enclose variant choices within C preprocessor conditional statements (`#if`).

Dependency

To display and enable this parameter, select the **Enable variant condition** check box.

Programmatic Use

Block parameter: `GeneratePreprocessorConditionals`

Type: character vector

Value: `'off'` | `'on'`

Default: 'off'

Function visibility — Select scope visibility of function

scoped (default) | global

Select scope of Simulink Function block within subsystem or model.

scoped

Limit accessibility of function to:

- Hierarchic level containing the Simulink Function block and levels below.
- One hierarchical level above with qualification.

global

Function accessible from any part of the model hierarchy.

Dependency

To display and enable this parameter, select `function-call` from the **Trigger type** list and then select the **Treat as a Simulink Function** check box..

Programmatic Use

Block parameter: FunctionVisibility

Type: character vector

Value: 'scoped' | 'global'

Default: 'scoped'

States when enabling — Select how to set block state values

held (default) | reset | inherit

Select how to set block state values when the subsystem or model is disabled.

held

Leave the block states at their current values.

reset

Reset the block state values.

inherit

Use the `held` or `reset` setting from the parent subsystem initiating the function-call. If the parent of the initiator is the model root, the inherited setting is `held`. If

the trigger has multiple initiators, set the parents of all initiators to either held or reset.

Dependencies

To enable this parameter, select `function-call` from the **Trigger Type** list.

This parameter setting applies only if the model explicitly enables and disables the function-call subsystem. For example:

- The function-call subsystem resides in an enabled subsystem. In this case, the model enables and disables the function-call subsystem along with the parent subsystem.
- The function-call initiator that controls the function-call subsystem resides in an enabled subsystem. In this case, the model enables and disables the function-call subsystem along with the enabled subsystem containing the function-call initiator.
- The function-call initiator is a Stateflow event bound to a particular state. See “Control Function-Call Subsystems Using Bind Actions” (Stateflow) in the Stateflow documentation.
- The function-call initiator is an S-function that explicitly enables and disables the function-call subsystem. See `ssEnableSystemWithTid` for an example.

Programmatic Use

Block parameter: `StatesWhenEnabling`

Type: character vector

Value: `'held' | 'reset' | 'inherit'`

Default: `'held'`

Propagate sizes of variable-size signals — Select when to propagate variable-size signals

`During execution (default) | Only when enabling`

Select when to propagate variable-size signals.

During execution

Propagate variable-size signals at each time step.

Only when enabling

Propagate variable-size signals when executing a Subsystem block or Model block containing an Enable port, Trigger port with **Trigger type** set to `function-call`, or Action Port block. When you select this option, sample time must be periodic.

Dependencies

To display and enable this parameter for a Trigger port block, select `Function-call` from the **Trigger type** list.

Programmatic Use

Block parameter: `PropagateVarSize`

Type: character vector

Value: `'During execution' | 'Only when enabling'`

Default: `'During execution'`

Show output port — Control display of output port

`off` (default) | `on`

Control display of an output port for a signal that identifies the trigger signal.

`off`

Remove the output port.

`on`

Display the output port and determine which signal caused the trigger. The width of the output port signal is the width of the triggering signal. The signal value is:

- 1 for a signal that causes a rising trigger
- -1 for a signal that causes a falling trigger
- 2 for a function-call trigger
- 0 in all other cases

Programmatic Use

Block parameter: `ShowOutputPort`

Type: character vector

Value: `'off' | 'on'`

Default: `'off'`

Output data type — Select output port data type

`auto` (default) | `double` | `int8`

Select output port data type for the signal that identifies the trigger signal.

auto

Data type is the same as the port connected to the output.

double

Double value.

int8

Integer value

Dependency

To enable this parameter, select the **Show output port** check box.

The Trigger block ignores the **Data type override** setting for the Fixed-Point Tool.

Programmatic Use

Block parameter: OutputDataType

Type: character vector

Value: 'auto' | 'double' | 'int8'

Default: 'auto'

Sample time type — Select calling rate

triggered (default) | periodic

Select the calling rate for a subsystem or model.

triggered

Apply to applications that do not have a periodic calling frequency.

periodic

Apply if the caller of the parent function-call subsystem calls the subsystem once per time step when the subsystem is active (enabled). A Stateflow chart is an example of a caller.

Dependency

To enable this parameter, select Function-call from the **Trigger type** list.

Programmatic Use

Block parameter: SampleTimeType

Type: character vector

Value: 'triggered' | 'periodic'

Default: 'triggered'

Sample time — Specify time interval

-1 (default) | Ts | [Ts, To]

Specify the time interval between function calls to a subsystem or model containing this Trigger port block. If the actual calling rate for the subsystem or model differs from the time interval this parameter specifies, Simulink displays an error.

-1

Inherit time interval from the trigger signal.

Ts

Scalar where Ts is the time interval.

[Ts, To]

Vector where Ts is the time interval and To is the initial time offset.

Dependencies

To enable this parameter, select `function-call` from the **Trigger type** list and `periodic` from the **Sample time type** list.

Programmatic Use

Block parameter: SampleTime

Type: character vector

Value: '-1' | 'Ts' | '[Ts, To]'

Default: '-1'

Enable zero-crossing detection — Control zero-crossing detection

on (default) | off

Control .



on

Detect zero crossings.



off

Do not detect zero crossings.

Dependencies

To enable this parameter, select rising, falling, or either from the **Trigger type** list.

Programmatic Use

Block parameter: ZeroCross

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Initial trigger signal state — Select the initial state of the trigger signal

compatibility (no trigger on first evaluation) (default) | zero | positive
| negative

Select the initial state of the trigger signal.

compatibility (no trigger on first evaluation)

No trigger at the first evaluation of trigger signal. If you choose this option and the Trigger block is in a subsystem where the states are reset, the block does not reset.

zero

Zero. Helps to evaluate a rising or falling trigger signal at the first time step.

positive

Positive value. Helps to evaluate a falling trigger signal at the first time step.

negative

Negative value. Helps to evaluate a rising trigger signal at the first time step.

Dependency

To display and activate this parameter, select rising, falling, or either from the **Trigger type** list.

Programmatic Use

Block parameter: InitialTriggerSignalState

Type: character vector

Value: 'compatibility (no trigger on first evaluation)' | 'zero' |
'positive' | 'negative'

Default: 'compatibility (no trigger on first evaluation)'

Signal Attribute

Port dimensions — Specify dimensions for the trigger signal

1 (default) | [n] | [m n]

Specify dimensions for the trigger signal attached externally to the Model block and passed to the inside of the block.

1

Scalar signal.

[n]

Vector signal of width n.

[m n]

Matrix signal having m rows and n columns.

Dependency

To display and enable this parameter for a Trigger port block at the root-level of a model, select *rising*, *falling*, or either from the **Trigger type** list.

Programmatic Use

Block parameter: PortDimensions

Type: character vector

Value: '1' | '[n]' | '[m n]'

Default: '1'

Trigger signal sample time — Specify time interval

-1 (default) | Ts | [Ts, To]

Specify time interval between block method executions for the block driving the trigger signal.

-1

Inherit time interval.

Ts

Scalar where Ts is the time interval.

[Ts, To]

Vector where Ts is the time interval and To is the initial time offset.

Dependency

To display and enable this parameter for a Trigger port block at the root-level of a model, select `rising`, `falling`, or `either` from the **Trigger type** list.

Programmatic Use

Block parameter: `TriggerSignalSampleTime`

Type: character vector

Value: `'-1'` | `'Ts'` | `'[Ts, To]'`

Default: `'-1'`

Minimum — Specify minimum output value for the trigger signal

`[]` (default) | real scalar

Specify minimum value for the trigger signal attached externally to a Model block and passed to the inside of the block.

Simulink uses this value to perform:

- Simulation range checking. See “Signal Ranges”.
- Automatic scaling of fixed-point data types.
- Optimization of generated code. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. See “Optimize using the specified minimum and maximum values”.

`[]`

Unspecified minimum value.

real scalar

Real scalar value.

Dependency

To display and enable this parameter for a Trigger port block at the root-level of a model, select `rising`, `falling`, or `either` from the **Trigger type** list.

Programmatic Use

Block parameter: `OutMin`

Type: character vector

Value: `'[]'` | `'<real scalar>'`

Default: `'[]'`

Maximum — Specify maximum output value for the trigger signal

`[]` (default) | real scalar

Specify maximum value for the trigger signal attached externally to a Model block and passed to the inside of the block.

Simulink uses this value to perform:

- Simulation range checking. See “Signal Ranges”.
- Automatic scaling of fixed-point data types.
- Optimization of generated code. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. See “Optimize using the specified minimum and maximum values”.

`[]`

Unspecified maximum value.

real scalar

Real scalar value.

Dependency

To display and enable this parameter for a Trigger port block at the root-level of a model, select `rising`, `falling`, or `either` from the **Trigger type** list.

Programmatic Use

Block parameter: `OutMax`

Type: character vector

Value: `'[]'` | `'<real scalar>'`

Default: `'[]'`

Data type — Select output data type for the trigger signal

`double` (default) | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `boolean` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^,0)` | `<data type expression>`

Select data type for the trigger signal attached externally to a Model block and passed to the inside of the block.

`double`

Double-precision floating point.

single

Single-precision floating point.

int8

Signed 8-bit integer.

uint8

Unsigned 8-bit integer.

int16

Signed 16-bit integer.

uint16

Unsigned 16-bit integer.

int32

Signed 32-bit integer.

uint32

Unsigned 32-bit integer.

boolean

Boolean with a value of `true` or `false`.

`fixdt(1,16)`

Signed 16-bit fixed point number with binary point undefined.

`fixdt(1,16,0)`

Signed 16-bit fixed point number with binary point set to zero.

`fixdt(1,16,2^0,0)`

Signed 16-bit fixed point number with slope set to 2^0 and bias set to 0.

<data type expression>

Data type object, for example `Simulink.NumericType`. Do not specify a bus object as the expression.

Dependency

To display and enable this parameter for a Trigger port block at the root-level of a model, select `rising`, `falling`, or either from the **Trigger type** list.

Programmatic Use

Block parameter: `OutDataTypeStr`

Type: character vector

Value: 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'boolean' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'

Default: 'double'

Mode — Select data type category

Build in (default) | Fixed point | Expression

Select data type category and display drop-down lists to help you define the data type.

Build in

Display drop-down lists for data type and **Data type override**.

Fixed point

Display drop-down lists for **Signedness**, **Scaling**, and **Data type override**.

Expression

Display text box for entering an expression.

Dependency

To enable this parameter, select the Show data type assistant button .

Programmatic Use

No equivalent command-line parameter.

Interpolate data — Control how missing workspace data is estimated

on (default) | off

Control how missing workspace data is estimated when loading data from the MATLAB workspace.

on

Linearly interpolate output at time steps for which no corresponding workspace data exists.

off

Do not interpolate output at time steps. The current output equals the output at the most recent time step for which data exists.

Dependency

To display and enable this parameter for a Trigger port block at the root-level of a model, select `rising`, `falling`, or either from the **Trigger type** list.

Programmatic Use

Block parameter: `Interpolate`

Type: character vector

Value: `'on'` | `'off'`

Default: `'on'`

See Also

[Enabled and Triggered Subsystem](#) | [Function-Call Subsystem](#) | [Subsystem](#) | [Triggered Subsystem](#)

Topics

[“Conditional Subsystems”](#)

[“Triggered Subsystems”](#)

[“Enabled and Triggered Subsystems”](#)

[“Using Function-Call Subsystems”](#)

[“Export-Function Models”](#)

Introduced before R2006a

Trigger-Based Linearization

Generate linear models in base workspace when triggered



Library

Model-Wide Utilities

Description

When triggered, this block calls `linmod` or `dlinmod` to create a linear model for the system at the current operating point. No trimming is performed. The linear model is stored in the base workspace as a structure, along with information about the operating point at which the snapshot was taken. Multiple snapshots are appended to form an array of structures.

The block sets the following model parameters to the indicated values:

- `BufferReuse` = 'off'
- `RTWInlineParameters` = 'on'
- `BlockReductionOpt` = 'off'

The name of the structure used to save the snapshots is the name of the model appended by `_Trigger_Based_Linearization`, for example, `vdp_Trigger_Based_Linearization`. The structure has the following fields:

Field	Description
a	The A matrix of the linearization
b	The B matrix of the linearization

Field	Description
c	The C matrix of the linearization
d	The D matrix of the linearization
StateName	Names of the model's states
OutputName	Names of the model's output ports
InputName	Names of the model's input ports
OperPoint	A structure that specifies the operating point of the linearization. The structure specifies the value of the model's states (<code>OperPoint.x</code>) and inputs (<code>OperPoint.u</code>) at the operating point time (<code>OperPoint.t</code>).
Ts	The sample time of the linearization for a discrete linearization

Use the Timed-Based Linearization block to generate linear models at predetermined times.

You can use state and simulation time logging to extract the model states at operating points. For example, suppose that you want to get the states of the vdp example model when the signal `x1` triggers the Trigger-Based Linearization block on a rising edge.

- 1 Open the model and drag an instance of this block from the Model-Wide Utilities library and drop the instance into the model.
- 2 Connect the block's trigger port to the signal labeled `x1`.
- 3 Open the model's **Model Configuration Parameters** dialog box.
- 4 Select the **Data Import/Export** pane.
- 5 Check **States** and **Time** on the **Save to Workspace** control panel
- 6 Select OK to confirm the selections and close the dialog box.
- 7 Simulate the model.

At the end of the simulation, the following variables appear in the MATLAB workspace: `vdp_Trigger_Based_Linearization`, `tout`, and `xout`.

- 8 Get the index to the first operating point time by entering the following at the MATLAB command line:

```
ind1 = find(vdp_Trigger_Based_Linearization(1).OperPoint.t==tout);
```

- 9 Get the state vector at this operating point.

```
x1 = xout(ind1, :);
```

Data Type Support

The trigger port accepts signals of any numeric data type that Simulink supports.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Trigger type

Type of event on the trigger input signal that triggers generation of a linear model. See the **Trigger type** parameter of the Trigger block for an explanation of the various trigger types that you can select.

Sample time (of linearized model)

Specify a sample time to create a discrete-time linearization of the model (see “Discrete-Time System Linearization” on page 2-47).

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	No

See Also

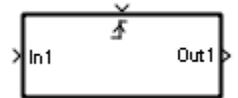
Timed-Based Linearization

Introduced before R2006a

Triggered Subsystem

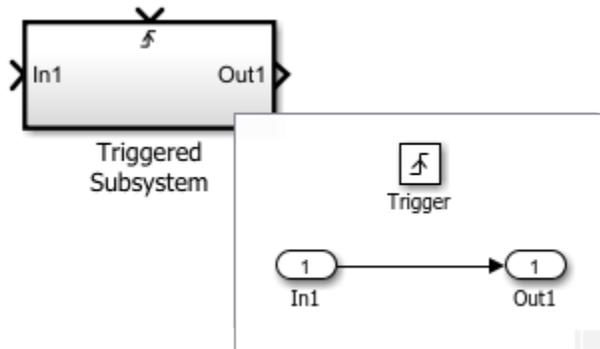
Subsystem whose execution is triggered by external input

Library: Ports & Subsystems



Description

The Triggered Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that executes each time the control signal has a trigger event.



Use Trigger Subsystem blocks to model:

- A task that runs with the detection of an event.
- An interrupt from I/O hardware.
- A processor request to handle an exception or error.

Ports

Input

In — Signal input to a subsystem block

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

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

Trigger — Control signal input to a subsystem block

scalar

Placing a Trigger block in a subsystem block adds an external input port to the block.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

Out — Signal output from a subsystem

scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

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

See Also

Blocks

Enabled Subsystem | Enabled and Triggered Subsystem | Function-Call Subsystem | Subsystem | Trigger

Topics

“Conditional Subsystems”

“Enabled Subsystems”

“Triggered Subsystems”

“Enabled and Triggered Subsystems”

“Using Function-Call Subsystems”

Introduced before R2006a

Trigonometric Function

Specified trigonometric function on input



Library

Math Operations

Description

Supported Functions

The Trigonometric Function block performs common trigonometric functions. You can select one of the following functions from the **Function** parameter list.

Function	Description	Mathematical Expression	MATLAB Equivalent
sin	Sine of the input	$\sin(u)$	sin
cos	Cosine of the input	$\cos(u)$	cos
tan	Tangent of the input	$\tan(u)$	tan
asin	Inverse sine of the input	$\text{asin}(u)$	asin
acos	Inverse cosine of the input	$\text{acos}(u)$	acos
atan	Inverse tangent of the input	$\text{atan}(u)$	atan

Function	Description	Mathematical Expression	MATLAB Equivalent
atan2	Four-quadrant inverse tangent of the input	$\text{atan2}(u)$	atan2
sinh	Hyperbolic sine of the input	$\text{sinh}(u)$	sinh
cosh	Hyperbolic cosine of the input	$\text{cosh}(u)$	cosh
tanh	Hyperbolic tangent of the input	$\text{tanh}(u)$	tanh
asinh	Inverse hyperbolic sine of the input	$\text{asinh}(u)$	asinh
acosh	Inverse hyperbolic cosine of the input	$\text{acosh}(u)$	acosh
atanh	Inverse hyperbolic tangent of the input	$\text{atanh}(u)$	atanh
sincos	Sine of the input; cosine of the input	—	—
cos + jsin	Complex exponential of the input	—	—

The block output is the result of applying the function to one or more inputs in radians. Each function supports:

- Scalar operations
- Element-wise vector and matrix operations

Note Not all compilers support the `asinh`, `acosh`, and `atanh` functions. If you use a compiler that does not support those functions, a warning appears and the generated code fails to link.

Block Appearance for the atan2 Function

If you select the `atan2` function, the block shows two inputs. The first input is the y -axis or imaginary part of the function argument. The second input is the x -axis or real part of the function argument. (See “Port Location After Rotating or Flipping” in the Simulink documentation for a description of the port order for various block orientations.)

Block Appearance for the sincos Function

If you select the `sincos` function, the block shows two outputs. The first output is the sine of the function argument, and the second output is the cosine of the function argument.

Definitions

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

Data Type Support

The block accepts input signals of the following data types:

Functions	Input Data Types
<ul style="list-style-type: none"> • sin • cos • sincos • cos + jsin • atan2 	<ul style="list-style-type: none"> • Floating point • Fixed point (only when Approximation method is CORDIC)
<ul style="list-style-type: none"> • tan • asin • acos • atan • sinh • cosh • tanh • asinh • acosh • atanh 	<ul style="list-style-type: none"> • Floating point

Complex input signals are supported for all functions in this block, except atan2.

You can use floating-point input signals when you set **Approximation method** to None or CORDIC. However, the block output data type depends on which of these approximation method options you choose.

Input Data Type	Approximation Method	Output Data Type
Floating point	None	Depends on your selection for Output signal type . Options are auto (same data type as input), real, or complex.
Floating point	CORDIC	Same as input. Output signal type is not available when you use the CORDIC approximation method to compute the block output.

You can use fixed-point input signals only when **Approximation method** is set to CORDIC. The CORDIC approximation is available for the sin, cos, sincos, cos +

`jsin`, and `atan2` functions. For the `atan2` function, the relationship between input and output data types depends also on whether the fixed-point input is signed or unsigned.

Input Data Type	Function	Output Data Type
Fixed point, signed or unsigned	<code>sin</code> , <code>cos</code> , <code>sincos</code> , and <code>cos + jsin</code>	<code>fixdt(1, WL, WL - 2)</code> where WL is the input word length This fixed-point type provides the best precision for the CORDIC algorithm.
Fixed point, signed	<code>atan2</code>	<code>fixdt(1, WL, WL - 3)</code>
Fixed point, unsigned	<code>atan2</code>	<code>fixdt(1, WL, WL - 2)</code>

For CORDIC approximations:

- Input must be real for the `sin`, `cos`, `sincos`, `cos + jsin`, and `atan2` functions.
- Output is real for the `sin`, `cos`, `sincos`, and `atan2` functions.
- Output is complex for the `cos + jsin` function.

Invalid Inputs for CORDIC Approximations

If you use the CORDIC approximation method (see “Definitions” on page 1-1701), the block input has some further requirements.

For the `sin`, `cos`, `sincos`, and `cos + jsin` functions:

- When you use signed fixed-point types, the input angle must fall within the range $[-2\pi, 2\pi)$ radians.
- When you use unsigned fixed-point types, the input angle must fall within the range $[0, 2\pi)$ radians.

For the `atan2` function:

- Inputs must be the same size, or at least one value must be a scalar value.
- Both inputs must have the same data type.
- When you use signed fixed-point types, the word length must be **126** or less.
- When you use unsigned fixed-point types, the word length must be **125** or less.

This table summarizes what happens for an invalid input.

Block Usage	Effect of Invalid Input
Simulation	An error appears.
Generated code	Undefined behavior occurs. Avoid relying on undefined behavior for generated code or Accelerator modes.
Accelerator modes	

Parameters

Function

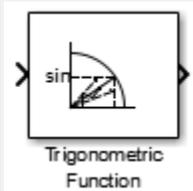
Specify the trigonometric function. The name of the function on the block icon changes to match your selection.

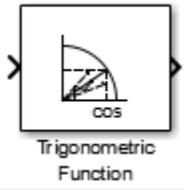
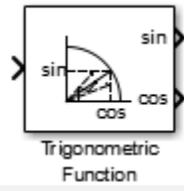
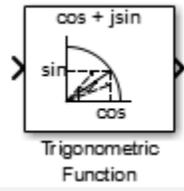
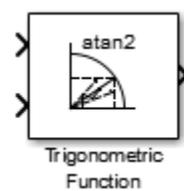
Approximation method

Specify the type of approximation for computing output. This parameter appears only when you set **Function** to `sin`, `cos`, `sincos`, `cos + jsin`, or `atan2`.

Approximation Method	Data Types Supported	When to Use This Method
None (default)	Floating point	You want to use the default Taylor series algorithm.
CORDIC	Floating point and fixed point	You want a fast, approximate calculation.

If you select `CORDIC` and enlarge the block from the default size, the block icon changes:

Function	Block Icon
<code>sin</code>	

Function	Block Icon
cos	
sincos	
cos + jsin	
atan2	

Number of iterations

Specify the number of iterations to perform the CORDIC algorithm. The default value is 11.

- When the block input uses a floating-point data type, the number of iterations can be a positive integer.
- When the block input is a fixed-point data type, the number of iterations cannot exceed the word length.

For example, if the block input is `fixdt(1, 16, 15)`, the word length is 16. In this case, the number of iterations cannot exceed 16.

This parameter appears when both of the following conditions hold:

- You set **Function** to `sin`, `cos`, `sincos`, `cos + jsin`, or `atan2`.
- You set **Approximation method** to `CORDIC`.

Output signal type

Specify the output signal type of the Trigonometric Function block as `auto`, `real`, or `complex`.

Function	Input Signal Type	Output Signal Type		
		Auto	Real	Complex
Any selection for the Function parameter	real	real	real	complex
	complex	complex	error	complex

Note When **Function** is `atan2`, complex input signals are not supported for simulation or code generation.

Setting **Approximation method** to `CORDIC` disables this parameter.

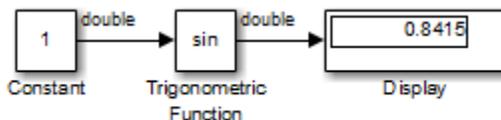
Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than `-1`. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Examples

sin Function with Floating-Point Input

Suppose that you have the following model:



The key block parameters for the Constant block are:

Parameter	Setting
Constant value	1
Output data type	Inherit: Inherit from 'Constant value'

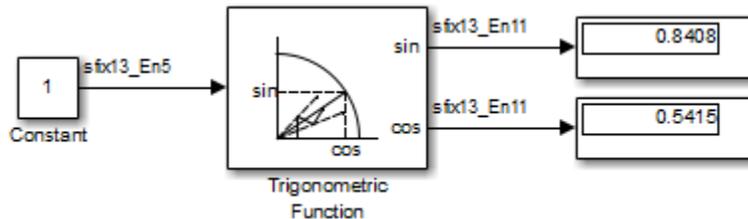
The block parameters for the Trigonometric Function block are:

Parameter	Setting
Function	sin
Approximation method	None
Output signal type	auto

The output type of the Trigonometric Function block is the same as the input because the input type is floating point and **Approximation method** is None.

sincos Function with Fixed-Point Input

Suppose that you have the following model:



The key block parameters for the Constant block are:

Parameter	Setting
Constant value	1 This value must fall within the range $[-2\pi, 2\pi)$ because the Trigonometric Function block uses the CORDIC algorithm and the block input uses a signed fixed-point type.
Output data type	fixdt(1, 13, 5)

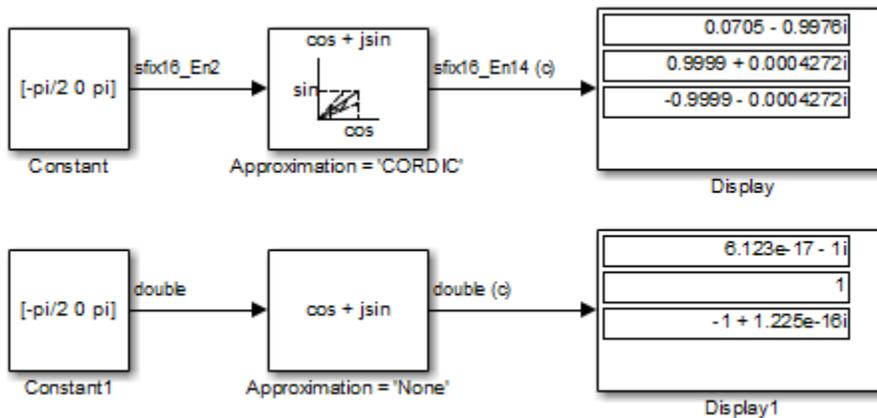
The block parameters for the Trigonometric Function block are:

Parameter	Setting
Function	sincos
Approximation method	CORDIC
Number of iterations	11

The output type of the Trigonometric Function block is `fixdt(1,13,11)` because the input type is fixed point and **Approximation method** is CORDIC. The output fraction length equals the input word length – 2.

Block Behavior for Complex Exponential Output

The following model compares the complex exponential output for the two different approximation methods:



The key block parameters for the Constant blocks are:

Block	Parameter	Setting
Constant	Constant value	$[-\pi/2 \ 0 \ \pi]$
	Output data type	<code>fixdt(1,16,2)</code>
Constant1	Constant value	$[-\pi/2 \ 0 \ \pi]$
	Output data type	<code>double</code>

The block parameters for the Trigonometric Function blocks are:

Block	Parameter	Setting
Approximation = 'CORDIC'	Function	cos + jsin
	Approximation method	CORDIC
	Number of iterations	11
Approximation = 'None'	Function	cos + jsin
	Approximation method	None

When the **Approximation method** is CORDIC, the input data type can be fixed point, in this case: `fixdt(1,16,2)`. The output data type is `fixdt(1,16,14)` because the output fraction length equals the input word length – 2.

When the **Approximation method** is None, the input data type must be floating point. The output data type is the same as the input.

Characteristics

Data Types	Double Single
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Math Function, Sqrt

References

- [1] Volder, JE. “The CORDIC Trigonometric Computing Technique.” *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

- [2] Andraka, R. “A survey of CORDIC algorithm for FPGA based computers.” *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. Feb. 22–24, 1998, pp. 191–200.
- [3] Walther, J.S. “A Unified Algorithm for Elementary Functions.” Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf
- [4] Schelin, Charles W. “Calculator Function Approximation.” *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

Introduced before R2006a

Unary Minus

Negate input



Library

Math Operations

Description

The Unary Minus block negates the input.

For signed-integer data types, the unary minus of the most negative value is not representable by the data type. In this case, the **Saturate on integer overflow** check box controls the behavior of the block:

If you...	The block...	And...
Select this check box	Saturates to the most positive value of the integer data type	<ul style="list-style-type: none"> For 8-bit signed integers, -128 maps to 127. For 16-bit signed integers, -32768 maps to 32767. For 32-bit signed integers, -2147483648 maps to 2147483647.
Do not select this check box	Wraps to the most negative value of the integer data type	<ul style="list-style-type: none"> For 8-bit signed integers, -128 remains -128. For 16-bit signed integers, -32768 remains -32768. For 32-bit signed integers, -2147483648 remains -2147483648.

Data Type Support

The Unary Minus block accepts and outputs signals of the following data types:

- Floating point
- Signed integer
- Fixed point

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Sample time

Note This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1 . To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Saturate on integer overflow

Select to have integer overflows saturate. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

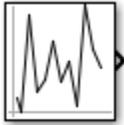
Data Types	Double Single Base Integer Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	No

Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Uniform Random Number

Generate uniformly distributed random numbers



Library

Sources

Description

The Uniform Random Number block generates uniformly distributed random numbers over an interval that you specify. To generate normally distributed random numbers, use the Random Number block.

You can generate a repeatable sequence using any Uniform Random Number block with the same nonnegative seed and parameters. The seed resets to the specified value each time a simulation starts.

Avoid integrating a random signal, because solvers must integrate relatively smooth signals. Instead, use the Band-Limited White Noise block.

The numeric parameters of this block must have the same dimensions after scalar expansion. If you select the **Interpret vector parameters as 1-D** check box and the numeric parameters are row or column vectors after scalar expansion, the block outputs a 1-D signal. If you clear the **Interpret vector parameters as 1-D** check box, the block outputs a signal of the same dimensionality as the parameters.

Data Type Support

The Uniform Random Number block accepts and outputs a real signal of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Minimum

Specify the minimum of the interval. The default is -1 .

Maximum

Specify the maximum of the interval. The default is 1 .

Seed

Specify the starting seed for the random number generator. The default is 0 .

The seed must be 0 or a positive integer. Output is repeatable for a given seed.

Sample time

Specify the time interval between samples. The default is 0.1 . See “Specify Sample Time” in the Simulink documentation for more information.

Interpret vector parameters as 1-D

If you select this check box and the other parameters are row or column vectors after scalar expansion, the block outputs a 1-D signal. Otherwise, the block outputs a signal of the same dimensionality as the other parameters. For more information, see “Determining the Output Dimensions of Source Blocks” in the Simulink documentation.

Characteristics

Data Types	Double
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

The generator algorithm is identical to the one used in MATLAB Version 4.0 by the `rand` and `randn` functions. For details on the `mcg16807` algorithm, see “Choosing a Random Number Generator” (MATLAB) in the MATLAB documentation.

To use other algorithms supported by MATLAB in a Simulink model, generate a set of random numbers in MATLAB, and store the output as a `.mat` file. Use this `.mat` file as the random number input for your simulation. For more information, see “Creating and Controlling a Random Number Stream” (MATLAB). To create multiple independent streams using MATLAB, see “Multiple Streams” (MATLAB)

Note Using multiple seeds to generate multiple parallel independent streams for a generator algorithm is not recommended for the `mcg16807` algorithm. Instead, use the method described above.

See Also

Random Number

Introduced before R2006a

Unit System Configuration

Configure units



Library

Ports & Subsystems

Description

The Unit System Configuration block specifies allowed and disallowed unit systems for the component. It restricts units systems for a subsystem or top model and all its children, unless you override it with another Unit System Configuration block in a child.

This block supports normal, accelerator, and rapid accelerator modes and fast restart.

Parameters

Disallowed unit systems

Displays disallowed unit system.

To designate a unit system as disallowed, select it in the **Allowed unit systems** column and click <<**Disallow**.

Default: None

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Allowed unit systems

Displays allowed unit system.

To designate a unit system as allowed, select it in the **Disallowed unit systems** column and click **Allow>>**.

Default: SI, English, SI (extended), CGS

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Allow all unit systems

Allow or restrict unit systems.

Default: On

On

Allow all unit systems.

Off

Restrict unit systems to those in **Allowed unit systems**.

Selecting the **Allow all unit systems** check box disables the **Disallowed unit systems** and **Allowed unit systems** parameters.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See Also

Inport | Outport | Unit Conversion

Topics

“Units in Simulink”

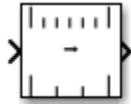
“Unit Specification in Simulink Models”

“Restricting Unit Systems”

Introduced in R2016a

Unit Conversion

Convert units



Library

Signal Attributes

Description

The Unit Conversion block converts the unit of the input signal to the output signal. The block can convert if the units are separated by a scaling factor or offset, or are inverse units, for example:

- $y=a*U$
- $y=a*U+b$, where a is the scale and b is the offset
- $y=a/U$

This block supports normal, accelerator, and rapid accelerator modes and fast restart.

Data Type Support

The Unit Conversion block accepts and outputs real or complex values of the following data types:

- Floating point
- Built-in integer
- Fixed point

For more information, see “Data Types Supported by Simulink”.

Parameters

Output data type

Specify the output data type.

Default: `Inherit`: `Inherit via internal rule`

`Inherit`: `Inherit via internal rule`

Simulink chooses intermediate and output data types to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change.

`Inherit`: `Inherit via back propagation`

Output data type is inherited via back propagation. Internal rules determine the intermediate data types and Simulink casts the final results to the output data type.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Converting Units” for more information.

Characteristics

Data Types	Double Single Base Integer Fixed-Point
Sample Time	Inherited from driving block
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

“Converting Units”

Topics

“Units in Simulink”

Introduced in R2016a

Unit Delay

Delay signal one sample period



Library

Discrete

Description

The Unit Delay block holds and delays its input by the sample period you specify. When placed in an iterator subsystem, it holds and delays its input by one iteration. This block is equivalent to the z^{-1} discrete-time operator. The block accepts one input and generates one output. Each signal can be scalar or vector. If the input is a vector, the block holds and delays all elements of the vector by the same sample period.

You specify the block output for the first sampling period with the **Initial conditions** parameter. Careful selection of this parameter can minimize unwanted output behavior. You specify the time between samples with the **Sample time** parameter. A setting of -1 means the block inherits the **Sample time**.

Note The Unit Delay block errors out if you use it to create a transition between blocks operating at different sample rates. Use the Rate Transition block instead.

Comparison with Similar Blocks

Blocks with Similar Functionality

The Unit Delay, Memory, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different. The sections that follow highlight some of these differences.

Recommended Usage for Each Block

Block	Purpose of the Block	Reference Examples
Unit Delay	Implement a delay using a discrete sample time that you specify. The block accepts and outputs signals with a discrete sample time.	<ul style="list-style-type: none"> sldemo_enginewc (Compression subsystem)
Memory on page 1-915	Implement a delay by one major integration time step. Ideally, the block accepts continuous (or fixed in minor time step) signals and outputs a signal that is fixed in minor time step.	<ul style="list-style-type: none"> sldemo_bounce sldemo_clutch (Friction Mode Logic/Lockup FSM subsystem)
Zero-Order Hold	Convert an input signal with a continuous sample time to an output signal with a discrete sample time.	<ul style="list-style-type: none"> sldemo_radar_eml aero_dap3dof

Overview of Block Capabilities

Capability	Block		
	Unit Delay	Memory	Zero-Order Hold
Specification of initial condition	Yes	Yes	No, because the block output at time $t = 0$ must match the input value.

Capability	Block		
	Unit Delay	Memory	Zero-Order Hold
Specification of sample time	Yes	No, because the block can only inherit sample time (from the driving block or the solver used for the entire model).	Yes
Support for frame-based signals	Yes	No	Yes
Support for state logging	Yes	No	No

Data Type Support

The Unit Delay block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types. If the data type of the input signal is user-defined, the initial condition must be zero.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name is resolved
- Min and Max values of the signal object

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder) in the Simulink Coder documentation.

Initial conditions

Specify the output of the simulation for the first sampling period, during which the output of the Unit Delay block is otherwise undefined.

Default: 0

The **Initial conditions** parameter is converted from a double to the input data type offline using round-to-nearest and saturation.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Input processing

Specify whether the Unit Delay block performs sample- or frame-based processing.

Default: Elements as channels (sample based)

Elements as channels (sample based)

Treat each element of the input as a separate channel (sample-based processing).

Columns as channels (frame based)

Treat each column of the input as a separate channel (frame-based processing).

Inherited

Sets the block to inherit the processing mode from the input signal and delay the input accordingly. You can identify whether the input signal is sample or frame based by looking at the signal line. Simulink represents sample-based signals with a single line and frame-based signals with a double line.

Note When you choose the `Inherited` option for the **Input processing** parameter, and the input signal is frame-based, Simulink® will generate a warning or error in future releases.

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input `u`. All other input signals must be sample based.

Input Signal <code>u</code>	Input Processing Mode	Block Works?
Sample based	Sample based	Yes
Frame based		No, produces an error

Input Signal u	Input Processing Mode	Block Works?
Sample based	Frame based	Yes
Frame based		Yes
Sample based	Inherited	Yes
Frame based		Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox) in the DSP System Toolbox documentation.

Frame-based processing requires a DSP System Toolbox license.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Sample time (-1 for inherited)

Enter the discrete interval between sample time hits or specify -1 to inherit the sample time.

Default: -1

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in “Types of Sample Time”.

See also “Specify Sample Time” in the online documentation for more information.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

State name

Use this parameter to assign a unique name to each state.

Default: ' '

- If left blank, no name is assigned.
- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click the **Apply** button.

For more information, see “Discrete Block State Naming in Generated Code” (Simulink Coder).

Parameter: StateName

Type: character vector

Value: ' '

Default: ' '

State name must resolve to Simulink signal object

Require that state name resolve to Simulink signal object.

Default: Off

On

Require that state name resolve to Simulink signal object.

Off

Do not require that state name resolve to Simulink signal object.

State name enables this parameter. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than `None`.

Selecting this check box disables **Code generation storage class**.

Parameter: StateMustResolveToSignalObject

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Signal object class

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

To programmatically set this parameter, use `StateSignalObject`.

For examples and more information about storage classes, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

`Simulink.Signal`

Use custom storage classes from the built-in package `Simulink`.

Package.Class

Use custom storage classes from the package that defines the class that you select.

If the class that you want does not appear in the list, select `Customize class lists`. For instructions, see “Apply Custom Storage Classes Directly to Signal Lines, Block States, and Outport Blocks” (Embedded Coder).

Code generation storage class

Select state storage class for code generation.

Default: `Auto`

`Auto`

`Auto` is the appropriate storage class for states that you do not need to interface to external code.

StorageClass

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Control Signals and States in Code by

Applying Storage Classes” (Simulink Coder). For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than Simulink.

State name enables this parameter.

TypeQualifier

Note **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

Specify a storage type qualifier such as `const` or `volatile`.

- **Default:** ' ' (empty character vector)
- `const`
- `volatile`

Setting **Code generation storage class** to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `SimulinkGlobal` enables this parameter. This parameter is hidden unless you previously set its value.

Parameter Name: `RTWStateStorageTypeQualifier`

Value Type: character vector

Default: ' ' (empty character vector)

Bus Support

The Unit Delay block is a bus-capable block. The input can be a virtual or nonvirtual bus signal subject to the following restrictions:

- **Initial conditions** must be zero, a nonzero scalar, or a finite numeric structure.
- If **Initial conditions** is zero or a structure, and you specify a **State name**, the input cannot be a virtual bus.
- If **Initial conditions** is a nonzero scalar, no **State name** can be specified.

For information about specifying an initial condition structure, see “Specify Initial Conditions for Bus Signals”.

All signals in a nonvirtual bus input to a Unit Delay block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “Specify Bus Signal Sample Times” and Bus-Capable Blocks for more information.

You can use an array of buses as an input signal to a Unit Delay block. You can specify the **Initial conditions** parameter with:

- The value 0. In this case, all of the individual signals in the array of buses use the initial value 0.
- An array of structures that specifies an initial condition for each of the individual signals in the array of buses.
- A single scalar structure that specifies an initial condition for each of the elements that the bus type defines. Use this technique to specify the same initial conditions for each of the buses in the array.

For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Examples

For an example of how to use the Unit Delay block, see the `sldemo_enginewc` model. The Unit Delay block appears in the `Compression` subsystem.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
------------	---------------------------------------------------------------------------

Sample Time	Specified in the Sample time parameter
Direct Feedthrough	No
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

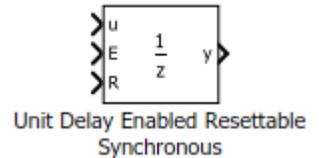
“Scalar Expansion of Inputs and Parameters” | Delay | Memory | Resettable Delay | Tapped Delay | Variable Integer Delay | Zero-Order Hold

Introduced before R2006a

Unit Delay Enabled Resettable Synchronous

Delay input signal by one sample period when external Enable signal is true and external Reset signal is false

Library: HDL Coder / Discrete / Unit Delay Enabled
Resettable Synchronous



Description

The Unit Delay Enabled Resettable Synchronous block combines the functionality of the Unit Delay Enabled Synchronous block and the Unit Delay Resettable Synchronous block.

The Unit Delay Enabled Resettable Synchronous block delays the input signal u by one sample period when the external Enable signal is true and when the external Reset signal is false. When the Enable signal is false, the state and output signal hold the previous value. When the Reset signal is true, the state and output signal take the value of the **Initial condition** parameter. The Enable and Reset signals are true when E and R are nonzero and false when E and R equal zero.

The Unit Delay Enabled Synchronous block implementation consists of a Synchronous Subsystem that contains an Enabled Delay block with a **Delay length** of one and a State Control block in *Synchronous* mode. When you use this block in your model and have HDL Coder installed, your model generates cleaner HDL code and uses fewer hardware resources due to the *Synchronous* behavior of the State Control block.

Limitations

- The block does not support vector inputs on the Reset and Enable ports.
- You cannot use the block inside Enabled Subsystem, Triggered Subsystem, or Resettable Subsystem blocks that use *Classic* semantics. The Subsystem must use *Synchronous* semantics.

Ports

Input

u — Input signal

Scalar | Vector | Matrix | Array | Bus

The Unit Delay Enabled Resettable Synchronous block accepts the input signal of the data types listed below. For more information, see “Data Types Supported by Simulink”.

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

Input

E — Enable signal

Scalar

The Unit Delay Enabled Synchronous block accepts the Enable signal of the data types listed below. For more information, see “Data Types Supported by Simulink”.

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

R — Reset signal

Scalar

The Unit Delay Resettable Synchronous block accepts the Reset signal of the data types listed below. For more information, see “Data Types Supported by Simulink”.

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

Output

y — Output signal

Scalar | Vector | Matrix | Array | Bus

Output data type always matches input.

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

Parameters

Initial condition — Initial output of simulation

0.0 (default) | `Scalar` | `Vector` | `Matrix` | `Array` | `Bus`

The **Initial condition** can take a scalar input or use the same data type as the input signal. You cannot run the simulation with NaN or Inf as the **Initial condition**.

Programmatic Use

Block parameter: `InitialCondition`

Type: character vector

Value: `'0'` | `'[n]'` | `'[m n]'`

Default: `'0'`

Sample time — Time interval between samples

-1 (default) | `Scalar` | `Vector`

The **Sample time** must be a real double scalar that specifies the period or a real double vector of length two that specifies the period and offset. The period and offset must be finite and non-negative with offset less than the period.

Programmatic Use

Block parameter: `SampleTime`

Type: character vector

Value: `'-1'` | `'[n]'` | `'[m n]'`

Default: `'-1'`

Block Characteristics

Data Types	<code>double</code> <code>single</code> <code>base integer</code> <code>fixed point</code> <code>bus</code>
Sample Time	<code>Inherit</code>
Direct Feedthrough	<code>Yes</code>

Multidimensional Signals	Scalar
Variable-Size Signals	Yes
Zero-Crossing Detection	No

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™.

Fixed-Point Conversion

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

See Also

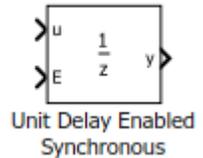
State Control | Unit Delay | Unit Delay Resettable Synchronous | Unit Delay Enabled Synchronous

Introduced in R2017b

Unit Delay Enabled Synchronous

Delay input signal by one sample period when external Enable signal is true

Library: HDL Coder / Discrete / Unit Delay Enabled
Synchronous



Description

The Unit Delay Enabled Synchronous block delays the input signal u by one sample period when the external Enable signal is true. When the Enable signal is false, the state and output signal hold the previous value. The Enable signal is true when E is not zero and false when E is zero.

The Unit Delay Enabled Synchronous block implementation consists of a Synchronous Subsystem that contains an Enabled Delay block with a **Delay length** of one and a State Control block in Synchronous mode. When you use this block in your model and have HDL Coder installed, your model generates cleaner HDL code and uses fewer hardware resources due to the Synchronous behavior of the State Control block.

Limitations

- The block does not support vector inputs on the Enable port.
- You cannot use the block inside Enabled Subsystem, Triggered Subsystem, or Resettable Subsystem blocks that use Classic semantics. The Subsystem must use Synchronous semantics.

Ports

Input

u — Input signal

Scalar | Vector | Matrix | Array | Bus

The Unit Delay Enabled Synchronous block accepts the input signal of the data types listed below. For more information, see “Data Types Supported by Simulink”.

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

Input

E — Enable signal

Scalar

The Unit Delay Enabled Synchronous block accepts the Enable signal of the data types listed below. For more information, see “Data Types Supported by Simulink”.

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

Output

y — Output signal

Scalar | Vector | Matrix | Array | Bus

Output data type always matches input.

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

Parameters

Initial condition — Initial output of simulation

0.0 (default) | Scalar | Vector | Matrix | Array | Bus

The **Initial condition** can take a scalar input or use the same data type as the input signal. You cannot run the simulation with NaN or Inf as the **Initial condition**.

Programmatic Use

Block parameter: InitialCondition

Type: character vector

Value: '0' | '[n]' | '[m n]'

Default: '0'

Sample time — Time interval between samples

-1 (default) | Scalar | Vector

The **Sample time** must be a real double scalar that specifies the period or a real double vector of length two that specifies the period and offset. The period and offset must be finite and non-negative with offset less than the period.

Programmatic Use

Block parameter: SampleTime

Type: character vector

Value: '-1' | '[n]' | '[m n]'

Default: '-1'

Block Characteristics

Data Types	double single base integer fixed point bus
Sample Time	Inherit
Direct Feedthrough	Yes
Multidimensional Signals	Scalar
Variable-Size Signals	Yes
Zero-Crossing Detection	No

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™.

Fixed-Point Conversion

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

See Also

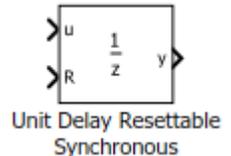
State Control | Unit Delay | Unit Delay Resettable Synchronous | Unit Delay Enabled
Resettable Synchronous

Introduced in R2017b

Unit Delay Resettable Synchronous

Delay input signal by one sample period when external Reset signal is false

Library: HDL Coder / Discrete / Unit Delay Resettable Synchronous



Description

The Unit Delay Resettable Synchronous block delays the input signal u by one sample period when the external Reset signal is false. When the Reset signal is true, the state and output signal take the value of the **Initial condition** parameter. The Reset signal is true when R is not zero and false when R is zero.

The Unit Delay Resettable Synchronous block implementation consists of a Synchronous Subsystem that contains a Resettable Delay block with a **Delay length** of one and a State Control block in Synchronous mode. When you use this block in your model and have HDL Coder installed, your model generates cleaner HDL code and uses fewer hardware resources due to the Synchronous behavior of the State Control block.

Limitations

- The block does not support vector inputs on the Reset port.
- You cannot use the block inside Enabled Subsystem, Triggered Subsystem, or Resettable Subsystem blocks that use Classic semantics. The Subsystem must use Synchronous semantics.

Ports

Input

u — Input signal

Scalar | Vector | Matrix | Array | Bus

The Unit Delay Resettable Synchronous block accepts the input signal of the data types listed below. For more information, see “Data Types Supported by Simulink”.

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

Input

R — Reset signal

Scalar

The Unit Delay Resettable Synchronous block accepts the Reset signal of the data types listed below. For more information, see “Data Types Supported by Simulink”.

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

Output

y — Output signal

Scalar | Vector | Matrix | Array | Bus

Output data type always matches input.

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

Parameters

Initial condition — Initial output of simulation

0.0 (default) | Scalar | Vector | Matrix | Array | Bus

The **Initial condition** can take a scalar input or use the same data type as the input signal. You cannot run the simulation with NaN or Inf as the **Initial condition**.

Programmatic Use

Block parameter: InitialCondition

Type: character vector

Value: '0' | '[n]' | '[m n]'

Default: '0'

Sample time — Time interval between samples

-1 (default) | Scalar | Vector

The **Sample time** must be a real double scalar that specifies the period or a real double vector of length two that specifies the period and offset. The period and offset must be finite and non-negative with offset less than the period.

Programmatic Use

Block parameter: SampleTime

Type: character vector

Value: '-1' | '[n]' | '[m n]'

Default: '-1'

Block Characteristics

Data Types	double single base integer fixed point bus
Sample Time	Inherit
Direct Feedthrough	Yes
Multidimensional Signals	Scalar
Variable-Size Signals	Yes
Zero-Crossing Detection	No

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™.

Fixed-Point Conversion

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

See Also

State Control | Unit Delay | Unit Delay Enabled Resettable Synchronous | Unit Delay Enabled Synchronous

Introduced in R2017b

Variable Integer Delay

Delay input signal by variable sample period

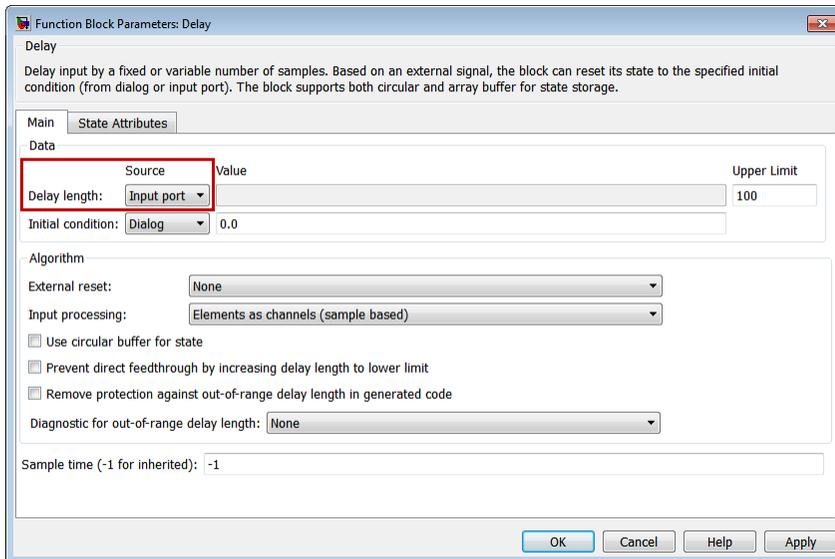


Library

Discrete

Description

The Variable Integer Delay block is a variant of the Delay block that has the source of the delay length set to `Input port`, by default.



Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Code Generation	Yes

See Also

Delay | Resettable Delay | Tapped Delay | Unit Delay

Introduced in R2012b

Variable Time Delay, Variable Transport Delay

Delay input by variable amount of time



Library

Continuous

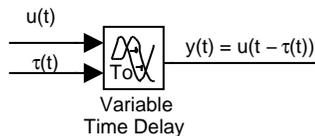
Description

The Variable Transport Delay and Variable Time Delay appear as two blocks in the Simulink block library. However, they are the same Simulink block with different settings of a **Select delay type** parameter. Use this parameter to specify the mode in which the block operates.

Variable Time Delay

In this mode, the block has a data input, a time delay input, and a data output. (See “Port Location After Rotating or Flipping” in the Simulink documentation for a description of the port order for various block orientations.) The output at the current time step equals the value of its data input at a previous time step. This time step is the current simulation time minus a delay time specified by the time delay input.

$$y(t) = u(t - t_0) = u(t - \tau(t))$$



During the simulation, the block stores time and input value pairs in an internal buffer. At the start of simulation, the block outputs the value of the **Initial output** parameter until the simulation time exceeds the time delay input. Then, at each simulation step,

the block outputs the signal at the time that corresponds to the current simulation time minus the delay time.

If you want the output at a time between input storing times and the solver is a continuous solver, the block interpolates linearly between points. If the time delay is smaller than the step size, the block extrapolates an output point from a previous point. For example, consider a fixed-step simulation with a step size of 1 and the current time at $t = 5$. If the delay is 0.5, the block needs to generate a point at $t = 4.5$, but the most recent stored time value is at $t = 4$. Thus, the block extrapolates the input at 4.5 from the input at 4 and uses the extrapolated value as its output at $t = 5$.

Extrapolating forward from the previous time step can produce a less accurate result than extrapolating back from the current time step. However, the block cannot use the current input to calculate its output value because the input port does not have direct feedthrough.

If the model specifies a discrete solver, the block does not interpolate between time steps. Instead, it returns the nearest stored value that precedes the required value.

Variable Transport Delay

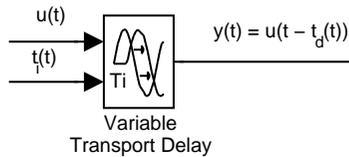
In this mode, the block output at the current time step is equal to the value of its data (top, or left) input at an earlier time step equal to the current time minus a transportation delay.

$$y(t) = u(t - t_d(t))$$

Simulink software finds the transportation delay, $t_d(t)$, by solving the following equation:

$$\int_{t-t_d(t)}^t \frac{1}{t_i(\tau)} d\tau = 1$$

This equation involves an instantaneous time delay, $t_i(t)$, given by the time delay (bottom, or right) input.



Suppose you want to use this block to model the fluid flow through a pipe where the fluid speed varies with time. In this case, the time delay input to the block is

$$t_i(t) = \frac{L}{v_i(t)}$$

where L is the length of the pipe and $v_i(t)$ is the speed of the fluid.

Data Type Support

The Variable Time Delay and Variable Transport Delay blocks accept and output real signals of type `double`.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Select delay type

Specify the mode in which the block operates.

Default: The Variable Time Delay block has a default value of `Variable time delay`. The Variable Transport Delay block has a default value of `Variable transport delay`.

`Variable time delay`

Specifies a Variable Time Delay block.

Variable transport delay

Specifies a Variable Transport Delay block.

Setting this parameter to `Variable time delay` enables the **Handle zero delay** parameter.

Setting this parameter to `Variable transport delay` enables the **Absolute tolerance** and **State Name** parameters.

Parameter: `VariableDelayType`

Type: character vector

Value: `'Variable transport delay' | 'Variable time delay'`

Default: `'Variable time delay'`

Maximum delay

Set the maximum value of the time delay input.

Default: 10

- This value defines the largest time delay input that this block allows. The block clips any delay that exceeds this value.
- This value cannot be negative. If the time delay becomes negative, the block clips it to zero and issues a warning message.

Parameter: `MaximumDelay`

Type: scalar or vector

Value: `'10'`

Default: `'10'`

Initial output

Specify the output that the block generates until the simulation time first exceeds the time delay input.

Default: Run-to-run tunable parameter

A Run-to-run tunable parameter cannot be changed during simulation run time. However, changing it before a simulation begins will not cause Accelerator or Rapid Accelerator to regenerate code. Also, the initial output of this block cannot be `inf` or `NaN`.

Parameter: `InitialOutput`

Type: scalar or vector

Value: '0'

Default: '0'

Initial buffer size

Define the initial memory allocation for the number of input points to store. The input points define the history of the input signal up to the current simulation time.

Default: 1024

- If the number of input points exceeds the initial buffer size, the block allocates additional memory.
- After simulation ends, a message displays if the buffer is not sufficient and more memory needs to be allocated.
- Because allocating memory slows down simulation, choose this value carefully if simulation speed is an issue.
- For long time delays, this block might use a large amount of memory, particularly for dimensionalized input.

Parameter: `MaximumPoints`

Type: scalar

Value: '1024'

Default: '1024'

Use fixed buffer size

Specify use of a fixed-size buffer to save input data from previous time steps.

Default: Off

On

The block uses a fixed-size buffer.

Off

The block does not use a fixed-size buffer.

The **Initial buffer size** parameter specifies the buffer size. If the buffer is full, new data replaces data already in the buffer. Simulink software uses linear extrapolation to estimate output values that are not in the buffer.

Note ERT or GRT code generation uses a fixed-size buffer even if you do not select this check box.

- If the input data is linear, selecting this check box can save memory.
- If the input data is nonlinear, do not select this check box. Doing so might yield inaccurate results.

Parameter: FixedBuffer

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Handle zero delay

Convert this block to a direct feedthrough block.

Default: Off

On

The block uses direct feedthrough.

Off

The block does not use direct feedthrough.

Setting **Select delay type** to `Variable time delay` enables this parameter.

Parameter: ZeroDelay

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Direct feedthrough of input during linearization

Cause the block to output its input during linearization and trim, which sets the block mode to direct feedthrough.

Default: Off

On

Enables direct feedthrough of input.

Off

Disables direct feedthrough of input.

- Selecting this check box can cause a change in the ordering of states in the model when you use the functions `linmod`, `dlinmod`, or `trim`. To extract this new state ordering:

- 1 Compile the model using the following command, where `model` is the name of the Simulink model.

```
[sizes, x0, x_str] = model([], [], [], 'lincompile');
```

- 2 Terminate the compilation with the following command.

```
model([], [], [], 'term');
```

- The output argument `x_str`, which is a cell array of the states in the Simulink model, contains the new state ordering. When you pass a vector of states as input to the `linmod`, `dlinmod`, or `trim` functions, the state vector must use this new state ordering.

Parameter: TransDelayFeedthrough

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Pade order (for linearization)

Set the order of the Pade approximation for linearization routines.

Default: 0

- The default value is 0, which results in a unity gain with no dynamic states.
- Setting the order to a positive integer n adds n states to your model, but results in a more accurate linear model of the transport delay.

Parameter: PadeOrder

Type: character vector

Value: '0'

Default: '0'

Absolute tolerance

Specify the absolute tolerance for computing the block state.

Default: auto

- You can enter `auto`, `-1`, or a positive real scalar or vector.
- If you enter `auto`, or `-1`, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute the block states.
- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

Setting **Select delay type** to `Variable transport delay` enables this parameter.

Parameter: AbsoluteTolerance

Type: character vector, scalar, or vector

Value: 'auto' | '-1' | any positive real scalar or vector

Default: 'auto'

State Name (e.g., 'position')

Assign a unique name to each state.

Default: ' '

If this field is blank, no name assignment occurs.

- To assign a name to a single state, enter the name between quotes, for example, 'velocity'.
- To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.
- The state names apply only to the selected block.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Setting **Select delay type** to Variable transport delay enables this parameter.

Parameter: ContinuousStateAttributes

Type: character vector

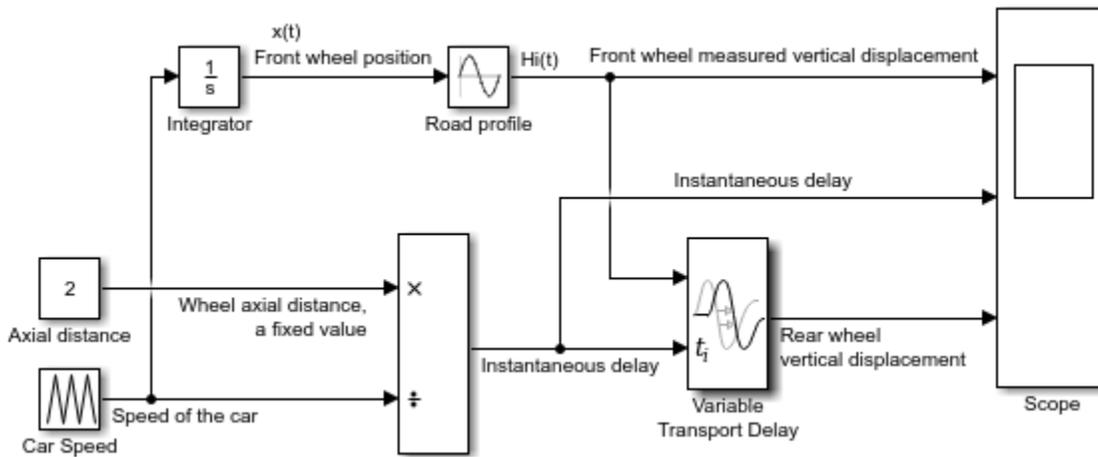
Value: ' ' | user-defined

Default: ' '

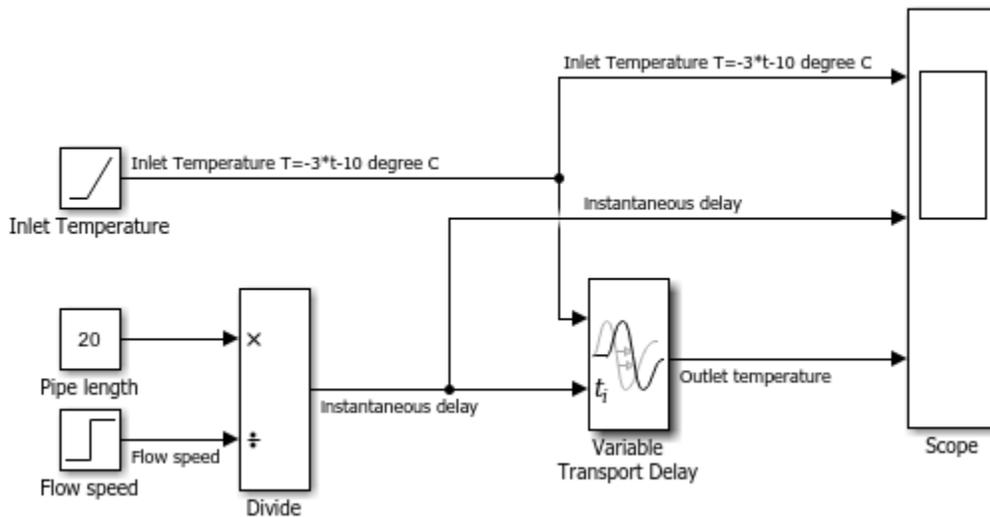
Examples

The `sldemo_VariableTransportDelay` and `sldemo_VariableTransportDelay_pipe` models show how you can use the Variable Transport Delay block.

The `sldemo_VariableTransportDelay` model shows how to model vertical wheel displacement on a one-dimensional car. The Variable Transport Delay block models the delay in vertical displacement of the rear wheel when the road profile changes:



The `sldemo_VariableTransportDelay_pipe` model shows how to model incompressible flow through a fixed-length pipe. The Variable Transport Delay block models the delay in temperature change at the outlet when fluid flow occurs:



Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	Yes, of the time delay (second) input
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

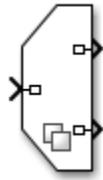
See Also

Transport Delay

Introduced in R2007a

Variant Sink

Route amongst multiple outputs using variants



Library

Signal Routing

Description

The Variant Sink block has one input port and one or more output ports. You can define variant choices as blocks that are connected to the output port so that, at most, one choice is active.

Each output port is associated with a variant control. The variant control that evaluates to `true`, determines which output port is active.

During simulation, Simulink connects the active choice directly to the input port of the Variant Sink block and ignores the inactive choices.

Parameters

Ports and associated conditions

A table of port choices, variant controls, and conditions. The **Variant control** that evaluates to `true` determines which input port must be active.

Default: The table has a row for each variant choice exiting in the Variant Sink block. See each column parameter for its default value:

- “Port” on page 1-1759
- “Variant Control” on page 1-1759
- “Condition (read-only)” on page 1-1760

- “Define, Configure, and Activate Variants”
- “Introduction to Variant Controls”

Port

Number of the input port that is connected to one variant choice upstream of the Variant Sink block

A read-only field, based on the input port that is connected to one variant choice upstream of the Variant Sink block.

Click  to add a port or  to delete an existing one.

Variant Control

Displays the variant controls available in the base workspace. The variant control can be a Boolean condition expression or a `Simulink.Variant` object representing a Boolean condition expression. If you want to generate code for your model, you must define the control variables as `Simulink.Parameter` objects.

Default: `Variant`

To enter a variant name, double-click a **Variant control** cell in a new row and type in the variant control expression. Click **Apply** after you edit a variant control name. If you add or delete a variant control without applying the changes, the previous edits on the variant control name are lost.

Parameter: `VariantControls`

Type: cell array of character vectors

Value: Variant control that is associated with the variant choice.

Default: ''

Condition (read-only)

Displays the **Condition** for the variant controls that are `Simulink.Variant` objects.

A read-only field, based on the condition for the associated variant control in the base workspace. Create or change a variant condition in the `Simulink.Variant` parameter dialog box or in the base workspace.

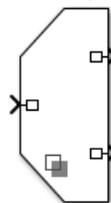
- “Create Variant Controls Programmatically”
- `Simulink.Variant`

Override variant conditions and use the following variant

Ignores the variant control value while determining the active variant and uses the overridden variant control as the active variant. The overriding variant control must be defined in the variant block dialog box.

If you do not select this option, Simulink determines the active variant choice based on the variant control that evaluates to `true` in the global workspace.

When you select this option, the variant badge indicates the change.



Default: Off

On

Use the specified variant instead of selecting the active variant from the variant control.

Off

Determine the active variant choice based on the variant control that evaluates to true.

This parameter enables **Variant**.

Parameter: OverrideUsingVariant

Type: character vector

Value: ' ' if no overriding variant is specified.

Default: ' '

- “Working with Variant Choices”

Variant

Select the variant control that Simulink uses if you select **Override variant conditions and use the following variant**.

Default: ' '

The **Variant** drop-down list displays all variant controls that are currently defined in the base workspace or a data dictionary. Use valid MATLAB identifiers to specify variant controls.

Override variant conditions and use the following variant enables this parameter.

Parameter: OverrideUsingVariant

Type: character vector

Value: Specified by the variant control expression.

- Simulink.Variant

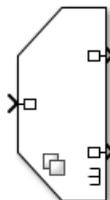
Allow zero active variant controls

To simulate a model (containing a variant block) without an active variant choice, select the **Allow zero active variant controls** option. When this option is selected and there

is no active variant choice, Simulink disables all the blocks connected to the input and output stream of Variant Sink block. The removed blocks are ignored from update diagram or simulation.

If you do not select this option, Simulink generates an error when there is no active variant choice.

When you select this option, the variant badge indicates the change.



Default: Disabled

Parameter: AllowZeroVariantControls

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Show variant condition on block

When you select this option, Simulink annotates each variant control (condition expression) on the Variant Sink block ports.

Default: Disabled

Parameter: ShowConditionOnBlock

Type: character vector

Value: 'off' | 'on'

Default: 'off'

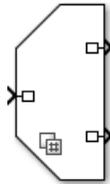
Analyze all choices during update diagram and generate preprocessor conditionals

When you select this option, Simulink analyzes all variant choices during an update diagram or simulation. This analysis helps Simulink to maintain consistency of all

variant branches during simulation and code generation. Simulink routes the output of the active and inactive regions to an internal `VariantMerge` block.

When this option is selected, the preprocessor conditionals (`#if`) are generated in the code with ERT-based targets.

When you select this option, the variant badge indicates the change.



Default: Disabled

- The check box is available for generating ERT targets only.
- **Override variant conditions and use following variant** is cleared ('off')

Parameter: `GeneratePreprocessorConditionals`

Type: character vector

Value: 'off' | 'on'

Default: 'off'

- “Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No

Code Generation	Yes
-----------------	-----

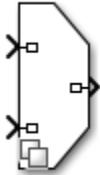
See Also

- “Define, Configure, and Activate Variants”
- “Variant Systems” (Embedded Coder)

Introduced in R2016a

Variant Source

Route amongst multiple inputs using variants



Library

Signal Routing

Description

The Variant Source block has one or more input ports and one output port. You can define variant choices as blocks that are connected to the input port so that, at most, one choice is active.

Each input port is associated with a variant control. The variant control that evaluates to `true`, determines which input port is active.

During simulation, Simulink connects the active choice directly to the output port of the Variant Source block and ignores the inactive choices.

Parameters

Ports and associated conditions

A table of port choices, variant controls, and conditions. The **Variant control** that evaluates to `true` determines which input port must be active.

Default: The table has a row for each variant choice feeding into the Variant Source block. See each column parameter for its default value:

- “Port” on page 1-1766
- “Variant Control” on page 1-1766
- “Condition (read-only)” on page 1-1767

- “Define, Configure, and Activate Variants”
- “Introduction to Variant Controls”

Port

Number of the input port that is connected to one variant choice upstream of the Variant Source block

A read-only field, based on the input port that is connected to one variant choice upstream of the Variant Source block.

Click  to add a port or  to delete an existing one.

Variant Control

Displays the variant controls available in the base workspace. The variant control can be a Boolean condition expression or a `Simulink.Variant` object representing a Boolean condition expression. If you want to generate code for your model, you must define the control variables as `Simulink.Parameter` objects.

Default: `Variant`

To enter a variant name, double-click a **Variant control** cell in a new row and type in the variant control expression. Click **Apply** after you edit a variant control name. If you add or delete a variant control without applying the changes, the previous edits on the variant control name are lost.

Parameter: `VariantControls`

Type: cell array of character vectors

Value: Variant control that is associated with the variant choice.

Default: ''

Condition (read-only)

Displays the **Condition** for the variant controls that are `Simulink.Variant` objects.

A read-only field, based on the condition for the associated variant control in the base workspace. Create or change a variant condition in the `Simulink.Variant` parameter dialog box or in the base workspace.

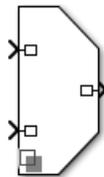
- “Create Variant Controls Programmatically”
- `Simulink.Variant`

Override variant conditions and use the following variant

Ignores the variant control value while determining the active variant and uses the overridden variant control as the active variant. The overriding variant control must be defined in the variant block dialog box.

If you do not select this option, Simulink determines the active variant choice based on the variant control that evaluates to `true` in the global workspace.

When you select this option, the variant badge indicates the change.



Default: Off

On

Use the specified variant instead of selecting the active variant from the variant control.

Off

Determine the active variant choice based on the variant control that evaluates to true.

This parameter enables **Variant**.

Parameter: OverrideUsingVariant

Type: character vector

Value: ' ' if no overriding variant is specified.

Default: ' '

- “Working with Variant Choices”

Variant

Select the variant control that Simulink uses if you select **Override variant conditions and use the following variant**.

Default: ' '

The **Variant** drop-down list displays all variant controls that are currently defined in the base workspace or a data dictionary. Use valid MATLAB identifiers to specify variant controls.

Override variant conditions and use the following variant enables this parameter.

Parameter: OverrideUsingVariant

Type: character vector

Value: Specified by the variant control expression.

- Simulink.Variant

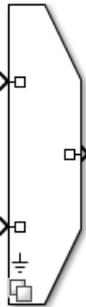
Allow zero active variant controls

To simulate a model (containing a variant block) without an active variant choice, select the **Allow zero active variant controls** option. When this option is selected and there

is no active variant choice, Simulink disables all the blocks connected to the input and output stream of Variant Source block. The removed blocks are ignored from update diagram or simulation.

If you do not select this option, Simulink generates an error when there is no active variant choice.

When you select this option, the variant badge indicates the change.



Default: Disabled

Parameter: AllowZeroVariantControls

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Show variant condition on block

When you select this option, Simulink annotates each variant control (condition expression) on the Variant Source block ports.

Default: Disabled

Parameter: ShowConditionOnBlock

Type: character vector

Value: 'off' | 'on'

Default: 'off'

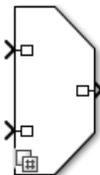
Analyze all choices during update diagram and generate preprocessor conditionals

When you select this option, Simulink analyzes all variant choices during an update diagram or simulation. This analysis helps Simulink to maintain consistency of all variant branches during simulation and code generation. Simulink routes the output of the active and inactive regions to an internal `VariantMerge` block.

When this option is selected, the preprocessor conditionals (`#if`) are generated in the code with ERT-based targets.

If this option is selected during code generation, the data type and the semantics at all input ports of the Variant Source block must be same to avoid failure.

When you select this option, the variant badge indicates the change.



Default: Disabled

- The check box is available for generating ERT targets only.
- **Override variant conditions and use following variant** is cleared ('off')

Parameter: `GeneratePreprocessorConditionals`

Type: character vector

Value: 'off' | 'on'

Default: 'off'

- “Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
------------	---------------------------------------------------------------------------

Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

See Also

- “Define, Configure, and Activate Variants”
- “Variant Systems” (Embedded Coder)

Introduced in R2016a

Variant Subsystem Variant Model

Template subsystem containing Subsystem blocks as variant choices

Library: Simulink / Ports & Subsystems



Description

The Variant Subsystem and Variant Subsystem blocks each can have one active choice for simulation. The Variant Subsystem block is a template preconfigured to contain two Subsystem blocks to use as variant systems. The Variant Model block is a template preconfigured to contain two Model blocks to use as variant systems. The Variant Subsystem and Variant Model blocks (variant system container blocks) can contain a mixture of Subsystem and Model blocks as variant systems. The variant system container blocks can also include Inport, Outport, and Connection Port blocks. There are no drawn connections inside the variant system container blocks.

Each variant system is associated with a variant control that is created in the base workspace. The variant control determines which variant system is active. The variant control can be a condition expression, a `Simulink.Variant` object specifying a condition expression, or a default variant. The variant control that evaluates to `true` determines the active variant.

When you select the **Specify output when source is unconnected** option in the Outport block that is in a variant subsystem container block, you can specify a nonground value as its output. Specify the data type in the **Signal Attributes** section of the Outport block dialog box.

Note If you need to use variant systems that are conditionally executed models (models with control ports), then use a Model block as a container for variant models. Otherwise, use a Subsystem Variant or Model Variant block to contain variant systems. Support for using a Model block to contain model variants will be removed in a future release.

For new models, use a Model block for model variants only if you need to use variants that are conditionally executed models (referenced models with control ports). Using a

Model block for variant models is supported for backward compatibility. However, support for using a Model block to contain model variants will be removed in a future release. For an example of a model that uses a Variant Subsystem block as a container for variant models, see “Model Reference Variants”.

Ports

During simulation, Simulink disables the inactive ports in a variant subsystem container block.

Input

Input_Port_1 — Input port corresponding to root-level Inport blocks contained in variant subsystem

same data types accepted by Inport blocks

Each Subsystem or Model block contained within a variant subsystem represents one variant system. If the inport names on a variant system are a subset of the inport names used by the variant subsystem container block, then variant system blocks can have different numbers of inports than the variant subsystem container block has.

Output

Output_Port_1 — Output port corresponding to root-level Outport blocks contained in variant subsystem

same data types accepted by Outport blocks

Each Subsystem or Model block contained within a variant subsystem represents one variant system. If the outport names on a variant system are a subset of the outport names used by the variant subsystem container block, then variant system blocks can have different numbers of outports than the variant subsystem container block has.

Parameters

Variant choices (table of variant systems) — Table of variant choices, variant controls, and conditions

empty table (default)

The table has a row for each variant system contained in the variant subsystem. If there are no variant systems, the table is empty.

You can use buttons to the left of the **Variant choices** table to modify the elements in the table.

To...	Click...
Create and add a new subsystem choice: Place a new subsystem variant choice in the table and create a Subsystem block in the variant subsystem container block diagram.	
Create and add a new model variant choice: Place a new model variant choice in the table and create a Model block in the variant subsystem container block.	
Create/Edit selected variant object: Create a Simulink.Variant object in the base workspace and open the Simulink.Variant object parameter dialog box to specify the variant Condition .	
Open selected variant choice block: Open the subsystem block diagram for the selected row in the Variant choices table.	
Refresh dialog information from Variant Subsystem contents: Update the Variant choices table according to the variant system and values of the variant control in the base workspace.	

Name (read-only) — Variant system name

' ' (default) | name of Subsystem or Model block contained in the variant subsystem

This read-only field is based on the variant system name. To add a subsystem variant choice, click . To add a model variant choice, click .

Variant Control — Variant control in base workspace

Variant (default) | boolean condition expression | a Simulink.Variant object representing a boolean condition expression | a Simulink.Parameter object (required for code generation)

To enter a variant name, double-click a **Variant control** cell in a new row and type in the variant control expression.

Programmatic Use

Structure field: Represented by the read-only `variant.Name` field in the `Variant` parameter structure

Type: character vector

Value: Variant control that is associated with the variant choice

Default: ''

Condition (read-only) — Condition for variant controls

'' (default)

This read-only field is based on the condition for the associated variant control in the base workspace. Create or change a variant condition in the `Simulink.Variant` parameter dialog box or in the base workspace.

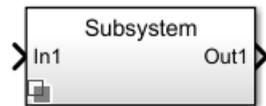
Override variant conditions and use the following variant — Override variant conditions and use specified variant

`off` (default) | `on`

To choose the active variant based on the evaluation of the variant conditions, use the default (`off`). To choose the active variant based on the value specified in the **Variant** parameter, select the **Override variant conditions and use following variant** parameter.

For **Override variant conditions and use following variant** option, the variant control need not be a Boolean condition expression or a `Simulink.Variant` object. Variant controls that start with a % symbol are ignored.

When you select this option, the variant badge indicates the change.

**Dependencies**

This parameter enables the **Variant** parameter.

Programmatic Use

Parameter: `OverrideUsingVariant`

Type: character vector

Value: '' if no overriding variant is specified

Default: ''

variant — Name of variant to use if overriding variant conditions

'' (default) | character vector

If you select **Override variant conditions and use following variant**, specify the name of the variant to use. The variant name must be valid MATLAB identifier.

For **Override variant conditions and use following variant** option, the variant control need not be a boolean condition expression, or a `Simulink.Variant` object. Variant controls that start with a % symbol are ignored.

Tip You can use the **Variant** drop-down list to see a list of variant controls that are specified in the variant choice section.

Dependencies

To enable this parameter, select the **Override variant conditions and use the following variant** parameter.

Programmatic Use

Parameter: `OverrideUsingVariant`

Type: character vector

Value: character vector specifying the variant control expression

Default: ''

Allow zero active variant controls — Simulate model without using active variant

off (default) | on

To simulate a model (containing a variant system) without an active variant choice, select the **Allow zero active variant controls** option. When you select this option and there is no active variant choice, Simulink disables all the blocks connected to the input and output stream of variant subsystem container block. The disabled blocks are ignored from update diagram or simulation.

If you do not select this option, Simulink generates an error when there is no active variant choice.

Dependencies

- No (default) option of variant is selected
- **Override variant conditions and use following variant** is cleared ('off')

Programmatic Use**Parameter:** AllowZeroVariantControls**Type:** character vector**Value:** 'off' 'on'**Default:** 'off'**Analyze all choices during update diagram and generate preprocessor conditionals — Generate preprocessor conditionals**

off (default) | on

When generating code for an ERT target, this parameter determines whether variant choices are enclosed within C preprocessor conditional statements (`#if`).

When you select this option, Simulink analyzes all variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness of all variant choices.

When you select this option, the variant badge indicates the change.

**Dependencies**

- The check box is available for generating only ERT targets.
- **Override variant conditions and use following variant** is cleared ('off').

Programmatic Use**Parameter:** GeneratePreprocessorConditionals**Type:** character vector**Value:** 'off' 'on'**Default:** 'off'**Propagate conditions outside of Variant Subsystem — Propagate variant conditions outside of variant subsystem container block**

off (default) | on

When you select this option, Simulink propagates the variant conditions outside of the variant subsystem container block to determine which components of the model are active during simulation.

When you select this option, the variant badge indicates the change.



Programmatic Use

Parameter: PropagateVariantConditions

Type: character vector

Value: 'off' 'on'

Default: 'off'

Model Examples

Block Characteristics

Data Types	double ^a single ^a Boolean ^a base integer ^a fixed point ^a enumerated ^a bus ^a
Multidimensional Signals	Yes ^a
Variable-Size Signals	Yes ^a

a. Actual data type or capability support depends on block implementation.

Extended Capabilities

C/C++ Code Generation

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

Actual data type or capability support depends on block implementation.

HDL Code Generation

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

Fixed-Point Conversion

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

Actual data type or capability support depends on block implementation.

See Also

[Inport](#) | [Output](#) | [Simulink.Variant](#)

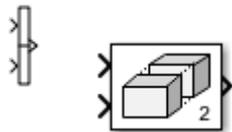
Topics

- “Variant Systems” (Embedded Coder)
- “Define, Configure, and Activate Variants”
- “Define, Configure, and Activate Variants”
- “Working with Variant Choices”
- “Introduction to Variant Controls”
- “Create Variant Controls Programmatically”

Introduced in R2010b

Vector Concatenate, Matrix Concatenate

Concatenate input signals of same data type to create contiguous output signal



Library

Math Operations, Signal Routing

Description

The Concatenate block concatenates the signals at its inputs to create an output signal whose elements reside in contiguous locations in memory.

Tip The Concatenate block is useful for creating an output signal that is nonvirtual. However, to create a vector of function calls, use a Mux block instead.

You use a Concatenate block to define an array of buses. For details about defining an array of buses, see “Combine Buses into an Array of Buses”.

The Concatenate block operates in either vector or multidimensional array concatenation mode, depending on the setting of its **Mode** parameter. In either case, the block concatenates the inputs from the top to bottom, or left to right, input ports.

Vector Mode

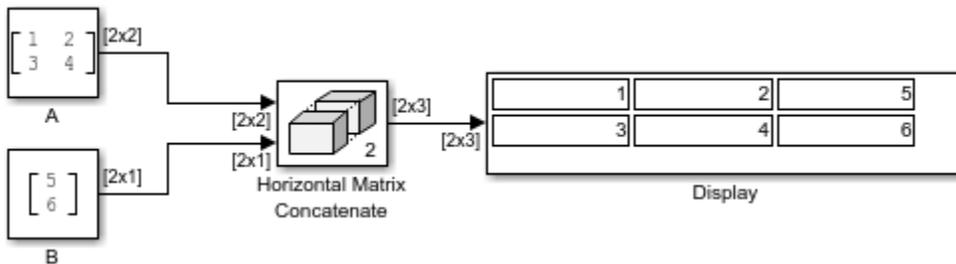
In vector mode, all input signals must be either vectors or row vectors [1xM matrices] or column vectors [Mx1 matrices] or a combination of vectors and either row or column vectors. The output is a vector if all inputs are vectors.

The output is a row or column vector if any of the inputs are row or column vectors, respectively.

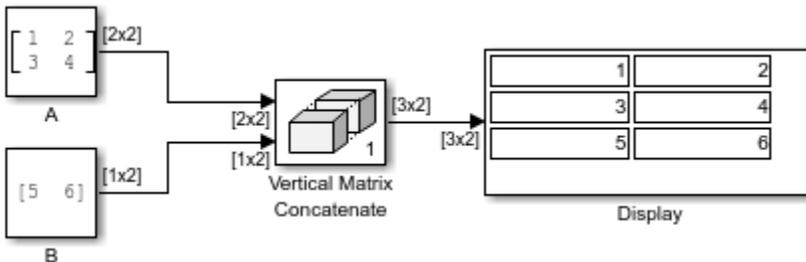
Multidimensional Array Mode

Multidimensional array mode accepts vectors and arrays of any size. It assumes that the trailing dimensions are all ones for input signals with lower dimensionality. For example, if the output is 4-D and the input is $[2 \times 3]$ (2-D) this block treats the input as $[2 \times 3 \times 1 \times 1]$. The output is always an array. The block's **Concatenate dimension** parameter allows you to specify the output dimension along which the block concatenates its input arrays.

If you set the **Concatenate dimension** parameter to 2 and inputs are 2-D matrices, the block performs horizontal matrix concatenation and places the input matrices side-by-side to create the output matrix, for example:



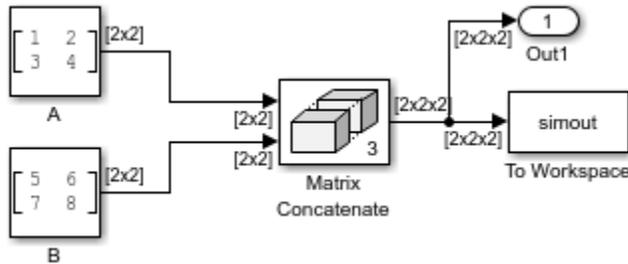
If you set the **Concatenate dimension** parameter to 1 and inputs are 2-D matrices, the block performs vertical matrix concatenation and stacks the input matrices on top of each other to create the output matrix, for example:



For horizontal concatenation, the input matrices must have the same column dimension. For vertical concatenation, the input matrices must have the same row dimension. All

input signals must have the same dimension for all dimensions other than the concatenation dimensions.

If you set the **Mode** parameter to `Multidimensional array`, the **Concatenate dimension** parameter to 3, and the inputs are 2-D matrices, the block performs multidimensional matrix concatenation, for example:



Data Type Support

Accepts signals of any data type that Simulink supports, including fixed-point, enumerated, and nonvirtual bus data types. All inputs must be of the same data type. Outputs have the same data type as the input.

For more information, see “Data Types Supported by Simulink”.

Parameters

Number of inputs

Specifies the number of inputs for the block.

Default: 2

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Mode

Select the type of concatenation that this block performs.

Default: `Vector` (for the Vector Concatenate block), `Multidimensional array` (for the Matrix Concatenate block)

`Vector`

Perform vector concatenation (see “Vector Mode” on page 1-1780 for details).

`Multidimensional array`

Perform matrix concatenation (see “Multidimensional Array Mode” on page 1-1781 for details).

This parameter enables **Concatenate dimension**.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Concatenate dimension

Specifies the output dimension along which to concatenate the input arrays.

Default: 2

- Enter 1 to concatenate input arrays vertically.
- Enter 2 to concatenate input arrays horizontally.
- Enter a higher dimension to perform multidimensional concatenation on the inputs.

Selecting `Multidimensional array` for **Mode** enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

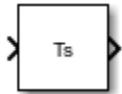
See Also

cat

Introduced in R2009b

Weighted Sample Time

Support calculations involving sample time



Library

Signal Attributes

Description

The Weighted Sample Time block is an implementation of the Weighted Sample Time Math block. See [Weighted Sample Time Math](#) for more information.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Multidimensional Signals	No
Variable-Size Signals	No
Code Generation	Yes

Introduced before R2006a

Waveform Generator

Output waveforms using signal notations



Library

Sources

Description

The Waveform Generator block outputs waveforms based on signal notations that you enter in the **Waveform Definition** table.

This block supports these syntaxes for the signal notations:

- Function syntax — Specify all arguments in the specific order for the signal syntax (see “Supported Waveforms” on page 1-1787).
- Name-value syntax — Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`. For more information, see “Supported Waveforms” on page 1-1787.

This block supports normal, accelerator, and rapid accelerator modes and fast restart.

Supported Operators

Operation	Operator
Absolute value	<code>abs()</code>
Addition	<code>+</code>
Division	<code>/</code>

Operation	Operator
Multiplication	*
Parentheses	()
Subtraction	-
Unary minus	-

The Waveform block observes the following rules of operator precedence:

- 1 ()
- 2 + - (unary)
- 3 * /
- 4 + -

Supported Waveforms

Enter signal notations in the **Waveform Definition** table, one waveform definition per line. To add a waveform definition, click **Add**. The new waveform appears as an empty character vector. The block interprets empty character vectors or white space character vectors as ground.

To remove a waveform definition, click **Remove**. You can select multiple waveforms using **Ctrl+click** or **Shift+click**.

- “Constant” on page 1-1787
- “Gaussian Noise” on page 1-1788
- “Pulse” on page 1-1789
- “Sawtooth” on page 1-1790
- “Sine Wave” on page 1-1791
- “Square” on page 1-1793
- “Step” on page 1-1794

Constant

Constant values can be:

- Numbers
- Workspace variables
 - Scalar, real variables only
- Built-in MATLAB constant, pi

- 1
- 1.1
- x
- pi

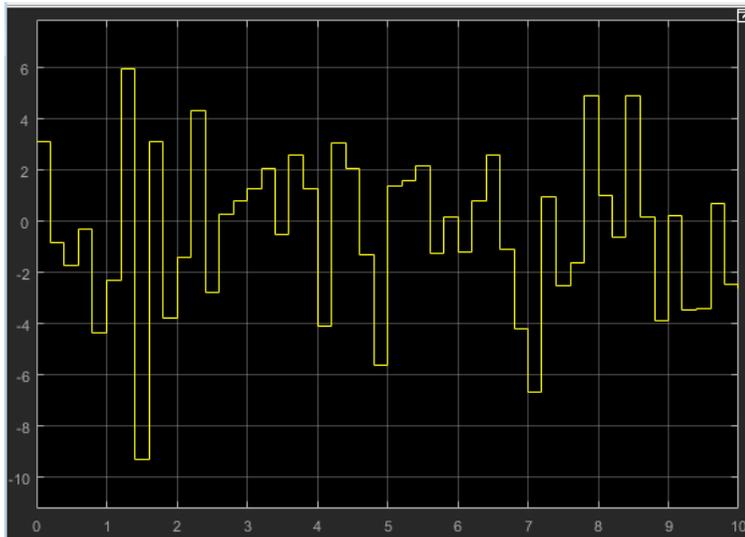
Gaussian Noise

```
gaussian(mean,variance,seed)
```

```
gaussian('Mean',mean,'Variance',variance,'Seed',seed)
```

- mean — Mean value of the random variable output.
 - Default: 0
- variance — Standard deviation squared of the random variable output.
 - Default: 1
 - Value: Positive constant or positive real scalar variable
- seed — Initial seed value for the random number generator.
 - Default: 0
 - Value: Constant or real scalar variable

```
gaussian('Mean',0,'Variance',10,'Seed',1)
```



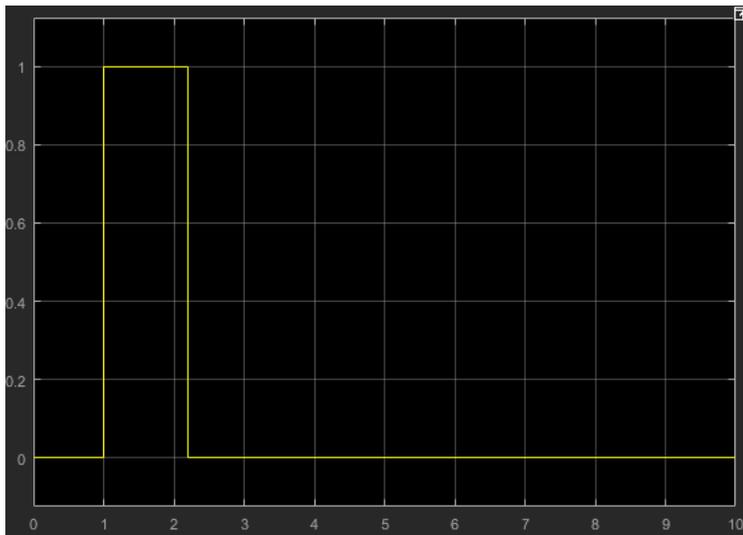
Pulse

```
pulse(amplitude,trigger_time,duration)
```

```
pulse('Amplitude',amplitude,'TriggerTime',trigger_time,'Duration',duration)
```

- `amplitude` — Value of signal when pulse is high.
 - Default: 1
- `trigger_time` — Elapsed simulation time when signal changes to amplitude, in seconds.
 - Default: 1
 - Value: Constant or real scalar variable
- `duration` — How long the signal remains at the given amplitude before returning to ground, in seconds.
 - Default: 1
 - Value: Positive constant or positive real scalar variable

```
pulse('Amplitude',1,'TriggerTime',1,'Duration',1)
```



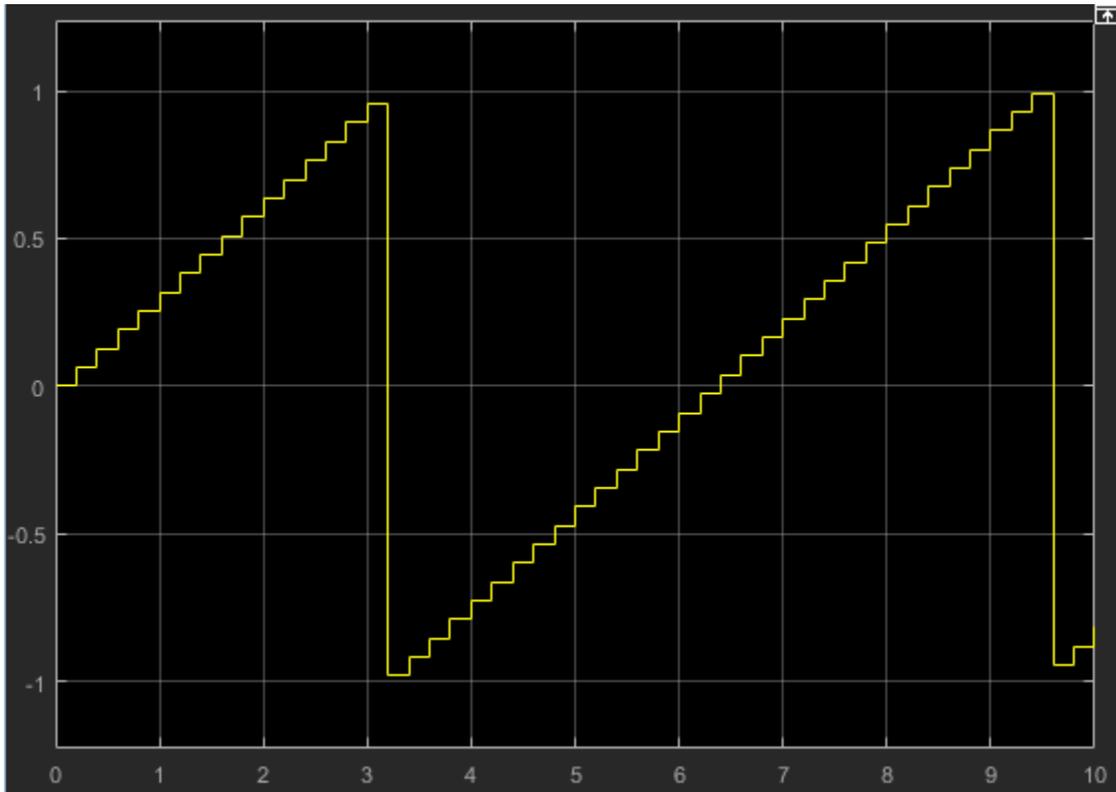
Sawtooth

```
sawtooth(amplitude,frequency,phase_offset)
```

```
sawtooth('Amplitude',amplitude,'Frequency',frequency,'Phase',phase_offset)
```

- `amplitude` — Sawtooth peak value.
 - Default: 1
- `frequency` — Waveform frequency, in rad/s.
 - Default: 1
- `phase_offset` — Horizontal signal shift, based on elapsed simulation time, in seconds.
 - Default: 0

```
sawtooth('Amplitude',1,'Frequency',1,'Phase',0)
```



Sine Wave

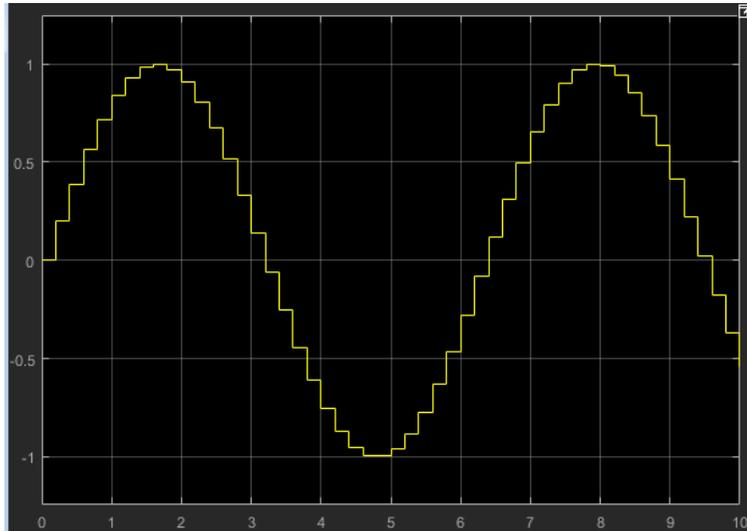
```
sin(amplitude,frequency,phase_offset)
```

```
sin('Amplitude',amplitude,'Frequency',frequency,'Phase',phase_offset)
```

- `amplitude` — Amplitude of sine wave.
 - Default: 1
- `frequency` — Waveform frequency, in rad/s.
 - Default: 1

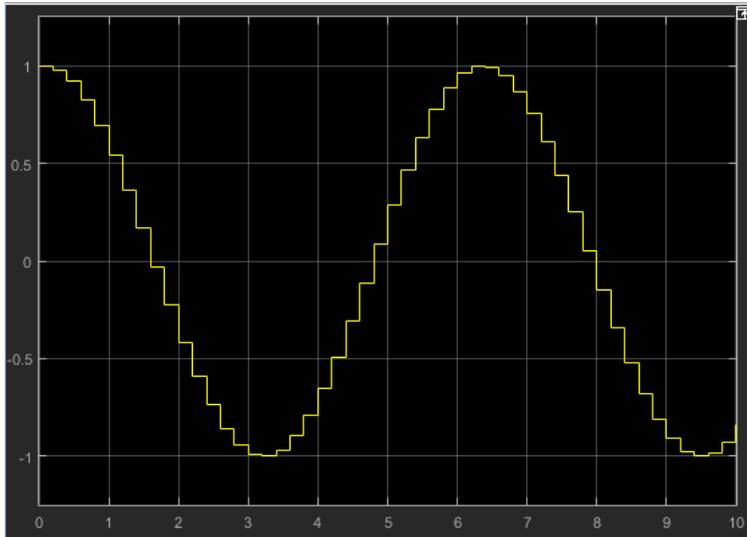
- `phase_offset` — Phase offset, in rads.
 - Default: 0

```
sin('Amplitude',1,'Frequency',1,'Phase',0)
```



To get the cosine waveform:

```
sin('Amplitude',1,'Frequency',1,'Phase',pi/2)
```



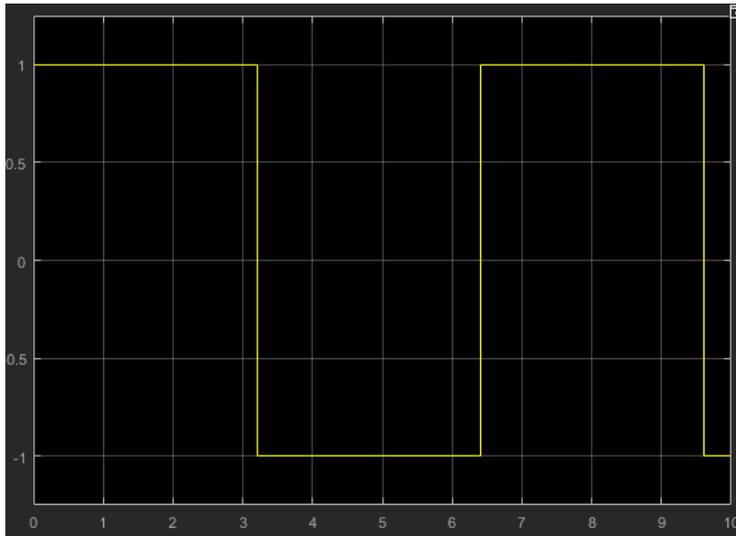
Square

```
square(amplitude,frequency,phase_delay,duty_cycle)
```

```
square('Amplitude',amplitude,'Frequency',frequency,'Phase',phase_delay,...  
'DutyCycle',duty_cycle)
```

- `amplitude` — Amplitude of signal.
 - Default: 1
- `frequency` — Waveform frequency in rad/s.
 - Default: 1
- `phase_delay` — Horizontal signal shift based on elapsed simulation time, in seconds.
 - Default: 0
- `duty_cycle` — Percentage of high signal per period (0–100%). The block clips the minimum signal to 0% and the maximum signal to 100%.
 - Default: 50

```
square('Amplitude',1,'Frequency',1,'Phase',0,'DutyCycle',50)
```



Step

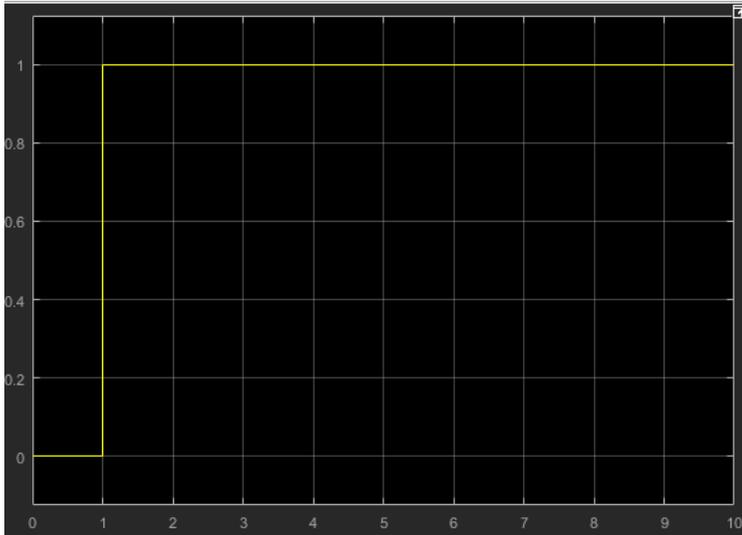
```
step(step_time,initial_value,final_value)
```

```
step('StepTime',step_time,'InitialValue',initial_value,'FinalValue',final_value)
```

- `step_time` — Elapsed simulation time when signal changes from `initial_value` to `final_value`, in seconds.
 - Default: 1
 - Value: Constant or positive real scalar variable.
- `initial_value` — Value of signal when elapsed simulation time is less than `step_time`, in seconds.
 - Default: 0
- `final_value` — Value of signal when elapsed simulation time is greater than or equal to `step_time`, in seconds.

- Default: 1

```
step('StepTime',1,'InitialValue',0,'FinalValue',1)
```

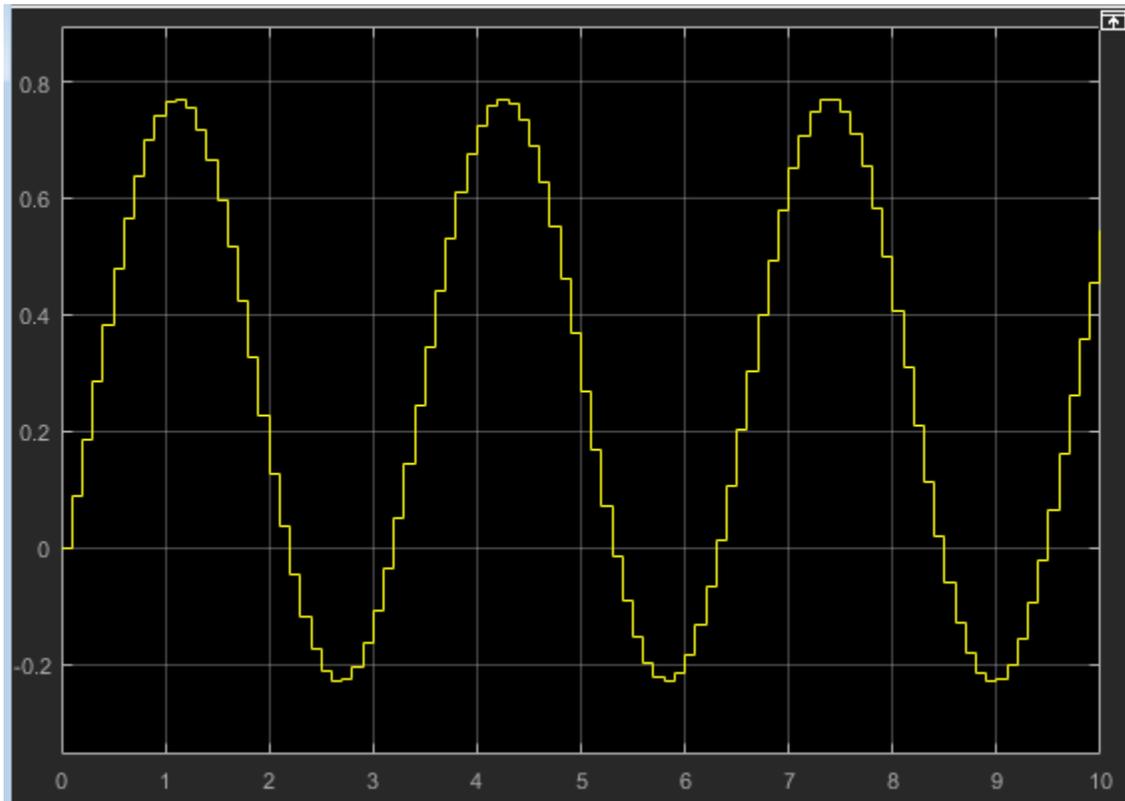


Supported Operations

The Waveform Generator block outputs one signal at a time. You can change this output signal. Express frequency and phase offset parameters in radians. You can also:

- Nest signal notations, for example:

```
sin('Amplitude',sin('Amplitude',1,'Frequency',1,'Phase',0),'Frequency',1,'Phase',1)
```



- Reference real scalar variables in the base or model workspace, for example:

```
sin('Amplitude',x,'Frequency',y,'Phase',z)
```

x , y , and z exist in the base workspace.

For more information on waveforms, see “Supported Waveforms” on page 1-1787.

To quickly determine the response of a system to different types of inputs, you can vary the output signal of the Waveform Generator block while a simulation is in progress.

Data Type Support

The Waveform Generator block outputs scalar values. The default data type is double. However, the block also supports a large subset of Simulink data types, including fixed point (see “Output data type” on page 1-695).

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Output Signal

Select the signal. The number corresponds to the line item in the **Waveform Definition** table. You can change this parameter while a simulation is running.

Default: 1

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Waveform Definition

Enter signal notations in the **Waveform Definition** table, one waveform definition per line. For more information on waveform definitions and how to enter them, see “Supported Waveforms” on page 1-1787.

Default: none

Output minimum

Lower value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMin
Type: character vector
Value: ' [] '
Default: ' [] '

Output maximum

Upper value of the output range that Simulink checks.

Default: [] (unspecified)

Specify this number as a finite, real, double, scalar value.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values”.

Note Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

Parameter: OutMax

Type: character vector

Value: ' [] '

Default: ' [] '

Output data type

Specify the output data type.

Default: Inherit: Inherit via back propagation

Inherit: Inherit via back propagation

Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. For example, if the block multiplies an input of type `int8` by a gain of `int16` and ASIC/FPGA is specified as the targeted hardware type, the output data type is `sfix24`. If `Unspecified` (assume 32-bit Generic), i.e., a generic 32-bit microprocessor, is specified as the target hardware, the output data type is `int32`. If none of the word lengths provided by the target microprocessor can accommodate the output range, Simulink software displays an error in the Diagnostic Viewer.

It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn't meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
- To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

Inherit: Inherit via back propagation

Use data type of the driving block.

double

Output data type is double.

single

Output data type is single.

int8

Output data type is int8.

uint8

Output data type is uint8.

int16

Output data type is int16.

uint16

Output data type is uint16.

int32

Output data type is int32.

uint32

Output data type is uint32.

boolean

Output data type is boolean.

fixdt(1,16)

Output data type is fixed point fixdt(1,16).

fixdt(1,16,0)

Output data type is fixed point fixdt(1,16,0).

fixdt(1,16,2^0,0)

Output data type is fixed point fixdt(1,16,2^0,0).

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Control Signal Data Types” for more information.

Data type override

Specify data type override mode for this signal.

Default: `double`

`Inherit`

Inherits the data type override setting from its context, that is, from the block, `Simulink.Signal` object or Stateflow chart in Simulink that is using the signal.

`Off`

Ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

This parameter appears only when the **Mode** is `Built in` or `Fixed point`.

Signedness

Specify whether you want the fixed-point data as signed or unsigned.

Default: `Signed`

`Signed`

Specify the fixed-point data as signed.

`Unsigned`

Specify the fixed-point data as unsigned.

Selecting **Mode** > `Fixed point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Word length

Specify the bit size of the word that holds the quantized integer.

Default: 16

Minimum: 0

Maximum: 32

Selecting **Mode** > Fixed point enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors.

Default: Binary point

Binary point

Specify binary point location.

Slope and bias

Enter slope and bias.

Selecting **Mode** > Fixed point enables this parameter.

Selecting Binary point enables:

- **Fraction length**

Selecting Slope and bias enables:

- **Slope**

- **Bias**

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specifying a Fixed-Point Data Type”.

Fraction length

Specify fraction length for fixed-point data type.

Default: 0

Binary points can be positive or negative integers.

Selecting **Scaling** > `Binary point` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Slope

Specify slope for the fixed-point data type.

Default: 2^0

Specify any positive real number.

Selecting **Scaling** > `Slope and bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Bias

Specify bias for the fixed-point data type.

Default: 0

Specify any real number.

Selecting **Scaling** > `Slope` and `bias` enables this parameter.

For more information, see “Specifying a Fixed-Point Data Type”.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Default: Off

On

Locks the output data type setting for this block.

Off

Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Parameter: `LockScale`

Type: character vector

Value: 'off' | 'on'

Default: 'off'

For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Default: `Floor`

Ceiling

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

Floor

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

Nearest

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Parameter: `RndMeth`

Type: character vector

Value: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`

Default: `'Floor'`

For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Default: Off

On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Parameter: SaturateOnIntegerOverflow

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Sample time

Enter the time interval between sample time hits or specify -1 to inherit the sample time.

Default: 0.1

This value cannot be 0 (continuous) or inf (constant). For more information, see “Sample Time”.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

Mode

Select the category of data to specify.

Default: `Inherit`

`Inherit`

Inheritance rules for data types. Selecting `Inherit` enables `Inherit` via back propagation.

`Built in`

Built-in data types. Selecting `Built in` enables a second menu/text box to the right. Select one of the following choices:

- `double (default)`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`

`Fixed point`

Fixed-point data types.

`Enumerated`

Enumerated data types. Selecting `Enumerated` enables a second menu/text box to the right, where you can enter the class name.

Expression

Expressions that evaluate to data types. Selecting `Expression` enables a second menu/text box to the right, where you can enter the expression.

Clicking the **Show data type assistant** button enables this parameter.

See “Block-Specific Parameters” on page 6-128 for the command-line information.

See “Specify Data Types Using Data Type Assistant”.

>> (Show data type assistant)

Displays the **Data Type Assistant**, to help you to set the **Output data type** parameter.

Characteristics

Data Types	See “Data Type Support” on page 1-1797
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Introduced in R2015b

Weighted Sample Time Math

Support calculations involving sample time



Library

Math Operations

Description

The Weighted Sample Time Math block adds, subtracts, multiplies, or divides its input signal, u , by a weighted sample time, T_s . If the input signal is continuous, T_s is the sample time of the Simulink model. Otherwise, T_s is the sample time of the discrete input signal. If the input signal is constant, Simulink assigns a finite sample time to the block based on its connectivity and context.

You specify the math operation with the **Operation** parameter. Additionally, you can specify to use only the weight with either the sample time or its inverse.

Enter the weighting factor in the **Weight value** parameter. If the weight, w , is 1, that value does not appear in the equation on the block icon.

The block computes its output using the precedence rules for MATLAB operators (see “Operator Precedence” (MATLAB) in the MATLAB documentation). For example, if the **Operation** parameter specifies $+$, the block calculates output using this equation:

$$u + (T_s * w)$$

However, if the **Operation** parameter specifies $/$, the block calculates output using this equation:

$$(u / T_s) / w$$

Data Type Support

The Weighted Sample Time Math block accepts signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Operation

Specify operation to use: +, -, *, /, Ts Only, or 1/Ts Only.

Weight value

Enter the weight of sample time.

Implement using

Select one of two modes: online calculations or offline scaling adjustment. This parameter is visible only when you set **Operation** to * or /.

Result of (Ts * w)	Output Data Type of Two Modes	Block Execution
A power of 2, or an integer value	The same, when Output data type is Inherit: Inherit via internal rule	Equally efficient in both modes
Not power of 2 and not an integer value	Different	More efficient for the offline scaling mode

Note When the **Implement using** parameter is not visible, operations default to online calculations.

Tip The **Saturate on integer overflow** parameter is visible only if:

- The **Operation** parameter specifies + or -.
- The **Operation** parameter specifies * or / and the **Implement using** parameter specifies Online Calculations.

Output data type

Specify whether the block inherits the output data type by an internal rule or back propagation.

Tip If you enter an expression in the edit field, the expression must evaluate to one of the two inherit rules.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer) in the Fixed-Point Designer documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Do not select this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</p>	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code> , which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code> , is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	For all math operations except T_s and $1/T_s$
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

While Iterator

Repeat execution of a subsystem while a logical condition signal is true

Library: Ports & Subsystems

```

> cond while {
> IC     ...
>       }

```

Description

When placed in a subsystem block, a While Iterator block controls execution of the subsystem. During a simulation time step, this block repeats execution of the subsystem while an input logical (Boolean) condition signal to the block is true (1).

Placing a While Iterator block in a Subsystem block makes it an atomic subsystem.

Ports

Input

cond — Logical condition signal

scalar

Signal with the result from evaluating a logical condition. Because the subsystem is not externally triggered during a time step, evaluating a condition as true (1) or false (0) must reside within the subsystem.

The data type and values of the signal can be:

- Logical (Boolean) — true (1) or false (0) .
- Numerical — true (any positive or negative number) or false (0).

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

ic — Initial logical condition

scalar

Signal with the initial logical condition. At the beginning of each time step:

- If IC is false (0), the subsystem does not execute during the time step.
- If IC is true (value not equal to 0), the subsystem starts executing and continues to repeat execution as long as the `cond` signal is true.

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

Output Arguments

Iteration number — Signal with the number of iterations

scalar

Signal with the number of While Iterator Subsystem block executions during each time step.

Data Types: `double` | `int8` | `int16` | `int32`

Parameters

Maximum number of iterations — Specify maximum number of iterations

-1 (default) | integer

Specify maximum number of iterations allowed during a time step.

-1

Any number of iterations as long as the `cond` signal is true (value not equal to 0) . If you specify -1 and the `cond` signal never becomes false (0), the simulation runs in an infinite loop. In this case, the only way to stop the simulation is to terminate MATLAB.

integer

Maximum number of iterations during a time step.

Programmatic Use**Block parameter:** MaxIters**Type:** character vector**Value:** '5' | '-1' | '<integer>'**Default:** '5'**while loop type — Select type of block**`while (default) | do-while`

Select type of block.

`while`

The While Iterator block has two inputs, a `cond` (logical condition) input and an `IC` (initial logical condition) input. The source of the `IC` signal must be external to the While Iterator Subsystem block.

At the beginning of each time step:

- If the `IC` input is true (value not equal to 0), the blocks in the subsystem repeat execution while the `cond` input is true. This process continues during a time step as long as the `cond` input is true and the number of iterations is less than or equal to the **Maximum number of iterations**.
- If the `IC` input is false, the While Iterator block does not execute the contents of the subsystem.

`do-while`The While Iterator block has one input, the `cond` (while condition) input.

At each time step, the blocks in the subsystem repeat execution while the `cond` input is true (value not equal to 0). This process continues as long as the `cond` input is true and the number of iterations is less than or equal to the **Maximum number of iterations**.

Programmatic Use**Block parameter:** WhileBlockType**Type:** character vector**Value:** 'while' | 'do-while'**Default:** 'while'**States when starting — Select block states between time steps**`held (default) | reset`

Select how to handle block states between time steps.

held

Hold block states between time steps. Block state values persist across time steps.

reset

Reset block states to their initial values at the beginning of each time step (in other words, before executing the first iteration loop in the current time step).

Programmatic Use

Block parameter: ResetStates

Type: character vector

Value: 'held' | 'reset'

Default: 'held'

Show iteration number port — Control display of output port

clear | select

Control display of output port for signal with number of block executions. The value of the signal from this port starts at 1 and is incremented by 1 for each succeeding iteration.

off

Remove output port.

on

Display output port for signal with iteration number.

Dependency

Selecting this parameter enables the **Output data type** parameter.

Programmatic Use

Block parameter: ShowIterationPort

Type: character vector

Value: 'off' | 'on'

Default: 'off'

Output data type — Select output data type for current iteration number

int32 (default) | int16 | int8 | double

Select output data type for iteration number signal. The value of this signal is the number of iterations during a time step and the total number of iterations at the end of a time step.

int32

Signed 32-bit integer.

int16

Signed 16-bit integer.

int8

Signed 8-bit integer.

double

Double-precision floating point.

Dependency

Select the **Show iteration number port** check box to enable this parameter.

Programmatic Use

Block parameter: OutputDataType

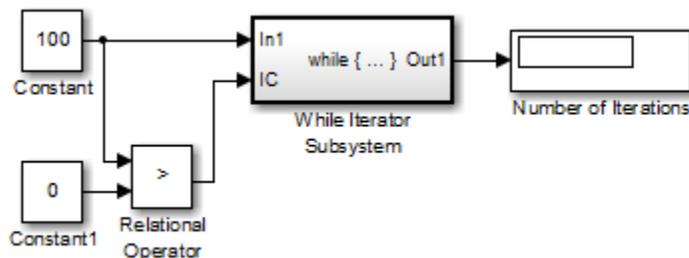
Type: character vector

Value: 'int32' | 'int16' | 'int8' | 'double'

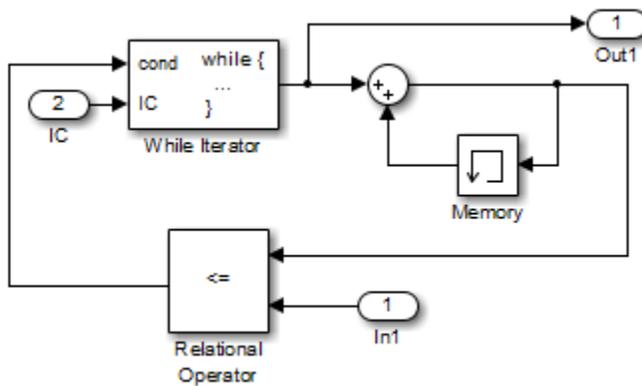
Default: 'int32'

Examples

The While Iterator block can output the current iteration number, starting at 1. This model computes N, where N is the first N integers whose sum is less than 100.



The While Iterator subsystem includes these blocks.



The While Iterator block uses these parameter settings:

- **Maximum number of iterations** set to 20
- **States when starting** set to reset

The model is the diagrammatic equivalent of this pseudocode.

```
max_sum = 100;
sum = 0;
iteration_number = 0;
cond = (max_sum > 0);
while (cond != 0) {
    iteration_number = iteration_number + 1;
    sum = sum + iteration_number;
    if (sum > max_sum OR iteration_number > max_iterations)
        cond = 0;
}
```

See Also

Blocks

[For Iterator](#) | [For Iterator Subsystem](#) | [Subsystem](#) | [While Iterator Subsystem](#)

Topics

[“Use Control Flow Logic”](#)

Introduced before R2006a

While Iterator Subsystem

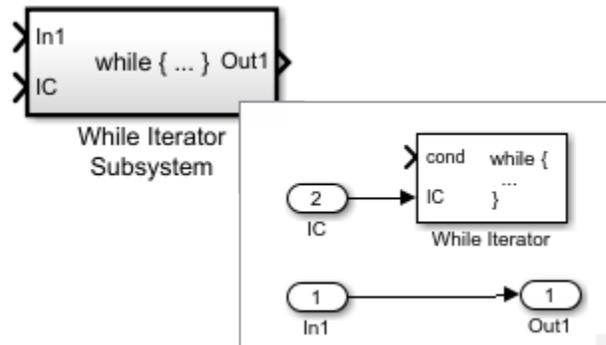
Subsystem that repeats execution during a simulation time step

Library: Ports & Subsystems



Description

The While Iterator Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that repeats execution during a simulation time step while a logical (Boolean) condition is true.



Use While Iterator Subsystem blocks to model:

- Block diagram equivalent of a program while or do-while loop.
- An iterative algorithm that converges on a more accurate solution after multiple iterations.

When using simplified initialization mode, if you place a block that needs elapsed time (such as a Discrete-Time Integrator block) in a While Iterator Subsystem block, Simulink displays an error.

If the output signal from a While Iterator Subsystem block is a function-call signal, Simulink displays an error when you simulate the model or update the diagram.

Ports

Input

In — Signal input to a subsystem block

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

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

IC (initial logical condition) — Control initial execution of a subsystem block

scalar

Placing a While Iterator block connected to an Input block in a Subsystem block adds this external input port to the block.

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

Output

Out — Signal output from a subsystem

scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

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

See Also

Blocks

For Iterator | For Iterator Subsystem | Subsystem | While Iterator

Topics

“Use Control Flow Logic”

Introduced before R2006a

Width

Output width of input vector



Library

Signal Attributes

Description

The Width block generates as output the width of its input vector.

You can use an array of buses as an input signal to a Width block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Data Type Support

The Width block accepts real or complex signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean
- Enumerated

The Width block also supports mixed-type signal vectors.

When the **Output data type mode** is not `Choose intrinsic data type`, the block supports only built-in numeric types. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Note The Width block ignores the **Data type override** setting of the Fixed-Point Tool.

Output data type mode

Specify the output data type to be the same as the input, or inherit the data type by back propagation. You can also choose to specify a built-in data type from the drop-down list in the **Output data type** parameter.

Output data type

This parameter is visible when you select `Choose intrinsic data type` for **Output data type mode**. Select a built-in data type from the drop-down list.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Constant
Multidimensional Signals	Yes
Variable-Size Signals	Yes
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

Wrap To Zero

Set output to zero if input is above threshold



Library

Discontinuities

Description

The Wrap To Zero block sets the output to zero when the input is above the **Threshold** value. However, the block outputs the input when the input is less than or equal to the **Threshold**.

Data Type Support

The Wrap To Zero block accepts inputs of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The block output has the same data type as the input. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Tip If the input data type cannot represent zero, parameter overflow occurs. To detect this overflow, go to the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box and set **Parameters > Detect overflow** to warning or error.

Parameters

Threshold

When the input exceeds the threshold, the block sets the output to zero.

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Direct Feedthrough	Yes
Multidimensional Signals	Yes
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

Introduced before R2006a

XY Graph

Display X-Y plot of signals using MATLAB figure window



Library

Sinks

Description

The XY Graph block displays an X-Y plot of its inputs in a MATLAB figure window.

The block has two scalar inputs. The block plots data in the first input (the x direction) against data in the second input (the y direction). (See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.) This block is useful for examining limit cycles and other two-state data. Data outside the specified range does not appear.

A figure window appears for each XY Graph block in the model at the start of simulation.

Data Type Support

The XY Graph block accepts real signals of the following data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Note The XY Graph block does not support stepping back in a simulation.

Parameters

x-min

Specify the minimum x -axis value. The default is -1 .

x-max

Specify the maximum x -axis value. The default is 1 .

y-min

Specify the minimum y -axis value. The default is -1 .

y-max

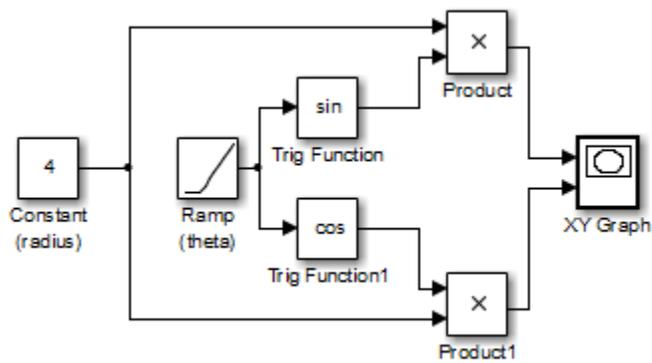
Specify the maximum y -axis value. The default is 1 .

Sample time

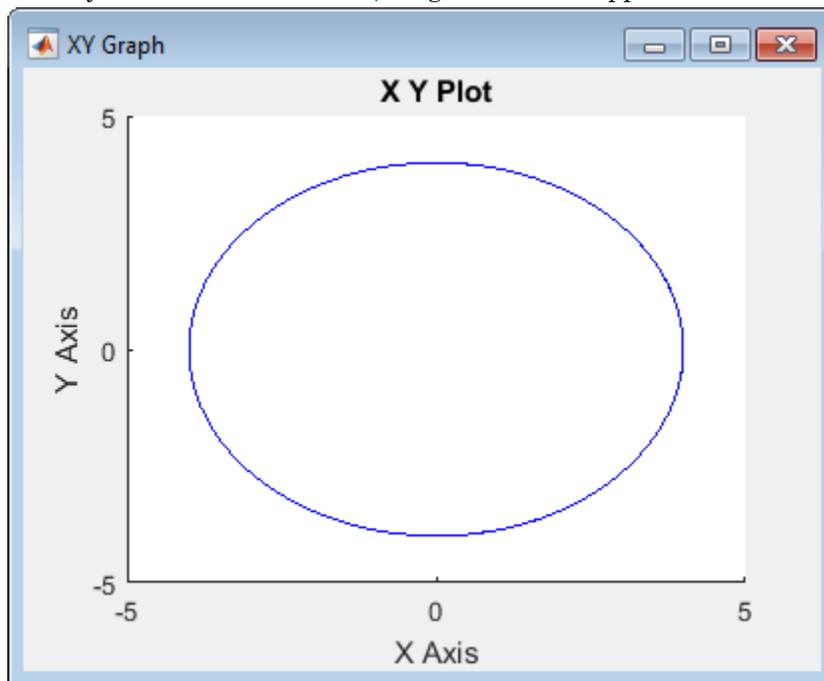
Specify the time interval between samples. To inherit the sample time, set this parameter to -1 . See “Specify Sample Time” in the Simulink documentation for more information.

Examples

The following model computes the points that define a circle of radius 4, centered at the origin of the x - y plane.



When you simulate the model, a figure window appears.



Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point
Sample Time	Specified in the Sample time parameter
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	No

Introduced before R2006a

Zero-Order Hold

Implement zero-order hold of one sample period



Library

Discrete

Description

The Zero-Order Hold block holds its input for the sample period you specify. The block accepts one input and generates one output. Each signal can be scalar or vector. If the input is a vector, the block holds all elements of the vector for the same sample period.

You specify the time between samples with the **Sample time** parameter. A setting of -1 means the block inherits the **Sample time**.

Tip Do not use the Zero-Order Hold block to create a fast-to-slow transition between blocks operating at different sample rates. Instead, use the Rate Transition block.

Comparison with Similar Blocks

Blocks with Similar Functionality

The Unit Delay, Memory, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different. The sections that follow highlight some of these differences.

Recommended Usage for Each Block

Block	Purpose of the Block	Reference Examples
Unit Delay	Implement a delay using a discrete sample time that you specify. The block accepts and outputs signals with a discrete sample time.	<ul style="list-style-type: none"> sldemo_enginewc (Compression subsystem)
Memory on page 1-915	Implement a delay by one major integration time step. Ideally, the block accepts continuous (or fixed in minor time step) signals and outputs a signal that is fixed in minor time step.	<ul style="list-style-type: none"> sldemo_bounce sldemo_clutch (Friction Mode Logic/Lockup FSM subsystem)
Zero-Order Hold	Convert an input signal with a continuous sample time to an output signal with a discrete sample time.	<ul style="list-style-type: none"> sldemo_radar_em1 aero_dap3dof

Overview of Block Capabilities

Capability	Block		
	Unit Delay	Memory	Zero-Order Hold
Specification of initial condition	Yes	Yes	No, because the block output at time $t = 0$ must match the input value.
Specification of sample time	Yes	No, because the block can only inherit sample time (from the driving block or the solver used for the entire model).	Yes
Support for frame-based signals	Yes	No	Yes
Support for state logging	Yes	No	No

Data Type Support

The Zero-Order Hold block accepts real or complex signals of any data type that Simulink supports, including fixed-point and enumerated data types.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Sample time (-1 for inherited)

Specify the time interval between samples. To inherit the sample time, set this parameter to -1 . See “Specify Sample Time” in the online documentation for more information.

Do not specify a continuous sample time, either 0 or $[0, 0]$. This block supports only discrete sample times. When this parameter is -1 , the inherited sample time must be discrete and not continuous.

Bus Support

The Zero-Order Hold block is a bus-capable block. The input can be a virtual or nonvirtual bus signal. No block-specific restrictions exist. All signals in a nonvirtual bus input to a Zero-Order Hold block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “Specify Bus Signal Sample Times” and “Bus-Capable Blocks” for more information.

You can use an array of buses as an input signal to a Zero-Order Hold block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Examples

The following models show how to use the Zero-Order Hold block:

- `sldemo_radar_eml`
- `aero_dap3dof`

Characteristics

Data Types	Double Single Boolean Base Integer Fixed-Point Enumerated Bus
Sample Time	Specified in the Sample time parameter
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

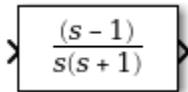
See Also

Memory, Unit Delay

Introduced before R2006a

Zero-Pole

Model system by zero-pole-gain transfer function



Library

Continuous

Description

The Zero-Pole block models a system that you define with the zeros, poles, and gain of a Laplace-domain transfer function. This block can model single-input single output (SISO) and single-input multiple-output (SIMO) systems.

Conditions for Using This Block

The Zero-Pole block assumes the following conditions:

- The transfer function has the form

$$H(s) = K \frac{Z(s)}{P(s)} = K \frac{(s - Z(1))(s - Z(2)) \dots (s - Z(m))}{(s - P(1))(s - P(2)) \dots (s - P(n))},$$

where Z represents the zeros, P the poles, and K the gain of the transfer function.

- The number of poles must be greater than or equal to the number of zeros.
- If the poles and zeros are complex, they must be complex-conjugate pairs.
- For a multiple-output system, all transfer functions must have the same poles. The zeros can differ in value, but the number of zeros for each transfer function must be the same.

Note You cannot use a Zero-Pole block to model a multiple-output system when the transfer functions have a differing number of zeros or a single zero each. Use multiple Zero-Pole blocks to model such systems.

Modeling a Single-Output System

For a single-output system, the input and the output of the block are scalar time-domain signals. To model this system:

- 1 Enter a vector for the zeros of the transfer function in the **Zeros** field.
- 2 Enter a vector for the poles of the transfer function in the **Poles** field.
- 3 Enter a 1-by-1 vector for the gain of the transfer function in the **Gain** field.

Modeling a Multiple-Output System

For a multiple-output system, the block input is a scalar and the output is a vector, where each element is an output of the system. To model this system:

- 1 Enter a matrix of zeros in the **Zeros** field.

Each *column* of this matrix contains the zeros of a transfer function that relates the system input to one of the outputs.

- 2 Enter a vector for the poles common to all transfer functions of the system in the **Poles** field.
- 3 Enter a vector of gains in the **Gain** field.

Each element is the gain of the corresponding transfer function in **Zeros**.

Each element of the output vector corresponds to a column in **Zeros**.

Transfer Function Display on the Block

The Zero-Pole block displays the transfer function depending on how you specify the zero, pole, and gain parameters.

- If you specify each parameter as an expression or a vector, the block shows the transfer function with the specified zeros, poles, and gain. If you specify a variable in parentheses, the block evaluates the variable.

For example, if you specify **Zeros** as $[3, 2, 1]$, **Poles** as $(poles)$, where $poles$ is $[7, 5, 3, 1]$, and **Gain** as $gain$, the block looks like this:

$$\frac{gain(s-3)(s-2)(s-1)}{(s-7)(s-5)(s-3)(s-1)}$$

- If you specify each parameter as a variable, the block shows the variable name followed by (s) if appropriate.

For example, if you specify **Zeros** as $zeros$, **Poles** as $poles$, and **Gain** as $gain$, the block looks like this:

$$\frac{gain \cdot zeros(s)}{poles(s)}$$

Data Type Support

The Zero-Pole block accepts real signals of type `double`. For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Parameters

Zeros

Define the matrix of zeros.

Default: $[1]$

- For a single-output system, enter a vector for the zeros of the transfer function.
- For a multiple-output system, enter a matrix. Each *column* of this matrix contains the zeros of a transfer function that relates the system input to one of the outputs.

Parameter: `Zeros`

Type: `vector`

Value: '[1]'

Default: '[1]'

Poles

Define the vector of poles.

Default: [0 -1]

- For a single-output system, enter a vector for the poles of the transfer function.
- For a multiple-output system, enter a vector for the poles common to all transfer functions of the system.

Parameter: Poles

Type: vector

Value: '[0 -1]'

Default: '[0 -1]'

Gain

Define the vector of gains.

Default: [1]

- For a single-output system, enter a 1-by-1 vector for the gain of the transfer function.
- For a multiple-output system, enter a vector of gains. Each element is the gain of the corresponding transfer function in **Zeros**.

Parameter: Gain

Type: vector

Value: '[1]'

Default: '[1]'

Absolute tolerance

Specify the absolute tolerance for computing block states.

Default: `auto`

- You can enter `auto`, `-1`, a positive real scalar or vector.
- If you enter `auto` or `-1`, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute block states.
- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

Parameter: `AbsoluteTolerance`

Type: character vector, scalar, or vector

Value: `'auto'` | `'-1'` | any positive real scalar or vector

Default: `'auto'`

State Name (e.g., 'position')

Assign a unique name to each state.

Default: `' '`

If this field is blank, no name assignment occurs.

- To assign a name to a single state, enter the name between quotes, for example, `'velocity'`.
- To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, `{'a', 'b', 'c'}`. Each name must be unique.
- The state names apply only to the selected block.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

Parameter: ContinuousStateAttributes

Type: character vector

Value: ' ' | user-defined

Default: ' '

Characteristics

Data Types	Double
Sample Time	Continuous
Direct Feedthrough	Only if the lengths of the Poles and Zeros parameters are equal
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

See Also

Discrete Zero-Pole

Introduced before R2006a

Functions — Alphabetical List

add_block

Add block to model

Syntax

```
h = add_block(source, dest)
h = add_block(source, dest, 'MakeNameUnique', 'on')
h = add_block(sourceIn, destIn, 'CopyOption', 'duplicate')
h = ( ____, Name, Value)
```

Description

`h = add_block(source, dest)` adds a copy of the block `source` from a library or model to the specified destination model and block name. This syntax creates the block at the same location as it appears in the model or the library model.

If you are copying between models or from a library, load the destination model first.

`h = add_block(source, dest, 'MakeNameUnique', 'on')` ensures that the destination block name is unique in the model. This syntax adds a number to the destination block name if a block with that name exists, incrementing to ensure a unique name.

`h = add_block(sourceIn, destIn, 'CopyOption', 'duplicate')` duplicates an inport block in a subsystem, giving the destination block the same port number as the source block. Duplicate an inport to branch a signal from an input port without creating a port or adding lines. For more information, see “Creating Duplicate Inports” on page 1-767.

`h = (____, Name, Value)` uses optional `Name, Value` arguments.

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

With the `add_block` function, you can use block parameter and value pairs. For a list of all the block parameters, see “Common Block Properties” on page 6-109 and “Block-Specific Parameters” on page 6-128.

Examples

Add Block to Model from a Library

Add the block from the Simulink library to the model `f14`.

Load or open the destination model.

```
open_system('f14');
```

Add the Scope block from the Simulink Sinks library to `f14`, naming the new block `MyScope`.

```
add_block('simulink/Sinks/Scope','f14/MyScope');
```

Add a Block from Another Model

Add a copy of a block from the model `f14` to `vdp`.

Load or open the destination model.

```
open_system('vdp');
```

Add the Actuator Model block from `f14` to `vdp`.

```
add_block('f14/Actuator Model','vdp/Actuator Model');
```

Add a Block Using a Unique Name

Add the block from the Simulink library to the model `vdp`. Because there is already a block named `Scope` in `vdp`, use the `MakeNameUnique` option to ensure that the new block name is unique.

Load or open the destination model.

```
open_system('vdp');
```

Add the Scope block from the Simulink Sinks library to vdp, ensuring that the name is unique.

```
add_block('simulink/Sinks/Scope','vdp/Scope','MakeNameUnique','on')
```

Duplicate an Inport Block in a Subsystem

Add an inport block in the f14/Controller subsystem that uses the same port number as another inport in that subsystem.

Duplicate the Stick Input (in) block in the Controller subsystem, naming the copy Stick Input (in)2. The resulting block uses the same port number as Stick Input (in) but does not add an inport on the parent subsystem. The signal that enters that port branches to both inports.

```
add_block('f14/Controller/Stick Input (in)',...  
'f14/Controller/Stick Input (in)2','CopyOption','duplicate')
```

Add a Block and Set Parameters

Add a block from a library to a model and set parameters using a Name, Value pair.

Load or open the destination model.

```
open_system('vdp');
```

Add a Gain block from the library to vdp, and set the Gain value to 5.

```
add_block('simulink/Math Operations/Gain','vdp/Five','Gain','5')
```

Input Arguments

source — Block to copy to model

block path name | library block path name

Block to copy to model, specified as:

- The full block path name if you are copying the block from another model, for example, 'vdp/Mu'. This usage copies the block and its settings.
- The library block path name if you want to add a block from a library, for example, 'simulink/Math Operations/Gain'.

To get the library block path name, you can hover over the block in the Library Browser. Alternatively, you can open the library model, select the block, and enter `gcb` at the command line. To open the library model, in the Library Browser, right-click the library name in the library list and select **Open *library_name* library**.

You can also use the syntax '`built-in/blocktype`' as the source block path name, where *blocktype* is the programmatic block name—the value of the `BlockType` parameter (see “Common Block Properties” on page 6-109). However, blocks added using '`built-in/blocktype`' sometimes have different default parameter values from library blocks.

For subsystems and masked blocks, use the library block path name. Using the `BlockType` value (`SubSystem`) creates an empty subsystem.

Example: 'vdp/Mu', 'simulink/Sinks/Scope'

dest — Name and location of new block

block path name

Name and location of the new block in the model, specified as the block path name.

Example: 'f14/Controller/MyNewBlock'

sourceIn — Inport block whose port number to copy

block path name

Inport block whose port number to copy, specified as the block path name.

Example: 'f14/Controller/Stick Input (in)', 'myModel/mySubsystem/In1'

destIn — Inport block to create

block path name

Inport block with duplicate port number to create, specified as the block path name. Create the destination block in the same system as the source block.

Example: `'myModel/mySubsystem/DupPortIn'`

Output Arguments

h — New block

handle

New block, returned as a handle.

See Also

`delete_block`

Topics

“Delete an Annotation Programmatically”

“Create Annotations Programmatically”

Introduced before R2006a

add_exec_event_listener

Register listener for block method execution event

Syntax

```
h = add_exec_event_listener(blk,event,listener);
```

Description

`h = add_exec_event_listener(blk,event,listener)` registers a listener for a block method execution event where the listener is a MATLAB program that performs some task, such as logging runtime data for a block, when the event occurs (see “Listen for Method Execution Events”). Simulink software invokes the registered listener whenever the specified event occurs during simulation of the model. You cannot register a listener for virtual blocks.

Note Simulink software can register a listener only while a simulation is running. Invoking this function when no simulation is running results in an error message. To ensure that a listener catches all relevant events triggered by a model's simulation, you should register the listener in the model's `StartFcn` callback function (see “Callbacks for Customized Model Behavior”).

Input Arguments

`blk`

Specifies the block whose method execution event the listener is intended to handle. May be one of the following:

- Full pathname of a block
- A block handle
- A block runtime object (see “Access Block Data During Simulation”).

event

Specifies the type of event for which the listener listens. It may be any of the following:

Event	Occurs...
'PreDerivatives'	Before a block's Derivatives method executes
'PostDerivatives'	After a block's Derivatives method executes
'PreOutputs'	Before a block's Outputs method executes.
'PostOutputs'	After a block's Outputs method executes
'PreUpdate'	Before a block's Update method executes
'PostUpdate'	After a block's Update method executes

listener

Specifies the listener to be registered. It may be either a character vector specifying a MATLAB expression, e.g., 'disp(''here'')' or a handle to a MATLAB function that accepts two arguments. The first argument is the block runtime object of the block that triggered the event. The second argument is an instance of `EventData` class that specifies the runtime object and the name of the event that just occurred.

Output Arguments

`add_exec_event_listener` returns a handle to the listener that it registered. To stop listening for an event, use the MATLAB `clear` command to clear the listener handle from the workspace in which the listener was registered.

Introduced before R2006a

add_line

Add line to Simulink model

Syntax

```
h = add_line(sys,out,in)
h = add_line(sys,out,in,'autorouting',autoOption)
h = add_line(sys,points)
```

Description

`h = add_line(sys,out,in)` adds a line in the model or subsystem `sys` that connects one block's output port `out` to another block's input port `in`. This syntax draws the most direct route from port to port, for example, diagonal lines or lines that go through other blocks.

You can connect ports when:

- The input port does not already have a connection.
- The blocks are compatible for connecting.

`h = add_line(sys,out,in,'autorouting',autoOption)` connects blocks, specifying whether to route the lines around other blocks.

`h = add_line(sys,points)` adds a line drawn by (x,y) coordinate `points` relative to the upper-left corner of the Simulink Editor canvas before any canvas resizing. If either end of the line is within five pixels of a corresponding port, the function connects the line to it. The line can have multiple segments.

Examples

Connect Blocks Using Port Numbers

Use the block port numbers to add a line to connect blocks.

Create a model and open it.

```
open_system(new_system('connect_model'));
```

Add and position a Constant block and a Gain block.

```
add_block('simulink/Commonly Used Blocks/Constant','connect_model/Constant');
set_param('connect_model/Constant','position',[140,80,180,120]);
add_block('simulink/Commonly Used Blocks/Gain','connect_model/Gain');
set_param('connect_model/Gain','position',[220,80,260,120]);
```

Connect the blocks. Each block has one port, so specify port 1.

```
add_line('connect_model','Constant/1','Gain/1');
```

Connect Blocks Using Port Handles

Get the port handles and connect the ports using `add_line`.

Open the model `vdp`.

```
open_system('vdp');
```

Delete the line that connects the Mu gain block to the Sum block.

```
delete_line('vdp','Mu/1','Sum/2');
```

Get the port handles from the Mu block and the Sum block.

```
h = get_param('vdp/Mu','PortHandles');
h1 = get_param('vdp/Sum','PortHandles');
```

Look at the `h1` structure. Notice the two handles for the `Inport` property.

```
h1
```

```
h1 =
```

```
struct with fields:
    Inport: [47.0002 54.0002]
    Outport: 39.0002
    Enable: []
    Trigger: []
```

```

    State: []
    LConn: []
    RConn: []
    Ifaction: []
    Reset: []

```

Index into the Outport and Inport properties on the port handles to get the handles you want and connect them. Connect to the second inport.

```
add_line('vdp',h.Outport(1),h1.Inport(2));
```

Add a Branched Line

You can branch a line by adding a connection programmatically. You can use the `points` syntax to draw the segment, or you can draw the line by specifying the ports to connect. When using the port, use automatic routing to improve the look of the branched line.

Add a scope to the vdp model above the outport.

```
vdp
add_block('simulink/Commonly Used Blocks/Scope','vdp/Scope1');
set_param('vdp/Scope1','position',[470,70,500,110]);
```

Connect the Integrator block x1 to Scope1. This code branches the existing line from the x1 output and connects it to the scope. With autorouting on, the resulting line is segmented.

```
add_line('vdp','x1/1','Scope1/1','autorouting','on')
```

Connect Blocks Using Points

You can use points on the canvas as the start and end of each segment. Get the port locations using `get_param` with the 'PortConnectivity' option.

Open the model vdp and delete the line that connects the Mu and Sum blocks.

```
vdp
delete_line('vdp','Mu/1','Sum/2')
```

Get the port locations for Mu. Mu has two ports. The first is the input port, and the second is the output port.

```
mu = get_param('vdp/Mu', 'PortConnectivity');  
mu.Position
```

```
ans =  
  
    190    150
```

```
ans =  
  
    225    150
```

Get the port locations for Sum, which has three ports. The second position is the lower input port.

```
s = get_param('vdp/Sum', 'PortConnectivity');  
s.Position
```

```
ans =  
  
    250    135
```

```
ans =  
  
    250    150
```

```
ans =  
  
    285    145
```

Connect the ports using the output and input points.

```
add_line('vdp', [225 150; 250 150])
```

Connect Blocks Using Autorouting Options

This example shows the effect of adding lines with and without autorouting options.

Create a model route. Display default block names.

```
open_system(new_system('route'));  
set_param('route', 'HideAutomaticNames', 'off')
```

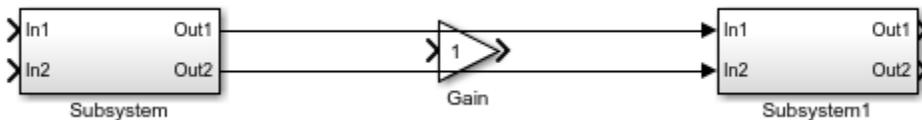
Add blocks as shown. Add an inport and output to each subsystem.



Add lines to connect the outputs from Subsystem to the inputs of Subsystem1.

```
add_line('route', {'Subsystem/1', 'Subsystem/2'}, ...
        {'Subsystem1/1', 'Subsystem1/2'})
```

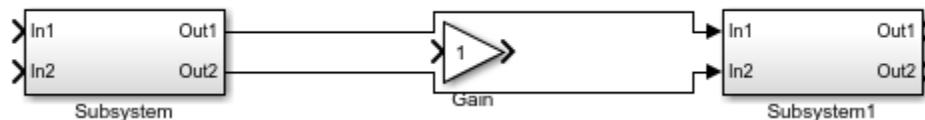
Because you did not use the autorouting options, the function draws straight lines, which pass through the Gain block.



Delete the lines. Add lines again, this time using the autorouting option set to 'on'.

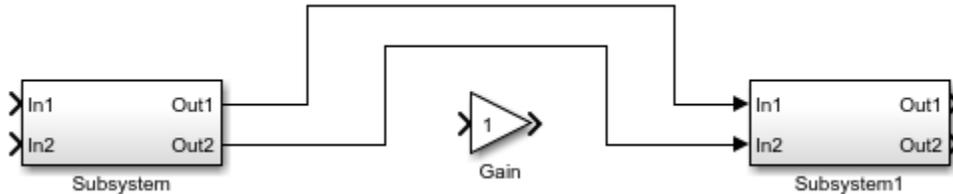
```
add_line('route', {'Subsystem/1', 'Subsystem/2'}, ...
        {'Subsystem1/1', 'Subsystem1/2'}, 'autorouting', 'on')
```

The lines route around the Gain block.



Delete the lines. Add lines again, using the smart autorouting option. When you use an array to connect two sets of inports and outports, 'smart' autorouting routes them together if doing so makes better use of the space.

```
add_line('route',{'Subsystem/1','Subsystem/2'},...
        {'Subsystem1/1','Subsystem1/2'},'autorouting','smart')
```



Input Arguments

sys — Model or subsystem to add line to
character vector

Model or subsystem to add the line to, specified as character vector.

Example: 'vdp', 'f14/Controller'

out — Block output port to connect line from
block/port name or number character vector | port handle | array of port designators

Block output port to connect line from, specified as:

- The block name, a slash, and the port name or number. Most block ports are numbered from top to bottom or from left to right. For a state port, use the port name State instead of a port number.
- The port handle that you want to connect from.
- An array of either of these port designators.

Use 'PortHandles' with `get_param` to get the handles.

Example: 'Mu/1', 'Subsystem/2', h.Outport(1) {'Subsystem/1', 'Subsystem/2'}

in — Block input port to connect line to
block/port name or number character vector | port handle | array of port designators

Block input port to connect line to, specified as:

- The block name, a slash, and the port name or number. The port name on:
 - An enabled subsystem is Enable.
 - A triggered subsystem is Trigger.
 - If Action and Switch Case Action subsystems is Action.
- The port handle that you want to add the line to.
- An array of either of these port designators.

Use the 'PortHandles' option with `get_param` to get handles.

```
Example: 'Mu/1', 'Subsystem/2', h.Inport(1), {'Subsystem/1', 'Subsystem/2'}
```

autoOption — Type of automatic line routing

'off' (default) | 'on' | 'smart'

Type of automatic line routing around other blocks, specified as:

- 'off' for no automatic line routing
- 'on' for automatic line routing
- 'smart' for automatic line routing that takes the best advantage of the blank spaces on the canvas and avoids overlapping other lines and labels

points — Points of the line to draw

matrix

Points of the line to draw, specified as at least a 2-by-2 matrix. Add a row for every segment you want to draw. Specify points as (x,y) coordinates from the upper-left corner of the Editor before any canvas resizing.

```
Example: [100 300; 200 300; 200 300; 200 500]
```

Output Arguments

h — Line

handle

Line created by `add_line`, returned as a handle.

See Also

`add_block` | `delete_block` | `delete_line` | `get_param` | `set_param`

Topics

“Create an Enabled Subsystem”

“Create a Triggered Subsystem”

Introduced before R2006a

add_param

Add parameter to Simulink system

Syntax

```
add_param('sys','parameter1',value1,'parameter2',value2,...)
```

Description

The `add_param` command adds the specified parameters to the specified system and initializes the parameters to the specified values. Case is ignored for parameter names. Value character vectors are case sensitive. The value of the parameter must be a character vector. Once the parameter is added to a system, `set_param` and `get_param` can be used on the new parameters as if they were standard Simulink parameters. Simulink software saves these new parameters with the model file.

Note If you attempt to add a parameter that has the same name as an existing parameter of the system, Simulink software displays an error.

Examples

This command

```
add_param('vdp','DemoName','VanDerPolEquation','EquationOrder','2')
```

adds the parameters `DemoName` and `EquationOrder` with `'VanDerPolEquation'` and `'2'` to the `vdp` system. Afterward, you can use the following command to retrieve the value of the `DemoName` parameter.

```
get_param('vdp','DemoName')
```

See Also

`delete_param` | `get_param` | `set_param`

Introduced before R2006a

addFile

Add file to Simulink Project

Syntax

```
addFile(proj, file)
```

Description

`addFile(proj, file)` adds a file to the project `proj`.

Examples

Add Files to a Project

Open the airframe project and create a project object.

```
sldemo_slproject_airframe  
proj = simulinkproject;
```

Remove a file.

```
removeFile(proj, 'models/AnalogControl.mdl')
```

Add the file back to the project.

```
addFile(proj, 'models/AnalogControl.mdl');
```

Create and save a new model.

```
new_system('mymodel');  
save_system('mymodel');
```

Add the new file to the project and return a project file object.

```
newPrjFile = addFile(proj, 'mymodel.slx');
```

Use the project file object to manipulate the file, for example, adding a label.

```
addLabel(newPrjFile, 'Classification', 'Design')
```

Input Arguments

proj — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

file — Path of file

character vector

Path of the file to add relative to the project root folder, including the file extension, specified as a character vector. The file must be within the root folder.

Example: 'models/myModelName.slx'

See Also

Functions

`addFolderIncludingChildFiles` | `removeFile` | `simulinkproject`

Introduced in R2013a

addFolderIncludingChildFiles

Add folder and child files to Simulink Project

Syntax

```
addFolderIncludingChildFiles(proj, folder)
```

Description

`addFolderIncludingChildFiles(proj, folder)` adds a folder and all child files to the project `proj`.

Examples

Add Folders to a Project

Open the airframe project and create a project object.

```
sldemo_slproject_airframe  
proj = simulinkproject;
```

Create a new folder in the project folder.

```
new_folder_path = fullfile(proj.RootFolder, 'new_folder')  
mkdir(new_folder_path);
```

Create a new folder in the previous folder.

```
new_sub_folder_path = fullfile(new_folder_path, 'new_sub_folder')  
mkdir(new_sub_folder_path);
```

Create a new file in the folder.

```
filepath = fullfile(new_sub_folder_path, 'new_model_in_subfolder.slx')  
new_system('new_model_in_subfolder');  
save_system('new_model_in_subfolder', filepath)
```

Add this new folder and child files to the project.

```
projectFile = addFolderIncludingChildFiles(proj, new_folder_path)
```

Input Arguments

proj — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

folder — Path of folder

character vector

Path of the folder to add relative to the project root folder, specified as a character vector. The folder must be within the root folder.

Example: 'models'

See Also

Functions

`addFile` | `removeFile` | `simulinkproject`

Introduced in R2015b

addterms

Add terminators to unconnected ports in model

Syntax

```
addterms('sys')
```

Description

`addterms('sys')` adds Terminator and Ground blocks to the unconnected ports in the Simulink block diagram `sys`.

See Also

`slupdate`

Introduced before R2006a

attachComponent

Attach a component to a configuration set

Syntax

```
attachComponent(cs, component)
```

Description

`attachComponent(cs, component)` attaches a component to a `Simulink.ConfigSet` object.

Examples

Replace Solver Component for Active Configuration Set

Replace the solver component of the active configuration set of one model with the solver component of another model.

Get the active configuration set for `modelB`.

```
hCs = getActiveConfigSet('modelB');
```

Get the 'Solver' component for this configuration set.

```
hSolverConfig = getComponent(hCs, 'Solver');
```

Create a copy of the component.

```
hSolverConfig = copy(hSolverConfig);
```

Get the active configuration set for `modelA`.

```
hCs = getActiveConfigSet('modelA');
```

Attach the copy of the 'Solver' component from modelB to modelA.

```
attachComponent(hCs, hSolverConfig);
```

Input Arguments

cs — Configuration set object

ConfigSet object

A configuration set object that you can attach a component to.

component — Component object

SimulinkConfigComponent object

A component that you can attach to configuration set.

See Also

Simulink.ConfigSet

Topics

“About Configuration Sets”

“Manage a Configuration Set”

Introduced before R2006a

attachConfigSet

Associate configuration set or configuration reference with model

Syntax

```
attachConfigSet(model, configObj)
```

```
attachConfigSet(model, configObj, allowRename)
```

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

`configObj`

A configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

`allowRename`

Boolean that determines how Simulink software handles a name conflict

Description

`attachConfigSet` associates the configuration set or configuration reference (configuration object) specified by `configObj` with `model`.

You cannot attach a configuration object to a model if the configuration object is already attached to another model, or has the same name as another configuration object attached to the same model. The optional Boolean argument `allowRename` determines how Simulink software handles a name conflict between configuration objects. If `allowRename` is `false` and the configuration object specified by `configObj` has the same name as a configuration object already attached to `model`, Simulink software generates an error. If `allowRename` is `true` and a name conflict occurs, Simulink software provides a unique name for `configObj` before associating `configObj` with `model`.

Examples

The following example creates a copy of the current model's active configuration object and attaches it to the model, changing its name if necessary to be unique. The code is the same whether the object is a configuration set or configuration reference.

```
myConfigObj = getActiveConfigSet(gcs);  
copiedConfig = myConfigObj.copy;  
copiedConfig.Name = 'DevConfig';  
attachConfigSet(gcs, copiedConfig, true);
```

See Also

[attachConfigSetCopy](#) | [closeDialog](#) | [detachConfigSet](#) | [getActiveConfigSet](#)
| [getConfigSet](#) | [getConfigSets](#) | [openDialog](#) | [setActiveConfigSet](#)

Topics

“Manage a Configuration Set”

“Manage a Configuration Reference”

Introduced before R2006a

attachConfigSetCopy

Copy configuration set or configuration reference and associate it with model

Syntax

```
myConfigObj = attachConfigSetCopy(model, configObj)
```

```
myConfigObj = attachConfigSetCopy(model, configObj, allowRename)
```

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

`configObj`

A configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

`allowRename`

Boolean that specifies how Simulink software handles a name conflict

Description

`attachConfigSetCopy` copies the configuration set or configuration reference (configuration object) specified by `configObj` and associates the copy with `model`. Simulink software returns the copied configuration object as `newConfigObj`.

You cannot attach a configuration object to a model if the configuration object has the same name as another configuration object attached to the same model. The optional Boolean argument `allowRename` determines how Simulink software handles a name conflict between configuration objects. If `allowRename` is `false` and the configuration object specified by `configObj` has the same name as a configuration object already attached to `model`, Simulink software generates an error. If `allowRename` is `true` and a name conflict occurs, Simulink software provides a unique name for the copy of `configObj` before associating it with `model`.

Examples

The following example creates a copy of ModelA's active configuration object and attaches it to ModelB, changing the name if necessary to be unique. The code is the same whether the object is a configuration set or configuration reference.

```
myConfigObj = getActiveConfigSet('ModelA');  
newConfigObj = attachConfigSetCopy('ModelB', myConfigObj, true);
```

See Also

[attachConfigSet](#) | [closeDialog](#) | [detachConfigSet](#) | [getActiveConfigSet](#) | [getConfigSet](#) | [getConfigSets](#) | [openDialog](#) | [setActiveConfigSet](#)

Topics

“Manage a Configuration Set”

“Manage a Configuration Reference”

Introduced in R2006b

addLabel

Attach label to Simulink Project file

Syntax

```
addLabel(file, categoryName, labelName)
addLabel(file, categoryName, labelName, labelData)
```

Description

`addLabel(file, categoryName, labelName)` attaches the specified label `labelName` in the category `categoryName` to the file.

`addLabel(file, categoryName, labelName, labelData)` attaches the label with data `labelData`.

Examples

Attach a Label to a Project File

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Get a particular file by name.

```
myfile = findFile(proj, 'models/AnalogControl.mdl')
```

```
myfile =
```

```
ProjectFile with properties:
```

```
Path: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.mdl'
Labels: [1x1 slproject.Label]
```

```
Revision: '2'  
SourceControlStatus: Unmodified
```

Get the Labels property of the file.

```
myfile.Labels  
  
ans =  
  
Label with properties:  
  
File: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.mdl'  
Data: []  
DataType: 'none'  
Name: 'Design'  
CategoryName: 'Classification'
```

Attach the label 'Artifact' to the file.

```
addLabel(myfile, 'Classification', 'Artifact')  
  
ans =  
  
Label with properties:  
  
File: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.mdl'  
Data: []  
DataType: 'none'  
Name: 'Artifact'  
CategoryName: 'Classification'
```

Index into the Labels property to get the label attached to this file.

```
reviewlabel = myfile.Labels(1)  
  
reviewlabel =  
  
Label with properties:  
  
File: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.mdl'  
Data: []  
DataType: 'none'  
Name: 'Artifact'  
CategoryName: 'Classification'
```

Detach the new label from the file.

```
removeLabel(myfile,reviewlabel)
```

Attach a Label to a Subset of Files

Attach the 'Classification' category label 'Utility' to all files in the project that have the .m file extension.

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

Get the list of files.

```
files = proj.Files;
```

Loop through each file. If a file has the extension .m, attach the label 'Utility'.

```
for fileIndex = 1:numel(files)  
    file = files(fileIndex);  
    [~, ~, fileExtension] = fileparts(file.Path);  
    if strcmp(fileExtension, '.m')  
        addLabel(file, 'Classification', 'Utility');  
    end  
end
```

In the Simulink Project **Files** view, the **Classification** column displays the label Utility for each .m file in the utilities folder.

Attach a Label and Label Data to a File

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

Create a new category 'Review'.

```
createCategory(proj, 'Review', 'char');
```

For the new category, create a label 'To Review'.

```
reviewCategory = findCategory(proj, 'Review');
createLabel(reviewCategory, 'To Review');
```

Get a particular file by name.

```
myfile = findFile(proj, 'models/AnalogControl.mdl')
```

```
myfile =
```

```
ProjectFile with properties:
```

```
Path: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.mdl'
Labels: [1x1 slproject.Label]
Revision: '2'
SourceControlStatus: Unmodified
```

Attach the label 'To Review' and a character vector of label data to the file.

```
addLabel(myfile, 'Review', 'To Review', 'Whole team design review')
```

Index into the Labels property to get the second label attached to this particular file, and see the label data.

```
myfile.Labels(2)
```

```
ans =
```

```
Label with properties:
```

```
File: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.mdl'
Data: 'Whole team design review'
DataType: 'char'
Name: 'To Review'
CategoryName: 'Review'
```

In the Simulink Project **Files** view, for the AnalogControl.mdl file, the **Review** column displays the To Review label with label data.

Alternatively, you can set or change label data using the data property.

```
mylabel = myfile.Labels(2);  
mylabel.Data = 'Final review';
```

Input Arguments

file — File to attach label to

file object

File to attach the label to, specified as a file object. You can get the file object by examining the project's `Files` property (`proj.Files`), or use `findFile` to find a file by name. The file must be in the project.

categoryName — Name of category for label

character vector

Name of the category for the label, specified as a character vector.

labelName — Name of label

character vector | label definition object

Name of the label to attach, specified as a character vector or a label definition object returned by the `file.Label` property or `findLabel`. You can specify a new label name that does not already exist in the project.

labelData — Data to attach to label

character vector | numeric

Data to attach to the label, specified as a character vector or numeric. Data type depends on the label definition. Get a label to examine its `DataType` property using `file.Label` or `findLabel`.

See Also

Functions

`createLabel` | `findFile` | `findLabel` | `removeLabel` | `simulinkproject`

Introduced in R2013a

bdclose

Close any or all Simulink system windows unconditionally

Syntax

```
bdclose  
bdclose('sys')  
  
bdclose('all')
```

Description

`bdclose` with no arguments closes the current system window unconditionally and without confirmation. Any changes made to the system since it was last saved are lost.

`bdclose('sys')` closes the specified system window.

`bdclose('all')` closes all system windows.

Examples

This command closes the `vdp` system.

```
bdclose('vdp')
```

See Also

`close_system` | `new_system` | `open_system` | `save_system`

Introduced before R2006a

bdIsDirty

Whether block diagram has unsaved changes

Syntax

```
isDirty = bdIsDirty(bdname)
```

Description

`isDirty = bdIsDirty(bdname)` returns whether or not the loaded block diagram `bdname` has unsaved changes.

Examples

Check Models for Unsaved Changes

Check if models contain unsaved changes using `bdIsDirty`.

Check if a single model is dirty.

```
vdp  
bdIsDirty('vdp')
```

```
ans =
```

```
    logical
```

```
    0
```

Check if a cell array of models are dirty.

```
vdp  
sf_car  
bdIsDirty({'sf_car', 'vdp'})
```

```
ans =
    1x2 logical array
    0     0
```

- “Manage Shadowed and Dirty Model Files”

Input Arguments

bdname — Loaded block diagram name

character vector | cell array of character vectors | double array

Loaded block diagram name, specified as a character vector, a cell array of character vectors, or a double array. All character vectors must be the names of loaded block diagrams. All doubles must be the handles of loaded block diagrams. It is an error to supply an invalid handle, a handle to anything other than a block diagram, a path to a block or subsystem, or a block diagram that is not loaded.

Data Types: `double` | `char` | `cell`

Output Arguments

isDirty — Whether block diagram has unsaved changes

logical scalar | logical array

Whether block diagram has unsaved changes, returned as a logical array with one entry for each block diagram. The logical value is true if the block diagram has been modified in memory since it was loaded or last saved, and false if there are no unsaved changes.

See Also

`bdIsLoaded`

Topics

“Manage Shadowed and Dirty Model Files”

Introduced in R2017a

bdIsLibrary

Whether block diagram is a library

Syntax

```
isLibrary = bdIsLibrary(bdnames)
```

Description

`isLibrary = bdIsLibrary(bdnames)` returns whether the loaded block diagrams specified by `bdnames` are libraries.

Examples

Check Whether Block Diagrams Are Libraries

Load some block diagrams and get a handle to one of them.

```
load_system({'sf_car', 'hydlib', 'vdp'})  
h = get_param('hydlib', 'Handle');
```

Check whether `sf_car` is a library. The returned value 0 indicates that it is not.

```
bdIsLibrary('sf_car')
```

```
ans =  
    0
```

Check whether `hydlib` and `vdp` are libraries. The returned value shows that `hydlib` is a library and `vdp` is not.

```
bdIsLibrary({'hydlib', 'vdp'})
```

```
ans =  
1    0
```

Using the handle to `hdlib`, check whether `hdlib` is a library. The value returned shows that it is.

```
bdIsLibrary(h)
```

```
ans =  
1
```

Input Arguments

bdnames — Names or handles of loaded block diagrams

character vector | cell array of character vectors | double | array of doubles

Names or handles of loaded block diagrams, specified as a character vector, a cell array of character vectors, a double, or a double array. All character vectors are names of loaded block diagrams. All doubles are handles of loaded block diagrams.

Data Types: `char` | `cell` | `double`

Output Arguments

isLibrary — Logical array showing whether block diagrams are libraries

logical scalar | logical array

Logical array showing whether block diagrams are libraries, returned as a logical scalar or array (1 for a library, 0 otherwise).

See Also

`bdIsLoaded` | `bdroot` | `find_system`

Introduced in R2015a

bdIsLoaded

Whether block diagram is in memory

Syntax

```
isLoaded = bdIsLoaded(bdnames)
```

Description

`isLoaded = bdIsLoaded(bdnames)` returns whether or not a block diagram is in memory. `bdnames` can be a character vector or a cell array of character vectors. All character vectors must be valid block diagram names. It is an error to supply a path to a block or subsystem.

`isLoaded` is a logical array with one entry for each block diagram name.

Examples

```
bdIsLoaded('sf_car')
```

returns a logical scalar.

```
bdIsLoaded({'sf_car', 'vdp'})
```

returns a 1-by-2 logical array.

See Also

`bdIsDirty` | `bdIsLibrary` | `find_system`

Introduced in R2008a

bdroot

Return name of top-level Simulink system

Syntax

```
bdroot  
bdroot(obj)  
bdroot(handle)  
bdroot(sys)
```

Description

`bdroot` with no arguments returns the name of the current top-level system.

`bdroot(obj)`, where *obj* is a character vector specifying a system or block path name, returns the name of the top-level system containing the specified object name. The `bdroot` of an empty character vector generates an error. Prior to issuing `bdroot`, make sure that the top-level system is loaded.

`bdroot(handle)`, where *handle* is the numeric handle for a system or block, returns the numeric handle of the top-level system containing the specified object. Prior to issuing `bdroot`, make sure that the top-level system for each element in the cell array is loaded. If you specify a vector of handles, Simulink returns a list of handles of the top-level systems.

`bdroot(sys)`, where *sys* is a cell array of system names or a vector of system handles. Prior to issuing `bdroot`, make sure that the top-level system for each element in the cell array is loaded. If you specify a vector of handles, Simulink returns a list of handles of the top-level systems.

Examples

This command returns the name of the top-level system that contains the current block.

```
bdroot (gcb)
```

This command returns the name of the top-level system that contains the current system.

```
bdroot (gcs)
```

This command returns the numeric handle of the top-level system that contains the current block.

```
bdroot (gcbh)
```

If `gcbh` is a cell array of system names or handles, it returns a cell array containing the corresponding top-level system names or handles.

If `gcbh` is a numeric array of system handles, it returns a numeric array containing the corresponding top-level system handles.

See Also

`find_system` | `gcb` | `gcbh` | `gcs` | `load_system`

Introduced before R2006a

dlinmod

Extract discrete-time linear state-space model around operating point

Syntax

```
argout = dlinmod('sys', Ts)
```

```
argout = dlinmod('sys', Ts, x, u)
```

```
argout = dlinmod('sys', Ts, x, u, para, 'v5')
```

```
argout = dlinmod('sys', Ts, x, u, para, xpert, upert, 'v5')
```

Arguments

<i>sys</i>	Name of the Simulink system from which the linear model is extracted.
<i>x</i> , <i>u</i>	State (<i>x</i>) and the input (<i>u</i>) vectors. If specified, they set the operating point at which the linear model is extracted. When a model has model references using the Model block, you must use the Simulink structure format to specify <i>x</i> . To extract the <i>x</i> structure from the model, use the following command: <pre><i>x</i> = Simulink.BlockDiagram.getInitialState('sys');</pre> <p>You can then change the operating point values within this structure by editing <code><i>x</i>.signals.values</code>.</p> <p>If the state contains different data types (for example, 'double' and 'uint8'), then you cannot use a vector to specify this state. You must use a structure instead. In addition, you can only specify the state as a vector if the state data type is 'double'.</p>
<i>Ts</i>	Sample time of the discrete-time linearized model
'v5'	An optional argument that invokes the perturbation algorithm created prior to MATLAB 5.3. Invoking this optional argument is equivalent to calling <code>linmodv5</code> .

para

A three-element vector of optional arguments:

- *para*(1) — Perturbation value of delta, the value used to perform the perturbation of the states and the inputs of the model. This is valid for linearizations using the 'v5' flag. The default value is 1e-05.
- *para*(2) — Linearization time. For blocks that are functions of time, you can set this parameter with a nonnegative value that gives the time (τ) at which Simulink evaluates the blocks when linearizing a model. The default value is 0.
- *para*(3) — Set *para*(3)=1 to remove extra states associated with blocks that have no path from input to output. The default value is 0.

xpert, upert

The perturbation values used to perform the perturbation of all the states and inputs of the model. The default values are

```
xpert = para(1) + 1e-3*para(1)*abs(x)
upert = para(1) + 1e-3*para(1)*abs(u)
```

When a model has model references using the Model block, you must use the Simulink structure format to specify *xpert*. To extract the *xpert* structure, use the following command:

```
xpert = Simulink.BlockDiagram.getInitialState('sys');
```

You can then change the perturbation values within this structure by editing *xpert.signals.values*.

The perturbation input arguments are only available when invoking the perturbation algorithm created prior to MATLAB 5.3, either by calling `linmodv5` or specifying the 'v5' input argument to `linmod`.

argout

`linmod`, `dlinmod`, and `linmod2` return state-space representations if you specify the output (left-hand) side of the equation as follows:

- `[A, B, C, D] = linmod('sys', x, u)` obtains the linearized model of `sys` around an operating point with the specified state variables `x` and the input `u`. If you omit `x` and `u`, the default values are zero.

`linmod` and `dlinmod` both also return a transfer function and MATLAB data structure representations of the linearized system, depending on how you specify the output (left-hand) side of the equation. Using `linmod` as an example:

- `[num, den] = linmod('sys', x, u)` returns the linearized model in transfer function form.
- `sys_struct = linmod('sys', x, u)` returns a structure that contains the linearized model, including state names, input and output names, and information about the operating point.

Description

`dlinmod` compute a linear state-space model for a discrete-time system by linearizing each block in a model individually.

`linmod` obtains linear models from systems of ordinary differential equations described as Simulink models. Inputs and outputs are denoted in Simulink block diagrams using Inport and Outport blocks.

The default algorithm uses preprogrammed analytic block Jacobians for most blocks which should result in more accurate linearization than numerical perturbation of block inputs and states. A list of blocks that have preprogrammed analytic Jacobians is available in the Simulink Control Design documentation along with a discussion of the block-by-block analytic algorithm for linearization.

The default algorithm also allows for special treatment of problematic blocks such as the Transport Delay and the Quantizer. See the mask dialog of these blocks for more information and options.

Discrete-Time System Linearization

The function `dlinmod` can linearize discrete, multirate, and hybrid continuous and discrete systems at any given sampling time. Use the same calling syntax for `dlinmod` as for `linmod`, but insert the sample time at which to perform the linearization as the second argument. For example,

```
[Ad,Bd,Cd,Dd] = dlinmod('sys', Ts, x, u);
```

produces a discrete state-space model at the sampling time T_s and the operating point given by the state vector x and input vector u . To obtain a continuous model approximation of a discrete system, set T_s to 0.

For systems composed of linear, multirate, discrete, and continuous blocks, `dlinmod` produces linear models having identical frequency and time responses (for constant inputs) at the converted sampling time T_s , provided that

- T_s is an integer multiple of all the sampling times in the system.
- The system is stable.

For systems that do not meet the first condition, in general the linearization is a time-varying system, which cannot be represented with the $[A,B,C,D]$ state-space model that `dlinmod` returns.

Computing the eigenvalues of the linearized matrix A_d provides an indication of the stability of the system. The system is stable if $T_s > 0$ and the eigenvalues are within the unit circle, as determined by this statement:

```
all(abs(eig(Ad))) < 1
```

Likewise, the system is stable if $T_s = 0$ and the eigenvalues are in the left half plane, as determined by this statement:

```
all(real(eig(Ad))) < 0
```

When the system is unstable and the sample time is not an integer multiple of the other sampling times, `dlinmod` produces A_d and B_d matrices, which can be complex. The eigenvalues of the A_d matrix in this case still, however, provide a good indication of stability.

You can use `dlinmod` to convert the sample times of a system to other values or to convert a linear discrete system to a continuous system or vice versa.

You can find the frequency response of a continuous or discrete system by using the `bode` command.

Notes

By default, the system time is set to zero. For systems that are dependent on time, you can set the variable `para` to a two-element vector, where the second element is used to set the value of `t` at which to obtain the linear model.

The ordering of the states from the nonlinear model to the linear model is maintained. For Simulink systems, a character vector variable that contains the block name associated with each state can be obtained using

```
[sizes,x0,xstring] = sys
```

where `xstring` is a vector of strings whose *i*th row is the block name associated with the *i*th state. Inputs and outputs are numbered sequentially on the diagram.

For single-input multi-output systems, you can convert to transfer function form using the routine `ss2tf` or to zero-pole form using `ss2zp`. You can also convert the linearized models to LTI objects using `ss`. This function produces an LTI object in state-space form that can be further converted to transfer function or zero-pole-gain form using `tf` or `zpk`.

The default algorithms in `linmod` and `dlinmod` handle Transport Delay blocks by replacing the linearization of the blocks with a Pade approximation. For the 'v5' algorithm, linearization of a model that contains Derivative or Transport Delay blocks can be troublesome. For more information, see “Linearizing Models”.

See Also

`linmod` | `linmod2` | `linmodv5`

Introduced in R2007a

close_system

Close Simulink system window or block dialog box

Syntax

```
close_system
close_system('sys')
close_system('sys', saveflag)
close_system('sys', 'newname')
close_system('sys', 'newname', 'ErrorIfShadowed', true)
```

Description

`close_system` with no arguments closes the current system or subsystem window. If the current system is the top-level system and it has been modified, `close_system` returns an error. The current system is defined in the description of the `gcs` command.

`close_system('sys')` closes the specified system, subsystem, or block window.

`close_system('sys')` unloads a model after specifying

- `load_system('sys')`.

'*sys*' can be a character vector (which can be a system, a subsystem, or a full block pathname), a cell array of character vectors, a numeric handle, or an array of numeric handles. This command displays an error if '*sys*' is a MATLAB keyword, 'simulink', or more than 63 characters long.

`close_system('sys', saveflag)`, if *saveflag* is 1, saves the specified top-level system to a file with its current name, then closes the specified top-level system window and removes it from memory. If *saveflag* is 0, the system is closed without saving. A single *saveflag* can be supplied, in which case it is applied to all block diagrams.

Alternatively, separate *saveflags* can be supplied for each block diagram, as a numeric array.

`close_system('sys', 'newname')` saves the specified top-level system to a file with the specified new name, then closes the system.

Additional arguments can be supplied when saving a block diagram. These are exactly the same as for `save_system`:

- `ErrorIfShadowed`: true or false (default: false)
- `BreakAllLinks`: true or false (default: false)
- `SaveAsVersion`: MATLAB version name (default: current)
- `OverwriteIfChangedOnDisk`: true or false (default: false)
- `SaveModelWorkspace`: true or false (default: false)

If you try to specify additional options when you are doing something other than saving a block diagram, they are ignored. You see a warning if you try to save when closing something other than a block diagram (e.g., a subsystem or a Block Properties dialog).

Examples

This command closes the current system.

```
close_system
```

This command closes the `vdp` system, unless it has been modified, in which case it returns an error.

```
close_system('vdp')
```

This command saves the engine system with its current name, then closes it.

```
close_system('engine', 1)
```

This command saves the `mymdl12` system under the new name `testsys`, then closes it.

```
close_system('mymdl12', 'testsys')
```

This command tries to save the `vdp` system to a file with the name `'max'`, but returns an error because `'max'` is the name of an existing MATLAB function.

```
close_system('vdp','max','ErrorIfShadowed', true)
```

All three of the following commands save and close mymodel (saved with the same name), and replace links to library blocks with copies of the library blocks in the saved file:

```
close_system('mymodel',1,'BreakAllLinks',true)
close_system('mymodel','mymodel','BreakAllLinks',true)
close_system('mymodel',[],'BreakAllLinks',true)
```

This command closes the dialog box of the Unit Delay block in the Combustion subsystem of the engine system.

```
close_system('engine/Combustion/Unit Delay')
```

Note The `close_system` command cannot be used in a block or menu callback to close the root-level model. Attempting to close the root-level model in a block or menu callback results in an error and discontinues the callback's execution.

See Also

`bdclose` | `gcs` | `load_system` | `new_system` | `open_system` | `save_system`

Introduced before R2006a

closeDialog

Close configuration parameters dialog

Syntax

```
closeDialog(configObj)
```

Arguments

configObj

A configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

Description

`closeDialog` closes an open configuration parameters dialog box. If *configObj* is a configuration set, the function closes the dialog box that displays the configuration set. If *configObj* is a configuration reference, the function closes the dialog box that displays the referenced configuration set, or generates an error if the reference does not specify a valid configuration set. If the dialog box is already closed, the function does nothing.

Examples

The following example closes a configuration parameters dialog box that shows the current parameters for the current model. The parameter values derive from the active configuration set or configuration reference (configuration object). The code is the same in either case; the only difference is which type of configuration object is currently active.

```
myConfigObj = getActiveConfigSet(gcs);  
closeDialog(myConfigObj);
```

See Also

`attachConfigSet` | `attachConfigSetCopy` | `detachConfigSet` |
`getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `openDialog` |
`setActiveConfigSet`

Topics

“Manage a Configuration Set”

“Manage a Configuration Reference”

Introduced in R2006b

close

Close Simulink Project

Syntax

```
close(proj)
```

Description

`close(proj)` closes the project `proj`.

Examples

Open and Close a Simulink Project

Open a specified project and get a project object to manipulate the project at the command line. For example,

```
proj = simulinkproject('C:/projects/project1/myproject.prj')
```

Close the project.

```
close(proj)
```

Input Arguments

proj — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

See Also

Functions

`simulinkproject`

Introduced in R2013a

coder.allowpcode

Package: coder

Control code generation from protected MATLAB files

Syntax

```
coder.allowpcode('plain')
```

Description

`coder.allowpcode('plain')` allows you to generate protected MATLAB code (P-code) that you can then compile into optimized MEX functions or embeddable C/C++ code. This function does not obfuscate the generated MEX functions or embeddable C/C++ code.

With this capability, you can distribute algorithms as protected P-files that provide code generation optimizations, providing intellectual property protection for your source MATLAB code.

Call this function in the top-level function before control-flow statements, such as `if`, `while`, `switch`, and function calls.

MATLAB functions can call P-code. When the `.m` and `.p` versions of a file exist in the same folder, the P-file takes precedence.

`coder.allowpcode` is ignored outside of code generation.

Examples

Generate optimized embeddable code from protected MATLAB code:

- 1 Write an function `p_abs` that returns the absolute value of its input:

```
function out = p_abs(in)    %#codegen
% The directive %#codegen indicates that the function
```

```
% is intended for code generation
coder.allowpcode('plain');
out = abs(in);
```

- 2 Generate protected P-code. At the MATLAB prompt, enter:

```
pcode p_abs
```

The P-file, `p_abs.p`, appears in the current folder.

- 3 Generate a MEX function for `p_abs.p`, using the `-args` option to specify the size, class, and complexity of the input parameter (requires a MATLAB Coder license). At the MATLAB prompt, enter:

```
codegen p_abs -args { int32(0) }
```

`codegen` generates a MEX function in the current folder.

- 4 Generate embeddable C code for `p_abs.p` (requires a MATLAB Coder license). At the MATLAB prompt, enter:

```
codegen p_abs -config:lib -args { int32(0) };
```

`codegen` generates C library code in the `codegen\lib\p_abs` folder.

See Also

`codegen` | `pcode`

Introduced in R2011a

coder.ceval

Call external C/C++ function

Syntax

```
coder.ceval(cfun_name)
coder.ceval(cfun_name,cfun_arguments)
coder.ceval('-global',cfun_name)
coder.ceval('-global',cfun_name,cfun_arguments)
cfun_return = coder.ceval(____)
```

Description

`coder.ceval(cfun_name)` executes the external C/C++ function specified by `cfun_name`. Define `cfun_name` in an external C/C++ source file or library.

`coder.ceval(cfun_name,cfun_arguments)` executes `cfun_name` with arguments `cfun_arguments`. `cfun_arguments` is a comma-separated list of input arguments in the order that `cfun_name` requires.

By default, `coder.ceval` passes arguments by value to the C/C++ function whenever C/C++ supports passing arguments by value. To make `coder.ceval` pass arguments by reference, use the constructs `coder.ref`, `coder.rref`, and `coder.wref`. If C/C++ does not support passing arguments by value, for example, if the argument is an array, `coder.ceval` passes arguments by reference. If you do not use `coder.ref`, `coder.rref` or `coder.wref`, a copy of the argument can appear in the generated code to enforce MATLAB semantics for arrays.

`coder.ceval('-global',cfun_name)` executes `cfun_name` and indicates that `cfun_name` uses one or more MATLAB global variables. The code generator can then produce code that is consistent with this global variable usage.

`coder.ceval('-global',cfun_name,cfun_arguments)` executes `cfun_name` with arguments `cfun_arguments` and indicates that `cfun_name` uses one or more MATLAB global variables.

`cfun_return = coder.ceval(____)` executes `cfun_name` and returns a single scalar value, `cfun_return`, corresponding to the value that the C/C++ function returns in the return statement. To be consistent with C/C++, `coder.ceval` can return only a scalar value. It cannot return an array. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Call External C Function

Call a C function `foo(u)` from a MATLAB function from which you intend to generate C code.

Create a C header file `foo.h` for a function `foo` that takes two input parameters of type `double` and returns a value of type `double`.

```
#ifndef MATLAB_MEX_FILE
#include <tmwtypes.h>
#else
#include "rtwtypes.h"
#endif

double foo(double in1, double in2);
```

Write the C function `foo.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include "foo.h"

double foo(double in1, double in2)
{
    return in1 + in2;
}
```

Write a function `callfoo` that calls `foo` by using `coder.ceval`.

```
function y = callfoo %#codegen
y = 0.0;
if coder.target('MATLAB')
    % Executing in MATLAB, call MATLAB equivalent of
```

```
% C function foo
y = 10 + 20;
else
% Executing in generated code, call C function foo
y = coder.ceval('foo', 10, 20);
end
end
```

Generate C library code for function `callfoo`. Pass `foo.c` and `foo.h` as parameters to include this custom C function in the generated code. `codegen` generates C code in the `codegen\lib\callfoo` subfolder.

```
codegen -config:lib callfoo foo.c foo.h

double callfoo(void)
{
/* Executing in generated code, call C function foo */
return foo(10.0, 20.0);
}
```

In this case, you have not specified the type of the input arguments, that is, the type of the constants 10 and 20. Therefore, the arguments are implicitly of double-precision, floating-point type by default because the default type for constants in MATLAB is `double`.

Call C Library Function

Call a C library function from MATLAB code.

Write a MATLAB function `absval`.

```
function y = absval(u) %#codegen
y = abs(u);
```

Generate the C library for `absval.m` by using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib absval -args {0.0}
```

`codegen` creates the library `absval.lib` and header file `absval.h` in the folder `/codegen/lib/absval`. It generates the functions `absval_initialize` and `absval_terminate` in the same folder.

Write a MATLAB function to call the generated C library functions by using `coder.ceval`.

```
function y = callabsval %#codegen
y = -2.75;
% Check the target. Do not use coder.ceval if callabsval is
% executing in MATLAB
if coder.target('MATLAB')
    % Executing in MATLAB, call function absval
    y = absval(y);
else
    % Executing in the generated code.
    % Call the initialize function before calling the
    % C function for the first time
    coder.ceval('absval_initialize');

    % Call the generated C library function absval
    y = coder.ceval('absval',y);

    % Call the terminate function after
    % calling the C function for the last time
    coder.ceval('absval_terminate');
end
```

Convert the code in `callabsval.m` to a MEX function so you can call the C library function `absval` directly from MATLAB.

```
codegen -config:mex callabsval codegen/lib/absval/absval.lib...
        codegen/lib/absval/absval.h
```

Call the C library by running the MEX function from MATLAB:

```
callabsval_mex
```

Call C Function That Uses Global Variable

Use the `'-global'` flag when you call a C function that modifies a global variable.

Write a MATLAB function `useGlobal` that calls a C function `addGlobal`. Use the `'-global'` flag to indicate to the code generator that the C function uses a global variable.

```
function y = useGlobal()
global g;
```

```
t = g;
% compare execution with/without '-global' flag
coder.ceval('-global', 'addGlobal');
y = t;
end
```

Create a C header file `addGlobal.h` for the function `addGlobal`.

```
void addGlobal(void);
```

Write the C function `addGlobal` in the file `addGlobal.c`. This function includes the header file `useGlobal_data.h` that the code generator creates when you generate code for the function `useGlobal`. This header file contains the global variable declaration for `g`.

```
#include "addGlobal.h"
#include "useGlobal_data.h"
void addGlobal(void) {
    g++;
}
```

Generate the MEX function for `useGlobal`. To define the input to the code generator, declare the global variable in the workspace.

```
global g;
g = 1;
codegen useGlobal -report addGlobal.h addGlobal.c
y = useGlobal_mex();
```

With the `'-global'` flag, the MEX function produces the result `y = 1`. The `'-global'` flag indicates to the code generator that the C function possibly modifies the global variable. For `useGlobal`, the code generator produces this code:

```
real_T useGlobal(const emlrtStack *sp)
{
    real_T y;
    (void)sp;
    y = g;
    addGlobal();
    return y;
}
```

Without the `'-global'` flag, the MEX function produces $y = 2$. Because there is no indication that the C function modifies `g`, the code generator assumes that `y` and `g` are identical. This C code is generated:

```
real_T useGlobal(const emlrtStack *sp)
{
    (void)sp;
    addGlobal();
    return g;
}
```

Input Arguments

cfun_name — C/C++ function name

character vector | string scalar

Name of external C/C++ function to call.

Example: `coder.ceval('foo')`

Data Types: `char` | `string`

cfun_arguments — C/C++ function arguments

scalar variable | array | element of an array | structure | structure field | object property

Comma-separated list of input arguments in the order that `cfun_name` requires.

Example: `coder.ceval('foo', 10, 20);`

Example: `coder.ceval('myFunction', coder.ref(x));`

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

Complex Number Support: Yes

Limitations

- You cannot use `coder.ceval` on functions that you declare extrinsic with `coder.extrinsic`.

- When the LCC compiler creates a library, it adds a leading underscore to the library function names. If the compiler for the library was LCC and your code generation compiler is not LCC, you must add the leading underscore to the function name, for example, `coder.ceval('_mylibfun')`. If the compiler for a library was not LCC, you cannot use LCC to generate code from MATLAB code that calls functions from that library. Those library function names do not have the leading underscore that the LCC compiler requires.
- If a property has a get method, a set method, or validators, or is a System object property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties”.

Tips

- For code generation, before calling `coder.ceval`, you must specify the type, size, and complexity data type of return values and output arguments.
- Use `coder.ceval` only in MATLAB for code generation. `coder.ceval` generates an error in uncompiled MATLAB code. To determine if a MATLAB function is executing in MATLAB, use `coder.target`. If the function is executing in MATLAB, call the MATLAB version of the C/C++ function.

See Also

`coder.extrinsic` | `coder.opaque` | `coder.ref` | `coder.rref` | `coder.target` | `coder.wref`

Topics

“Data Definition Basics”

“Unknown Output Type for `coder.ceval`”

Introduced in R2011a

coder.cinclude

Include header file in generated code

Syntax

```
coder.cinclude(headerfile)
coder.cinclude(headerfile, 'InAllSourceFiles', allfiles)
```

Description

`coder.cinclude(headerfile)` includes a header file in generated C/C++ source code.

MATLAB Coder generates the include statement in the C/C++ source files that are generated from the MATLAB code that contains the `coder.cinclude` call.

In a Simulink model, when a `coder.cinclude` call appears in a MATLAB Function block, the code generator puts the include statement in the model header file.

`coder.cinclude(headerfile, 'InAllSourceFiles', allfiles)` uses the `allfiles` option to determine whether to include the header file in almost all C/C++ source files.

If `allfiles` is `true`, MATLAB Coder generates the include statement in almost all C/C++ source files, except for some utility files. This behavior is the `coder.cinclude` behavior from R2016a and earlier releases. The presence of the include statement in these additional files can increase compile time and make the generated code less readable. Use this option only if your code depends on the legacy behavior. If `allfiles` is `false`, the behavior is the same as the behavior of `coder.cinclude(headerfile)`.

In a MATLAB Function block, `coder.cinclude(headerfile, 'InAllSourceFiles', allfiles)` is the same as `coder.cinclude(headerfile)`.

Examples

Include Header File in C/C++ Code Generated by Using the MATLAB Coder codegen Command

Generate code from a MATLAB function that calls an external C function. Use `coder.cinclude` to include the required header file in the generated C code.

In a writable folder, create a subfolder `mycfiles`.

Write a C function `myMult2.c` that doubles its input. Save it in `mycfiles`.

```
#include "myMult2.h"
double myMult2(double u)
{
    return 2 * u;
}
```

Write the header file `myMult2.h`. Save it in `mycfiles`.

```
#if !defined(MYMULT2)
#define MYMULT2
extern double myMult2(double);
#endif
```

Write a MATLAB function, `myfunc`, that includes `myMult2.h` and calls `myMult2` for code generation only.

```
function y = myfunc
%#codegen
y = 21;
if ~coder.target('MATLAB')
    % Running in generated code
    coder.cinclude('myMult2.h');
    y = coder.ceval('myMult2', y);
else
    % Running in MATLAB
    y = y * 2;
end
end
```

Create a code configuration object for a static library. Specify the locations of `myMult2.h` and `myMult2.c`

```
cfg = coder.config('lib');
cfg.CustomInclude = fullfile(pwd, 'mycfiles');
cfg.CustomSource = fullfile(pwd, 'mycfiles', 'myMult2.c');
```

Generate the code.

```
codegen -config cfg myfunc -report
```

The file `myfunc.c` contains this statement:

```
#include "myMult2.h"
```

The include statement does not appear in any other file.

Include Header File in C/C++ Code Generated from a MATLAB Function Block in a Simulink Model

Generate code from a MATLAB Function block that calls an external C function. Use `coder.cinclude` to include the required header file in the generated C code.

In a writable folder, create a subfolder `mycfiles`.

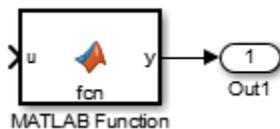
Write a C function `myMult2.c` that doubles its input. Save it in `mycfiles`.

```
#include "myMult2.h"
double myMult2(double u)
{
    return 2 * u;
}
```

Write the header file `myMult2.h`. Save it in `mycfiles`.

```
#if !defined(MYMULT2)
#define MYMULT2
extern double myMult2(double);
#endif
```

Create a Simulink model that contains a MATLAB Function block connected to an Outport block.



In the MATLAB Function block, add the function `myfunc` that includes `myMult2.h` and calls `myMult2`.

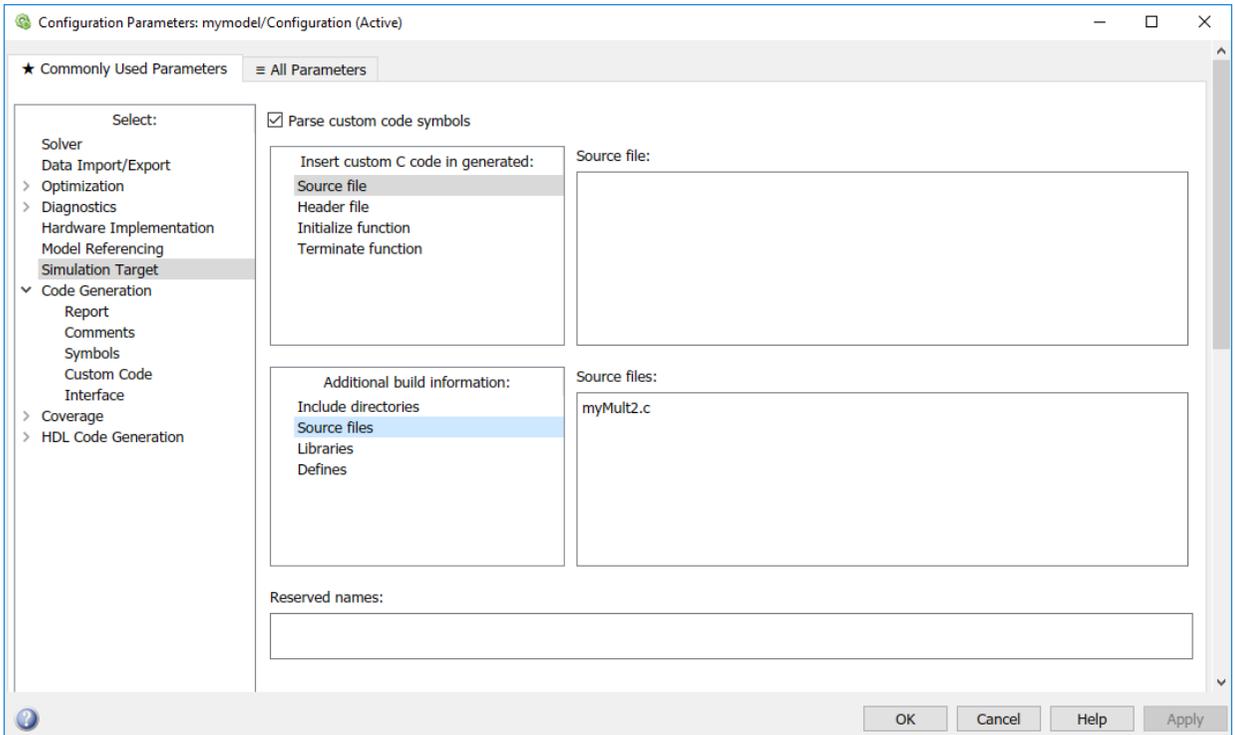
```
function y = myfunc
%#codegen
y = 21;
coder.cinclude('myMult2.h');
y = coder.ceval('myMult2', y);
end
```

Open the Configuration Parameters dialog box.

On the **Solver** pane, select a fixed-step solver.

Specify the locations of `myMult2.h` and `myMult2.c` for simulation and code generation.

- On the **Simulation Target** pane, under **Additional build information**, for **Include directories**, enter `mycfiles`.
- For **Source files**, enter `myMult2.c`.



- On the **Code Generation > Custom Code** pane, select the **Use the same custom code settings as Simulation Target** check box.

Save the model as mymodel1.

Build the model.

The file mymodel1.h contains this statement:

```
#include "myMult2.h"
```

Input Arguments

headerfile — Name of header file

character vector | string scalar

Name of a header file specified as a character vector or string scalar. `headerfile` must be a compile-time constant.

Enclose a system header file name in angle brackets `< >`. The generated `#include` statement for a system header file has the format `#include <sysheader>`. A system header file must be in a standard location or on the include path. Specify the include path by using code generation custom code parameters.

Example: `coder.cinclude ('<sysheader.h>')`

For a header file that is not a system header file, omit the angle brackets. The generated `#include` statement for a header file that is not a system header file has the format `#include "myHeader"`. The header file must be in the current folder or on the include path. Specify the include path by using code generation custom code parameters.

Example: `coder.cinclude ('myheader.h')`

Data Types: `char`

allfiles — All source files option

`true` | `false`

Option to include header file in all generated C/C++ source files. If `allfiles` is `true`, MATLAB Coder generates the include statement in almost all of the C/C++ source files, except for some utility files. If `allfiles` is `false`, the behavior is the same as the behavior of `coder.cinclude (headerfile)`.

In a MATLAB Function block, the code generator ignores the all source files option.

Data Types: `logical`

Limitations

- Do not call `coder.cinclude` inside run-time conditional constructs such as `if` statements, `switch` statements, `while`-loops, and `for`-loops. You can call `coder.cinclude` inside compile-time conditional statements, such as `coder.target`. For example:

```
...
if ~coder.target('MATLAB')
    coder.cinclude('foo.h');
```

```
    coder.ceval('foo');  
end  
...
```

Tips

- Before a `coder.ceval` call, call `coder.cinclude` to include the header file required by the external function that `coder.ceval` calls.
- Extraneous include statements in generated C/C++ code can increase compile time and reduce code readability. To avoid extraneous include statements in code generated by MATLAB Coder, follow these best practices:
 - Place a `coder.cinclude` call as close as possible to the `coder.ceval` call that requires the header file.
 - Do not set `allfiles` to `true`.

For the MATLAB Function block, the code generator generates the include statement in the model header file.

- In R2016a and earlier releases, for any `coder.cinclude` call, MATLAB Coder included the header file in almost all generated C/C++ source files, except for some utility files. If you have code that depends on this legacy behavior, you can preserve the legacy behavior by using this syntax:

```
coder.cinclude(headerfile, 'InAllSourceFiles', true)
```

See Also

`coder.ceval` | `coder.target`

Topics

“Model Configuration Parameters: Code Generation Custom Code” (Simulink Coder)

Introduced in R2013a

coder.const

Fold expressions into constants in generated code

Syntax

```
out = coder.const(expression)
[out1,...,outN] = coder.const(handle, arg1, ..., argN)
```

Description

`out = coder.const(expression)` evaluates `expression` and replaces `out` with the result of the evaluation in generated code.

`[out1,...,outN] = coder.const(handle, arg1, ..., argN)` evaluates the multi-output function having handle `handle`. It then replaces `out1, ..., outN` with the results of the evaluation in the generated code.

Examples

Specify Constants in Generated Code

This example shows how to specify constants in generated code using `coder.const`.

Write a function `AddShift` that takes an input `Shift` and adds it to the elements of a vector. The vector consists of the square of the first 10 natural numbers. `AddShift` generates this vector.

```
function y = AddShift(Shift) %#codegen
y = (1:10).^2+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generator produces code for creating the vector. It adds `Shift` to each element of the vector during vector creation. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
    int k;
    for (k = 0; k < 10; k++) {
        y[k] = (double)((1 + k) * (1 + k)) + Shift;
    }
}
```

Replace the statement

```
y = (1:10).^2+Shift;
```

with

```
y = coder.const((1:10).^2)+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generator creates the vector containing the squares of the first 10 natural numbers. In the generated code, it adds `Shift` to each element of this vector. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
    int i0;
    static const signed char iv0[10] = { 1, 4, 9, 16, 25, 36,
                                         49, 64, 81, 100 };

    for (i0 = 0; i0 < 10; i0++) {
        y[i0] = (double)iv0[i0] + Shift;
    }
}
```

Create Lookup Table in Generated Code

This example shows how to fold a user-written function into a constant in generated code.

Write a function `getsine` that takes an input `index` and returns the element referred to by `index` from a lookup table of sines. The function `getsine` creates the lookup table using another function `gettable`.

```
function y = getsine(index) %#codegen
    assert(isa(index, 'int32'));
    persistent tbl;
    if isempty(tbl)
        tbl = gettable(1024);
    end
    y = tbl(index);

function y = gettable(n)
    y = zeros(1,n);
    for i = 1:n
        y(i) = sin((i-1)/(2*pi*n));
    end
```

Generate code for `getsine` using an argument of type `int32`. Open the Code Generation Report.

```
codegen -config:lib -launchreport getsine -args int32(0)
```

The generated code contains instructions for creating the lookup table.

Replace the statement:

```
tbl = gettable(1024);
```

with:

```
tbl = coder.const(gettable(1024));
```

Generate code for `getsine` using an argument of type `int32`. Open the Code Generation Report.

The generated code contains the lookup table itself. `coder.const` forces the expression `gettable(1024)` to be evaluated during code generation. The generated code does not

contain instructions for the evaluation. The generated code contains the result of the evaluation itself.

Specify Constants in Generated Code Using Multi-Output Function

This example shows how to specify constants in generated code using a multi-output function in a `coder.const` statement.

Write a function `MultiplyConst` that takes an input `factor` and multiplies every element of two vectors `vec1` and `vec2` with `factor`. The function generates `vec1` and `vec2` using another function `EvalConsts`.

```
function [y1,y2] = MultiplyConst(factor) %#codegen
    [vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
    y1=vec1.*factor;
    y2=vec2.*factor;

function [f1,f2]=EvalConsts(z,n)
    f1=z.^(2*n)/factorial(2*n);
    f2=z.^(2*n+1)/factorial(2*n+1);
```

Generate code for `MultiplyConst` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport MultiplyConst -args 0
```

The code generator produces code for creating the vectors.

Replace the statement

```
[vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
```

with

```
[vec1,vec2]=coder.const(@EvalConsts,pi.*(1./2.^(1:10)),2);
```

Generate code for `MultiplyConst` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport MultiplyConst -args 0
```

The code generator does not generate code for creating the vectors. Instead, it calculates the vectors and specifies the calculated vectors in generated code.

Read Constants by Processing XML File

This example shows how to call an extrinsic function using `coder.const`.

Write an XML file `MyParams.xml` containing the following statements:

```
<params>
  <param name="hello" value="17"/>
  <param name="world" value="42"/>
</params>
```

Save `MyParams.xml` in the current folder.

Write a MATLAB function `xml2struct` that reads an XML file. The function identifies the XML tag `param` inside another tag `params`.

After identifying `param`, the function assigns the value of its attribute `name` to the field name of a structure `s`. The function also assigns the value of attribute `value` to the value of the field.

```
function s = xml2struct(file)

s = struct();
doc = xmlread(file);
els = doc.getElementsByTagName('params');
for i = 0:els.getLength-1
    it = els.item(i);
    ps = it.getElementsByTagName('param');
    for j = 0:ps.getLength-1
        param = ps.item(j);
        paramName = char(param.getAttribute('name'));
        paramValue = char(param.getAttribute('value'));
        paramValue = evalin('base', paramValue);
        s.(paramName) = paramValue;
    end
end
```

Save `xml2struct` in the current folder.

Write a MATLAB function `MyFunc` that reads the XML file `MyParams.xml` into a structure `s` using the function `xml2struct`. Declare `xml2struct` as extrinsic using `coder.extrinsic` and call it in a `coder.const` statement.

```
function y = MyFunc(u) %#codegen
    assert(isa(u, 'double'));
    coder.extrinsic('xml2struct');
    s = coder.const(xml2struct('MyParams.xml'));
    y = s.hello + s.world + u;
```

Generate code for `MyFunc` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:dll -launchreport MyFunc -args 0
```

The code generator executes the call to `xml2struct` during code generation. It replaces the structure fields `s.hello` and `s.world` with the values 17 and 42 in generated code.

Input Arguments

expression — MATLAB expression or user-written function

expression with constants | single-output function with constant arguments

MATLAB expression or user-defined single-output function.

The expression must have compile-time constants only. The function must take constant arguments only. For instance, the following code leads to a code generation error, because `x` is not a compile-time constant.

```
function y=func(x)
    y=coder.const(log10(x));
```

To fix the error, assign `x` to a constant in the MATLAB code. Alternatively, during code generation, you can use `coder.Constant` to define input type as follows:

```
codegen -config:lib func -args coder.Constant(10)
```

Example: `2*pi`, `factorial(10)`

handle — Function handle

function handle

Handle to built-in or user-written function.

Example: @log, @sin

Data Types: `function_handle`

arg1, . . . , argN — Arguments to the function with handle handle

function arguments that are constants

Arguments to the function with handle `handle`.

The arguments must be compile-time constants. For instance, the following code leads to a code generation error, because `x` and `y` are not compile-time constants.

```
function y=func(x,y)
    y=coder.const(@nchoosek,x,y);
```

To fix the error, assign `x` and `y` to constants in the MATLAB code. Alternatively, during code generation, you can use `coder.Constant` to define input type as follows:

```
codegen -config:lib func -args {coder.Constant(10),coder.Constant(2)}
```

Output Arguments

out — Value of expression

value of the evaluated expression

Value of `expression`. In the generated code, MATLAB Coder replaces occurrences of `out` with the value of `expression`.

out1, . . . , outN — Outputs of the function with handle handle

values of the outputs of the function with handle `handle`

Outputs of the function with handle `handle`. MATLAB Coder evaluates the function and replaces occurrences of `out1, . . . , outN` with constants in the generated code.

Tips

- The code generator constant-folds expressions automatically when possible. Typically, automatic constant-folding occurs for expressions with scalars only. Use `coder.const` when the code generator does not constant-fold expressions on its own.

See Also

Introduced in R2013b

coder.cstructname

Package: coder

Name structure in generated code

Syntax

```
coder.cstructname(var, 'structName')  
coder.cstructname(var, 'structName', 'extern')  
coder.cstructname(var, 'structName', 'extern', Name, Value)
```

Description

`coder.cstructname(var, 'structName')` specifies the name of the structure type that represents `var` in the generated C/C++ code. `var` is a structure or cell array variable. `structName` is the name for the structure type in the generated code. Call `coder.cstructname` before the first use of the variable. If `var` is a cell array element, call `coder.cstructname` after the first assignment to the element.

`coder.cstructname(var, 'structName', 'extern')` declares an externally defined structure. It does not generate the definition of the structure type. Provide the definition in a custom include file.

`coder.cstructname(var, 'structName', 'extern', Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments.

Limitations

- You cannot use `coder.cstructname` with global variables.
- If `var` is a cell array, the field names of externally defined structures must be `f1`, `f2`, and so on.
- If `var` is a cell array element, call `coder.cstructname` after the first assignment to the element. For example:

```

...
x = cell(2,2);
x{1} = struct('a', 3);
coder.cstructname(x{1}, 'mytype');
...

```

Tips

- The code generator represents a heterogeneous cell array as a structure in the generated C/C++ code. To specify the name of the generated structure type, use `coder.cstructname`. Using `coder.cstructname` with a cell array variable makes the cell array heterogeneous.
- To use `coder.cstructname` on arrays, use single indexing. For example, you cannot use `coder.cstructname(x(1,2))`. Instead, use single indexing, for example `coder.cstructname(x(n))`.
- If you use `coder.cstructname` on an array, it sets the name of the base type of the array, not the name of the array. Therefore, you cannot use `coder.cstructname` on the base element and then on the array. For example, the following code does not work. The second `coder.cstructname` attempts to set the name of the base type to `myStructArrayName`, which conflicts with the previous `coder.cstructname`, `myStructName`.

```

% Define scalar structure with field a
myStruct = struct('a', 0);
coder.cstructname(myStruct, 'myStructName');
% Define array of structure with field a
myStructArray = repmat(myStruct, k, n);
coder.cstructname(myStructArray, 'myStructArrayName');

```

- If you are using custom structure types, specify the name of the header file that includes the external definition of the structure. Use the `HeaderFile` input argument.
- If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. To take advantage of target-specific function implementations that require data to be aligned, use the `Alignment` input argument.
- You can also use `coder.cstructname` to assign a name to a substructure, which is a structure that appears as a field of another structure. For more information, see “Assign a Name to a SubStructure” on page 2-85.

Input Arguments

structName

The name of the structure type in the generated code.

var

Structure or cell array variable.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Alignment

The run-time memory alignment of structures of this type in bytes. If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. This capability allows you to take advantage of target-specific function implementations that require data to be aligned. By default, the structure is not aligned on a specific boundary. Hence it is not matched by CRL functions that require alignment.

Alignment must be either -1 or a power of 2 that is not greater than 128.

Default: -1

HeaderFile

Name of the header file that contains the external definition of the structure, for example, 'mystruct.h'.

By default, the generated code contains `#include` statements for custom header files after the standard header files. If a standard header file refers to the custom structure type, then the compilation fails. By specifying the `HeaderFile` option, MATLAB Coder includes that header file exactly at the point where it is required.

Must be a non-empty character vector or string scalar.

Examples

Apply `coder.cstructname` to Top-Level Inputs

Generate code for a MATLAB function that takes structure inputs.

- 1 Write a MATLAB function, `topfun`, that assigns the name `MyStruct` to its input parameter.

```
function y = topfun(x)    %#codegen
% Assign the name 'MyStruct' to the input variable in
% the generated code
    coder.cstructname(x, 'MyStruct');
    y = x;
end
```

- 2 Declare a structure `s` in MATLAB. `s` is the structure definition for the input variable `x`.

```
s = struct('a',42,'b',4711);
```

- 3 Generate a MEX function for `topfun`, using the `-args` option to specify that the input parameter is a structure.

```
codegen topfun.m -args { s }
```

`codegen` generates a MEX function in the default folder `codegen\mex\topfun`. In this folder, the structure definition is in `topfun_types.h`.

```
typedef struct
{
    double a;
    double b;
} MyStruct;
```

Assign a Name to a Structure and Pass It to a Function

Assign the name `MyStruct` to the structure `var`. Pass the structure to a C function `use_struct`.

- 1 Create a C header file, `use_struct.h`, for a `use_struct` function that takes a parameter of type `MyStruct`. Define a structure of type `MyStruct` in the header file.

```
#ifndef MATLAB_MEX_FILE
#include <tmwtypes.h>
#else
#include "rtwtypes.h"
#endif

typedef struct MyStruct
{
    double s1;
    double s2;
} MyStruct;

void use_struct(struct MyStruct *my_struct);
```

2 Write the C function `use_struct.c`.

```
#include <stdio.h>
#include <stdlib.h>

#include "use_struct.h"

void use_struct(struct MyStruct *my_struct)
{
    double x = my_struct->s1;
    double y = my_struct->s2;
}
```

3 Write a `m_use_struct` compliant with MATLAB that declares a structure. Have the function assign the name `MyStruct` to the structure. Then, have the function call the C function `use_struct` using `coder.ceval`.

```
function m_use_struct %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
% Declare a MATLAB structure
var.s1 = 1;
var.s2 = 2;

% Assign the name MyStruct to the structure variable.
% extern indicates this is an externally defined
% structure.
coder.cstructname(var, 'MyStruct', 'extern');

% Call the C function use_struct. The type of var
% matches the signature of use_struct.
% Use coder.rref to pass the variable var by
```

```
% reference as a read-only input to the external C
% function use_struct
coder.ceval('use_struct', coder.rref(var));
```

- 4 Generate C library code for function `m_use_struct`, passing `use_struct.h` to include the structure definition.

```
codegen -config:lib m_use_struct use_struct.c use_struct.h
```

codegen generates C code in the default folder `codegen\lib\m_use_struct`. The generated header file `m_use_struct_types.h` in this folder does not contain a definition of the structure `MyStruct` because `MyStruct` is an external type.

Assign a Name to a SubStructure

Use `coder.cstructname` to assign a name to a substructure.

- 1 Define a MATLAB structure, `top`, that has another structure, `lower`, as a field.

```
% Define structure top with field lower,
% which is a structure with fields a and b
top.lower = struct('a',1,'b',1);
top.c = 1;
```

- 2 Define a function, `MyFunc`, which takes an argument, `TopVar`, as input. Mark the function for code generation using `codegen`.

```
function out = MyFunc(TopVar) %codegen
```

- 3 Inside `MyFunc`, include the following lines

```
coder.cstructname(TopVar, 'topType');
coder.cstructname(TopVar.lower, 'lowerType');
```

- 4 So that `TopVar` has the same type as `top`, generate C code for `MyFunc` with an argument having the same type as `top`.

```
codegen -config:lib MyFunc -args coder.typeof(top)
```

In the generated C code, the field variable `TopVar.lower` is assigned the type name `lowerType`. For instance, the structure declaration of the variable `TopVar.lower` appears in the C code as:

```
typedef struct
{
```

```
    /* Definitions of a and b appear here */  
} lowerType;
```

and the structure declaration of the variable `TopVar` appears as:

```
typedef struct  
{  
    lowerType lower;  
    /* Definition of c appears here */  
} topType;
```

Assign a Name to a Structure That Is an Element of a Cell Array

Write a function `struct_in_cell` that has a cell array `x{1}` that contains a structure. The `coder.cstructname` call follows the assignment to `x{1}`.

```
function z = struct_in_cell()  
x = cell(2,2);  
x{1} = struct('a', 3);  
coder.cstructname(x{1}, 'mytype');  
z = x{1};  
end
```

Generate a static library for `struct_in_cell`.

```
codegen -config:lib struct_in_cell -report
```

The type for `a` has the name `mytype`.

```
typedef struct {  
    double a;  
} mytype;
```

See Also

Topics

“Code Generation for Cell Arrays”

Introduced in R2011a

coder.extrinsic

Package: coder

Declare extrinsic function or functions

Syntax

```
coder.extrinsic('function_name');  
  
coder.extrinsic('function_name_1', ... , 'function_name_n');  
  
coder.extrinsic('-sync:on', 'function_name');  
  
coder.extrinsic('-sync:on', 'function_name_1', ... ,  
    'function_name_n');  
  
coder.extrinsic('-sync:off', 'function_name');  
  
coder.extrinsic('-sync:off', 'function_name_1', ... ,  
    'function_name_n');
```

Arguments

function_name
function_name_1, ... , function_name_n

Declares *function_name* or *function_name_1* through *function_name_n* as extrinsic functions.

-sync:on

function_name or *function_name_1* through *function_name_n*.

Enables synchronization of global data between MATLAB and MEX functions before and after calls to the extrinsic functions, *function_name* or *function_name_1* through *function_name_n*. If only a few extrinsic calls modify global data, turn off synchronization before and after all extrinsic function calls by setting the global synchronization mode to `At MEX-function entry and exit`. Use the *-sync:on*

option to turn on synchronization for only the extrinsic calls that *do* modify global data.

`-sync:off`

Disables synchronization of global data between MATLAB and MEX functions before and after calls to the extrinsic functions, `function_name` or `function_name_1` through `function_name_n`. If most extrinsic calls modify global data, but a few do not, you can use the `-sync:off` option to turn off synchronization for the extrinsic calls that *do not* modify global data.

Description

`coder.extrinsic` declares extrinsic functions. During simulation, the code generator produces code for the call to an extrinsic function, but does not produce the function's internal code. Therefore, simulation can run only on platforms where MATLAB software is installed. During standalone code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable. Provided that there is no change to the output, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, compilation errors occur.

You cannot use `coder.ceval` on functions that you declare extrinsic by using `coder.extrinsic`.

`coder.extrinsic` is ignored outside of code generation.

Limitations

- Extrinsic function calls have some overhead that can affect performance. Input data that is passed in an extrinsic function call must be provided to MATLAB, which requires making a copy of the data. If the function has any output data, this data must be transferred back into the MEX function environment, which also requires a copy.

Tips

- The code generator detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions, but you do not have to declare them extrinsic using the `coder.extrinsic` function.
- Use the `coder.screener` function to detect which functions you must declare extrinsic. This function opens the code generations readiness tool that detects code generation issues in your MATLAB code.

Examples

The following code declares the MATLAB function `patch` as extrinsic in the MATLAB local function `create_plot`.

```
function c = pythagoras(a,b,color) %#codegen
% Calculates the hypotenuse of a right triangle
% and displays the triangle as a patch object.

c = sqrt(a^2 + b^2);

create_plot(a, b, color);

function create_plot(a, b, color)

%Declare patch as extrinsic
coder.extrinsic('patch');

x = [0;a;a];
y = [0;0;b];
patch(x, y, color);
axis('equal');
```

By declaring `patch` as extrinsic, you instruct the code generator not to compile or produce code for `patch`. Instead, the code generator dispatches `patch` to MATLAB for execution.

See Also

`coder.ceval` | `coder.screener`

Topics

“Extrinsic Functions”

“Controlling Synchronization for Extrinsic Function Calls” (MATLAB Coder)

“Resolution of Function Calls for Code Generation”

“Restrictions on Extrinsic Functions for Code Generation”

Introduced in R2011a

coder.ignoreConst

Prevent use of constant value of expression for function specializations

Syntax

```
coder.ignoreConst(expression)
```

Description

`coder.ignoreConst(expression)` prevents the code generator from using the constant value of `expression` to create function specializations on page 2-93.

`coder.ignoreConst(expression)` returns the value of `expression`.

Examples

Prevent Function Specializations Based on Constant Input Values

Use `coder.ignoreConst` to prevent function specializations for a function that is called with constant values.

Write the function `call_myfcn`, which calls `myfcn`.

```
function [x, y] = call_myfcn(n)
    %#codegen
    x = myfcn(n, 'mode1');
    y = myfcn(n, 'mode2');
end
```

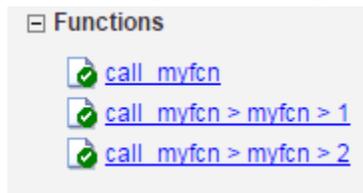
```
function y = myfcn(n,mode)
    coder.inline('never');
    if strcmp(mode,'mode1')
        y = n;
    else
        y = -n;
    end
end
```

```
end  
end
```

Generate standalone C code. For example, generate a static library. Enable the code generation report.

```
codegen -config:lib call_myfcn -args {1} -report
```

In the code generation report, you see two function specializations for `call_myfcn`.



The code generator creates `call_myfcn>myfcn>1` for mode with a value of 'mode1'. It creates `call_myfcn>myfcn>2` for mode with a value of 'mode2'.

In the generated C code, you see the specializations `my_fcn` and `b_my_fcn`.

```
static double b_myfcn(double n)  
{  
    return -n;  
}  
  
static double myfcn(double n)  
{  
    return n;  
}
```

To prevent the function specializations, instruct the code generator to ignore that values of the mode argument are constant.

```
function [x, y] = call_myfcn(n)  
    %#codegen  
    x = myfcn(n, coder.ignoreConst('mode1'));  
    y = myfcn(n, coder.ignoreConst('mode2'));  
end  
  
function y = myfcn(n,mode)  
    coder.inline('never');  
    if strcmp(mode, 'mode1')  
        y = n;
```

```

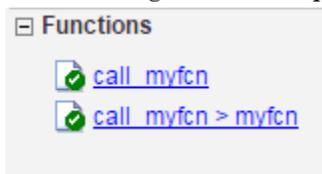
else
    y = -n;
end
end

```

Generate the C code.

```
codegen -config:lib call_myfcn -args {1} -report
```

In the code generation report, you do not see multiple function specializations.



In the generated C code, you see one function for `my_fcn`.

Input Arguments

expression — Expression whose value is to be treated as a nonconstant
MATLAB expression

Definitions

Function Specialization

Version of a function in which an input type, size, complexity, or value is customized for a particular invocation of the function.

Function specialization produces efficient C code at the expense of code duplication. The code generation report shows all MATLAB function specializations that the code generator creates. However, the specializations might not appear in the generated C/C++ code due to later transformations or optimizations.

Tips

- For some recursive function calls, you can use `coder.ignoreConst` to force run-time recursion. See “Force Code Generator to Use Run-Time Recursion”.
- `coder.ignoreConst(expression)` prevents the code generator from using the constant value of `expression` to create function specializations. It does not prevent other uses of the constant value during code generation.

See Also

`coder.inline`

Topics

“Force Code Generator to Use Run-Time Recursion”

“Compile-Time Recursion Limit Reached”

Introduced in R2017a

coder.inline

Package: coder

Control inlining in generated code

Syntax

```
coder.inline('always')
coder.inline('never')
coder.inline('default')
```

Description

`coder.inline('always')` forces inlining on page 2-96 of the current function in the generated code. Place the `coder.inline` directive inside the function to which it applies. The code generator does not inline entry-point functions, inline functions into `parfor` loops, or inline functions called from `parfor` loops.

`coder.inline('never')` prevents inlining of the current function in the generated code. Prevent inlining when you want to simplify the mapping between the MATLAB source code and the generated code.

`coder.inline('default')` uses internal heuristics to determine whether to inline the current function. Usually, the heuristics produce highly optimized code. Use `coder.inline` only when you need to fine-tune these optimizations.

Examples

- “Prevent Function Inlining” on page 2-95
- “Use `coder.inline` in Control Flow Statements” on page 2-96

Prevent Function Inlining

In this example, function `f00` is not inlined in the generated code:

```
function y = foo(x)
    coder.inline('never');
    y = x;
end
```

Use `coder.inline` in Control Flow Statements

You can use `coder.inline` in control flow code. If the software detects contradictory `coder.inline` directives, the generated code uses the default inlining heuristic and issues a warning.

Suppose that you want to generate code for a division function used by a system with limited memory. To optimize memory use in the generated code, the `inline_division` function manually controls inlining based on whether it performs scalar division or vector division:

```
function y = inline_division(dividend, divisor)

% For scalar division, inlining produces smaller code
% than the function call itself.
if isscalar(dividend) && isscalar(divisor)
    coder.inline('always');
else
% Vector division produces a for-loop.
% Prohibit inlining to reduce code size.
    coder.inline('never');
end

if any(divisor == 0)
    error('Cannot divide by 0');
end

y = dividend / divisor;
```

Definitions

Inlining

Technique that replaces a function call with the contents (body) of that function. Inlining eliminates the overhead of a function call, but can produce larger C/C++ code. Inlining can create opportunities for further optimization of the generated C/C++ code.

See Also

Introduced in R2011a

coder.load

Load compile-time constants from MAT-file or ASCII file into caller workspace

Syntax

```
S = coder.load(filename)
S = coder.load(filename,var1,...,varN)
S = coder.load(filename,'-regexp',expr1,...,exprN)
S = coder.load(filename,'-ascii')
S = coder.load(filename,'-mat')
S = coder.load(filename,'-mat',var1,...,varN)
S = coder.load(filename,'-mat','-regexp', expr1,...,exprN)
```

Description

`S = coder.load(filename)` loads compile-time constants from `filename`.

- If `filename` is a MAT-file, then `coder.load` loads variables from the MAT-file into a structure array.
- If `filename` is an ASCII file, then `coder.load` loads data into a double-precision array.

`S = coder.load(filename,var1,...,varN)` loads only the specified variables from the MAT-file `filename`.

`S = coder.load(filename,'-regexp',expr1,...,exprN)` loads only the variables that match the specified regular expressions.

`S = coder.load(filename,'-ascii')` treats `filename` as an ASCII file, regardless of the file extension.

`S = coder.load(filename,'-mat')` treats `filename` as a MAT-file, regardless of the file extension.

`S = coder.load(filename,'-mat',var1,...,varN)` treats `filename` as a MAT-file and loads only the specified variables from the file.

`S = coder.load(filename, '-mat', '-regexp', expr1, ..., exprN)` treats `filename` as a MAT-file and loads only the variables that match the specified regular expressions.

Examples

Load compile-time constants from MAT-file

Generate code for a function `edgeDetect1` which given a normalized image, returns an image where the edges are detected with respect to the threshold value. `edgeDetect1` uses `coder.load` to load the edge detection kernel from a MAT-file at compile time.

Save the Sobel edge-detection kernel in a MAT-file.

```
k = [1 2 1; 0 0 0; -1 -2 -1];
```

```
save sobel.mat k
```

Write the function `edgeDetect1`.

```
function edgeImage = edgeDetect1(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

S = coder.load('sobel.mat', 'k');
H = conv2(double(originalImage), S.k, 'same');
V = conv2(double(originalImage), S.k', 'same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

Create a code generation configuration object for a static library.

```
cfg = coder.config('lib');
```

Generate a static library for `edgeDetect1`.

```
codegen -report -config cfg edgeDetect1
```

`codegen` generates C code in the `codegen\lib\edgeDetect1` folder.

Load compile-time constants from ASCII file

Generate code for a function `edgeDetect2` which given a normalized image, returns an image where the edges are detected with respect to the threshold value. `edgeDetect2` uses `coder.load` to load the edge detection kernel from an ASCII file at compile time.

Save the Sobel edge-detection kernel in an ASCII file.

```
k = [1 2 1; 0 0 0; -1 -2 -1];  
save sobel.dat k -ascii
```

Write the function `edgeDetect2`.

```
function edgeImage = edgeDetect2(originalImage, threshold) %#codegen  
assert(all(size(originalImage) <= [1024 1024]));  
assert(isa(originalImage, 'double'));  
assert(isa(threshold, 'double'));  
  
k = coder.load('sobel.dat');  
H = conv2(double(originalImage), k, 'same');  
V = conv2(double(originalImage), k', 'same');  
E = sqrt(H.*H + V.*V);  
edgeImage = uint8((E > threshold) * 255);
```

Create a code generation configuration object for a static library.

```
cfg = coder.config('lib');
```

Generate a static library for `edgeDetect2`.

```
codegen -report -config cfg edgeDetect2
```

`codegen` generates C code in the `codegen\lib\edgeDetect2` folder.

Input Arguments

filename — Name of file

character vector | string scalar

Name of file. `filename` must be a compile-time constant.

`filename` can include a file extension and a full or partial path. If `filename` has no extension, `load` looks for a file named `filename.mat`. If `filename` has an extension other than `.mat`, `load` treats the file as ASCII data.

ASCII files must contain a rectangular table of numbers, with an equal number of elements in each row. The file delimiter (the character between elements in each row) can be a blank, comma, semicolon, or tab character. The file can contain MATLAB comments (lines that begin with a percent sign, `%`).

Example: `'myFile.mat'`

`var1, ..., varN` — Names of variables to load

character vector | string scalar

Names of variables, specified as one or more character vectors or string scalars. Each variable name must be a compile-time constant. Use the `*` wildcard to match patterns.

Example: `coder.load('myFile.mat', 'A*')` loads all variables in the file whose names start with `A`.

`expr1, ..., exprN` — Regular expressions indicating which variables to load

character vector | string scalar

Regular expressions indicating which variables to load specified as one or more character vectors or string scalars. Each regular expression must be a compile-time constant.

Example: `coder.load('myFile.mat', '-regexp', '^A')` loads only variables whose names begin with `A`.

Output Arguments

`s` — Loaded variables or data

structure array | m-by-n array

If `filename` is a MAT-file, `S` is a structure array.

If `filename` is an ASCII file, `S` is an m-by-n array of type `double`. `m` is the number of lines in the file and `n` is the number of values on a line.

Limitations

- `coder.load` does not support loading objects.
- Arguments to `coder.load` must be compile-time constants.
- The output `S` must be the name of a structure or array without any subscripting. For example, `S(i) = coder.load('myFile.mat')` is not allowed.
- You cannot use `save` to save workspace data to a file inside a function intended for code generation. The code generator does not support the `save` function. Furthermore, you cannot use `coder.extrinsic` with `save`. Prior to generating code, you can use `save` to save workspace data to a file.

Tips

- `coder.load` loads data at compile time, not at run time. If you are generating MEX code or code for Simulink simulation, you can use the MATLAB function `load` to load run-time values.
- If the MAT-file contains unsupported constructs, use `coder.load(filename, var1, ..., varN)` to load only the supported constructs.
- If you generate code in a MATLAB Coder project, the code generator practices incremental code generation for the `coder.load` function. When the MAT-file or ASCII file used by `coder.load` changes, the software rebuilds the code.

See Also

`matfile` | `regexp` | `save`

Topics

“Regular Expressions” (MATLAB)

Introduced in R2013a

coder.nullcopy

Package: coder

Declare uninitialized variables

Syntax

```
X = coder.nullcopy(A)
```

Description

`X = coder.nullcopy(A)` copies type, size, and complexity of `A` to `X`, but does not copy element values. Preallocates memory for `X` without incurring the overhead of initializing memory.

`coder.nullcopy` does not support MATLAB classes as inputs.

Use With Caution

Use this function with caution. See “How to Eliminate Redundant Copies by Defining Uninitialized Variables”.

Examples

The following example shows how to declare variable `X` as a 1-by-5 vector of real doubles without performing an unnecessary initialization:

```
function X = foo

N = 5;
X = coder.nullcopy(zeros(1,N)) ;
for i = 1:N
    if mod(i,2) == 0
        X(i) = i;
```

```
    else
        X(i) = 0;
    end
end
```

Using `coder.nullcopy` with `zeros` lets you specify the size of vector X without initializing each element to zero.

See Also

Topics

“Eliminate Redundant Copies of Variables in Generated Code”

Introduced in R2011a

coder.opaque

Declare variable in generated code

Syntax

```
y = coder.opaque(type)
y = coder.opaque(type,value)
y = coder.opaque( ____, 'Size',Size)
y = coder.opaque( ____, 'HeaderFile',HeaderFile)
```

Description

`y = coder.opaque(type)` declares a variable `y` with the specified type and no initial value in the generated code.

- `y` can be a variable or a structure field.
- MATLAB code cannot set or access `y`, but external C functions can accept `y` as an argument.
- `y` can be an:
 - Argument to `coder.rref`, `coder.wref`, or `coder.ref`
 - Input or output argument to `coder.ceval`
 - Input or output argument to a user-written MATLAB function
 - Input to a subset of MATLAB toolbox functions supported for code generation
- Assignment from `y` declares another variable with the same type in the generated code. For example:

```
y = coder.opaque('int');
z = y;
```

declares a variable `z` of type `int` in the generated code.

- You can assign `y` from another variable declared using either `coder.opaque` or assignment from a variable declared using `coder.opaque`. The variables must have identical types.

- You can compare `y` to another variable declared using either `coder.opaque` or assignment from a variable declared using `coder.opaque`. The variables must have identical types.

`y = coder.opaque(type,value)` specifies the type and initial value of `y`.

`y = coder.opaque(____, 'Size',Size)` specifies the size, in bytes, of `y`. You can specify the size with any of the previous syntaxes.

`y = coder.opaque(____, 'HeaderFile',HeaderFile)` specifies the header file that contains the type definition. The code generator produces the `#include` statement for the header file where the statement is required in the generated code. You can specify the header file with any of the previous syntaxes.

Examples

Declare Variable Specifying Initial Value

Generate code for a function `valtest` which returns 1 if the call to `myfun` is successful. This function uses `coder.opaque` to declare a variable `x1` with type `int` and initial value 0. The assignment `x2 = x1` declares `x2` to be a variable with the type and initial value of `x1`.

Write a function `valtest`.

```
function y = valtest
%codegen
%declare x1 to be an integer with initial value '0'
x1 = coder.opaque('int','0');
%Declare x2 to have same type and initial value as x1
x2 = x1;
x2 = coder.ceval('myfun');
%test the result of call to 'myfun' by comparing to value of x1
if x2 == x1
    y = 0;
else
    y = 1;
```

```
end
end
```

Declare Variable Specifying Initial Value and Header File

Generate code for a MATLAB function `filetest` which returns its own source code using `fopen/fread/fclose`. This function uses `coder.opaque` to declare the variable that stores the file pointer used by `fopen/fread/fclose`. The call to `coder.opaque` declares the variable `f` with type `FILE *`, initial value `NULL`, and header file `<stdio.h>`.

Write a MATLAB function `filetest`.

```
function buffer = filetest
%#codegen

% Declare 'f' as an opaque type 'FILE *' with initial value 'NULL'
%Specify the header file that contains the type definition of 'FILE *';

f = coder.opaque('FILE *', 'NULL', 'HeaderFile', '<stdio.h>');
% Open file in binary mode
f = coder.ceval('fopen', cstring('filetest.m'), cstring('rb'));

% Read from file until end of file is reached and put
% contents into buffer
n = int32(1);
i = int32(1);
buffer = char(zeros(1,8192));
while n > 0
    % By default, MATLAB converts constant values
    % to doubles in generated code
    % so explicit type conversion to int32 is inserted.
    n = coder.ceval('fread', coder.ref(buffer(i)), int32(1), ...
        int32(numel(buffer)), f);
    i = i + n;
end
coder.ceval('fclose', f);

buffer = strip_cr(buffer);

% Put a C termination character '\0' at the end of MATLAB character vector
function y = cstring(x)
    y = [x char(0)];

% Remove all character 13 (CR) but keep character 10 (LF)
function buffer = strip_cr(buffer)
j = 1;
for i = 1:numel(buffer)
    if buffer(i) ~= char(13)
        buffer(j) = buffer(i);
        j = j + 1;
    end
end
```

```
    end
end
buffer(i) = 0;
```

Compare Variables Declared Using `coder.opaque`

Compare variables declared using `coder.opaque` to test for successfully opening a file.

Use `coder.opaque` to declare a variable `null` with type `FILE *` and initial value `NULL`.

```
null = coder.opaque('FILE *', 'NULL', 'HeaderFile', '<stdio.h>');
```

Use assignment to declare another variable `ftmp` with the same type and value as `null`.

```
ftmp = null;
ftmp = coder.ceval('fopen', ['testfile.txt', char(0)], ['r', char(0)]);
```

Compare the variables.

```
if ftmp == null
    %error condition
end
```

Cast to and from Types of Variables Declared Using `coder.opaque`

This example shows how to cast to and from types of variables that are declared using `coder.opaque`. The function `castopaque` calls the C run-time function `strncmp` to compare at most `n` characters of the strings `s1` and `s2`. `n` is the number of characters in the shorter of the strings. To generate the correct C type for the `strncmp` input `nsizet`, the function casts `n` to the C type `size_t` and assigns the result to `nsizet`. The function uses `coder.opaque` to declare `nsizet`. Before using the output `retval` from `strncmp`, the function casts `retval` to the MATLAB type `int32` and stores the results in `y`.

Write this MATLAB function:

```
function y = castopaque(s1,s2)

% <0 - the first character that does not match has a lower value in s1 than in s2
%  0 - the contents of both strings are equal
% >0 - the first character that does not match has a greater value in s1 than in s2
%
%#codegen
```

```

coder.cinclude('<string.h>');
n = min(numel(s1), numel(s2));

% Convert the number of characters to compare to a size_t
nsizet = cast(n, 'like', coder.opaque('size_t', '0'));

% The return value is an int
retval = coder.opaque('int');
retval = coder.ceval('strncmp', cstr(s1), cstr(s2), nsizet);

% Convert the opaque return value to a MATLAB value
y = cast(retval, 'int32');

%-----
function sc = cstr(s)
% NULL terminate a MATLAB character vector for C
sc = [s, char(0)];

```

Generate the MEX function.

```
codegen castopaque -args {blanks(3), blanks(3)} -report
```

Call the MEX function with inputs 'abc' and 'abc'.

```
castopaque_mex('abc', 'abc')
```

```
ans =
```

```
0
```

The output is 0 because the strings are equal.

Call the MEX function with inputs 'abc' and 'abd'.

```
castopaque_mex('abc', 'abd')
```

```
ans =
```

```
-1
```

The output is -1 because the third character d in the second string is greater than the third character c in the first string.

Call the MEX function with inputs 'abd' and 'abc'.

```
castopaque_mex('abd','abc')  
  
ans =  
  
     1
```

The output is 1 because the third character `d` in the first string is greater than the third character `c` in the second string.

In the MATLAB workspace, you can see that the type of `y` is `int32`.

Declare Variable Specifying Initial Value and Size

Declare `y` to be a 4-byte integer with initial value 0.

```
y = coder.opaque('int','0','Size',4);
```

Input Arguments

type — Type of variable

character vector | string scalar

Type of variable in generated code. `type` must be a compile-time constant. The type must be a:

- Built-in C data type or a type defined in a header file
- C type that supports copy by assignment
- Legal prefix in a C declaration

Example: `'FILE *'`

value — Initial value of variable

character vector | string scalar

Initial value of variable in generated code. `value` must be a compile-time constant. Specify a C expression not dependent on MATLAB variables or functions.

If you do not provide the initial value in `value`, initialize the value of the variable before using it. To initialize a variable declared using `coder.opaque`:

- Assign a value from another variable with the same type declared using either `coder.opaque` or assignment from a variable declared using `coder.opaque`.
- Assign a value from an external C function.
- Pass the address of the variable to an external function using `coder.wref`.

Specify a value that has the type that `type` specifies. Otherwise, the generated code can produce unexpected results.

Example: `'NULL'`

size — Size of variable

integer

Number of bytes for the variable in the generated code, specified as an integer. If you do not specify the size, the size of the variable is 8 bytes.

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

HeaderFile — Name of header file

character vector | string scalar

Name of header file that contains the definition of `type`. `HeaderFile` must be a compile-time constant.

For a system header file, use angle brackets.

Example: `'<stdio.h>'` generates `#include <stdio.h>`

For an application header file, use double quotes.

Example: `"foo.h"` generates `#include "foo.h"`

If you omit the angle brackets or double quotes, the code generator produces double quotes.

Example: `'foo.h'` generates `#include "foo.h"`

Specify the include path in the build configuration parameters.

Example: `cfg.CustomInclude = 'c:\myincludes'`

Tips

- Specify a value that has the type that `type` specifies. Otherwise, the generated code can produce unexpected results. For example, the following `coder.opaque` declaration can produce unexpected results.

```
y = coder.opaque('int', '0.2')
```

- `coder.opaque` declares the type of a variable. It does not instantiate the variable. You can instantiate a variable by using it later in the MATLAB code. In the following example, assignment of `fp1` from `coder.ceval` instantiates `fp1`.

```
% Declare fp1 of type FILE *
fp1 = coder.opaque('FILE *');
%Create the variable fp1
fp1 = coder.ceval('fopen', ['testfile.txt', char(0)], ['r', char(0)]);
```

- In the MATLAB environment, `coder.opaque` returns the value specified in `value`. If `value` is not provided, it returns an empty character vector.
- You can compare variables declared using either `coder.opaque` or assignment from a variable declared using `coder.opaque`. The variables must have identical types. The following example demonstrates how to compare these variables. “Compare Variables Declared Using `coder.opaque`” on page 2-108
- To avoid multiple inclusions of the same header file in generated code, enclose the header file in the conditional preprocessor statements `#ifndef` and `#endif`. For example:

```
#ifndef MyHeader_h
#define MyHeader_h
<body of header file>
#endif
```

- You can use the MATLAB `cast` function to cast a variable to or from a variable that is declared using `coder.opaque`. Use `cast` with `coder.opaque` only for numeric types.

To cast a variable declared by `coder.opaque` to a MATLAB type, you can use the `B = cast(A, type)` syntax. For example:

```
x = coder.opaque('size_t', '0');
x1 = cast(x, 'int32');
```

You can also use the `B = cast(A, 'like', p)` syntax. For example:

```
x = coder.opaque('size_t','0');  
x1 = cast(x, 'like', int32(0));
```

To cast a MATLAB variable to the type of a variable declared by `coder.opaque`, you must use the `B = cast(A, 'like', p)` syntax. For example:

```
x = int32(12);  
x1 = coder.opaque('size_t', '0');  
x2 = cast(x, 'like', x1);
```

Use `cast` with `coder.opaque` to generate the correct data types for:

- Inputs to C/C++ functions that you call using `coder.ceval`.
- Variables that you assign to outputs from C/C++ functions that you call using `coder.ceval`.

Without this casting, it is possible to receive compiler warnings during code generation.

See Also

`coder.ceval` | `coder.ref` | `coder.rref` | `coder.wref`

Introduced in R2011a

coder.ref

Pass argument by reference

Syntax

```
coder.ref(arg)
```

Description

`coder.ref(arg)` passes `arg` by reference to an external C/C++ function that you call by using `coder.ceval`. Use `coder.ref` inside a `coder.ceval` call only. The C/C++ function can read from or write to the variable passed by reference. Use a separate `coder.ref` construct for each argument that you pass by reference to the function.

Examples

Pass Scalar Variable by Reference

Pass a variable by reference to a C function that adds 1 to the variable.

Suppose that you have a C function `addone`.

```
void addone(double* p) {  
    *p = *p + 1;  
}
```

The C function prototype defines the input as a pointer because it is passed by reference.

Pass a variable by reference to `addone`.

```
...  
u = 42;
```

```

coder.ceval('addone', coder.ref(u));
...

```

Pass Multiple Arguments by Reference

```

...
u = 1;
v = 2;
y = coder.ceval('my_fcn', coder.ref(u), coder.ref(v));
...

```

Pass Class Property by Reference

```

...
x = myClass;
x.prop = 1;
coder.ceval('foo', coder.ref(x.prop));
...

```

Pass a Structure by Reference

To indicate that the structure type is defined in a C header file, use `coder.cstructname`.

Suppose that you have the C function `incr_struct`.

```

void incr_struct(struct MyStruct *my_struct)
{
    my_struct->f1 = my_struct->f1 + 1;
    my_struct->f2 = my_struct->f2 + 1;
}

```

Suppose that a C header file, `MyStruct.h`, defines a structure type named `MyStruct`.

```

...
typedef struct MyStruct
{
    double f1;
    double f2;
} MyStruct;

```

```
void incr_struct(struct MyStruct *my_struct);  
  
...
```

In your MATLAB function, pass a structure by reference to `incr_struct`. To indicate that the structure type for `s` has the name `MyStruct` that is defined in a C header file `MyStruct.h`, use `coder.cstructname`.

```
...  
s = struct('f1', 1, 'f2', 2);  
coder.cstructname(s, 'MyStruct', 'extern', 'HeaderFile', 'MyStruct.h');  
coder.ceval('incr_struct', coder.ref(s));  
...
```

Pass Structure Field by Reference

```
...  
s = struct('s1', struct('a', [0 1]));  
coder.ceval('foo', coder.ref(s.s1.a));  
...
```

Input Arguments

arg — Argument to pass by reference

scalar variable | array | element of an array | structure | structure field | object
property

Argument to pass by reference to an external C/C++ function. The argument cannot be a class, a System object, a cell array, or an index into a cell array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` |
`uint32` | `uint64` | `logical` | `char` | `struct`
Complex Number Support: Yes

Limitations

- You cannot pass these data types by reference:
 - Class or System object

- Cell array or index into a cell array
- If a property has a get method, a set method, or validators, or is a System object property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties”.

Tips

- When you pass a structure by reference to an external C/C++ function, use `coder.cstructname` to provide the name of a C structure type that is defined in a C header file.
- In MATLAB, `coder.ref` results in an error. To parameterize your MATLAB code so that it can run in MATLAB and in generated code, use `coder.target`.
- You can use `coder.opaque` to declare variables that you pass to and from an external C/C++ function.

See Also

`coder.ceval` | `coder.cstructname` | `coder.opaque` | `coder.rref` | `coder.wref`

Introduced in R2011a

coder.rref

Pass argument by reference as read-only input

Syntax

```
coder.rref(arg)
```

Description

`coder.rref(arg)` passes `arg` by reference to an external C/C++ function that you call by using `coder.ceval`. Use `coder.rref` inside a `coder.ceval` call only. The code generator assumes that the variable that you pass in `coder.rref` is read-only. To avoid unpredictable results, the C/C++ function must not write to this variable.

Use a separate `coder.rref` construct for each argument that you pass by reference to the function.

Examples

Pass Scalar Variable as a Read-Only Reference

Pass a variable as a read-only reference to a C function that returns the value of the variable plus one.

Consider the C function `addone`:

```
double addone(const double* p) {  
    return *p + 1;  
}
```

The C function prototype defines the input as a pointer because it is passed by reference.

Pass a variable by reference to `addone`.

```

...
u = 42;
y = 0;
y = coder.ceval('addone', coder.rref(u));
...

```

Pass Multiple Arguments as a Read-Only Reference

```

...
u = 1;
v = 2;
y = coder.ceval('my_fcn', coder.rref(u), coder.rref(v));
...

```

Pass Class Property as a Read-Only Reference

```

...
x = myClass;
x.prop = 1;
y = coder.ceval('foo', coder.rref(x.prop));
...

```

Pass Structure as a Read-Only Reference

To indicate that the structure type is defined in a C header file, use `coder.cstructname`.

Suppose that you have the C function `use_struct`.

```

double use_struct(struct MyStruct *my_struct)
{
    return my_struct->f1 + my_struct->f2;
}

```

Suppose that a C header file, `MyStruct.h`, defines a structure type named `MyStruct`.

```

...
typedef struct MyStruct
{
    double f1;
    double f2;
}

```

```
} MyStruct;

double use_struct(struct MyStruct *my_struct);

...
```

In your MATLAB function, pass a structure as a read-only reference to `use_struct`. To indicate that the structure type for `s` has the name `MyStruct` that is defined in a C header file `MyStruct.h`, use `coder.cstructname`.

```
...
s = struct('f1', 1, 'f2', 2);
coder.cstructname(s, 'MyStruct', 'extern', 'HeaderFile', 'MyStruct.h');
y = 0;
y = coder.ceval('use_struct', coder.rref(s));
...
```

Pass Structure Field as a Read-Only Reference

```
...
s = struct('s1', struct('a', [0 1]));
y = coder.ceval('foo', coder.rref(s.s1.a));
...
```

Input Arguments

arg — Argument to pass by reference

scalar variable | array | element of an array | structure | structure field | object property

Argument to pass by reference to an external C/C++ function. The argument cannot be a class, a System object, a cell array, or an index into a cell array.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | struct
Complex Number Support: Yes

Limitations

- You cannot pass these data types by reference:
 - Class or System object
 - Cell array or index into a cell array
- If a property has a get method, a set method, or validators, or is a System object property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties”.

Tips

- When you pass a structure by reference to an external C/C++ function, use `coder.cstructname` to provide the name of a C structure type that is defined in a C header file.
- In MATLAB, `coder.rref` results in an error. To parameterize your MATLAB code so that it can run in MATLAB and in generated code, use `coder.target`.
- You can use `coder.opaque` to declare variables that you pass to and from an external C/C++ function.

See Also

`coder.ceval` | `coder.cstructname` | `coder.opaque` | `coder.ref` | `coder.wref`

Introduced in R2011a

coder.screener

Determine if function is suitable for code generation

Syntax

```
coder.screener(fcn)
coder.screener(fcn_1, ..., fcn_n )
```

Description

`coder.screener(fcn)` analyzes the entry-point MATLAB function, `fcn`. It identifies unsupported functions and language features as code generation compliance issues. It displays the code generation compliance issues in a report. If `fcn` calls other functions directly or indirectly that are not MathWorks® functions, `coder.screener` analyzes these functions. It does not analyze MathWorks functions. It is possible that `coder.screener` does not detect all code generation issues. Under certain circumstances, it is possible that `coder.screener` reports false errors.

`coder.screener(fcn_1, ..., fcn_n)` analyzes entry-point functions (`fcn_1, ..., fcn_n`).

Input Arguments

fcn

Name of entry-point MATLAB function that you want to analyze. Specify as a character vector or a string scalar.

fcn_1, ..., fcn_n

Comma-separated list of names of entry-point MATLAB functions that you want to analyze. Specify as character vectors or string scalars.

Examples

Identify Unsupported Functions

The `coder.screener` function identifies calls to functions that are not supported for code generation. It checks both the entry-point function, `foo1`, and the function `foo2` that `foo1` calls.

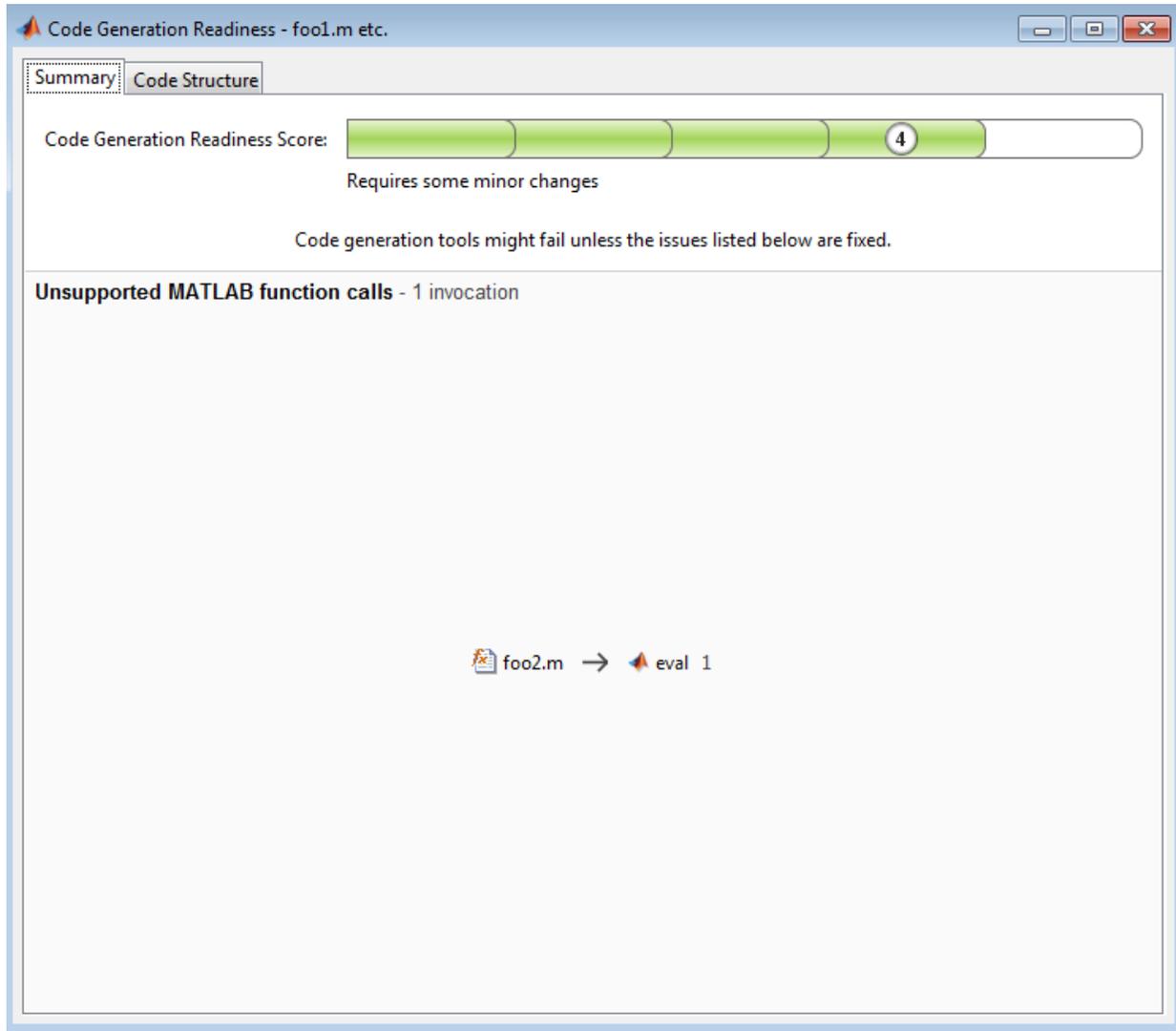
Analyze the MATLAB function `foo1` that calls `foo2`. Put `foo1` and `foo2` in separate files.

```
function out = foo1(in)
    out = foo2(in);
    disp(out);
end
```

```
function out = foo2(in)
    out = eval(in);
end
```

```
coder.screener('foo1')
```

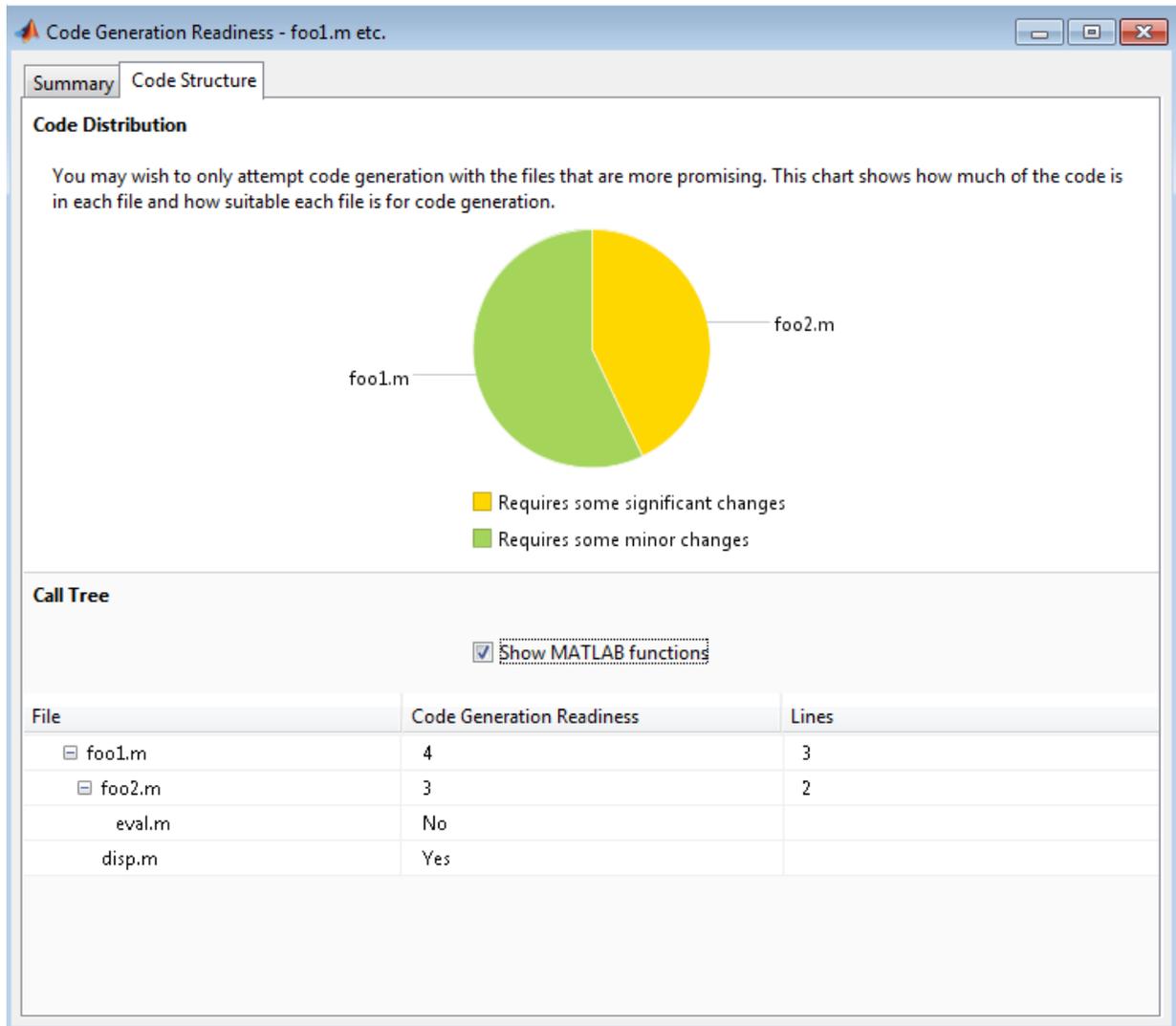
The code generation readiness report displays a summary of the unsupported MATLAB function calls. The function `foo2` calls one unsupported MATLAB function.



In the report, click the **Code Structure** tab and select the **Show MATLAB functions** check box.

This tab displays a pie chart showing the relative size of each file and how suitable each file is for code generation. In this case, the report:

- Colors `foo1.m` green to indicate that it is suitable for code generation.
- Colors `foo2.m` yellow to indicate that it requires significant changes.
- Assigns `foo1.m` a code generation readiness score of 4 and `foo2.m` a score of 3. The score is based on a scale of 1–5. 1 indicates that significant changes are required; 5 indicates that the code generation readiness tool does not detect issues.
- Displays a call tree.



The report **Summary** tab indicates that `foo2.m` contains one call to the `eval` function, which code generation does not support. To generate a MEX function for `foo2.m`, modify the code to make the call to `eval` extrinsic.

```
function out = foo2(in)
    coder.extrinsic('eval');
```

```
    out = eval(in);  
end
```

Rerun the code generation readiness tool.

```
coder.screener('foo1')
```

The report no longer flags that code generation does not support the `eval` function. When you generate a MEX function for `foo1`, the code generator dispatches `eval` to MATLAB for execution. For standalone code generation, the code generator does not generate code for `eval`.

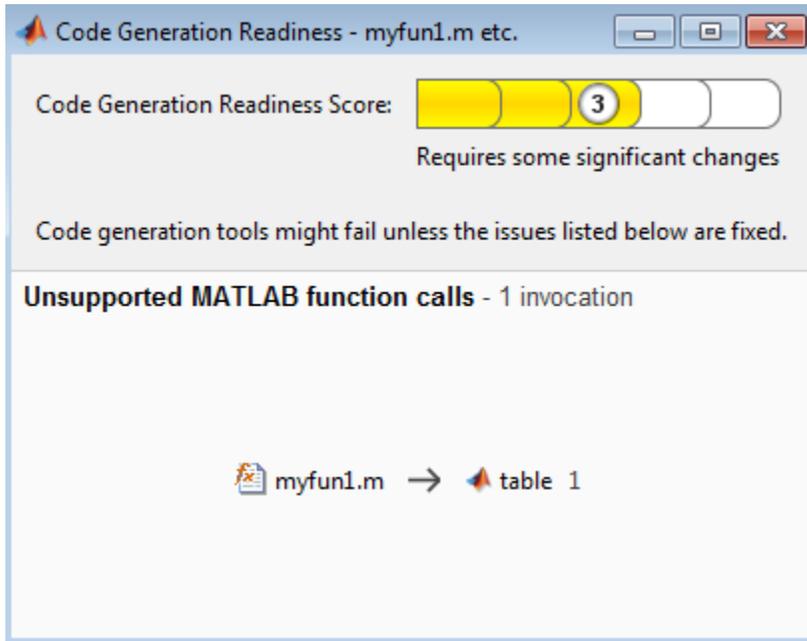
Identify Unsupported Data Types

The `coder.screener` function identifies MATLAB data types that code generation does not support.

Analyze the MATLAB function `myfun` that contains a MATLAB table.

```
function outTable = myfun1(A)  
outTable = table(A);  
end  
  
coder.screener('myfun1');
```

The code generation readiness report indicates that table data types are not supported for code generation.



The report assigns `myfun1` a code readiness score of 3. Before generating code, you must fix the reported issues.

Tips

- Before using `coder.screener`, fix issues that the Code Analyzer identifies.
- Before generating code, use `coder.screener` to check that a function is suitable for code generation. Fix all the issues that it detects.

Alternatives

- “Run the Code Generation Readiness Tool From the Current Folder Browser”

See Also

Topics

“Functions and Objects Supported for C/C++ Code Generation — Alphabetical List”

“Functions and Objects Supported for C/C++ Code Generation — Category List”

“Code Generation Readiness Tool”

Introduced in R2012b

coder.target

Determine if code generation target is specified target

Syntax

```
tf = coder.target(target)
```

Description

`tf = coder.target(target)` returns true (1) if the code generation target is target. Otherwise, it returns false (0).

If you generate code for MATLAB classes, MATLAB computes class initial values at class loading time before code generation. If you use `coder.target` in MATLAB class property initialization, `coder.target('MATLAB')` returns true.

Examples

Use `coder.target` to parameterize a MATLAB function

Parameterize a MATLAB function so that it works in MATLAB or generated code. When the function runs in MATLAB, it calls the MATLAB function `myabsval`. The generated code, however, calls a C library function `myabsval`.

Write a MATLAB function `myabsval`.

```
function y = myabsval(u)    %#codegen
y = abs(u);
```

Generate the C library for `myabsval.m`, using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib myabsval -args {0.0}
```

codegen creates the library `myabsval.lib` and header file `myabsval.h` in the folder `/codegen/lib/myabsval`. It also generates the functions `myabsval_initialize` and `myabsval_terminate` in the same folder.

Write a MATLAB function to call the generated C library function using `coder.ceval`.

```
function y = callmyabsval %#codegen
y = -2.75;
% Check the target. Do not use coder.ceval if callmyabsval is
% executing in MATLAB
if coder.target('MATLAB')
    % Executing in MATLAB, call function myabsval
    y = myabsval(y);
else
    % Executing in the generated code.
    % Call the initialize function before calling the
    % C function for the first time
    coder.ceval('myabsval_initialize');

    % Call the generated C library function myabsval
    y = coder.ceval('myabsval', y);

    % Call the terminate function after
    % calling the C function for the last time
    coder.ceval('myabsval_terminate');
end
```

Convert `callmyabsval.m` to the MEX function `callmyabsval_mex`.

```
codegen -config:mex callmyabsval codegen/lib/myabsval/myabsval.lib...
        codegen/lib/myabsval/myabsval.h
```

Run the MATLAB function `callmyabsval`.

```
callmyabsval

ans =

    2.7500
```

Run the MEX function `callmyabsval_mex` which calls the library function `myabsval`.

```
callmyabsval_mex
```

```
ans =  
  
    2.7500
```

Input Arguments

target — code generation target

'MATLAB' | 'MEX' | 'Sfun' | 'Rtw' | 'HDL' | 'Custom'

Code generation target, specified as a character vector or a string scalar. Specify one of these targets.

'MATLAB'	Running in MATLAB (not generating code)
'MEX'	Generating a MEX function
'Sfun'	Simulating a Simulink model
'Rtw'	Generating a LIB, DLL, or EXE target
'HDL'	Generating an HDL target
'Custom'	Generating a custom target

Example: `tf = coder.target('MATLAB')`

Example: `tf = coder.target("MATLAB")`

See Also

`coder.ceval`

Introduced in R2011a

coder.unroll

Unroll `for`-loop by making a copy of the loop body for each loop iteration

Syntax

```
coder.unroll()  
coder.unroll(flag)  
for i = coder.unroll(range)  
for i = coder.unroll(range, flag)
```

Description

`coder.unroll()` unrolls a `for`-loop. The `coder.unroll` call must be on a line by itself immediately preceding the `for`-loop that it unrolls.

Instead of producing a `for`-loop in the generated code, loop unrolling produces a copy of the `for`-loop body for each loop iteration. In each iteration, the loop index becomes constant. To unroll a loop, the code generator must be able to determine the bounds of the `for`-loop.

For small, tight loops, unrolling can improve performance. However, for large loops, unrolling can increase code generation time significantly and generate inefficient code.

`coder.unroll` is ignored outside of code generation.

`coder.unroll(flag)` unrolls a `for`-loop if `flag` is `true`. `flag` is evaluated at code generation time. The `coder.unroll` call must be on a line by itself immediately preceding the `for`-loop that it unrolls.

`for i = coder.unroll(range)` is a legacy syntax that generates the same code as `coder.unroll()`.

`for i = coder.unroll(range, flag)` is a legacy syntax that generates the same code as `coder.unroll(flag)`.

Examples

Unroll a `for`-loop

To produce copies of a `for`-loop body in the generated code, use `coder.unroll`.

In one file, write the entry-point function `call_getrand` and a local function `getrand`. `getrand` unrolls a `for`-loop that assigns random numbers to an `n`-by-1 array. `call_getrand` calls `getrand` with the value 3.

```
function z = call_getrand
    %#codegen
    z = getrand(3);
end

function y = getrand(n)
    coder.inline('never');
    y = zeros(n, 1);
    coder.unroll();
    for i = 1:n
        y(i) = rand();
    end
end
```

Generate a static library.

```
codegen -config:lib call_getrand -report
```

In the generated code, the code generator produces a copy of the `for`-loop body for each of the three loop iterations.

```
static void getrand(double y[3])
{
    y[0] = b_rand();
    y[1] = b_rand();
    y[2] = b_rand();
}
```

Control `for`-loop Unrolling with Flag

Control loop unrolling by using `coder.unroll` with the `flag` argument.

In one file, write the entry-point function `call_getrand_unrollflag` and a local function `getrand_unrollflag`. When the number of loop iterations is less than 10, `getrand_unrollflag` unrolls the for-loop. `call_getrand` calls `getrand` with the value 50.

```
function z = call_getrand_unrollflag
    %#codegen
    z = getrand_unrollflag(50);
end

function y = getrand_unrollflag(n)
    coder.inline('never');
    unrollflag = n < 10;
    y = zeros(n, 1);
    coder.unroll(unrollflag)
    for i = 1:n
        y(i) = rand();
    end
end
```

Generate a static library.

```
codegen -config:lib call_getrand_unrollflag -report
```

The number of iterations is not less than 10. Therefore, the code generator does not unroll the for-loop. It produces a for-loop in the generated code.

```
static void getrand_unrollflag(double y[50])
{
    int i;
    for (i = 0; i < 50; i++) {
        y[i] = b_rand();
    }
}
```

Use Legacy Syntax to Unroll for-Loop

```
function z = call_getrand
    %#codegen
    z = getrand(3);
end

function y = getrand(n)
```

```
coder.inline('never');
y = zeros(n, 1);
for i = coder.unroll(1:n)
    y(i) = rand();
end
end
```

Use Legacy Syntax to Control `for`-Loop Unrolling

```
function z = call_getrand_unrollflag
    %#codegen
    z = getrand_unrollflag(50);
end

function y = getrand_unrollflag(n)
    coder.inline('never');
    unrollflag = n < 10;
    y = zeros(n, 1);
    for i = coder.unroll(1:n, unrollflag)
        y(i) = rand();
    end
end
```

Input Arguments

flag — Indicates whether to unroll the `for`-loop

true (default) | false

When `flag` is true, the code generator unrolls the `for`-loop. When `flag` is false, the code generator produces a `for`-loop in the generated code. `flag` is evaluated at code generation time.

Tips

Sometimes, the code generator unrolls a `for`-loop even though you do not use `coder.unroll`. For example, if a `for`-loop indexes into a heterogeneous cell array or into `varargin` or `varargout`, the code generator unrolls the loop. By unrolling the loop, the code generator can determine the value of the index for each loop iteration. The code generator uses heuristics to determine when to unroll a `for`-loop. If the heuristics fail to

identify that unrolling is warranted, or if the number of loop iterations exceeds a limit, code generation fails. In these cases, you can force loop unrolling by using `coder.unroll`. See “Nonconstant Index into varargin or varargout in a for-Loop”.

See Also

`coder.inline`

Topics

“Unroll for-Loops”

“Nonconstant Index into varargin or varargout in a for-Loop”

Introduced in R2011a

coder.updateBuildInfo

Update build information object RTW.BuildInfo

Syntax

```
coder.updateBuildInfo('addCompileFlags', options)
coder.updateBuildInfo('addLinkFlags', options)
coder.updateBuildInfo('addDefines', options)
coder.updateBuildInfo( ____, group)
```

```
coder.updateBuildInfo('addLinkObjects', filename, path)
coder.updateBuildInfo('addLinkObjects', filename, path, priority,
precompiled)
coder.updateBuildInfo('addLinkObjects', filename, path, priority,
precompiled, linkonly)
coder.updateBuildInfo( ____, group)
```

```
coder.updateBuildInfo('addNonBuildFiles', filename)
coder.updateBuildInfo('addSourceFiles', filename)
coder.updateBuildInfo('addIncludeFiles', filename)
coder.updateBuildInfo( ____, path)
coder.updateBuildInfo( ____, path, group)
```

```
coder.updateBuildInfo('addSourcePaths', path)
coder.updateBuildInfo('addIncludePaths', path)
coder.updateBuildInfo( ____, group)
```

Description

`coder.updateBuildInfo('addCompileFlags', options)` adds compiler options to the build information object.

`coder.updateBuildInfo('addLinkFlags', options)` adds link options to the build information object.

`coder.updateBuildInfo('addDefines', options)` adds preprocessor macro definitions to the build information object.

`coder.updateBuildInfo(____, group)` assigns a group name to options for later reference.

`coder.updateBuildInfo('addLinkObjects', filename, path)` adds a link object from a file to the build information object.

`coder.updateBuildInfo('addLinkObjects', filename, path, priority, precompiled)` specifies if the link object is precompiled.

`coder.updateBuildInfo('addLinkObjects', filename, path, priority, precompiled, linkonly)` specifies if the object is to be built before being linked or used for linking alone. If the object is to be built, it specifies if the object is precompiled.

`coder.updateBuildInfo(____, group)` assigns a group name to the link object for later reference.

`coder.updateBuildInfo('addNonBuildFiles', filename)` adds a nonbuild-related file to the build information object.

`coder.updateBuildInfo('addSourceFiles', filename)` adds a source file to the build information object.

`coder.updateBuildInfo('addIncludeFiles', filename)` adds an include file to the build information object.

`coder.updateBuildInfo(____, path)` adds the file from specified path.

`coder.updateBuildInfo(____, path, group)` assigns a group name to the file for later reference.

`coder.updateBuildInfo('addSourcePaths', path)` adds a source file path to the build information object.

`coder.updateBuildInfo('addIncludePaths', path)` adds an include file path to the build information object.

`coder.updateBuildInfo(____, group)` assigns a group name to the path for later reference.

Examples

Add Multiple Compiler Options

Add the compiler options `-Zi` and `-Wall` during code generation for function, `func`.

Anywhere in the MATLAB code for `func`, add the following line:

```
coder.updateBuildInfo('addCompileFlags','-Zi -Wall');
```

Generate code for `func` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport func
```

Add Source File Name

Add a source file to the project build information while generating code for a function, `calc_factorial`.

- 1 Write a header file `fact.h` that declares a C function `factorial`.

```
double factorial(double x);
```

`fact.h` will be included as a header file in generated code. This inclusion ensures that the function is declared before it is called.

Save the file in the current folder.

- 2 Write a C file `fact.c` that contains the definition of `factorial`. `factorial` calculates the factorial of its input.

```
#include "fact.h"
```

```
double factorial(double x)
{
    int i;
    double fact = 1.0;
    if (x == 0 || x == 1) {
        return 1.0;
    } else {
        for (i = 1; i <= x; i++) {
```

```

        fact *= (double)i;
    }
    return fact;
}
}

```

fact.c is used as a source file during code generation.

Save the file in the current folder.

- 3 Write a MATLAB function `calc_factorial` that uses `coder.ceval` to call the external C function `factorial`.

Use `coder.updateBuildInfo` with option `'addSourceFiles'` to add the source file `fact.c` to the build information. Use `coder.cinclude` to include the header file `fact.h` in the generated code.

```

function y = calc_factorial(x) %#codegen

    coder.cinclude('fact.h');
    coder.updateBuildInfo('addSourceFiles', 'fact.c');

    y = 0;
    y = coder.ceval('factorial', x);

```

- 4 Generate code for `calc_factorial` using the `codegen` command.

```

codegen -config:dll -launchreport calc_factorial -args 0

```

Add Link Object

Add a link object `LinkObj.lib` to the build information while generating code for a function `func`. For this example, you must have a link object `LinkObj.lib` saved in a local folder, for example, `c:\Link_Objects`.

Anywhere in the MATLAB code for `func`, add the following lines:

```

libPriority = '';
libPreCompiled = true;
libLinkOnly = true;
libName = 'LinkObj.lib';
libPath = 'c:\Link_Objects';

```

```
coder.updateBuildInfo('addLinkObjects', libName, libPath, ...  
    libPriority, libPreCompiled, libLinkOnly);
```

Generate a MEX function for `func` using the `codegen` command. Open the Code Generation Report.

```
codegen -launchreport func
```

Add Include Paths

Add an include path to the build information while generating code for a function, `adder`. Include a header file, `adder.h`, existing on the path.

When header files do not reside in the current folder, to include them, use this method:

- 1 Write a header file `mysum.h` that contains the declaration for a C function `mysum`.

```
double mysum(double, double);
```

Save it in a local folder, for example `c:\coder\myheaders`.

- 2 Write a C file `mysum.c` that contains the definition of the function `mysum`.

```
#include "mysum.h"
```

```
double mysum(double x, double y)  
{  
    return(x+y);  
}
```

Save it in the current folder.

- 3 Write a MATLAB function `adder` that adds the path `c:\coder\myheaders` to the build information.

Use `coder.cinclude` to include the header file `mysum.h` in the generated code.

```
function y = adder(x1, x2) %#codegen  
    coder.updateBuildInfo('addIncludePaths', 'c:\coder\myheaders');  
    coder.updateBuildInfo('addSourceFiles', 'mysum.c');  
    %Include the source file containing C function definition  
    coder.cinclude('mysum.h');
```

```

y = 0;
if coder.target('MATLAB')
    % This line ensures that the function works in MATLAB
    y = x1 + x2;
else
    y = coder.ceval('mysum', x1, x2);
end
end
end

```

- 4 Generate code for adder using the codegen command.

```
codegen -config:lib -launchreport adder -args {0,0}
```

Input Arguments

options — Build options

character vector | string scalar

Build options, specified as a character vector or string scalar. The value must be a compile-time constant.

Depending on the leading argument, options specifies the relevant build options to be added to the project's build information.

Leading Argument	Values in options
'addCompileFlags'	Compiler options
'addLinkFlags'	Link options
'addDefines'	Preprocessor macro definitions

The function adds the options to the end of an option vector.

Example: `coder.updateBuildInfo('addCompileFlags','-Zi -Wall')`

group — Group name

character vector | string scalar

Name of user-defined group, specified as a character vector or string scalar. The value must be a compile-time constant.

The group option assigns a group name to the parameters in the second argument.

Leading Argument	Second Argument	Parameters Named by group
'addCompileFlags'	options	Compiler options
'addLinkFlags'	options	Link options
'addLinkObjects'	filename	Name of file containing linkable objects
'addNonBuildFiles'	filename	Name of nonbuild-related file
'addSourceFiles'	filename	Name of source file
'addSourcePaths'	path	Name of source file path

You can use `group` to:

- Document the use of specific parameters.
- Retrieve or apply multiple parameters together as one group.

filename — File name

character vector | string scalar

File name, specified as a character vector or string scalar. The value must be a compile-time constant.

Depending on the leading argument, `filename` specifies the relevant file to be added to the project's build information.

Leading Argument	File Specified by filename
'addLinkObjects'	File containing linkable objects
'addNonBuildFiles'	Nonbuild-related file
'addSourceFiles'	Source file

The function adds the file name to the end of a file name vector.

path — Full path name

character vector | string scalar

Full path name, specified as a character vector or string scalar. The value must be a compile-time constant.

Depending on the leading argument, `path` specifies the relevant path name to be added to the project's build information.

Leading Argument	Path Specified by <code>path</code>
'addLinkObjects'	Path to linkable objects
'addNonBuildFiles'	Path to nonbuild-related files
'addSourceFiles', 'addSourcePaths'	Path to source files

The function adds the path to the end of a path name vector.

priority — Relative priority of link object

''

Priority of link objects.

This feature applies only when several link objects are added. Currently, only a single link object file can be added for every `coder.updateBuildInfo` statement. Therefore, this feature is not available for use.

To use the succeeding arguments, include '' as a placeholder argument.

precompiled — Variable indicating if link objects are precompiled

logical value

Variable indicating if the link objects are precompiled, specified as a logical value. The value must be a compile-time constant.

If the link object has been prebuilt for faster compiling and linking and exists in a specified location, specify `true`. Otherwise, the MATLAB Coder build process creates the link object in the build folder.

If `linkonly` is set to `true`, this argument is ignored.

Data Types: `logical`

linkonly — Variable indicating if objects must be used for linking only

logical value

Variable indicating if objects must be used for linking only, specified as a logical value. The value must be a compile-time constant.

If you want that the MATLAB Coder build process must not build or generate rules in the makefile for building the specified link object, specify `true`. Instead, when linking

the final executable, the process should just include the object. Otherwise, rules for building the link object are added to the makefile.

You can use this argument to incorporate link objects for which source files are not available.

If `linkonly` is set to `true`, the value of `precompiled` is ignored.

Data Types: `logical`

See Also

Introduced in R2013b

coder.varsize

Package: coder

Declare variable-size array

Syntax

```
coder.varsize('var1', 'var2', ...)
coder.varsize('var1', 'var2', ..., ubound)
coder.varsize('var1', 'var2', ..., ubound, dims)
coder.varsize('var1', 'var2', ..., [], dims)
```

Description

`coder.varsize('var1', 'var2', ...)` declares one or more variables as variable-size data, allowing subsequent assignments to extend their size. Each `'varn'` is the name of a variable or structure field enclosed in quotes. If the structure field belongs to an array of structures, use colon (`:`) as the index expression to make the field variable-size for all elements of the array. For example, the expression `coder.varsize('data(:).A')` declares that the field `A` inside each element of `data` is variable sized.

`coder.varsize('var1', 'var2', ..., ubound)` declares one or more variables as variable-size data with an explicit upper bound specified in `ubound`. The argument `ubound` must be a constant, integer-valued vector of upper bound sizes for every dimension of each `'varn'`. If you specify more than one `'varn'`, each variable must have the same number of dimensions.

`coder.varsize('var1', 'var2', ..., ubound, dims)` declares one or more variables as variable size with an explicit upper bound and a mix of fixed and varying dimensions specified in `dims`. The argument `dims` is a logical vector, or double vector containing only zeros and ones. Dimensions that correspond to zeros or `false` in `dims` have fixed size; dimensions that correspond to ones or `true` vary in size. If you specify more than one variable, each fixed dimension must have the same value across all `'varn'`.

`coder.varsize('var1', 'var2', ..., [], dims)` declares one or more variables as variable size with a mix of fixed and varying dimensions. The empty vector `[]` means that you do not specify an explicit upper bound.

When you do *not* specify *ubound*, the upper bound is computed for each `'varn'` in generated code.

When you do *not* specify *dims*, dimensions are assumed to be variable except the singleton ones. A singleton dimension is a dimension for which `size(A, dim) = 1`.

You must add the `coder.varsize` declaration before each `'varn'` is used (read). You can add the declaration before the first assignment to each `'varn'`. However, for a cell array element, the `coder.varsize` declaration must follow the first assignment to the element. For example:

```
...
x = cell(3, 3);
x{1} = [1 2];
coder.varsize('x{1}');
...
```

You cannot use `coder.varsize` outside the MATLAB code intended for code generation. For example, the following code does not declare the variable, `var`, as variable-size data:

```
coder.varsize('var',10);
codegen -config:lib MyFile -args var
```

Instead, include the `coder.varsize` statement inside `MyFile` to declare `var` as variable-size data.

Examples

Develop a Simple Stack That Varies in Size up to 32 Elements as You Push and Pop Data at Run Time.

Write primary function `test_stack` to issue commands for pushing data on and popping data from a stack.

```
function test_stack %#codegen
    % The directive %#codegen indicates that the function
```

```

% is intended for code generation
stack('init', 32);
for i = 1 : 20
    stack('push', i);
end
for i = 1 : 10
    value = stack('pop');
    % Display popped value
    value
end
end

```

Write local function `stack` to execute the push and pop commands.

```

function y = stack(command, varargin)
    persistent data;
    if isempty(data)
        data = ones(1,0);
    end
    y = 0;
    switch (command)
    case {'init'}
        coder.varsize('data', [1, varargin{1}], [0 1]);
        data = ones(1,0);
    case {'pop'}
        y = data(1);
        data = data(2:size(data, 2));
    case {'push'}
        data = [varargin{1}, data];
    otherwise
        assert(false, ['Wrong command: ', command]);
    end
end

```

The variable `data` is the stack. The statement `coder.varsize('data', [1, varargin{1}], [0 1])` declares that:

- `data` is a row vector
- Its first dimension has a fixed size
- Its second dimension can grow to an upper bound of 32

Generate a MEX function for `test_stack`:

```
codegen -config:mex test_stack
```

codegen generates a MEX function in the current folder.

Run `test_stack_mex` to get these results:

```
value =  
    20
```

```
value =  
    19
```

```
value =  
    18
```

```
value =  
    17
```

```
value =  
    16
```

```
value =  
    15
```

```
value =  
    14
```

```
value =  
    13
```

```
value =  
    12
```

```
value =  
    11
```

At run time, the number of items in the stack grows from zero to 20, and then shrinks to 10.

Declare a Variable-Size Structure Field.

Write a function `struct_example` that declares an array `data`, where each element is a structure that contains a variable-size field:

```

function y=struct_example() %#codegen

    d = struct('values', zeros(1,0), 'color', 0);
    data = repmat(d, [3 3]);
    coder.varsize('data(:).values');

    for i = 1:numel(data)
        data(i).color = rand-0.5;
        data(i).values = 1:i;
    end

    y = 0;
    for i = 1:numel(data)
        if data(i).color > 0
            y = y + sum(data(i).values);
        end;
    end
end

```

The statement `coder.varsize('data(:).values')` marks as variable-size the field values inside each element of the matrix `data`.

Generate a MEX function for `struct_example`:

```
codegen -config:mex struct_example
```

Run `struct_example`.

Each time you run `struct_example` you get a different answer because the function loads the array with random numbers.

Make a Cell Array Variable-Size

Write the function `make_varsz_cell` that defines a local cell array variable `c` whose elements have the same class, but different sizes. Use `coder.varsize` to indicate that `c` has variable-size.

```

function y = make_varsz_cell()
    c = {1 [2 3]};
    coder.varsize('c', [1 3], [0 1]);
    y = c;
end

```

Generate a C static library.

```
codegen -config:lib make_varsz_cell -report
```

In the report, view the MATLAB variables.

`c` is a 1x:3 homogeneous cell array whose elements are 1x:2 double.

Make the Elements of a Cell Array Variable-Size

Write the function `mycell` that defines a local cell array variable `c`. Use `coder.versize` to make the elements of `c` variable-size.

```
function y = mycell()
c = {1 2 3};
coder.versize('c{:}', [1 5], [0 1]);
y = c;
end
```

Generate a C static library.

```
codegen -config:lib mycell -report
```

In the report, view the MATLAB variables.

The elements of `c` are 1-by-:5 arrays of doubles.

Limitations

- If you use the `cell` function to create a cell array, you cannot use `coder.versize` with that cell array.
- If you use `coder.versize` with a cell array element, the `coder.versize` declaration must follow the first assignment to the element. For example:

```
...
x = cell(3, 3);
x{1} = [1 2];
coder.versize('x{1}');
...
```

- You cannot use `coder. varsize` with global variables.
- You cannot use `coder. varsize` with MATLAB class properties.
- You cannot use `coder. varsize` with string scalars.

Tips

- If you use input variables (or result of a computation using input variables) to specify the size of an array, it is declared as variable-size in the generated code. Do not use `coder. varsize` on the array again, unless you also want to specify an upper bound for its size.
- Using `coder. varsize` on an array without explicit upper bounds causes dynamic memory allocation of the array. This dynamic memory allocation can reduce the speed of generated code. To avoid dynamic memory allocation, use the syntax `coder. varsize('var1', 'var2', ..., ubound)` to specify an upper bound for the array size (if you know it in advance).
- A cell array can be variable size only if it is homogeneous. When you use `coder. varsize` with a cell array, the code generator tries to make the cell array homogeneous. It tries to find a class and size that apply to all elements of the cell array. For example, if the first element is double and the second element is 1x2 double, all elements can be represented as 1x2 double. If the code generator cannot find a common class and size, code generation fails. For example, suppose that the first element of a cell array is char and the second element is double. The code generator cannot find a class that can represent both elements.

See Also

`size`

Topics

“Code Generation for Variable-Size Arrays”

“Incompatibilities with MATLAB in Variable-Size Support for Code Generation”

“Code Generation for Cell Arrays”

Introduced in R2011a

coder.wref

Pass argument as write-only reference

Syntax

```
coder.wref(arg)
```

Description

`coder.wref(arg)` passes `arg` by reference to an external C/C++ function that you call by using `coder.ceval`. Use `coder.wref` inside a `coder.ceval` call only. The code generator assumes that the variable that you pass in `coder.wref` is write-only. The C/C++ function must write to this variable. If the variable is a vector or an array, the C/C++ function must write to every element of the variable. If the MATLAB code initializes `arg` before the `coder.ceval` call, the code generator might remove the initial values. To avoid unpredictable results, the C/C++ function must not read the variable before writing to it.

Use a separate `coder.wref` construct for each argument that you pass by reference to the function.

Examples

Pass Array by Reference as Write-Only

Pass a variable by reference to a C function that initializes an array.

Suppose that you have a C function `init_array`.

```
void init_array(double* array, int numel) {
    for(int i = 0; i < numel; i++) {
        array[i] = 42;
    }
}
```

The C function prototype defines `array` as a pointer because it is passed by reference.

Call the C function `init_array` to initialize all elements of `y` to 42.

```
...
Y = zeros(5, 10);
coder.ceval('init_array', coder.wref(Y), int32(numel(Y)));
...
```

Pass Multiple Arguments as a Write-Only Reference

```
...
U = zeros(5, 10);
V = zeros(5, 10);
coder.ceval('my_fcn', coder.wref(U), int32(numel(U)), coder.wref(V), int32(numel(V)));
...
```

Pass Class Property as a Write-Only Reference

```
...
x = myClass;
x.prop = 1;
coder.ceval('foo', coder.wref(x.prop));
...
```

Pass Structure as a Write-Only Reference

To indicate that the structure type is defined in a C header file, use `coder.cstructname`.

Suppose that you have the C function `init_struct`.

```
double init_struct(struct MyStruct *my_struct)
{
    my_struct->f1 = 1;
    my_struct->f2 = 2;
}
```

Suppose that a C header file, `MyStruct.h`, defines a structure type named `MyStruct`.

```
...
typedef struct MyStruct
```

```
{
    double f1;
    double f2;

} MyStruct;

double init_struct(struct MyStruct *my_struct);

...
```

In your MATLAB function, pass a structure as a write-only reference to `init_struct`. To indicate that the structure type for `s` has the name `MyStruct` that is defined in a C header file `MyStruct.h`, use `coder.cstructname`.

```
...
s = struct('f1', 1, 'f2', 2);
coder.cstructname(s, 'MyStruct', 'extern', 'HeaderFile', 'MyStruct.h');
coder.ceval('init_struct', coder.wref(s));
...
```

Pass Structure Field by Reference

```
...
s = struct('s1', struct('a', [0 1]));
coder.ceval('foo', coder.wref(s.s1.a));
...
```

Input Arguments

arg — Argument to pass by reference

scalar variable | array | element of an array | structure | structure field | object property

Argument to pass by reference to an external C/C++ function. The argument cannot be a class, a System object, a cell array, or an index into a cell array.

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

Complex Number Support: Yes

Limitations

- You cannot pass these data types by reference:
 - Class or System object
 - Cell array or index into a cell array
- If a property has a get method, a set method, or validators, or is a System object property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties”.

Tips

- When you pass a structure by reference to an external C/C++ function, use `coder.cstructname` to provide the name of a C structure type that is defined in a C header file.
- In MATLAB, `coder.wref` results in an error. To parameterize your MATLAB code so that it can run in MATLAB and in generated code, use `coder.target`.
- You can use `coder.opaque` to declare variables that you pass to and from an external C/C++ function.

See Also

`coder.ceval` | `coder.cstructname` | `coder.opaque` | `coder.ref` | `coder.rref`

Introduced in R2011a

createInputDataset

Generate dataset object for root-level Inport blocks in model

Syntax

```
[inports_dataset] = createInputDataset(model)
```

Description

`[inports_dataset] = createInputDataset(model)` generates a `Simulink.SimulationData.Dataset` object from the root-level Inport blocks in a model. Signals in the generated dataset have the properties of the root inports and the corresponding ground values at model start and stop times. You can create `timetable` or `timeseries` objects for the time and values for signals for which you want to load data for simulation. The other signals use ground values.

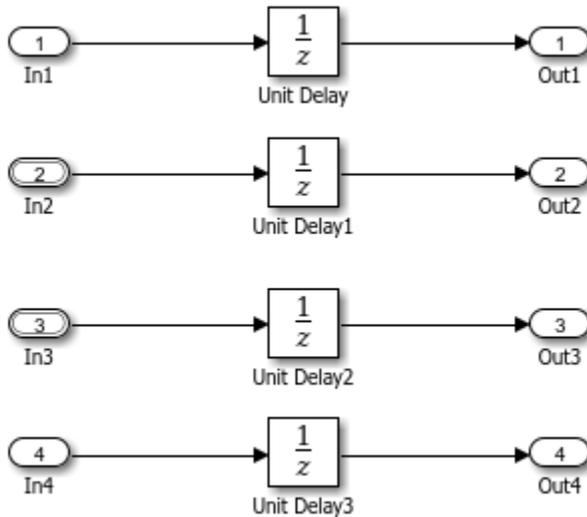
Examples

Generate and Populate Dataset for Root-Level Inport Blocks

Create a dataset with elements for the four root-level Inport blocks in a model. Use that dataset as a basis for creating a dataset to load signal data into the model.

Open the model. The `In1` block outputs a double, `In2` and `In3` each output a nonvirtual bus, and `In4` outputs an `int16`.

```
mdl = 'ex_dataset_for_inports';  
open_system(mdl)
```



Create a Dataset object for the root-level Inport blocks.

```
ds = createInputDataset mdl)
```

```
Simulink.SimulationData.Dataset '' with 4 elements
```

		Name	BlockPath
1	[1x1 timeseries]	In1	''
2	[1x1 struct]	In2	''
3	[1x1 struct]	In3	''
4	[1x1 timeseries]	In4	''

- Use braces { } to access, modify, or add elements using index.

Replace the placeholder value for the first signal in the Dataset with actual signal values that you want to load into the model.

```
ds{1} = ds{1}.delsample('Index',[1,2]);
ds{1} = ds{1}.addsample('time',[1 3 3 10]','data',[1 1 5 5]');
```

Examine the In2 signal.

```
ds{2}
```

```
ans =  
  
    struct with fields:  
  
        a: [1×1 timeseries]  
        b: [1×1 timeseries]
```

For In2, create data for bus elements a and b.

```
ds{2}.a = ds{2}.a.delsample('Index', [1,2]);  
ds{2}.a = addsample(ds{2}.a, 'time', [1:10]', 'data', [1:10]');  
ds{2}.b = timeseries((1:10)', 0.1:.1:1, 'Name', 'sig2_b');
```

For In3, specify data for element a of the bus, and use ground values for element b.

```
ds{3}.a = timeseries((1:10)', 0.1:.1:1, 'Name', 'sig3_a');
```

Plot ds. For more information, see the `Simulink.SimulationData.Dataset.plot` documentation.

```
plot(ds)
```

Set the **Input** configuration parameter to ds.

```
set_param mdl, 'LoadExternalInput', 'on';  
set_param mdl, 'ExternalInput', 'ds';
```

Tip Alternatively, you can use the Root Inport Mapper tool to set the **Input** parameter. For details, see “Map Root Inport Signal Data”.

Run the simulation. The Inport blocks use the signal data specified in ds or ground values for elements that do not have specified signal data.

```
sim mdl)
```

Input Arguments

model — Model for which to generate dataset for root-level Inport blocks
character vector | model handle

Model for which to generate a dataset with an element for each root-level Inport block, specified as a character vector or model handle.

Output Arguments

inports_dataset — Dataset with an element for each root-level Inport block

a `Simulink.SimulationData.Dataset` object

Dataset with an element for each root-level Inport block, returned as a `Simulink.SimulationData.Dataset` object.

Related Links

`Simulink.SimulationData.Dataset` MATLAB timeseries timetable “Create a Dataset Object for Root-Level Inports”

Introduced in R2017a

getHardwareImplementation

Class: `coder.BuildConfig`

Package: `coder`

Get handle of copy of hardware implementation object

Syntax

```
hw = bldcfg.getHardwareImplementation()
```

Description

`hw = bldcfg.getHardwareImplementation()` returns the handle of a copy of the hardware implementation object.

Input Arguments

bldcfg

`coder.BuildConfig` object.

Output Arguments

hw

Handle of copy of hardware implementation object.

See Also

getStdLibInfo

Class: coder.BuildConfig

Package: coder

Get standard library information

Syntax

```
[linkLibPath,linkLibExt,execLibExt,libPrefix]=  
bldcfg.getStdLibInfo()
```

Description

[linkLibPath,linkLibExt,execLibExt,libPrefix]=
bldcfg.getStdLibInfo() returns character vectors representing the:

- Standard MATLAB architecture-specific library path
- Platform-specific library file extension for use at link time
- Platform-specific library file extension for use at run time
- Standard architecture-specific library name prefix

Input Arguments

bldcfg

coder.BuildConfig object.

Output Arguments

linkLibPath

Standard MATLAB architecture-specific library path specified as a character vector. The character vector can be empty.

linkLibExt

Platform-specific library file extension for use at link time, specified as a character vector. The value is one of `'.lib'`, `'.dylib'`, `'.so'`, `''`.

execLibExt

Platform-specific library file extension for use at run time, specified as a character vector. The value is one of `'.dll'`, `'.dylib'`, `'.so'`, `''`.

linkPrefix

Standard architecture-specific library name prefix, specified as a character vector. The character vector can be empty.

getTargetLang

Class: coder.BuildConfig

Package: coder

Get target code generation language

Syntax

```
lang = bldcfg.getTargetLang()
```

Description

`lang = bldcfg.getTargetLang()` returns a character vector containing the target code generation language.

Input Arguments

bldcfg

`coder.BuildConfig` object.

Output Arguments

lang

A character vector containing the target code generation language. The value is 'C' or 'C++'.

getToolchainInfo

Class: `coder.BuildConfig`

Package: `coder`

Returns handle of copy of toolchain information object

Syntax

```
tc = bldcfg.getToolchainInfo()
```

Description

`tc = bldcfg.getToolchainInfo()` returns a handle of a copy of the toolchain information object.

Input Arguments

bldcfg

`coder.BuildConfig` object.

Output Arguments

tc

Handle of copy of toolchain information object.

See Also

isCodeGenTarget

Class: coder.BuildConfig

Package: coder

Determine if build configuration represents specified target

Syntax

```
tf = bldcfg.isCodeGenTarget(target)
```

Description

`tf = bldcfg.isCodeGenTarget(target)` returns true (1) if the code generation target of the current build configuration represents the code generation target specified by `target`. Otherwise, it returns false (0).

Input Arguments

bldcfg

coder.BuildConfig object.

target

Code generation target specified as a character vector or cell array of character vectors.

Specify	For code generation target
'rtw'	C/C++ dynamic Library, C/C++ static library, or C/C++ executable
'sfun'	S-function (Simulation)
'mex'	MEX-function

Specify `target` as a cell array of character vectors to test if the code generation target of the build configuration represents one of the targets specified in the cell array.

For example:

```
...  
mytarget = {'sfun','mex'};  
tf = bldcfg.isCodeGenTarget(mytarget);  
...
```

tests whether the build context represents an S-function target or a MEX-function target.

Output Arguments

tf

The value is true (1) if the code generation target of the build configuration represents the code generation target specified by `target`. Otherwise, the value is false (0).

See Also

`coder.target`

isMatlabHostTarget

Class: coder.BuildConfig

Package: coder

Determine if hardware implementation object target is MATLAB host computer

Syntax

```
tf = bldcfg.isMatlabHostTarget()
```

Description

`tf = bldcfg.isMatlabHostTarget()` returns true (1) if the current hardware implementation object targets the MATLAB host computer. Otherwise, it returns false (0).

Input Arguments

bldcfg

coder.BuildConfig object.

Output Arguments

tf

Value is true (1) if the current hardware implementation object targets the MATLAB host computer. Otherwise, the value is false (0).

See Also

coder.ExternalDependency.getDescriptiveName

Class: coder.ExternalDependency

Package: coder

Return descriptive name for external dependency

Syntax

```
extname = coder.ExternalDependency.getDescriptiveName (bldcfg)
```

Description

`extname = coder.ExternalDependency.getDescriptiveName (bldcfg)` returns the name that you want to associate with an “external dependency” on page 2-171. The code generator uses the external dependency name for error messages.

Input Arguments

bldcfg

`coder.BuildConfig` object. Use `coder.BuildConfig` methods to get information about the “build context” on page 2-171

You can use this information when you want to return different names based on the build context.

Output Arguments

extname

External dependency name returned as a character vector.

Examples

Return external dependency name

Define a method that always returns the same name.

```
function myextname = getDescriptiveName(~)
    myextname = 'MyLibrary'
end
```

Return external library name based on the code generation target

Define a method that uses the build context to determine the name.

```
function myextname = getDescriptiveName(context)
    if context.isMatlabHostTarget()
        myextname = 'MyLibrary_MatlabHost';
    else
        myextname = 'MyLibrary_Local';
    end
end
```

Definitions

external dependency

External code interface represented by a class derived from a `coder.ExternalDependency` class. The external code can be a library, object files, or C/C++ source.

build context

Information used by the build process including:

- Target language
- Code generation target

- Target hardware
- Build toolchain

coder.ExternalDependency.isSupportedContext

Class: coder.ExternalDependency

Package: coder

Determine if build context supports external dependency

Syntax

```
tf = coder.ExternalDependency.isSupportedContext(bldcfg)
```

Description

`tf = coder.ExternalDependency.isSupportedContext(bldcfg)` returns true (1) if you can use the “external dependency” on page 2-174 in the current “build context” on page 2-174 . You must provide this method in the class definition for a class that derives from `coder.ExternalDependency`.

If you cannot use the “external dependency” on page 2-174 in the current “build context” on page 2-174, display an error message and stop code generation. The error message must describe why you cannot use the external dependency in this build context. If the method returns false (0), the code generator uses a default error message. The default error message uses the name returned by the `getDescriptiveName` method of the `coder.ExternalDependency` class.

Use `coder.BuildConfig` methods to determine if you can use the external dependency in the current build context.

Input Arguments

bldcfg

`coder.BuildConfig` object. Use `coder.BuildConfig` methods to get information about the “build context” on page 2-174.

Output Arguments

tf

Value is true (1) if the build context supports the external dependency.

Examples

Report error when build context does not support external library

This method returns true(1) if the code generation target is a MATLAB host target. Otherwise, the method reports an error and stops code generation.

Write `isSupportedContext` method.

```
function tf = isSupportedContext(ctx)
    if ctx.isMatlabHostTarget()
        tf = true;
    else
        error('adder library not available for this target');
    end
end
```

Definitions

external dependency

External code interface represented by a class derived from `coder.ExternalDependency` class. The external code can be a library, object file, or C/C++ source.

build context

Information used by the build process including:

- Target language

- Code generation target
- Target hardware
- Build toolchain

coder.ExternalDependency.updateBuildInfo

Class: coder.ExternalDependency

Package: coder

Update build information

Syntax

```
coder.ExternalDependency.updateBuildInfo(buildInfo, bldcfg)
```

Description

`coder.ExternalDependency.updateBuildInfo(buildInfo, bldcfg)` updates the build information object whose handle is `buildInfo`. After code generation, the build information object has standard information. Use this method to provide additional information required to link to external code. Use `coder.BuildConfig` methods to get information about the “build context” on page 2-177.

You must implement this method in a subclass of `coder.ExternalDependency`.

Input Arguments

buildInfo

Handle of build information object.

bldcfg

`coder.BuildConfig` object. Use `coder.BuildConfig` methods to get information about the “build context” on page 2-177.

Limitations

- The build information method `AddIncludeFiles` has no effect in a `coder.ExternalDependency.updateBuildInfo` method.

Definitions

build context

Information used by the build process including:

- Target language
- Code generation target
- Target hardware
- Build toolchain

See Also

convertToSLDataset

Convert contents of MAT-file to Simulink.SimulationData.Dataset object

Syntax

```
success=convertToSLDataset(source,destination)
success=convertToSLDataset(source,destination,datasetname)
```

Description

`success=convertToSLDataset(source,destination)` converts the contents of a MAT-file (`source`) to a destination MAT-file (`destination`).

`success=convertToSLDataset(source,destination,datasetname)` names the dataset `datasetname`.

When converting structure signal data, the function names the signal using the value contained in the label field of the structure signal field, such as:
`mySignal.signal(1).label`.

This function ignores time expressions in `source`.

Examples

Save Signals to Dataset in file2.mat

Save signals from `file1.mat` to a dataset named `file1` in `file2.mat`.

```
success=convertToSLDataset('file1.mat','file2.mat')
```

Save Signals to Dataset Named myDataset in file2.mat

Save signals from file1.mat to a dataset named myDataset in file2.mat.

```
success=convertToSLDataset('file1.mat','file2.mat','myDataset')
```

Input Arguments

source — MAT-file

character vector

MAT-file that contains Simulink inputs.

destination — MAT-file

character vector

MAT-file to contain Simulink.SimulationData.Dataset converted from contents of source.

datasetname — Data set name

character vector

Data set name for new Simulink.SimulationData.Dataset object.

Output Arguments

success — Outcome of conversion

binary

Outcome of conversion, specified as binary:

- 1
Conversion is successful.
- 0
Conversion is not successful.

See Also

Introduced in R2016a

copy

Copy a configuration set

Syntax

```
copyCs = copy(cs)
```

Description

`copyCs = copy(cs)` returns a copy of a configuration set.

Examples

Attach New Configuration Set to a Model

Create a copy of a configuration set and attach it to a model.

Get the active configuration set for your model.

```
activeConfig = getActiveConfigSet ('vdp');
```

Copy the active configuration set.

```
develConfig = copy(activeConfig);
```

Give the copied configuration set a name.

```
develConfig.Name = 'develConfig';
```

Attach the new configuration set to the model.

```
attachConfigSet ('vdp', develConfig);
```

Input Arguments

cs — Configuration set

`ConfigSet` object

Configuration set object to copy, specified as a `ConfigSet` object.

Output Arguments

copyCs — Copy of configuration set

`ConfigSet` object

A copy of the configuration set, specified as a `ConfigSet` object.

See Also

Topics

“About Configuration Sets”

“Manage a Configuration Set”

Introduced before R2006a

createCategory

Create category of Simulink Project labels

Syntax

```
createCategory(proj, categoryName)
createCategory(proj, categoryName, dataType)
createCategory(proj, categoryName, dataType, single-valued)
```

Description

`createCategory(proj, categoryName)` creates a new category of labels `categoryName` in the project `proj`.

`createCategory(proj, categoryName, dataType)` specifies the class of data to store in labels of the new category.

`createCategory(proj, categoryName, dataType, single-valued)` specifies a single-valued category, where you can attach only one label from the category to a file. If you do not specify `single-valued`, then you can attach multiple labels from the category to a file.

Examples

Create a New Category of Labels for File Ownership

Create a new category of labels for file ownership, and attach a new label and label data to a file.

Open the `airframe` project and create a project object.

```
sldemo_slproject_airframe
proj = simulinkproject;
```

Create a new category of labels, called `Engineers`, to denote file ownership in a project. These labels have the `char` datatype for attaching character vector data.

```
createCategory(proj, 'Engineers', 'char');
```

Use `findCategory` to get the new category.

```
engineersCategory = findCategory(proj, 'Engineers');
```

Create labels in the new category.

```
createLabel(engineersCategory, 'Tom');  
createLabel(engineersCategory, 'Dick')  
createLabel(engineersCategory, 'Harry')
```

Attach one of the new labels to a file in the project.

```
myfile = findFile(proj, 'models/AnalogControl.mdl')  
addLabel(myfile, 'Engineers', 'Tom');
```

Get the label and add data.

```
label = findLabel(file, 'Engineers', 'Tom');  
label.Data = 'Maintenance responsibility';  
disp(label)
```

Label with properties:

```
File: [1x80 char]  
Data: 'Maintenance responsibility'  
DataType: 'char'  
Name: 'Tom'  
CategoryName: 'Engineers'
```

Create a New Category of Labels with Datatype Double

Open the airframe project and create a project object.

```
sldemo_slproject_airframe  
proj = simulinkproject;
```

Create a new category of labels.

```
createCategory(proj, 'Coverage', 'double')
```

```
category =
    Category with properties:
        Name: 'Coverage'
        DataType: 'double'
        LabelDefinitions: []
```

Find out what you can do with the new category.

```
category = findCategory(proj, 'Coverage');
methods(category)
```

Methods for class slproject.Category:

```
findLabel  removeLabel  createLabel
```

Create a Single-Valued Category

Open the airframe project and create a project object.

```
sldemo_slproject_airframe
proj = simulinkproject;
```

Create a category of labels for file ownership, and specify single-valued to restrict only one label in the category per file.

```
createCategory(proj, 'Engineers', 'char', 'single-valued');
```

Input Arguments

proj — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

categoryName — Name of category

character vector

Name of the category of labels to create, specified as a character vector.

dataType — Class of data to store in labels

character vector

The class of data to store in labels in the new category, specified as a character vector.

single-valued — Single-valued category

character vector

Single-valued category, specified as a character vector. Single-valued means you can attach only one label from the category to a file. If you do not specify single-valued, then you can attach multiple labels from the category to a file.

Tips

After you create a new category, you can create labels in the new category. See `createLabel`.

See Also

Functions

`createLabel` | `simulinkproject`

Introduced in R2013a

createLabel

Define Simulink Project label

Syntax

```
createLabel(category, newLabelName)
```

Description

`createLabel(category, newLabelName)` creates a new label, `newLabelName`, in a category. Use this syntax if you previously got a category by accessing a `Categories` property, e.g., using a command like `proj.Categories(1)`.

Use `addLabel` instead to create and attach a new label in an existing category using a single step.

Use `createCategory` first if you want to make a new category of labels.

Examples

Create a New Label

Open the airframe project and create a project object.

```
sldemo_slproject_airframe  
proj = simulinkproject;
```

Examine the first existing category.

```
cat = proj.Categories(1)
```

```
cat =
```

```
Category with properties:
```

```
Name: 'Classification'  
DataType: 'none'  
LabelDefinitions: [1x8 slproject.LabelDefinition]
```

Define a new label in the category.

```
createLabel(cat, 'Future');
```

Create a New Category of Labels for File Ownership

Open the airframe project and create a project object.

```
sldemo_slproject_airframe  
proj = simulinkproject;
```

Create creates a new category of labels called Engineers which can be used to denote file ownership in a project. These labels have the char datatype for attaching character vector data.

```
createCategory(proj, 'Engineers', 'char');
```

Use findCategory to get the new category.

```
engineersCategory = findCategory(proj, 'Engineers');
```

Create labels in the new category.

```
createLabel(engineersCategory, 'Tom');  
createLabel(engineersCategory, 'Dick');  
createLabel(engineersCategory, 'Harry');
```

Attach one of the new labels to a file in the project.

```
myfile = findFile(proj, 'models/AnalogControl.mdl')  
addLabel(myfile, 'Engineers', 'Tom');
```

Get the label and add data.

```
label = findLabel(myfile, 'Engineers', 'Tom');  
label.Data = 'Maintenance responsibility';  
disp(label)
```

Label with properties:

```
File: [1x80 char]
Data: 'Maintenance responsibility'
DataType: 'char'
Name: 'Tom'
CategoryName: 'Engineers'
```

Input Arguments

category — Category

category object

Category for the new label, specified as a category object. Get the category by accessing a `Categories` property, e.g. with a command like `proj.Categories(1)`, or use `findCategory`. To create a new category, use `createCategory`.

newLabelName — The name of the new label to define

character vector

The name of the new label to define, specified as a character vector.

See Also

Functions

`addLabel` | `createCategory`

Introduced in R2013a

addStartupFile

Add startup file to project

Syntax

```
addStartupFile(proj, file)
```

Description

`addStartupFile(proj, file)` adds a startup file to the project `proj`. Startup files automatically run (.m and .p files), load (.mat files), or open (Simulink models) when you open the project.

Examples

Add a Startup File

Open the airframe project and create a project object.

```
sldemo_slproject_airframe  
proj = simulinkproject;
```

Create a new file.

```
filepath = fullfile(proj.RootFolder, 'new_model.slx')  
    new_system('new_model');  
    save_system('new_model', filepath)
```

Add the new model to the project.

```
projectFile = addFile(proj, filepath)
```

Automatically open the model when the project opens, by making it a startup file.

```
addStartupFile(proj, filepath);
```

Input Arguments

proj — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

file — Path of file

character vector

Path of the file to add relative to the project root folder, including the file extension, specified as a character vector. The file must be within the root folder.

Example: 'models/myModelName.slx'

See Also

`simulinkproject`

Introduced in R2017b

addShutdownFile

Add shutdown file to project

Syntax

```
addShutdownFile(proj, file)
```

Description

`addShutdownFile(proj, file)` adds a shutdown file to the project `proj`. When you close the project, it runs the shutdown file automatically. Use the shutdown list to specify executable MATLAB code to run as the project shuts down.

Examples

Add a Shutdown File

Open the airframe project and create a project object.

```
sldemo_slproject_airframe  
proj = simulinkproject;
```

Specify executable MATLAB code to run as the project shuts down.

```
filepath = fullfile('utilities', 'rebuild_s_functions.m');
```

Automatically run the file when the project closes, by making it a shutdown file.

```
addShutdownFile(project, filepath);
```

Input Arguments

proj — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

file — Path of file

character vector

Path of the MATLAB file to add relative to the project root folder, including the file extension, specified as a character vector. The file must be within the root folder.

Example: 'utilities/myscript.m'

See Also

`simulinkproject`

Introduced in R2017b

removeStartupFile

Remove startup file from project startup list

Syntax

```
removeStartupFile(proj, file)
```

Description

`removeStartupFile(proj, file)` removes the startup file from the startup list in the project `proj`.

Examples

Remove a Startup File

Open the airframe project and create a project object.

```
sldemo_slproject_airframe  
proj = simulinkproject;
```

Create a new file.

```
filepath = fullfile(proj.RootFolder, 'new_model.slx')  
new_system('new_model');  
save_system('new_model', filepath)
```

Add the new model to the project.

```
projectFile = addFile(proj, filepath)
```

Automatically open the model when the project opens, by making it a startup file.

```
addStartupFile(proj, filepath);
```

Remove the startup file.

```
removeStartupFile(proj, filepath);
```

Input Arguments

proj — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

file — Path of file

character vector

Path of the file to add relative to the project root folder, including the file extension, specified as a character vector. The file must be within the root folder.

Example: 'models/myModelName.slx'

See Also

`simulinkproject`

Introduced in R2017b

removeShutdownFile

Remove shutdown file from project shutdown list

Syntax

```
removeShutdownFile(proj, file)
```

Description

`removeShutdownFile(proj, file)` removes the shutdown file from the shutdown list in the project `proj`.

Examples

Remove a Shutdown File

Open the airframe project and create a project object.

```
sldemo_slproject_airframe  
proj = simulinkproject;
```

Specify executable MATLAB code to run as the project shuts down.

```
filepath = fullfile('utilities', 'rebuild_s_functions.m');
```

Automatically run the file when the project closes, by making it a shutdown file.

```
addShutdownFile(project, filepath);
```

Remove the shutdown file.

```
removeShutdownFile(project, filepath);
```

Input Arguments

proj — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

file — Path of file

character vector

Path of the MATLAB file to add relative to the project root folder, including the file extension, specified as a character vector. The file must be within the root folder.

Example: 'utilities/myscript.m'

See Also

`simulinkproject`

Introduced in R2017b

delete_block

Delete blocks from Simulink system

Syntax

```
delete_block(blockArg)
```

Description

`delete_block(blockArg)` deletes the specified blocks from a system. Open the system before you delete blocks.

Examples

Delete Block Using Full Path Name

Delete the block Mu from the vdp system.

```
open_system('vdp')  
delete_block('vdp/Mu')
```

Delete Block Using Block Handle

Delete the block Out2 from the vdp system using the block handle.

Open the vdp system.

```
open_system('vdp')
```

Interactively select the block Out1. Get the block's handle and assign it to the variable `Out1_handle`. Delete the block using the handle.

```
Out1_handle = get_param(gcb, 'Handle');  
delete_block(Out1_handle)
```

Delete Blocks Using Vector of Handles

Delete three blocks from the vdp system.

Open the vdp system. Add three blocks and assign their handles to variables.

```
open_system('vdp')  
Constant_handle = add_block('built-in/Constant', 'vdp/MyConstant');  
Gain_handle = add_block('built-in/Gain', 'vdp/MyGain');  
Outputport_handle = add_block('built-in/Outputport', 'vdp/MyOutputport');
```

Delete the blocks you added using a vector of handles.

```
delete_block([Constant_handle Gain_handle Outputport_handle])
```

Input Arguments

blockArg — Blocks to delete

full path name | handle | vector of handles | 1-D cell array of handles or block path names

Blocks to delete, specified as the full block path name, a handle, a vector of handles, or a 1-D cell array of handles or block path names.

Example: 'vdp/Mu'

Example: [handle1 handle2]

Example: {'vdp/Mu' 'vdp/Out1' 'vdp/Out2'}

See Also

`add_block`

Introduced before R2006a

delete_line

Delete line from Simulink model

Syntax

```
delete_line(sys,out,in)
delete_line(sys,point)
delete_line(lineHandle)
```

Description

`delete_line(sys,out,in)` deletes the line from the model or subsystem `sys` that connects the output port `out` to the input port `in`.

`delete_line(sys,point)` deletes the line that includes the point `point`.

`delete_line(lineHandle)` deletes the line using the line handle.

Examples

Remove Line Using Block Port Names

For the model `vdp`, remove the line connecting the Product block with the Gain block.

```
load_system('vdp');
delete_line('vdp','Product/1','Mu/1');
```

Remove Line Using Line Handle

For the model `vdp`, remove a line using the line handle. You can get the line handle using different techniques.

```
load_system('vdp');
h = get_param('vdp/Mu', 'LineHandles');
delete_line(h.Outputport(1));
```

Get a line handle when you create the line. Delete the line using that handle.

```
a = add_line('vdp', 'Mu/1', 'Sum/2');
delete_line(a)
```

Delete a Line Using a Point

You can use a point on the line to delete the whole line.

Find the port coordinates for the block Mu in the model vdp.

```
open_system('vdp');
mu = get_param('vdp/Mu', 'PortConnectivity');
mu.Position
```

```
ans =
    190    150
```

```
ans =
    225    150
```

The line that connects the Mu block to the Sum block starts at the output port, which is at (225,150). You can use any point to the right of that point along the same x -axis to delete the line.

```
delete_line('vdp', [230,150]);
```

Delete Segments of Branched Lines

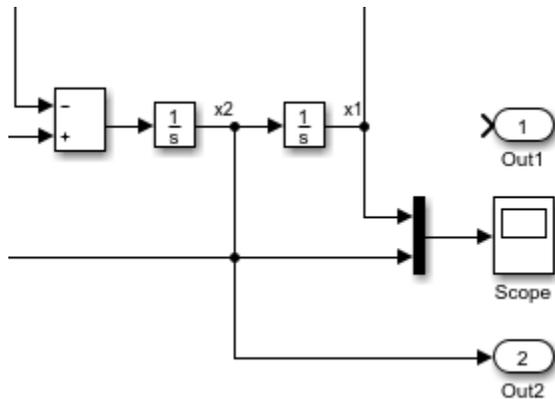
Use `delete_line` with branched lines to remove the segment for any connection.

Open the model vdp.

```
open_system('vdp');
```

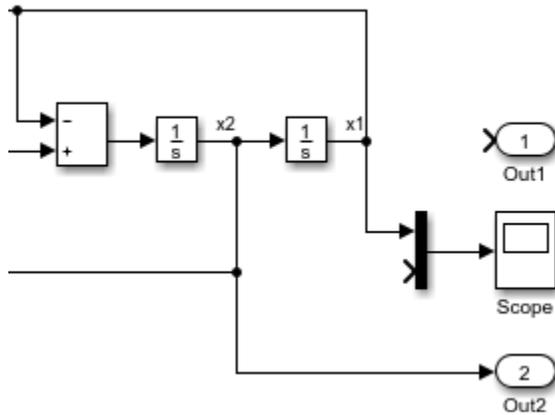
Delete the line from x1 to the Out1 block. This command deletes only the segment of the line that connects the branch to the specified block.

```
delete_line('vdp','x1/1','Out1/1')
```



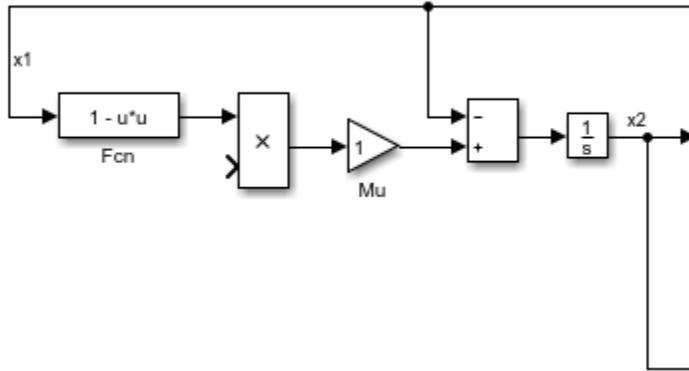
Delete the line segment from x2 to the Mux.

```
delete_line('vdp','x2/1','Mux/2')
```



Delete the line segment from x2 to the Product block.

```
delete_line('vdp','x2/1','Product/2')
```



Input Arguments

sys — Model or subsystem to delete line from
character vector

Model or subsystem to delete the line from, specified as a character vector.

Example: 'vdp', 'f14/Controller'

out — Block output port to delete line from
block/port name or number character vector | port handle

Block output port to delete line from, specified as either:

- The block name, a slash, and the port name or number. Most block ports are numbered from top to bottom or from left to right. For a state port, use the port name State instead of a port number.
- The port handle that you want to delete the line from.

Use 'PortHandles' with `get_param` to get the handles.

Example: 'Mu/1', 'Subsystem/2'

in — Block input port to delete line from
block/port name or number character vector | port handle

Block input port to delete line from, specified as either:

- The block name, a slash, and the port name or number. The port name on:
 - An enabled subsystem is Enable.
 - A triggered subsystem is Trigger.
 - If Action and Switch Case Action subsystems is Action.
- The port handle that you want to delete the line from.

Use 'PortHandles' with `get_param` to get handles.

Example: 'Mu/1', 'Subsystem/2'

point — Point on the line you want to delete

1-by-2 matrix

Point that falls on the line you want to delete, specified as a 1-by-2 matrix.

Example: [150 200]

lineHandle — Handle of the line you want to delete

handle

Handle of the line you want to delete. You can get the line handle by using `get_param` with the 'LineHandles' option or by assigning the line to a handle when you create it programmatically.

See Also

`add_line` | `get_param`

Introduced before R2006a

delete_param

Delete system parameter added via `add_param` command

Syntax

```
delete_param('sys', 'parameter1', 'parameter2', ...)
```

Description

This command deletes parameters that were added to the system using the `add_param` command. The command displays an error message if a specified parameter was not added with the `add_param` command.

Examples

The following example

```
add_param('vdp', 'DemoName', 'VanDerPolEquation', 'EquationOrder', '2')
delete_param('vdp', 'DemoName')
```

adds the parameters `DemoName` and `EquationOrder` to the `vdp` system, then deletes `DemoName` from the system.

See Also

`add_param`

Introduced before R2006a

dependencies.fileDependencyAnalysis

Find model file dependencies

Syntax

```
files = dependencies.fileDependencyAnalysis('modelname')
[files, missing] = dependencies.fileDependencyAnalysis('modelname')
[files, missing, depfile] = dependencies.fileDependencyAnalysis('
modelname')
[files, missing, depfile, manifestfile] =
dependencies.fileDependencyAnalysis('modelname', 'manifestfile')
```

Description

`files = dependencies.fileDependencyAnalysis('modelname')` returns `files`, a cell array of character vectors containing the full paths of all existing files referenced by the model `modelname`.

`[files, missing] = dependencies.fileDependencyAnalysis('modelname')` returns `files`, all existing files referenced by the model `modelname`, and any referenced files that cannot be found in *missing*.

`[files, missing, depfile] = dependencies.fileDependencyAnalysis('modelname')` also returns `depfile`, the full path of the user dependencies (.smd) file, if it exists, that stores the names of any files you manually added or excluded.

`[files, missing, depfile, manifestfile] = dependencies.fileDependencyAnalysis('modelname', 'manifestfile')` also creates a manifest file with the name and path specified in `manifestfile`.

Input Arguments

modelName

Character vector specifying the name of the model to analyze for dependencies.

manifestfile

(Optional) Character vector to specify the name of the manifest file to create. You can specify a full path or just a file name (in which case the file is created in the current folder). The function adds the suffix `.smf` to the user-specified name.

Output Arguments

files

A cell array of character vectors containing the full-paths of all existing files referenced by the model `modelName`. If there is only one dependency, the return is a character vector. If there are no dependencies, the return is empty.

Default: []

missing

A cell array of character vectors containing the names of any files that are referenced by the model `modelName`, but cannot be found.

Default: []

depfile

Character vector containing the full path of a user dependencies (`.smd`) file, if it exists, that stores the names of any files you manually added or excluded. Simulink uses the `.smd` file to remember your changes the next time you generate a manifest. See “Edit Manifests”.

Default: []

manifestfile

Character vector containing the name and path of the new manifest file.

Default: []

Examples

The following code analyses the model *mymodel* for file dependencies:

```
files = dependencies.fileDependencyAnalysis('mymodel')
```

If you try dependency analysis on an example model, it returns an empty list of required files because the standard MathWorks installation includes all the files required for the example models.

Tip

If you try dependency analysis on an example model, it returns an empty list of required files because the standard MathWorks installation includes all the files required for the example models.

Alternatives

If your file is in a Simulink project, use `listRequiredFiles` instead.

You can interactively run dependency analysis in Simulink Project. See “Run Dependency Analysis”.

To create a report to identify where dependencies arise, find required toolboxes, and for more control over dependency analysis options, you can interactively generate a manifest and report. See “Analyze Model Dependencies”.

To programmatically check which *toolboxes* are required, see `dependencies.toolboxDependencyAnalysis`.

See Also

`dependencies.toolboxDependencyAnalysis` | `listRequiredFiles`

Topics

“What Are Model Dependencies?”

Introduced in R2012a

dependencies.toolboxDependencyAnalysis

Find toolbox dependencies

Syntax

```
names = dependencies.toolboxDependencyAnalysis(files_in)
[names, folders] = dependencies.toolboxDependencyAnalysis(files_in)
```

Description

`names = dependencies.toolboxDependencyAnalysis(files_in)` returns `names`, a cell array of toolbox names required by the files in `files_in`.

`[names, folders] = dependencies.toolboxDependencyAnalysis(files_in)` returns toolbox names and also a cell array of the toolbox folders.

Tip In a Simulink project, you can interactively run dependency analysis. You can find the required toolboxes for the whole project or for selected files. You can see which products a new team member requires to use the project, or find which file is introducing a product dependency. See “Find Required Toolboxes”.

Input Arguments

files_in

Cell array of character vectors containing `.m`, `.mdl`, or `.slx` files on the MATLAB path. Simulink model names (without file extension) are also allowed.

Default: `[]`

Output Arguments

names

Cell array of toolbox names required by the files in `files_in`.

folders

(Optional) Cell-array of the required toolbox folders.

Examples

The following code reports the detectable required toolboxes for the model `vdp`:

```
files_in={'vdp'};
names = dependencies.toolboxDependencyAnalysis(files_in)

names =

    'MATLAB'    'Simulink'    'Simulink Coder'
```

To find all detectable toolbox dependencies of your model *and* the files it depends on:

- 1 Call `fileDependencyAnalysis` on your model.

For example:

```
files = dependencies.fileDependencyAnalysis('mymodel')

files =

    'C:\Work\foo.m'
    'C:\Work\mymodel.mdl'
```

- 2 Call `toolboxDependencyAnalysis` on the files output of step 1.

For example:

```
tbxes = dependencies.toolboxDependencyAnalysis(files)

tbxes =

[1x24 char]    'MATLAB'    'Simulink Coder'    'Simulink'
```

To view long product names examine the `tbxes` cell array as follows:

```
tbxes{:}

ans =
Image Processing Toolbox

ans =
MATLAB

ans =
Simulink Coder

ans =
Simulink
```

Tips

The function `dependencies.toolboxDependencyAnalysis` looks for toolbox dependencies of the files in `files_in` but does *not* analyze any subsequent dependencies. See “Examples” on page 2-211.

For command-line dependency analysis, the analysis uses the default settings for analysis scope to determine required toolboxes. For example, if you have code generation products, then the check **Find files required for code generation** is on by default and Simulink Coder is always reported as required. See “Required Toolboxes” in the manifest documentation for more examples of how your installed products and analysis scope settings can affect reported toolbox requirements.

Alternatives

In a Simulink project, you can interactively run dependency analysis and find the required toolboxes for the whole project or for selected files. See “Find Required Toolboxes”.

For a model that is not in a project, you can interactively generate a manifest and report. You can create a report to identify where dependencies arise, and control dependency analysis options. See “Analyze Model Dependencies”.

To programmatically check which *files* are required, see `dependencies.fileDependencyAnalysis`.

See Also

`dependencies.fileDependencyAnalysis`

Topics

“Dependency Analysis”

“What Is Dependency Analysis?”

Introduced in R2012a

detachConfigSet

Dissociate configuration set or configuration reference from model

Syntax

```
detachConfigSet(model, configObjName)
```

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

`configObjName`

The name of a configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

Description

`detachConfigSet` detaches the configuration set or configuration reference (configuration object) specified by `configObjName` from `model`. If no such configuration object is attached to the model, an error occurs.

Examples

The following example detaches the configuration object named `DevConfig` from the current model. The code is the same whether `DevConfig` is a configuration set or configuration reference.

```
detachConfigSet(gcs, 'DevConfig');
```

See Also

attachConfigSet | attachConfigSetCopy | closeDialog | getActiveConfigSet
| getConfigSet | getConfigSets | openDialog | setActiveConfigSet

Topics

“Manage a Configuration Set”

“Manage a Configuration Reference”

Introduced in R2006a

removeLabel(was detachLabelFromFile)

REMOVE — RENAMED TO REMOVELABEL — consolidate with existing removeLabel page. was: Detach label from Simulink Project file

Syntax

```
removeLabel(file,labelDefinition)
```

Description

`removeLabel(file,labelDefinition)` detaches the specified label `labelDefinition` from the file. Before you can detach the label, you need to get the label from the file.Label property or by using `findLabel`.

Examples

Detach a Label from a File

Remove a label from a particular project file.

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

Get a particular file by name.

```
myfile = findFile(proj,'models/AnalogControl.mdl')
```

```
myfile =
```

```
ProjectFile with properties:
```

```
Path: [1x86 char]  
Labels: [1x1 slproject.Label]
```

Get the Labels property of the file.

```
myfile.Labels
```

```
ans =
```

```
Label with properties:
```

```
File: 'C:\work\airframe\models\AnalogControl.mdl'  
Data: []  
DataType: 'none'  
Name: 'Design'  
CategoryName: 'Classification'
```

Attach the label 'To Review' to the file.

```
addLabel(myfile, 'Review', 'To Review')
```

Get the label you want to remove. Index into the Labels property to get the second label attached to the file.

```
labeltoremove = myfile.Labels(2)
```

```
labeltoremove =
```

```
Label with properties:
```

```
File: [1x86 char]  
Data: []  
DataType: 'char'  
Name: 'To Review'  
CategoryName: 'Review'
```

Remove the label from the file.

```
removeLabel(myfile, labeltoremove)  
myfile.Labels
```

```
ans =
```

```
Label with properties:
```

```
File: [1x86 char]  
Data: []  
DataType: 'none'
```

```
Name: 'Design'  
CategoryName: 'Classification'
```

Input Arguments

file — File to detach label from

file object

File to detach the label from, specified as a file object. You can get the file object by examining the project's Files property (`proj.Files`), or use `findFile` to find a file by name. The file must be within the root folder.

labelDefinition — Label to detach

label definition object

Name of the label to detach, specified as a label definition object returned by the `file.Label` property or `findLabel`.

See Also

Functions

`addLabel` | `createLabel` | `findFile` | `findLabel` | `simulinkproject`

disableimplicitsignalresolution

Convert model to use only explicit signal resolution

Syntax

```
retVal = disableimplicitsignalresolution('model')  
retVal = disableimplicitsignalresolution('model', displayOnly)
```

Description

`retVal = disableimplicitsignalresolution('model')` inputs a model, reports all signals and states that implicitly resolve to signal objects, and converts the model to resolve only signals and states that explicitly require it. The report and any changes are limited to the model itself; they do not include blocks that are library links.

Before executing this function, ensure that all relevant Simulink data objects are defined in the base workspace or a data dictionary to which the model is linked. The function ignores any data objects that are defined elsewhere.

The function scans `model`, returns a structure of handles to signals and states that resolve implicitly to signal objects, and performs the following operations on `model`:

- Search the model for all output ports and block states that resolve to Simulink signal objects.
- Modify these ports and blocks to enforce signal object resolution in the future.
- Set the model's `SignalResolutionControl` parameter to 'UseLocalSettings' (GUI: **Explicit Only**).

If `SignalResolutionControl` is already set to 'UseLocalSettings' or to 'None', the function takes no action and returns a warning.

- If any Stateflow output data resolves to a Simulink signal object:
 - Turn off hierarchical scoping of signal objects from within the Stateflow chart.
 - Explicitly label the output signal of the Stateflow chart.

- Enforce signal object resolution for this signal in the future.

Any changes made by `disableimplicitsignalresolution` permanently change the model. Be sure to back up the model before calling the function with `displayOnly` defaulted to or specified as `false`.

`retVal = disableimplicitsignalresolution('model', displayOnly)` is equivalent to `disableimplicitsignalresolution(model)` if `displayOnly` is `false`.

If `displayOnly` is `true`, the function returns a structure of handles to signals and states that resolve implicitly to signal objects, but leaves the model unchanged.

Input Arguments

`displayOnly`

Boolean specifying whether to change the model (`false`) or just generate a report (`true`)

Default: `false`

`model`

Model name or handle

Output Arguments

`retVal`

A MATLAB structure containing:

Signals

Handles to ports with signal names that resolve to signal objects

States

Handles to blocks with states that resolve to signal objects

See Also

`Simulink.Signal`

Topics

“Data Validity Diagnostics Overview”

“Symbol Resolution”

Introduced in R2007a

docblock

Get or set editor invoked by Simulink DocBlock

Syntax

```
docblock(setEditorType,command)
```

```
editCommand = docblock(getEditorType)
```

Description

`docblock(setEditorType,command)` uses the specified command to set the editor opened by double-clicking a DocBlock block.

By default, a DocBlock block opens Microsoft Word to edit HTML or RTF files. If Word is not available on your system, the block opens these file types using the text editor specified on the **Editor/Debugger Preferences** pane of the MATLAB Preferences dialog box. For text files, the default editor is the text editor specified in the MATLAB preferences.

`editCommand = docblock(getEditorType)` returns the current command to open the specified editor from a DocBlock block.

Input Arguments

setEditorType — File type whose editor command to set

'setEditorHTML' | 'setEditorDOC' | 'setEditorTXT'

File type whose editor command you want to set, specified as 'setEditorHTML', 'setEditorDOC', or 'setEditorTXT'.

command — Command to open file in editor

character vector | ''

Command to open the specified file type in an editor from the MATLAB command prompt, specified as a character vector. Use ' ' to reset to the default editor for that file type.

In the command, use the "%<FileName>" token to represent the full pathname to the document.

getEditorType — File type of the editor command to return

'getEditorHTML' | 'getEditorDOC' | 'getEditorTXT'

File type of the editor command to return, specified as 'getEditorHTML', 'getEditorDOC', or 'getEditorTXT'.

Output Arguments

editCommand — Command to open the editor

character vector

Command to open the editor, returned as a character vector.

Examples

Set DocBlock Text Editor

Specify Notepad as the DocBlock editor for text files.

```
docblock('setEditorTXT','system(''notepad "%<FileName>"'');')
```

Set and Get Current HTML Editor

You can use the docblock command to get the current editor.

Set your HTML editor for the DocBlock block to Mozilla Composer. The ampersand executes the command in the background.

```
docblock('setEditorHTML',...
    'system('/usr/local/bin/mozilla -edit "%<FileName>" &');')
```

Get the current HTML editor.

```
htmlEd = docblock('getEditorHTML')  
  
htmlEd =  
  
    'system('/usr/local/bin/mozilla -edit "%<FileName>" &');'
```

Reset Text Editor to Default

Specify Notepad as the DocBlock editor for text files.

```
docblock('setEditorTXT', 'system(''notepad "%<FileName>"');')
```

Get the current text editor.

```
txtEd = docblock('getEditorTXT')  
  
txtEd =  
  
    'system('notepad "%<FileName>");'
```

Reset the editor to the default editor.

```
docblock('setEditorTXT', '')
```

- “Use a Simulink DocBlock to Add a Comment” (Embedded Coder)

See Also

DocBlock

Topics

“Use a Simulink DocBlock to Add a Comment” (Embedded Coder)

Introduced in R2007a

export

Export Simulink Project to zip

Syntax

```
export(proj, zipFileName)
export(proj, zipFileName, definitionType)
```

Description

`export(proj, zipFileName)` exports the project `proj` to a zip file specified by `zipFileName`. The zip archive preserves the project files, structure, labels, and shortcuts, and does not include any source control information. You can use the zip archive to send the project to customers, suppliers, or colleagues who do not have access to your source control repository. Recipients can create a new project from the zip archive by clicking **New** in the Simulink Project Tool, and then in the start page, clicking **Archive**.

`export(proj, zipFileName, definitionType)` exports the project using the specified `definitionType` for the project definition files, single or multiple. If you do not specify `definitionType`, the project's current setting is used. Use the `definitionType` `export` option if you want to change project definition file management from the type selected when the project was created. You can control project definition file management in the preferences.

Examples

Export a Project to a Zip File

Open the `airframe` project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Export the project to a zip file.

```
export(proj, 'airframeproj.zip')
```

- “Archive Projects in Zip Files”

Input Arguments

proj — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

zipFileName — Zip file name or path

character vector

Zip file name or path, specified as a character vector ending in the file extension `.zip`. If `zipFileName` is a filename, Simulink exports the file to the current folder. You can also specify a fully qualified path name.

Example: `'project.zip'`

Data Types: `char`

definitionType — Definition file type

`slproject.DefinitionFiles.SingleFile` | `slproject.DefinitionFiles.MultiFile`

Definition file type, specified as `slproject.DefinitionFiles.SingleFile`, `slproject.DefinitionFiles.MultiFile`, or `slproject.DefinitionFiles.FixedPathMultiFile`. Use the `definitionType` export option if you want to change project definition file management from the type selected when the project was created. `MultiFile` is better for avoiding merging issues on shared projects. `SingleFile` is faster but is likely to cause merge issues when two users submit changes in the same project to a source control tool. If you need to work with long file paths, use `FixedPathMultiFile`.

Example: `export(proj, 'proj.zip', slproject.DefinitionFiles.SingleFile)`

See Also

Topics

“Archive Projects in Zip Files”

Introduced in R2013a

findCategory

Get Simulink Project category of labels

Syntax

```
category = findCategory(proj, categoryName)
```

Description

`category = findCategory(proj, categoryName)` returns the project category specified by `categoryName`. You need to get a category before you can use `createLabel` or `removeLabel`.

Examples

Get a Category of Project Labels

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

Use `findCategory` to get a category of labels by name.

```
category = findCategory(proj, 'Classification')  
  
category =
```

```
Category with properties:
```

```
    Name: 'Classification'  
    DataType: 'none'  
    LabelDefinitions: [1x8 slproject.LabelDefinition]
```

Alternatively, you can examine categories by index. Get the first category.

```
proj.Categories(1)

ans =

    Category with properties:
        Name: 'Classification'
        DataType: 'none'
        LabelDefinitions: [1x8 slproject.LabelDefinition]
```

Find out what you can do with the category.

```
methods(category)

Methods for class slproject.Category:

createLabel      findLabel  removeLabel
```

Input Arguments

proj — Project
project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

categoryName — Name of category
character vector

Name of the category to get, specified as a character vector.

Output Arguments

category — Category of labels
category object

Category of labels, returned as a category object that you can query or modify. If the specified category is not found, the function returns an empty array.

See Also

Functions

`createLabel` | `removeLabel` | `simulinkproject`

Introduced in R2013a

findFile

Get Simulink Project file by name

Syntax

```
file = findFile(proj, fileName)
```

Description

`file = findFile(proj, fileName)` returns a specific project file by name. You need to get a file before you can query labels, or use `addLabel` or `removeLabel`.

Examples

Find a File By Name

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

Use `findFile` to get a file by name. You need to know the path if it is inside subfolders under the project root.

```
myfile = findFile(proj, 'models/AnalogControl.mdl')
```

```
myfile =
```

```
ProjectFile with properties:
```

```
    Path: [1x86 char]  
    Labels: [1x1 slproject.Label]  
    Revision: '2'  
    SourceControlStatus: Unmodified
```

Alternatively, you can examine files by index. Get the first file.

```
file = proj.Files(1);
```

Find out what you can do with the file.

```
methods(file)
```

```
Methods for class slproject.ProjectFile:
```

```
addLabel    removeLabel  findLabel
```

Input Arguments

proj — Project

project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

fileName — Path of file

character vector

Path of the file to find, including any subfolders under the project root, specified as a character vector.

Output Arguments

file — Project file

file object

Project file, returned as a file object that you can query or modify.

See Also

Functions

`addLabel` | `findCategory` | `findLabel` | `removeLabel` | `simulinkproject`

Introduced in R2013a

findLabel

Get Simulink Project file label

Syntax

```
label = findLabel(file, categoryName, labelName)
label = findLabel(file, labelDefinition)
label = findLabel(category, labelName)
```

Description

`label = findLabel(file, categoryName, labelName)` returns the label and its attached data for the label `labelName` in the category `categoryName` that is attached to the specified file or files. Use this syntax when you know the label name and category.

`label = findLabel(file, labelDefinition)` returns the file label and its attached data for the label name and category specified by `labelDefinition`. Use this syntax if you previously got a `labelDefinition` by accessing a `Labels` property, e.g., using a command like `myfile.Labels(1)`.

`label = findLabel(category, labelName)` returns the label definition of the label in this category specified by `labelName`. Returns an empty array if the label is not found.

Examples

Find Files with the Label Utility

Find all project files with a particular label.

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Get the list of project files.

```
files = proj.Files;
```

Loop through each file. If the file has the extension .m, attach the label Utility.

```
for fileIndex = 1:numel(files)
    file = files(fileIndex);
    [~, ~, fileExtension] = fileparts(file.Path);
    if strcmp(fileExtension, '.m')
        addLabel(file, 'Classification', 'Utility');
    end
end
```

Find all the files with the label Utility and add them to a list returned in utility_files_to_review.

```
utility_files_to_review = {};
for jj=1:numel(files)
    this_file = files(jj);

    label = findLabel(this_file, 'Classification', 'Utility');

    if ( ~isempty(label))
        % This is a file labeled 'Utility'. Add to the
        % list of utility files.
        utility_files_to_review = [utility_files_to_review; this_file];
    end
end
```

Find a Label by Name or Definition

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;
proj = simulinkproject;
```

Get a particular file by name.

```
myfile = findFile(proj, 'models/AnalogControl.mdl');
```

Get a label by name.

```
label = findLabel(myfile, 'Classification', 'Design');
```

Alternatively, examine the `Labels` property of the file to get an array of `Label` objects, one for each label attached to the file.

```
labels = myfile.Labels
```

Index into the `Labels` property to get the label attached to the particular file.

```
labeldefinition = myfile.Labels(1)
```

After you get the label definition from the `Labels` property, you can use it with `findLabel`.

```
label = findLabel(myfile,labeldefinition);
```

Find Labels by Name or Definition

Open the `airframe` project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

Get a category.

```
category = proj.Categories(1)
```

```
category =
```

```
Category with properties:
```

```
    Name: 'Classification'  
    DataType: 'none'  
    LabelDefinitions: [1x8 slproject.LabelDefinition]
```

Get a label definition.

```
ld = findLabel(category,'Design')
```

```
ld =
```

```
LabelDefinition with properties:
```

```
Name: 'Design'  
CategoryName: 'Classification'
```

Input Arguments

file — File to search labels of

file object

File to search the labels of, specified as a file object or an array of file objects. You can get the file object by examining the project's Files property (`proj.Files`), or use `findFile` to get a file by name. The file must be in the project.

categoryName — Name of category

character vector

Name of the parent category for the label, specified as a character vector.

labelName — Name of label

character vector

Name of the label to get, specified as a character vector.

labelDefinition — Name of label

label definition object

Name of the label to get, specified as a label definition object returned by the `file.Label` property.

category — Category of labels

category object

Category of labels, specified as a category object. Get a category object from the `proj.Categories` property or by using `findCategory`.

Output Arguments

label — Label

label object

Label, returned as a label object.

See Also

Functions

`addLabel` | `createLabel` | `findFile` | `simulinkproject`

Introduced in R2013a

findLabelDefinition(renamed to findLabel)

Get Simulink Project label definition

Syntax

```
labelDefinition = findLabelDefinition(category, labelName)
```

Description

`labelDefinition = findLabelDefinition(category, labelName)` returns the label definition of the label in this category specified by `labelName`. Returns an empty array if the label is not found.

Examples

Find Labels by Name or Definition

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

Get a category.

```
category = proj.Categories(1)
```

```
category =
```

```
Category with properties:
```

```
    Name: 'Review'  
    DataType: 'char'  
    LabelDefinitions: [1x4 slproject.LabelDefinition]
```

Get a label definition.

```
ld = findLabelDefinition(category, 'To Review')

ld =

    LabelDefinition with properties:

        Name: 'To Review'
        CategoryName: 'Review'
```

Alternatively, get a file and examine the `Labels` property to get an array of `Label` objects, one for each label attached to the file.

```
myfile = findFile(proj, 'models/AnalogControl.mdl');
labels = myfile.Labels
```

Index into the `Labels` property to get the second label attached to the particular file.

```
labeldefinition = myfile.Labels(1)
```

After you get the label definition from the `Labels` property, you can use it with `findLabel`.

```
label = findLabel(myfile, labeldefinition);
```

Alternatively, get a particular file by name, and then get one of its labels by name.

```
myfile = findFile(proj, 'models/AnalogControl.mdl');
label = findLabel(myfile, 'Review', 'To Review');
```

Input Arguments

labelName — Name of label

character vector

Name of the label to get, specified as a character vector.

category — Category of labels

category object

Category of labels, specified as a category object. Get a category object from the `proj.Categories` property or by using `findCategory`.

Output Arguments

labelDefinition — Label definition

label definition object

Label definition, returned as a label definition object. Query the label definition properties to find the label data type.

See Also

Functions

`addLabel` | `createLabel` | `findCategory` | `simulinkproject`

Introduced in R2013a

find_mdrefs

Find Model blocks and referenced models at all levels or at top level only

Syntax

```
[refMdls,mdlBlks] = find_mdrefs(system)
[refMdls,mdlBlks] = find_mdrefs(system,Name,Value)
[refMdls,mdlBlks] = find_mdrefs(system,allLevels)
```

Description

`[refMdls,mdlBlks] = find_mdrefs(system)` finds all referenced models and Model blocks contained by the subsystem or model reference hierarchy that `system` is the top level of.

`[refMdls,mdlBlks] = find_mdrefs(system,Name,Value)` finds referenced models and Model blocks with additional options specified by one or more `Name, Value` pair arguments.

`[refMdls,mdlBlks] = find_mdrefs(system,allLevels)` specifies the levels of the system to search.

Tip The `find_mdrefs` function provides two different ways to specify the levels of the system to search. Both techniques give the same results, but only the name and value technique allows you to control inclusion of protected and variant models in `refMdls`.

Examples

Find Referenced Models in Model Reference Hierarchy

Find referenced models and Model blocks for all models referenced by the specified model. Include all model reference variants.

```
load_system('sldemo_mdhref_variants_enum');
[myModels,myModelBlks] = find_mdrefs('sldemo_mdhref_variants_enum',...
'AllLevels',true,'Variants','AllVariants')
```

```
myModels =
```

```
9x1 cell array
```

```
{'sldemo_mrv_linear_controller'      }
{'sldemo_mrv_nonlinear_controller'  }
{'sldemo_mrv_sig_filter1_production'}
{'sldemo_mrv_sig_filter1_prototype' }
{'sldemo_mrv_sig_filter2_production'}
{'sldemo_mrv_sig_filter2_prototype' }
{'sldemo_mrv_sig_filter3_production'}
{'sldemo_mrv_sig_filter3_prototype' }
{'sldemo_mdhref_variants_enum'      }
```

```
myModelBlks =
```

```
8x1 cell array
```

```
{'sldemo_mdhref_variants_enum/Controller/Linear'    }
{'sldemo_mdhref_variants_enum/Controller/Nonlinear'}
{'sldemo_mdhref_variants_enum/Filter1/Production'   }
{'sldemo_mdhref_variants_enum/Filter1/Prototype'   }
{'sldemo_mdhref_variants_enum/Filter2/Production'   }
{'sldemo_mdhref_variants_enum/Filter2/Prototype'   }
{'sldemo_mdhref_variants_enum/Filter3/Production'   }
{'sldemo_mdhref_variants_enum/Filter3/Prototype'   }
```

- “Set up Model Variants Using a Model Block”
- “Protected Model”

Input Arguments

system — System to search

character vector | handle

System to search, specified as a character vector or a handle.

- The character vector can be the path to a Model block, subsystem, or a model in a model reference hierarchy.
- The handle can be for a Model block, subsystem, or model in a model reference hierarchy.

allLevels — Levels to search

`true` (default) | `false`

Levels to search, specified as `true` or `false`.

- `true` — Search all Model blocks in the model reference hierarchy for which the system is the top model.
- `false` — Search only the top-level system.

Data Types: `logical`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `refModels = find_mdirefs(top_model, 'Variants', true)`

AllLevels — Levels to search

`true` (default) | `false`

Levels to search, specified as a `true` or `false`.

- `true` — Search all Model blocks in the model reference hierarchy for which the system is the top model.
- `false` — Search only the top-level system.

Data Types: `logical`

IncludeProtectedModels — Include protected models in search results

`false` (default) | `true`

Include protected models in search, specified as `true` or `false`. This setting does not affect the list of Model blocks returned.

Data Types: `logical`

Variants — Include variants in search

'ActivePlusCodeVariants' (default) | 'ActiveVariants' | 'AllVariants'

Include variants in search, specified as 'ActivePlusCodeVariants', 'ActiveVariants', or 'AllVariants'.

- 'ActivePlusCodeVariants' — Include all variants for Model Variants blocks for which you select the **Generate preprocessor conditionals** option.
- 'ActiveVariants' — Include the active variant for Model Variants blocks.
- 'AllVariants' — Include all variants for Model Variants blocks.

IncludeCommented — Include commented blocks in search

'off' (default) | 'on'

Include commented blocks in search, specified as 'off' or 'on'.

KeepModelsLoaded — Keep loaded models that function loads

false (default) | true

The `find_mdhrefs` function loads the models in the model reference hierarchy of the model that you specify. By default, the function closes those models, except for models that were already loaded before execution of the function. To keep all the models loaded that the function loads, set this argument to `true`.

Data Types: `logical`

Output Arguments

refMdlS — Names of referenced models

cell array of character vectors

Names of referenced models, returned as a cell array of character vectors. The last element is the system you specified in the `system` input argument or the parent model of that system.

mdlBlks — Names of Model blocks

cell array of character vectors

Names of Model blocks, returned as a cell array of character vectors.

See Also

Model | `find_system`

Topics

“Set up Model Variants Using a Model Block”

“Protected Model”

“Model Referencing”

Introduced before R2006a

find_system

Find systems, blocks, lines, ports, and annotations

Syntax

```
Objects = find_system
Objects = find_system(System)
Objects = find_system(Name, Value)
Objects = find_system(System, Name, Value)
```

Description

`Objects = find_system` returns loaded systems and their blocks, including subsystems.

`Objects = find_system(System)` returns the specified system and its blocks.

`Objects = find_system(Name, Value)` returns loaded systems and the objects in those systems that meet the criteria specified by one or more `Name, Value` pair arguments. You can use this syntax to specify search constraints and to search for specific parameter values. Specify the search constraints before the parameter and value pairs.

`Objects = find_system(System, Name, Value)` returns the objects in the specified system that meet the specified criteria.

Input Arguments

System — System to search

path name | cell array of path names | handle | vector of handles

System to search, specified as the full system path name, a cell array of system path names, a handle, or a vector of handles.

Example: 'MyModel/Subsystem1'

Example: `{'vdp', 'fuelsys'}`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

When you use the `find_system` function, `Name`, `Value` pair arguments can include search constraints and parameter name and value pairs. You can specify search constraints in any order, but they must precede the parameter name and value pairs.

See “Model Parameters” on page 6-2 and “Block-Specific Parameters” on page 6-128 for the lists of model and block parameters.

Example: `'SearchDepth', '0', 'LookUnderMasks', 'none', 'BlockType', 'Goto'` searches in loaded systems, excluding masked subsystems, for `Goto` blocks.

The table lists the possible search constraint pairs. Braces indicate default values.

Name	Value Type	Description
'BlockDialogParams'	character vector	Search block dialog box parameters for the specified value. This pair, like parameter and value pairs, must follow the other search constraint pairs.
'CaseSensitive'	{'on'} 'off'	If 'on', search considers case when matching.
'FindAll'	'on' {'off'}	If 'on', search includes lines, ports, and annotations within systems. <code>find_system</code> returns a vector of handles when this option is set to 'on', regardless of how you specify <code>System</code> .
'FirstResultOnly'	'on' {'off'}	Return only the first result found and stop the search.

Name	Value Type	Description
'FollowLinks'	'on' {'off'}	If 'on', search follows links into library blocks. If you do not specify a system to search, <code>find_system</code> includes loaded libraries in the results, whether you set 'FollowLinks' to 'on' or 'off'. You can use 'FollowLinks' with 'LookUnderMasks' to update library links in subsystems. See “Update Library Links in a Subsystem” on page 2-256.
'IncludeCommented'	'on' {'off'}	Specify whether to include commented blocks in the search.
'LoadFullyIfNeeded'	{'on'} 'off'	If 'on', attempts to load any partially loaded models. If 'off', disables model loading. Use this pair, for example, to prevent load warnings.
'LookUnderMasks'	{'graphical'}	Search includes masked subsystems that have no workspaces and no dialogs.
	'none'	Search skips masked subsystems.
	'functional'	Search includes masked subsystems that do not have dialogs.
	'all'	Search includes all masked subsystems.
	'on' 'off'	If 'on', search includes all masked subsystems. If 'off', search skips masked subsystem.
'RegExp'	'on' {'off'}	If 'on', search treats search expressions as regular expressions. To learn more about MATLAB regular expressions, see “Regular Expressions” (MATLAB).

Name	Value Type	Description
'SearchDepth'	positive integer character vector	Restricts the search depth to the specified level ('0' for loaded systems only, '1' for blocks and subsystems of the top-level system, '2' for the top-level system and its children, etc.). The default is all levels.
'Variants' This search constraint applies only to variant subsystems and model variants.	{'ActiveVariants'}	Search in only the active variant subsystems.
	'AllVariants'	Search in all variants.
	'ActivePlusCodeVariants'	Search all variants if any generate preprocessor conditionals. Otherwise, search only the active variant.

Output Arguments

Objects — Matching objects

cell array of path names | vector of handles

Matching objects found, returned as:

- A cell array of path names if you specified `System` as a path name or cell array of path names, or if you did not specify a system
- A vector of handles if you specified `System` as a handle or vector of handles

Examples

Find Loaded Systems and Their Blocks

Return the names of all loaded systems and their blocks.

```
load_system('vdp')
find_system
```

```

ans =

    13x1 cell array

    {'vdp'
     {'vdp/Fcn'
     {'vdp/More Info'
     {'vdp/More Info/Model Info'
     {'vdp/Mu'
     {'vdp/Mux'
     {'vdp/Product'
     {'vdp/Scope'
     {'vdp/Sum'
     {'vdp/x1'
     {'vdp/x2'
     {'vdp/Out1'
     {'vdp/Out2'
  
```

Returns loaded systems and libraries, including vdp.

Find Specific System and Its Blocks

Return vdp system and its blocks.

```

load_system({'vdp', 'fuelsys'})
find_system('vdp')
  
```

```

ans =

    13x1 cell array

    {'vdp'
     {'vdp/Fcn'
     {'vdp/More Info'
     {'vdp/More Info/Model Info'
     {'vdp/Mu'
     {'vdp/Mux'
     {'vdp/Product'
     {'vdp/Scope'
     {'vdp/Sum'
  
```

```
{ 'vdp/x1'           }  
{ 'vdp/x2'           }  
{ 'vdp/Out1'        }  
{ 'vdp/Out2'        }
```

Return Names of Loaded Models

Return the names of only the loaded models, i.e., block diagrams. This command returns library names as well, because libraries are treated as models.

```
load_system('vdp');  
open_bd = find_system('type','block_diagram')  
  
open_bd =  
  
    1x1 cell array  
  
    {'vdp'}
```

Search Children of Subsystem

Return the names of all Goto blocks that are children of the Unlocked subsystem in the `sldemo_clutch` system.

```
load_system('sldemo_clutch');  
find_system('sldemo_clutch/Unlocked','SearchDepth',1,'BlockType','Goto')  
  
ans =  
  
    2x1 cell array  
  
    {'sldemo_clutch/Unlocked/Goto' }  
    {'sldemo_clutch/Unlocked/Goto1'}
```

Search Using Multiple Criteria

Search in the vdp system and return the names of all Gain blocks whose **Gain** value is set to 1.

```
load_system('vdp');  
find_system('vdp','BlockType','Gain','Gain','1')
```

```
ans =  
  
    1x1 cell array  
  
    {'vdp/Mu'}
```

Return Lines and Annotations as Handles

Get the handles of all lines and annotations in the vdp system. With 'FindAll', the function returns handles regardless of how you specify the system to search.

```
load_system('vdp');  
L = find_system('vdp','FindAll','on','type','line')  
A = find_system('vdp','FindAll','on','type','annotation')
```

```
L =  
  
    33.0201  
    32.0181  
    31.0338  
    30.0575  
    29.0746  
    28.1221  
    27.0188  
    26.0209  
    25.0227  
    24.0176  
    23.0206  
    22.0188  
    21.0189  
    20.0187  
    19.0199
```

```
18.0149
17.0195
```

```
A =
```

```
35.0153
34.0171
```

Search for Specific Block Parameter Value

Find any block dialog box parameters with a value of 0 in the vdp and fuelsys systems.

```
load_system({'vdp', 'fuelsys'})
find_system({'vdp', 'fuelsys'}, 'BlockDialogParams', '0')
```

```
ans =
```

```
'vdp/x2'
'vdp/Out1'
'vdp/Out2'
'fuelsys/Constant2'
'fuelsys/Constant4'
'fuelsys/Constant5'
'fuelsys/engine ...'
'fuelsys/engine ...'
'fuelsys/engine ...'
'fuelsys/engine ...'
.
.
.
```

Search Using Regular Expressions

Find all blocks in the top level of the currently loaded systems with a block dialog parameter value that starts with 3.

```
load_system({'fuelsys', 'vdp'});
find_system('SearchDepth', '1', 'regex', 'on', 'BlockDialogParams', '^3')
```

```
ans =

3x1 cell array

{'vdp/Scope'          }
{'vdp/Scope'          }
{'fuelsys/Nominal...'}

```

Regular Expression Search for Partial Match

When you search using regular expressions, you can specify a part of the character vector you want to match to return all objects that contain that character vector. Find all the inport and outport blocks in the `sldemo_clutch` model.

```
load_system('sldemo_clutch');
find_system('sldemo_clutch','regexp','on','blocktype','port')

```

```
ans =

39x1 cell array

{'sldemo_clutch/Friction...' }
{'sldemo_clutch/Friction...' }
{'sldemo_clutch/Friction...' }
{'sldemo_clutch/Friction Mode Logic/Tin' }
{'sldemo_clutch/Friction Mode Logic/Tfmaxs' }
{'sldemo_clutch/Friction Mode Logic/Break Apart...' }
{'sldemo_clutch/Friction Mode Logic/Break Apart...' }
{'sldemo_clutch/Friction Mode Logic/Break Apart...' }
{'sldemo_clutch/Friction Mode Logic/Lockup...' }
{'sldemo_clutch/Friction Mode Logic/Lockup FSM/lock' }
{'sldemo_clutch/Friction Mode Logic/Lockup FSM/unlock' }
{'sldemo_clutch/Friction Mode Logic/Lockup FSM/locked' }

```

```
{'sldemo_clutch/Friction Mode Logic/Requisite Friction/Tin'}
{'sldemo_clutch/Friction Mode Logic/Requisite Friction/Tf' }
{'sldemo_clutch/Friction Mode Logic/locked' }
{'sldemo_clutch/Friction Mode Logic/lock' }
{'sldemo_clutch/Friction Mode Logic/unlock' }
{'sldemo_clutch/Friction Mode Logic/Tf' }
{'sldemo_clutch/Locked/Tin' }
{'sldemo_clutch/Locked/w' }
{'sldemo_clutch/Unlocked/Tfmaxk' }
{'sldemo_clutch/Unlocked/Tin' }
{'sldemo_clutch/Unlocked/we' }
{'sldemo_clutch/Unlocked/wv' }
{'sldemo_clutch/we' }
{'sldemo_clutch/wv' }
{'sldemo_clutch/w' }
{'sldemo_clutch/Locked Flag' }
{'sldemo_clutch/Lockup Flag' }
{'sldemo_clutch/Break-Apart Flag' }
{'sldemo_clutch/FrictionTorque...' }
{'sldemo_clutch/Max Static...' }
```

Update Library Links in a Subsystem

In this example, `myModel` contains a single subsystem, which is a library link. After the model was last opened, a Gain block was added to the corresponding subsystem in the library.

Open the model. Use `find_system` with `'FollowLinks'` set to `'off'`. The command does not follow the library links into the subsystem and returns only the subsystem at the top level.

```
open_system('myModel')
find_system(bdroot, 'LookUnderMasks', 'on', 'FollowLinks', 'off')

ans =

    'myModel'
    'myModel/Subsystem'
```

Use `find_system` with `'FollowLinks'` set to `'on'`. `find_system` updates the library links and returns the block in the subsystem.

```
find_system(bdroot, 'LookUnderMasks', 'on', 'FollowLinks', 'on')
```

```
Updating Link: myModel/Subsystem/Gain
```

```
Updating Link: myModel/Subsystem/Gain
```

```
ans =
```

```
    'myModel'  
    'myModel/Subsystem'  
    'myModel/Subsystem/Gain'
```

Return Values as Handles

Provide the system to `find_system` as a handle. Search for block dialog box parameters with a value of 0. If you make multiple calls to `get_param` for the same block, then using the block handle is more efficient than specifying the full block path as a character vector.

```
load_system('vdp');  
sys = get_param('vdp', 'Handle');  
find_system(sys, 'BlockDialogParams', '0')
```

```
ans =
```

```
    10.0027  
    13.0029  
    14.0028  
    14.0028  
    15.0026  
    15.0026
```

- “Find Model Elements in Simulink Models”
- “Edit and Manage Workspace Variables by Using Model Explorer”

See Also

[find_mdrefs](#) | [getSimulinkBlockHandle](#) | [get_param](#) | [set_param](#)

Topics

“Find Model Elements in Simulink Models”

“Edit and Manage Workspace Variables by Using Model Explorer”

“Regular Expressions” (MATLAB)

“Model Parameters” on page 6-2

“Block-Specific Parameters” on page 6-128

Introduced before R2006a

fixdt

Create `Simulink.NumericType` object describing fixed-point or floating-point data type

Syntax

```
a = fixdt(Signed, WordLength)
a = fixdt(Signed, WordLength, FractionLength)
a = fixdt(Signed, WordLength, TotalSlope, Bias)
a = fixdt(Signed, WordLength, SlopeAdjustmentFactor, FixedExponent,
Bias)
a = fixdt(DataTypeNameString)
a = fixdt(..., 'DataTypeOverride', 'Off')
[DataType, IsScaledDouble] = fixdt(DataTypeNameString)
[DataType, IsScaledDouble] = fixdt(DataTypeNameString,
'DataTypeOverride', 'Off')
```

Description

`a = fixdt(Signed, WordLength)` returns a `Simulink.NumericType` object describing a fixed-point data type with unspecified scaling. The scaling would typically be determined by another block parameter. `Signed` can be 0 (false) for unsigned or 1 (true) for signed.

`a = fixdt(Signed, WordLength, FractionLength)` returns a `Simulink.NumericType` object describing a fixed-point data type with binary point scaling. `FractionLength` can be greater than `WordLength`. For more information, see “Binary Point Interpretation” (Fixed-Point Designer).

`a = fixdt(Signed, WordLength, TotalSlope, Bias)` or `a = fixdt(Signed, WordLength, SlopeAdjustmentFactor, FixedExponent, Bias)` returns a `Simulink.NumericType` object describing a fixed-point data type with slope and bias scaling.

`a = fixdt(DataTypeNameString)` returns a `Simulink.NumericType` object describing an integer, fixed-point, or floating-point data type specified by a data type

name. The data type name can be either the name of a built-in Simulink data type or the name of a fixed-point data type that conforms to the naming convention for fixed-point names established by the Fixed-Point Designer product. For more information, see “Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer).

`a = fixdt(..., 'DataTypeOverride', 'Off')` returns a `Simulink.NumericType` object with its `DataTypeOverride` parameter set to `Off`. The default value for this property is `Inherit`. You can specify the `DataTypeOverride` parameter after any combination of other input parameters.

`[DataType, IsScaledDouble] = fixdt(DataTypeNameString)` returns a `Simulink.NumericType` object describing an integer, fixed-point, or floating-point data type specified by a data type name and a flag that indicates whether the specified data type name was the name of a scaled double data type.

`[DataType, IsScaledDouble] = fixdt(DataTypeNameString, 'DataTypeOverride', 'Off')` returns:

- A `Simulink.NumericType` object describing an integer, fixed-point, or floating-point data type specified by a data type name. The `DataTypeOverride` parameter of the `Simulink.NumericType` object is set to `Off`.
- A flag that indicates whether the specified data type name was the name of a scaled double data type.

Examples

Return a `Simulink.NumericType` object describing a fixed-point data type with unspecified scaling:

```
a = fixdt(1,16)
```

```
a =
```

```
Simulink.NumericType
  DataTypeMode: 'Fixed-point: unspecified scaling'
  Signedness: 'Signed'
  WordLength: 16
  IsAlias: false
  HeaderFile: ''
  Description: ''
```

Return a `Simulink.NumericType` object describing a fixed-point data type with binary point scaling :

```
a = fixdt(1,16,2)
```

```
a =
```

```
Simulink.NumericType
  DataTypeMode: 'Fixed-point: binary point scaling'
  Signedness: 'Signed'
  WordLength: 16
  FractionLength: 2
  IsAlias: false
  HeaderFile: ''
  Description: ''
```

Return a `Simulink.NumericType` object describing a fixed-point data type with slope and bias scaling:

```
a = fixdt(1, 16, 2^-2, 4)
```

```
a =
```

```
Simulink.NumericType
  DataTypeMode: 'Fixed-point: slope and bias scaling'
  Signedness: 'Signed'
  WordLength: 16
  Slope: 0.25
  Bias: 4
  IsAlias: false
  HeaderFile: ''
  Description: ''
```

Return a `Simulink.NumericType` object describing an integer, fixed-point, or floating-point data type specified by a data type name:

```
[DataType, IsScaledDouble] = fixdt('ufix8')
```

```
DataType =
```

```
Simulink.NumericType
  DataTypeMode: 'Fixed-point: binary point scaling'
  Signedness: 'Unsigned'
  WordLength: 8
```

```
FractionLength: 0
    IsAlias: false
    HeaderFile: ''
    Description: ''
IsScaledDouble =
    0
```

Return a `Simulink.NumericType` object with its `DataTypeOverride` property set to `Off`:

```
a = fixdt(0, 8, 2, 'DataTypeOverride', 'Off')

a =

Simulink.NumericType
    DataTypeMode: 'Fixed-point: binary point scaling'
    Signedness: 'Unsigned'
    WordLength: 8
    FractionLength: 2
    DataTypeOverride: Off
    IsAlias: false
    HeaderFile: ''
    Description: ''
```

See Also

`float` | `sfix` | `sfrac` | `sint` | `ufix` | `ufrac` | `uint`

Topics

“Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer)

Introduced before R2006a

fixpt_evenspace_cleanup

Modify breakpoints of lookup table to have even spacing

Syntax

```
xdata_modified = fixpt_evenspace_cleanup(xdata,xdt,xscale)
```

Description

xdata_modified = fixpt_evenspace_cleanup(*xdata*,*xdt*,*xscale*) modifies breakpoints of a lookup table to have even spacing after quantization. By adjusting breakpoints to have even spacing after quantization, Simulink Coder generated code can exclude breakpoints from memory.

xdata is the breakpoint vector of a lookup table to make evenly spaced, such as 0:0.005:1. *xdt* is the data type of the breakpoints, such as `sfixed(16)`. *xscale* is the scaling of the breakpoints, such as 2^{-12} . Using these three inputs, fixpt_evenspace_cleanup returns the modified breakpoints in *xdata_modified*.

This function works only with nontunable data and considers data to have even spacing relative to the scaling slope. For example, the breakpoint vector [0 2 5], which has spacing value 2 and 3, appears to have uneven spacing. However, the difference between the maximum spacing 3 and the minimum spacing 2 equals 1. If the scaling slope is 1 or greater, a spacing variation of 1 represents a 1-bit change or less. In this case, the fixpt_evenspace_cleanup function considers a spacing variation of 1 bit or less to be even.

Modifications to breakpoints can change the numerical behavior of a lookup table. To check for changes, test the model using simulation, rapid prototyping, or other appropriate methods.

Examples

Modify breakpoints of a lookup table to have even spacing after quantization:

```
xdata = 0:0.005:1;  
xdt = sfix(16);  
xscale = 2^-12;  
xdata_modified = fixpt_evenspace_cleanup(xdata,xdt,xscale)
```

See Also

[fixdt](#) | [fixpt_interpl](#) | [fixpt_look1_func_approx](#) | [fixpt_look1_func_plot](#)
| [sfix](#) | [ufix](#)

Topics

“Effects of Spacing on Speed, Error, and Memory Usage” (Fixed-Point Designer)

“Create Lookup Tables for a Sine Function” (Fixed-Point Designer)

Introduced before R2006a

fixpt_interp1

Implement 1-D lookup table

Syntax

```
y = fixpt_interp1(xdata, ydata, x, xdt, xscale, ydt, yscale, rndmeth)
```

Description

`y = fixpt_interp1(xdata, ydata, x, xdt, xscale, ydt, yscale, rndmeth)` implements a one-dimensional lookup table to find output y for input x . If x falls between two $xdata$ values (breakpoints), y is the result of interpolating between the corresponding $ydata$ values. If x is greater than the maximum value in $xdata$, y is the maximum $ydata$ value. If x is less than the minimum value in $xdata$, y is the minimum $ydata$ value.

If the input data type xdt or the output data type ydt is floating point, `fixpt_interp1` performs the interpolation using floating-point calculations. Otherwise, `fixpt_interp1` uses integer-only calculations. These calculations handle the input scaling $xscale$ and the output scaling $yscale$ and obey the rounding method $rndmeth$.

Input Arguments

xdata

Vector of breakpoints for the lookup table, such as `linspace(0, 8, 33)`.

ydata

Vector of table data that correspond to the breakpoints for the lookup table, such as `sin(xdata)`.

x

Vector of input values for the lookup table to process, such as `linspace(-1, 9, 201)`.

xdt

Data type of input x , such as `sfixed(8)`.

xscale

Scaling for input x , such as 2^{-3} .

ydt

Data type of output y , such as `sfixed(16)`.

yscale

Scaling for output y , such as 2^{-14} .

rndmeth

Rounding mode supported by fixed-point Simulink blocks:

'Ceiling'	Round to the nearest representable number in the direction of positive infinity.
'Floor' (default)	Round to the nearest representable number in the direction of negative infinity.
'Nearest'	Round to the nearest representable number.
'Toward Zero'	Round to the nearest representable number in the direction of zero.

Examples

Interpolate outputs for x using a 1-D lookup table that approximates the sine function:

```
xdata = linspace(0,8,33).';
ydata = sin(xdata);
% Define input x as a vector of 201 evenly
% spaced points between -1 and 9 (includes
% values both lower and higher than the range
% of breakpoints in xdata)
x = linspace(-1,9,201).';
% Interpolate output values for x
```

```
y = fixpt_interp1(xdata,ydata,x,sfix(8),2^-3,sfix(16),...  
  2^-14,'Floor')
```

See Also

[fixpt_evenspace_cleanup](#) | [fixpt_look1_func_approx](#) |
[fixpt_look1_func_plot](#)

Topics

“Producing Lookup Table Data” (Fixed-Point Designer)

Introduced before R2006a

fixpt_look1_func_approx

Optimize fixed-point approximation of nonlinear function by interpolating lookup table data points

Syntax

```
[xdata,ydata,errworst] = fixpt_look1_func_approx('func',...  
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,nptsmax)  
[xdata,ydata,errworst] = fixpt_look1_func_approx('func',...  
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[])  
[xdata,ydata,errworst] = fixpt_look1_func_approx('func',...  
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,[],nptsmax)  
[xdata,ydata,errworst] = fixpt_look1_func_approx('func',...  
xmin,xmax,xdt,xscale,ydtydt,yscale,rndmeth,errmax,nptsmax,spacing)
```

Description

[xdata,ydata,errworst] = fixpt_look1_func_approx('func',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,nptsmax) returns the optimal breakpoints of a lookup table, an ideal function applied to the breakpoints, and the worst-case approximation error. The lookup table satisfies the maximum acceptable error and maximum number of points that you specify.

[xdata,ydata,errworst] = fixpt_look1_func_approx('func',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[]) returns the optimal breakpoints of a lookup table, an ideal function applied to the breakpoints, and the worst-case approximation error. The lookup table satisfies the maximum acceptable error that you specify.

[xdata,ydata,errworst] = fixpt_look1_func_approx('func',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,[],nptsmax) returns the optimal breakpoints of a lookup table, an ideal function applied to the breakpoints, and the worst-case approximation error. The lookup table satisfies the maximum number of points that you specify.

[xdata,ydata,errworst] = fixpt_look1_func_approx('func',...)

xmin, xmax, xdt, xscale, ydtydt, yscale, rndmeth, errmax, nptsmax, spacing) returns the optimal breakpoints of a lookup table, an ideal function applied to the breakpoints, and the worst-case approximation error. The lookup table satisfies the maximum acceptable error, maximum number of points, and breakpoint spacing that you specify.

In each case, `fixpt_look1_func_approx` interpolates between lookup table data points to optimize the fixed-point approximation. The inputs *xmin* and *xmax* specify the range over which to approximate the breakpoints. The inputs *xdt*, *xscale*, *ydt*, *yscale*, and *rndmeth* follow conventions used by fixed-point Simulink blocks.

The inputs *errmax*, *nptsmax*, and *spacing* are optional. Of these inputs, you must specify at least *errmax* or *nptsmax*. If you omit one of those two inputs, you must use brackets, [], in place of the omitted input. `fixpt_look1_func_approx` ignores that requirement for the lookup table.

If you do not specify *spacing*, and more than one spacing satisfies *errmax* and *nptsmax*, `fixpt_look1_func_approx` chooses in this order: power-of-2 spacing, even spacing, uneven spacing. This behavior applies when you specify both *errmax* and *nptsmax*, but not when you specify just one of the two.

Input Arguments

func

Function of *x* for which to approximate breakpoints. Enclose this expression in single quotes, for example, `'sin(2*pi*x)'`.

xmin

Minimum value of *x*.

xmax

Maximum value of *x*.

xdt

Data type of *x*.

xscale

Scaling for the x values.

ydt

Data type of y .

yscale

Scaling for the y values.

rndmeth

Rounding mode supported by fixed-point Simulink blocks:

'Ceiling'	Round to the nearest representable number in the direction of positive infinity.
'Floor' (default)	Round to the nearest representable number in the direction of negative infinity.
'Nearest'	Round to the nearest representable number.
'Toward Zero'	Round to the nearest representable number in the direction of zero.

errmax

Maximum acceptable error between the ideal function and the approximation given by the lookup table.

nptsmax

Maximum number of points for the lookup table.

spacing

Spacing of breakpoints for the lookup table:

'even'	Even spacing
'pow2'	Even, power-of-2 spacing
'unrestricted' (default)	Uneven spacing

If you specify...	The breakpoints of the lookup table...
<i>errmax</i> and <i>nptsmax</i>	Meet both criteria, if possible. The <i>errmax</i> requirement has higher priority than <i>nptsmax</i> . If the breakpoints cannot meet both criteria with the specified spacing, <i>nptsmax</i> does not apply.
<i>errmax</i> only	Meet the error criteria, and <code>fixpt_look1_func_approx</code> returns the fewest number of points.
<i>nptsmax</i> only	Meet the points criteria, and <code>fixpt_look1_func_approx</code> returns the smallest worst-case error.

Output Arguments

xdata

Vector of breakpoints for the lookup table.

ydata

Vector of values from applying the ideal function to the breakpoints.

errworst

Worst-case error, which is the maximum absolute error between the ideal function and the approximation given by the lookup table.

Examples

Approximate a fixed-point sine function using a lookup table:

```
func = 'sin(2*pi*x)';
% Define the range over which to optimize breakpoints
xmin = 0;
xmax = 0.25;
% Define the data type and scaling for the inputs
```

```
xdt = ufix(16);
xscale = 2^-16;
% Define the data type and scaling for the outputs
ydt = sfix(16);
yscale = 2^-14;
% Specify the rounding method
rndmeth = 'Floor';
% Define the maximum acceptable error
errmax = 2^-10;
% Choose even, power-of-2 spacing for breakpoints
spacing = 'pow2';
% Create the lookup table
[xdata,ydata,errworst] = fixpt_look1_func_approx(func,...
    xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[],spacing);
```

See Also

[fixpt_evenspace_cleanup](#) | [fixpt_interpl](#) | [fixpt_look1_func_plot](#)

Topics

“Producing Lookup Table Data” (Fixed-Point Designer)

“Use Lookup Table Approximation Functions” (Fixed-Point Designer)

Introduced before R2006a

fixpt_look1_func_plot

Plot fixed-point approximation function for lookup table

Syntax

```
fixpt_look1_func_plot(xdata, ydata, 'func', ...  
xmin, xmax, xdt, xscale, ydt, yscale, rndmeth)  
errworst = fixpt_look1_func_plot(xdata, ydata, 'func', ...  
xmin, xmax, xdt, xscale, ydt, yscale, rndmeth)
```

Description

`fixpt_look1_func_plot(xdata, ydata, 'func', ...
xmin, xmax, xdt, xscale, ydt, yscale, rndmeth)` plots a lookup table approximation function and the error from the ideal function.

`errworst = fixpt_look1_func_plot(xdata, ydata, 'func', ...
xmin, xmax, xdt, xscale, ydt, yscale, rndmeth)` plots a lookup table approximation function and the error from the ideal function. The output `errworst` is the maximum absolute error.

You can use `fixpt_look1_func_approx` to generate `xdata` and `ydata`, the breakpoints and table data for the lookup table, respectively. `fixpt_look1_func_approx` applies the ideal function to the breakpoints in `xdata` to produce `ydata`. While this method is the easiest way to generate `ydata`, you can choose other values for `ydata` as input for `fixpt_look1_func_plot`. Choosing different values for `ydata` can, in some cases, produce a lookup table with a smaller maximum absolute error.

Input Arguments

xdata

Vector of breakpoints for the lookup table.

ydata

Vector of values from applying the ideal function to the breakpoints.

func

Function of x for which to approximate breakpoints. Enclose this expression in single quotes, for example, 'sin(2*pi*x) '.

xmin

Minimum value of x .

xmax

Maximum value of x .

xdt

Data type of x .

xscale

Scaling for the x values.

ydt

Data type of y .

yscale

Scaling for the y values.

rndmeth

Rounding mode supported by fixed-point Simulink blocks:

'Ceiling'

Round to the nearest representable number in the direction of positive infinity.

'Floor' (default)

Round to the nearest representable number in the direction of negative infinity.

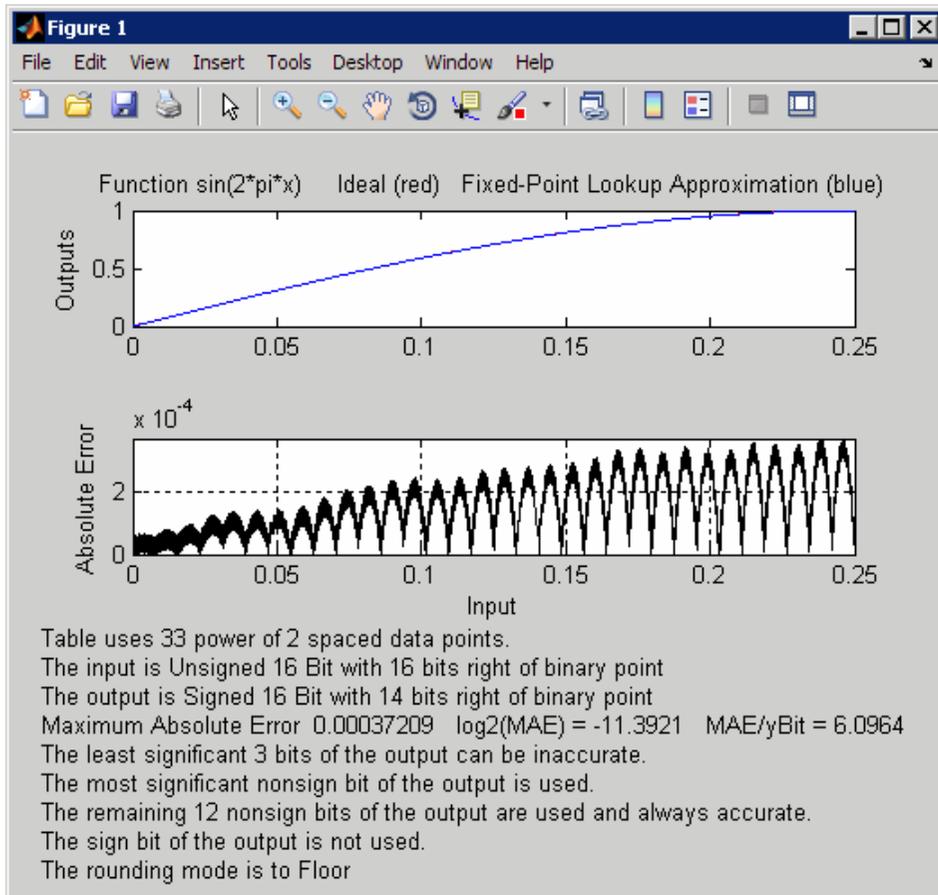
'Nearest'	Round to the nearest representable number.
'Toward Zero'	Round to the nearest representable number in the direction of zero.

Examples

Plot a fixed-point approximation of the sine function using data points generated by `fixpt_look1_func_approx`:

```
func = 'sin(2*pi*x)';
% Define the range over which to optimize breakpoints
xmin = 0;
xmax = 0.25;
% Define the data type and scaling for the inputs
xdt = ufix(16);
xscale = 2^-16;
% Define the data type and scaling for the outputs
ydt = sfix(16);
yscale = 2^-14;
% Specify the rounding method
rndmeth = 'Floor';
% Define the maximum acceptable error
errmax = 2^-10;
% Choose even, power-of-2 spacing for breakpoints
spacing = 'pow2';
% Generate data points for the lookup table
[xdata,ydata,errworst]=fixpt_look1_func_approx(func,...
    xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[],spacing);
% Plot the sine function (ideal and fixed-point) & errors
fixpt_look1_func_plot(xdata,ydata,func,xmin,xmax,...
    xdt,xscale,ydt,yscale,rndmeth);
```

`fixpt_look1_func_plot` plots the fixed-point sine function, using generated data points, and plots the error between the ideal function and the fixed-point function. The maximum absolute error and the number of points required appear on the plot. The error drops to zero at a breakpoint, but increases between breakpoints due to curvature differences between the ideal function and the line drawn between breakpoints.



The lookup table requires 33 points to achieve a maximum absolute error of $2^{-11.3922}$.

See Also

[fixpt_evenspace_cleanup](#) | [fixpt_interp1](#) | [fixpt_look1_func_approx](#)

Topics

“Producing Lookup Table Data” (Fixed-Point Designer)

“Use Lookup Table Approximation Functions” (Fixed-Point Designer)

Introduced before R2006a

fixpt_set_all

Set property for each fixed-point block in subsystem

Syntax

```
fixpt_set_all(SystemName, fixptPropertyName, fixptPropertyValue)
```

Description

`fixpt_set_all(SystemName, fixptPropertyName, fixptPropertyValue)` sets the property `fixptPropertyName` of every applicable block in the model or subsystem `SystemName` to the value `fixptPropertyValue`

Examples

Set each fixed-point block in a model `Filter_1` to round towards the floor and saturate upon overflow:

```
% Round towards the floor
fixpt_set_all('Filter_1', 'RndMeth', 'Floor')

% Saturate upon overflow
fixpt_set_all('Filter_1', 'DoSatur', 'on')
```

Introduced before R2006a

fixptbestexp

Exponent that gives best precision for fixed-point representation of value

Syntax

```
out = fixptbestexp(RealWorldValue, TotalBits, IsSigned)
out = fixptbestexp(RealWorldValue, FixPtDataType)
```

Description

`out = fixptbestexp(RealWorldValue, TotalBits, IsSigned)` returns the exponent that gives the best precision for the fixed-point representation of *RealWorldValue*. *TotalBits* specifies the number of bits for the fixed-point number. *IsSigned* specifies whether the fixed-point number is signed: 1 indicates the number is signed and 0 indicates the number is not signed.

`out = fixptbestexp(RealWorldValue, FixPtDataType)` returns the exponent that gives the best precision based on the data type *FixPtDataType*.

Examples

Get the exponent that gives the best precision for the real-world value $4/3$ using a signed, 16-bit number:

```
out = fixptbestexp(4/3,16,1)

out =
    -14
```

Alternatively, specify the fixed-point data type:

```
out = fixptbestexp(4/3,sfix(16))

out =
    -14
```

This shows that the maximum precision representation of $4/3$ is obtained by placing 14 bits to the right of the binary point:

```
01.01010101010101
```

You can specify the precision of this representation in fixed-point blocks by setting the scaling to 2^{-14} or `2^fixptbestexp(4/3,16,1)`.

See Also

`fixptbestprec`

Introduced before R2006a

fixptbestprec

Determine maximum precision available for fixed-point representation of value

Syntax

```
out = fixptbestprec(RealWorldValue, TotalBits, IsSigned)
out = fixptbestprec(RealWorldValue, FixPtDataType)
```

Description

`out = fixptbestprec(RealWorldValue, TotalBits, IsSigned)` determines the maximum precision for the fixed-point representation of the real-world value specified by `RealWorldValue`. You specify the number of bits for the fixed-point number with `TotalBits`, and you specify whether the fixed-point number is signed with `IsSigned`. If `IsSigned` is 1, the number is signed. If `IsSigned` is 0, the number is not signed. The maximum precision is returned to `out`.

`out = fixptbestprec(RealWorldValue, FixPtDataType)` determines the maximum precision based on the data type specified by `FixPtDataType`.

Examples

Example 1

The following command returns the maximum precision available for the real-world value $4/3$ using a signed, 8-bit number:

```
out = fixptbestprec(4/3, 8, 1)
out =
    0.015625
```

Alternatively, you can specify the fixed-point data type:

```
out = fixptbestprec(4/3,sfix(8))  
  
out =  
    0.015625
```

This value means that the maximum precision available for $4/3$ is obtained by placing six bits to the right of the binary point since 2^{-6} equals 0.015625:

```
01.010101
```

Example 2

You can use the maximum precision as the scaling in fixed-point blocks. This enables you to use `fixptbestprec` to perform a type of autoscaling if you would like to designate a known range of your simulation. For example, if your known range is -13 to 22, and you are using a safety margin of 30%:

```
knownMax = 22;  
knownMin = -13;  
localSafetyMargin = 30;  
slope = max( fixptbestprec( (1+localSafetyMargin/100)* ...  
    [knownMax,knownMin], sfix(16) ) );
```

The variable `slope` can then be used in the expression that you specify for the **Output data type** parameter in a block mask. Be sure to select the **Lock output data type setting against changes by the fixed-point tools** check box in the same block to prevent the Fixed-Point Tool from overriding the scaling. If you know the range, you can use this technique in place of relying on a model simulation to provide the range to the autoscaling tool, as described in `autofixexp` in the Fixed-Point Designer documentation.

See Also

`fixptbestexp`

Introduced before R2006a

float

Create `Simulink.NumericType` object describing floating-point data type

Syntax

```
a = float('single')
a = float('double')
```

Description

`a = float('single')` returns a `Simulink.NumericType` object that describes the data type of an IEEE single (32 total bits, 8 exponent bits).

`a = float('double')` returns a `Simulink.NumericType` object that describes the data type of an IEEE double (64 total bits, 11 exponent bits).

Note `float` is a legacy function. In new code, use `fixdt` instead. In existing code, replace `float('single')` with `fixdt('single')` and `float('double')` with `fixdt('double')`.

Examples

Define an IEEE single data type.

```
>> a = float('single')
a =
    NumericType with properties:
        DataTypeMode: 'Single'
        IsAlias: 0
        DataScope: 'Auto'
        HeaderFile: ''
        Description: ''
```

See Also

`Simulink.NumericType` | `fixdt` | `sfix` | `sfrac` | `sint` | `ufix` | `ufrac` | `uint`

Introduced before R2006a

frameedit

Edit print frames for Simulink and Stateflow block diagrams

Syntax

```
frameedit  
frameedit filename
```

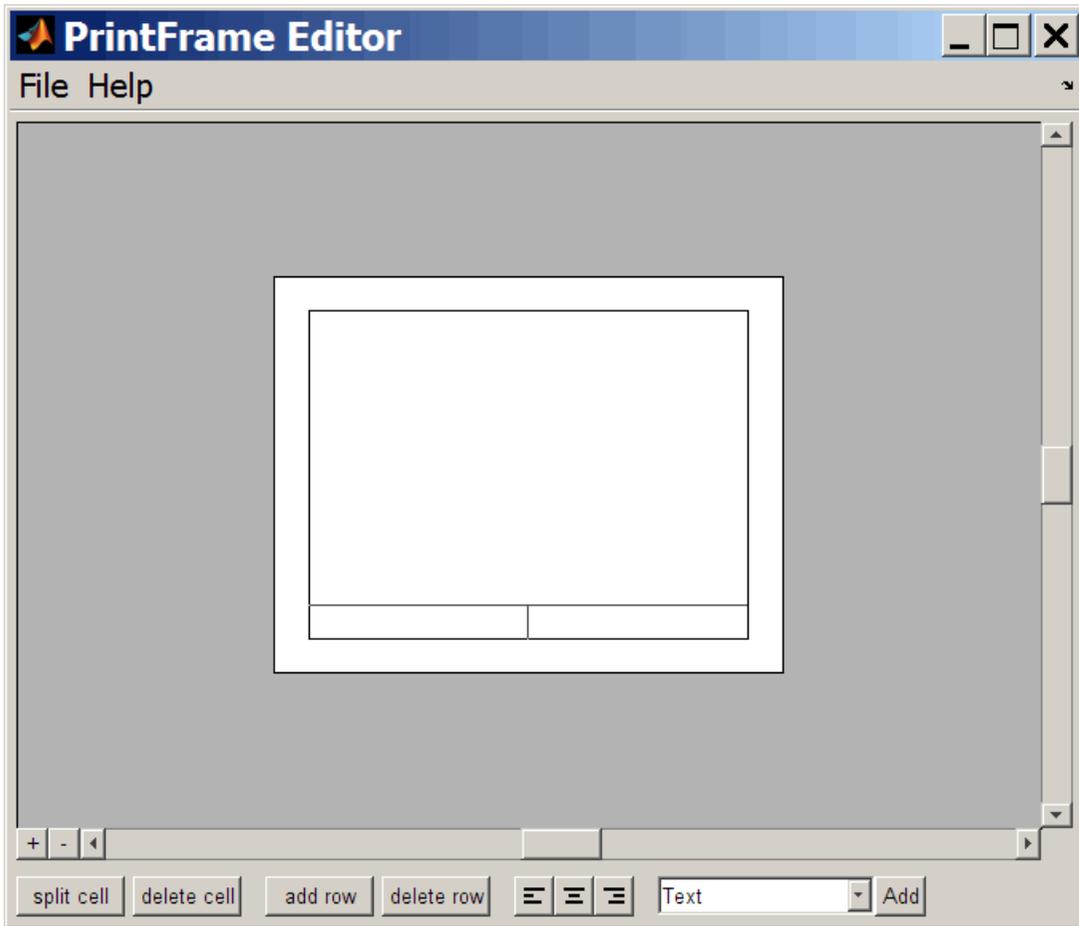
Description

`frameedit` starts the **PrintFrame Editor**, a graphical user interface you use to create borders for Simulink and Stateflow block diagrams. With no argument, `frameedit` opens the **PrintFrame Editor** window with a new file.

`frameedit filename` opens the **PrintFrame Editor** window with the specified `filename`, where `filename` is a figure file (`.fig`) previously created and saved using `frameedit`.

Tips

This illustrates the main features of the **PrintFrame Editor**.



Closing the PrintFrame Editor

To close the **PrintFrame Editor** window, click the close box in the upper right corner, or select **Close** from the **File** menu.

Printing Simulink Block Diagrams with Print Frames

Select **Print** from the Simulink **File** menu. Check the **Frame** box and supply the filename for the print frame you want to use. Click **OK** in the **Print** dialog box.

Getting Help for the PrintFrame Editor

For further instructions on using the PrintFrame Editor, select **PrintFrame Editor Help** from the **Help** menu in the PrintFrame Editor.

Introduced in R2008b

fxptdlg

Start Fixed-Point Tool

Syntax

```
fxptdlg('modelname')
```

Description

`fxptdlg('modelname')` starts the Fixed-Point Tool for the Simulink model specified by `modelname`. You can also access this tool by the following methods:

- From the Simulink **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- From a subsystem context (right-click) menu, select **Fixed-Point Tool**.

In conjunction with Fixed-Point Designer software, the Fixed-Point Tool provides convenient access to:

- Model and subsystem parameters that control the signal logging, fixed-point instrumentation mode, and data type override. (see “Model Parameters” on page 6-2)
- Plotting capabilities that enable you to plot data that resides in the MATLAB workspace, namely, simulation results associated with Scope, To Workspace, and root-level Outport blocks, in addition to logged signal data (see “Signal Logging” in the *Simulink User's Guide*)
- An interactive automatic data typing feature that proposes fixed-point data types for appropriately configured objects in your model, and then allows you to selectively accept and apply the data type proposals

You can launch the Fixed-Point Tool for any system or subsystem, and the tool controls the object selected in its **System under design** pane. If Fixed-Point Designer software is installed, the Fixed-Point Tool displays the name, data type, design minimum and maximum values, minimum and maximum simulation values, and scaling of each model object that logs fixed-point data. Additionally, if a signal saturates or overflows, the tool displays the number of times saturation or overflow occurred.

Note The Fixed-Point Tool works only for models that simulate in Normal mode. The tool does not work when you simulate your model in Accelerator or Rapid Accelerator mode.

Overriding Fixed-Point Specifications

Most of the functionality in the Fixed-Point Tool is for use with the Fixed-Point Designer software. However, even if you do not have Fixed-Point Designer software, you can configure data type override settings to simulate a model that specifies fixed-point data types. In this mode, the Simulink software temporarily overrides fixed-point data types with floating-point data types when simulating the model.

Note If you use `fi` objects or embedded numeric data types in your model or workspace, you might introduce fixed-point data types into your model. You can set `fipref` to prevent the checkout of a Fixed-Point Designer license.

To simulate a model without using Fixed-Point Designer:

- 1 Enter the following at the command line.

```
set_param(gcs, 'DataTypeOverride', 'Double', ...
'DataTypeOverrideAppliesTo', 'AllNumericTypes', ...
'MinMaxOverflowLogging', 'ForceOff')
```

- 2 If you use `fi` objects or embedded numeric data types in your model, set the `fipref` `DataTypeOverride` property to `TrueDoubles` or `TrueSingles` (to be consistent with the model-wide data type override setting) and the `DataTypeOverrideAppliesTo` property to `All numeric types`.

For example, at the MATLAB command line, enter:

```
p = fipref('DataTypeOverride', 'TrueDoubles', ...
'DataTypeOverrideAppliesTo', 'AllNumericTypes');
```

See Also

“Propose Fraction Lengths Using Simulation Range Data” (Fixed-Point Designer) | Fixed-Point Tool

Introduced before R2006a

gcb

Get path name of current block

Syntax

```
b1 = gcb
b1 = gcb(sys)
```

Description

`b1 = gcb` returns the full block path name of the current block in the current system. The current block is:

- The most recently clicked block
- The S-Function block currently executing its corresponding MATLAB function
- The block whose callback routine is being executed
- The block whose mask is being evaluated as part of the `MaskInitialization` parameter evaluation
- The last block loaded after opening a model

`b1 = gcb(sys)` returns the full block path name of the current block in the specified system. Load the system first.

Input Arguments

sys — System that contains the block

character vector

System that contains the block, specified as a character vector.

Example: `'vdp' 'sldemo_fuelsys/fuel_rate_control'`

Examples

Get Path Name of Most Recently Selected Block

Open a model.

```
vdp
```

In the model, select a block. For example, select the Gain block. Then, enter `gcb` at the command prompt.

```
gcb
```

```
ans =
```

```
vdp/Mu
```

Get Parameters of Current Block

Open a model.

```
vdp
```

Select the Gain block.

Use the value of `gcb` with `get_param` to get the value of the Gain parameter.

```
x = get_param(gcb, 'Gain')
```

```
x =
```

```
1
```

Get Current Block in Specified System

Load the model.

```
load_system('sldemo_fuelsys');
```

Get the current block in the `fuel_rate_control` subsystem.

```
b1 = gcb('sldemo_fuelsys/fuel_rate_control')  
  
b1 =  
  
sldemo_fuelsys/fuel_rate_control/validate_sample_time
```

See Also

gcbh | gcs | get_param

Topics

“Mask Callback Code”

Introduced before R2006a

gcbh

Get handle of current block

Syntax

```
gcbh
```

Description

`gcbh` returns the handle of the current block in the current system.

You can use this command to identify or address blocks that have no parent system. The command should be most useful to blockset authors.

Examples

This command returns the handle of the most recently selected block.

```
gcbh
```

```
ans =
```

```
    281.0001
```

See Also

`gcb` | `getSimulinkBlockHandle`

Introduced before R2006a

gcs

Get pathname of current system

Syntax

```
gcs
```

Description

`gcs` returns the full pathname of the current system.

The current system is one of these:

- During editing, the current system is the system or subsystem most recently clicked.
- During simulation of a system that contains S-Function blocks, the current system is the system or subsystem containing the S-Function block that is currently being evaluated.
- During callbacks, the current system is the system containing any block whose callback routine is being executed.
- During evaluation of `MaskInitialization`, the current system is the system containing the block whose mask is being evaluated.

The current system is always the current model or a subsystem of the current model. Use `bdroot` to get the current model.

Examples

This example returns the path of the system that contains the most recently selected block.

```
gcs
ans =
    clutch/Locked
```

See Also

`bdroot` | `gcb`

Introduced before R2006a

get_param

Get parameter names and values

Syntax

```
ParamValue = get_param(Object,Parameter)
```

Description

`ParamValue = get_param(Object,Parameter)` returns the name or value of the specified parameter for the specified model or block object. Open or load the Simulink model first.

Tip If you make multiple calls to `get_param` for the same block, then specifying the block using a numeric handle is more efficient than using the full block path. Use `getSimulinkBlockHandle` to get a block handle.

For parameter names, see:

- “Model Parameters” on page 6-2
- “Block-Specific Parameters” on page 6-128
- “Common Block Properties” on page 6-109

Examples

Get a Block Parameter Value and a Model Parameter Value

Load the `vdp` model.

```
load_system('vdp');
```

Get the value for the `Expression` block parameter.

```
BlockParameterValue = get_param('vdp/Fcn', 'Expression')
```

```
BlockParameterValue =  
    1 - u*u
```

Get the value for the SolverType model parameter.

```
SolverType = get_param('vdp', 'SolverType')
```

```
SolverType =  
    Variable-step
```

Get Root Parameter Names and Values

Get a list of global parameter names by finding the difference between the Simulink root parameter names and the model parameter names.

```
RootParameterNames = fieldnames(get_param(0, 'ObjectParameters'));  
load_system('vdp')  
ModelParameterNames = fieldnames(get_param('vdp', 'ObjectParameters'));  
GlobalParameterNames = setdiff(RootParameterNames, ModelParameterNames)
```

```
GlobalParameterNames =  
    'AutoSaveOptions'  
    'CacheFolder'  
    'CallbackTracing'  
    'CharacterEncoding'  
    . . .  
    'CurrentSystem'
```

Get the value of a global parameter.

```
GlobalParameterValue = get_param(0, 'CurrentSystem')
```

```
GlobalParameterValue =  
    vdp
```

Get Model Parameter Names and Values

Get a list of model parameters for the vdp model .

```
load_system('vdp')
ModelParameterNames = get_param('vdp','ObjectParameters')

ModelParameterNames =
    Name: [1x1 struct]
    Tag: [1x1 struct]
    Description: [1x1 struct]
    Type: [1x1 struct]
    Parent: [1x1 struct]
    Handle: [1x1 struct]
    .
    .
    .
    Version: [1x1 struct]
```

Get the current value of the ModelVersion model parameter for the vdp model.

```
ModelParameterValue = get_param('vdp','ModelVersion')

ModelParameterValue =
    1.6
```

Get All Blocks and a Parameter Value

Get a list of block paths and names for the vdp model.

```
load_system('vdp')
BlockPaths = find_system('vdp','Type','Block')

BlockPaths =
    'vdp/Fcn'
    'vdp/More Info'
    'vdp/More Info/Model Info'
    'vdp/Mu'
    'vdp/Mux'
    'vdp/Product'
    'vdp/Scope'
    'vdp/Sum'
    'vdp/x1'
    'vdp/x2'
    'vdp/Out1'
    'vdp/Out2'
```

Get a list of block dialog parameters for the Fcn block.

```
BlockDialogParameters = get_param('vdp/Fcn','DialogParameters')
```

```
BlockDialogParameters =  
    Expr: [1x1 struct]  
    SampleTime: [1x1 struct]
```

Get the value for the Expr block parameter.

```
BlockParameterValue = get_param('vdp/Fcn', 'Expr')
```

```
BlockParameterValue =  
    1 - u*u
```

Get a Block Parameter Value Using a Block Handle

If you make multiple calls to `get_param` for the same block, then using the block handle is more efficient than specifying the full block path as a character vector, e.g., `'vdp/Fcn'`.

You can use the block handle in subsequent calls to `get_param` or `set_param`. If you examine the handle, you can see that it contains a double. Do not try to use the number of a handle alone (e.g., `5.007`) because you usually need to specify many more digits than MATLAB displays. Instead, assign the handle to a variable and use that variable name to specify a block.

Use `getSimulinkBlockHandle` to load the vdp model if necessary (by specifying `true`), and get a handle to the FCN block.

```
fcnblockhandle = getSimulinkBlockHandle('vdp/Fcn', true);
```

Use the block handle with `get_param` and get the value for the Expr block parameter.

```
BlockParameterValue = get_param(fcnblockhandle, 'Expression')
```

```
BlockParameterValue =  
    1 - u*u
```

Display Block Types for all Blocks in a Model

Get a list of block paths and names for the vdp model.

```
load_system('vdp')  
BlockPaths = find_system('vdp', 'Type', 'Block')
```

```
BlockPaths =
    'vdp/Fcn'
    'vdp/More Info'
    'vdp/More Info/Model Info'
    'vdp/Mu'
    'vdp/Mux'
    'vdp/Product'
    'vdp/Scope'
    'vdp/Sum'
    'vdp/x1'
    'vdp/x2'
    'vdp/Out1'
    'vdp/Out2'
```

Get the value for the `BlockType` parameter for each of the blocks in the `vdp` model.

```
BlockTypes = get_param(BlockPaths, 'BlockType')
```

```
BlockTypes =
    'Fcn'
    'SubSystem'
    'SubSystem'
    'Gain'
    'Mux'
    'Product'
    'Scope'
    'Sum'
    'Integrator'
    'Integrator'
    'Outport'
    'Outport'
```

- “Associating User Data with Blocks”
- “Use MATLAB Commands to Change Workspace Data”

Input Arguments

Object — Name or handle of a model or block, or root

handle | character vector | cell array of character vectors | 0

Handle or name of a model or block, or root, specified as a numeric handle or a character vector, a cell array of character vectors for multiple blocks, or 0 for root. A numeric

handle must be a scalar. You can also get parameters of lines and ports, but you must use numeric handles to specify them.

Tip If you make multiple calls to `get_param` for the same block, then specifying a block using a numeric handle is more efficient than using the full block path. Use `getSimulinkBlockHandle` to get a block handle. Do not try to use the number of a handle alone (e.g., 5.007) because you usually need to specify many more digits than MATLAB displays. Assign the handle to a variable and use that variable name to specify a block.

Specify 0 to get root parameter names, including global parameters and model parameters for the current Simulink session.

- Global parameters include Editor preferences and Simulink Coder parameters.
- Model parameters include configuration parameters, Simulink Coder parameters, and Simulink Code Inspector™ parameters.

Example: 'vdp/Fcn'

Parameter — Parameter of model or block, or root
character vector

Parameter of model or block, or root, specified as a character vector or 0 for root. The table shows special cases.

Specified Parameter	Result
'ObjectParameters'	Returns a structure array with the parameter names of the specified object (model, block, or root) as separate fields in the structure.
'DialogParameters'	Returns a structure array with the block dialog box parameter names as separate fields in the structure. If the block has a mask, the function instead returns the mask parameters.

Specified Parameter	Result
Parameter name, e.g., 'BlockType'. Specify any model or block parameter, or block dialog box parameter.	Returns the value of the specified model or block parameter. If you specified multiple blocks as a cell array, returns a cell array with the values of the specified parameter common to all blocks. All of the specified blocks in the cell array must contain the parameter, otherwise the function returns an error.

Example: 'ObjectParameters'

Data Types: char

Output Arguments

ParamValue — The name or value of the specified parameter for the specified model or block, or root

any data type, depending on the parameter

The name or value of the specified parameter for the specified model or block, or root. If you specify multiple objects, the output is a cell array of objects. The table shows special cases.

Specified Parameter	ParamValue Returned
'ObjectParameters'	A structure array with the parameter names of the specified object (model, block, or root) as separate fields in the structure.
'DialogParameters'	A structure array with the block dialog box parameter names as separate fields in the structure. If the block has a mask, the function instead returns the mask parameters.

Specified Parameter	ParamValue Returned
Parameter name, e.g., 'BlockType'	The value of the specified model or block parameter. If multiple blocks are specified as a cell array, returns a cell array with the values of the specified parameter common to all blocks.

If you get the root parameters by specifying `get_param(0, 'ObjectParameters')`, then the output `ParamValue` is a structure array with the root parameter names as separate fields in the structure. Each parameter field is a structure containing these fields:

- **Type** — Parameter type values are: 'boolean', 'string', 'int', 'real', 'point', 'rectangle', 'matrix', 'enum', 'ports', or 'list'
- **Enum** — Cell array of enumeration character vector values that applies only to 'enum' parameter types
- **Attributes** — Cell array of character vectors defining the attributes of the parameter. Values are: 'read-write', 'read-only', 'read-only-if-compiled', 'write-only', 'dont-eval', 'always-save', 'never-save', 'nondirty', or 'simulation'

See Also

`bdroot` | `find_system` | `gcb` | `gcs` | `getSimulinkBlockHandle` | `set_param`

Topics

“Associating User Data with Blocks”

“Use MATLAB Commands to Change Workspace Data”

“Model Parameters” on page 6-2

“Block-Specific Parameters” on page 6-128

“Common Block Properties” on page 6-109

Introduced before R2006a

getActiveConfigSet

Get model's active configuration set or configuration reference

Syntax

```
myConfigObj = getActiveConfigSet(model)
```

Arguments

model

The name of an open model, or `gcs` to specify the current model

Description

`getActiveConfigSet` returns the configuration set or configuration reference (configuration object) that is the active configuration object of model.

Examples

The following example returns the active configuration object of the current model. The code is the same whether the object is a configuration set or configuration reference.

```
myConfigObj = getActiveConfigSet(gcs);
```

See Also

`attachConfigSet` | `attachConfigSetCopy` | `closeDialog` | `detachConfigSet` | `getConfigSet` | `getConfigSets` | `openDialog` | `setActiveConfigSet`

Topics

“Manage a Configuration Set”

“Manage a Configuration Reference”

Introduced before R2006a

getCallbackAnnotation

Get annotation executing callback

Syntax

```
ann = getCallbackAnnotation
```

Description

`ann = getCallbackAnnotation` gets the annotation from which a callback was invoked. Invoke a callback from an annotation click function. After you get the annotation, you can, for example, get text or parameters from the annotation to use someplace else in your model.

For information on click functions, see “Associate a Click Function with an Annotation”.

Examples

Click Annotation to Change Parameter Value

Invoke a callback by way of an annotation click function. This example shows how to change a parameter value on a block to the value shown on an annotation.

Open `vdp`. Add and position two annotations. Each annotation displays a different value.

```
open_system('vdp');  
an1 = Simulink.Annotation('vdp/1');  
an1.position = [100,300];  
an2 = Simulink.Annotation('vdp/3');  
an2.position = [150,300];
```

Assign a click function to each annotation. The click function uses `getAnnotationCallback` to get the annotation instance. Get the text from each annotation and use it to set the parameter on the Gain block (`Mu`).

```
an1.ClickFcn = 'ann = getCallbackAnnotation; v = ann.Text; set_param(''vdp/Mu'', ''Gain'  
an2.ClickFcn = 'ann = getCallbackAnnotation; v = ann.Text; set_param(''vdp/Mu'', ''Gain'
```

Click each annotation. When you click, the gain value on the Mu block changes to the number shown on the annotation.

Output Arguments

ann — Annotation

`Simulink.Annotation` instance

Annotation, returned as a `Simulink.Annotation` instance.

See Also

`Simulink.Annotation`

Topics

“Associate a Click Function with an Annotation”

Introduced before R2006a

getComponent

Get a configuration set component

Syntax

```
component = getComponent(cs, componentName)
```

Description

`component = getComponent(cs, componentName)` returns the specified component from a configuration set. With no component name specified, returns a list of the components contained in the configuration set.

Examples

Get a Component for a Configuration Set

Get the solver component for the active configuration set.

Get the active configuration set of the currently selected model.

```
hCs = getActiveConfigSet(gcs);
```

Get the solver component of the active configuration set.

```
hSolverConfig = getComponent(hCs, 'Solver');
```

Get List of Components for an Active Configuration Set

Get a list of the components contained in the active configuration set.

Get the active configuration set of the currently selected model.

```
hCs = getActiveConfigSet(gcs);
```

Get the list of components contained in the configuration set.

```
getComponent(hCs)
```

The code displays the names of the components at the MATLAB command line.

Input Arguments

cs — Configuration set object

`ConfigSet` object

Configuration set, specified as a `ConfigSet` object

componentName — Component name

character vector

Name of a component object, specified as a character vector. If a component name is not specified, the function displays a list of the components contained in the configuration set at the MATLAB command line.

Example: 'Solver'

Output Arguments

component — Component

`Simulink.ConfigComponent` object

A component in the configuration set, returned as an instance of a `Simulink.ConfigComponent` object

See Also

Topics

“About Configuration Sets”

“Manage a Configuration Set”

Introduced before R2006a

getConfigSet

Get one of model's configuration sets or configuration references

Syntax

```
myConfigObj = getConfigSet(model, configObjName)
```

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

`configObjName`

The name of a configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

Description

`getConfigSet` returns the configuration set or configuration reference (configuration object) that is attached to `model` and is named `configObjName`. If no such object exists, an error occurs.

Examples

The following example returns the configuration object that is named `DevConfig` and attached to the current model. The code is the same whether `DevConfig` is a configuration set or configuration reference.

```
myConfigObj = getConfigSet(gcs, 'DevConfig');
```

See Also

`attachConfigSet` | `attachConfigSetCopy` | `closeDialog` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSets` | `openDialog` | `setActiveConfigSet`

Topics

“Manage a Configuration Set”

“Manage a Configuration Reference”

Introduced before R2006a

getConfigSets

Get names of all of model's configuration sets or configuration references

Syntax

```
myConfigObjNames = getConfigSets(model)
```

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

Description

`getConfigSets` returns a cell array of character vectors specifying the names of all configuration sets and configuration references (configuration objects) attached to `model`.

Examples

The following example obtains the names of the configuration objects attached to the current model.

```
myConfigObjNames = getConfigSets(gcs)
```

See Also

`attachConfigSet` | `attachConfigSetCopy` | `closeDialog` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `openDialog` | `setActiveConfigSet`

Topics

“Manage a Configuration Set”

“Manage a Configuration Reference”

Introduced before R2006a

getfullname

Get pathname of block or line

Syntax

```
path=getfullname(handle)
```

Description

`path=getfullname(handle)` returns the full pathname of the block or line specified by `handle`.

Examples

`getfullname(gcb)` returns the pathname of the block currently selected in the model editor's window.

The following code returns the pathname of the line currently selected in the model editor's window.

```
line = find_system(gcs, 'SearchDepth', 1, 'FindAll', 'on', ...  
    'Type', 'line', 'Selected', 'on');  
path = getfullname(line);
```

See Also

`find_system` | `gcb`

Introduced in R2007a

getInputString

Create comma-separated list of variables to map

Syntax

```
externalInputString = getInputString(inputmap, 'base')
```

```
externalInputString = getInputString(inputmap, filename)
```

```
externalInputString = getInputString(inputmap)
```

Description

`externalInputString = getInputString(inputmap, 'base')` creates an input character vector using the supplied mapping `inputmap` and the variables loaded in the base workspace ('base').

This function generates a comma-separated list of variables (input character vector) to be mapped. You can then use this list:

- As input to the `sim` command. Load the variables in the base workspace first.
- As input for the **Configuration Parameters > Data Import/Export > Input** parameter. Copy the contents of the input character vector into the text field.

This function is most useful if you have created a custom mapping.

`externalInputString = getInputString(inputmap, filename)` creates an input character vector using the supplied mapping `inputmap` and the variables defined in `filename`.

`externalInputString = getInputString(inputmap)` creates an input character vector using the signals from the most recently created mapping.

Examples

Create an input character vector from the base workspace

Create an input character vector from the base workspace and simulate a model.

Open the model

```
slexAutotransRootInportsExample;
```

Create signal variables in the base workspace

```
Throttle = timeseries(ones(10,1)*10);  
Brake     = timeseries(zeros(10,1));
```

Create a mapping (inputMap) for the model.

```
inputMap = getRootInportMap('model', ...  
    'slexAutotransRootInportsExample', ...  
    'signalName', {'Throttle', 'Brake'}, ...  
    'blockName', {'Throttle', 'Brake'});
```

Call `getInputString` with `inputMap` and 'base' as inputs.

```
externalInputString = getInputString(inputMap, 'base')  
  
externalInputString =
```

```
Throttle, Brake
```

Simulate the model with the input character vector.

```
sim('slexAutotransRootInportsExample', 'ExternalInput', ...  
externalInputString);
```

Create an external input character vector from variables in a MAT-file

Create an external input character vector from variables in a MAT-file named `input.mat`.

In a writable folder, create a MAT-file with input variables.

```
Throttle = timeseries(ones(10,1)*10);
Brake     = timeseries(zeros(10,1));
save('input.mat','Throttle','Brake');
```

Open the model.

```
slexAutotransRootInportsExample;
```

Create map object.

```
inputMap = getRootInportMap('model',...
'slexAutotransRootInportsExample',...
'signalName',{ 'Throttle','Brake'},...
'blockName',{ 'Throttle','Brake'});
```

Get the resulting input character vector.

```
externalInputString = getInputString(inputMap,'input.mat')
```

```
externalInputString =
```

```
Throttle,Brake
```

Load variables from the base workspace for the simulation.

```
load('input.mat');
```

Simulate the model.

```
sim('slexAutotransRootInportsExample','ExternalInput',...
externalInputString);
```

Create an external input character vector from only an input map

Create an input character vector from only an input map vector and simulate the model.

Open the model.

```
slexAutotransRootInportsExample;
```

Create signal variables in the base workspace

```
Throttle = timeseries(ones(10,1)*10);
Brake     = timeseries(zeros(10,1));
```

Create a mapping vector for the model.

```
inputMap = getS1RootInportMap('model', 'slexAutotransRootInportsExample', ...  
    'MappingMode', 'BlockName', ...  
    'signalName', {'Throttle', 'Brake'}, ...  
    'signalValue', {Throttle, Brake});
```

Get the resulting input character vector.

```
externalInputString = getInputString(inputMap)
```

Simulate the model with the input character vector.

```
sim('slexAutotransRootInportsExample', 'ExternalInput', ...  
externalInputString);
```

Alternatively, if you want to input the list of variables through the Configuration Parameters dialog, copy the contents of `externalInputString` (`Throttle, Brake`) into the **Data Import/Export > Input** parameter. Apply the changes, and then simulate the model.

Input Arguments

inputmap — Map object

character vector

Map object, as returned from the `getRootInportMap` or `getS1RootInportMap` functions.

filename — Input variables

MAT-file name as character vector

Input variables, contained in a MAT-file. The file contains variables to map.

Example: `'data.mat'`

Data Types: `char`

Output Arguments

externalInputString — External input

comma-separated character vector

External input, returned as a comma-separated character vector. The character vector contains root inport information that you can specify to the `sim` command or the **Configuration Parameters > Data Import/Export > Input** parameter.

See Also

`getRootInportMap` | `getSlRootInportMap`

Topics

“Map Root Inport Signal Data”

Introduced in R2013a

getRootInportMap

Create custom object to map signals to root-level inports

Syntax

```
map = getRootInportMap('Empty');  
map = getRootInportMap(model, mdl, Name, Value);  
map = getRootInportMap(inputmap, map, Name, Value);
```

Description

`map = getRootInportMap('Empty');` creates an empty map object, *map*. Use this map object to set up an empty custom mapping object. Load the model before using this function. If you do not load the model first, the function loads the model to make the mapping and then closes the model afterwards.

`map = getRootInportMap(model, mdl, Name, Value);` creates a map object for *model*, *mdl*, with block names and signal names specified. Load the model before using this function. If you do not load the model first, the function loads the model to make the mapping and then closes the model afterwards. To create a comma-separated list of variables to map from this object, use the `getInputString` function.

`map = getRootInportMap(inputmap, map, Name, Value);` overrides the mapping object with the specified property. You can override only the properties *model*, *blockName*, and *signalName*. Load the model before using this function. If you do not load the model first, the function loads the model to make the mapping and then closes the model afterwards. To create a comma-separated list of variables to map from this object, use the `getInputString` function.

Use the `getRootInportMap` function when creating a custom mapping mode to map data to root-level inports. See the example file `BlockNameIgnorePrefixMap.m`, located in `matlabroot/help/toolbox/simulink/examples`, for an example of a custom mapping algorithm.

Input Arguments

Empty

Create an empty map object.

Default: none

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

model

Name of model to associate with the root inport map.

Default: None

blockName

Block names of root-level input ports. The tool assigns data to ports according to the name of the root-inport block. If the tool finds a data element whose name matches the name of a root-inport block, it maps the data to the corresponding port.

The value for this argument can be:

Block name of root-level input ports.

Cell array containing multiple block names of root-level input ports.

Default: None

signalName

Signal names to be mapped. The tool assigns data to ports according to the name of the signal on the port. If the tool finds a data element whose name matches the name of a signal at a port, it maps the data to the corresponding port.

The value for this argument can be:

Signal name to be mapped.

Cell array containing multiple signal names of signals to be mapped.

Default: None

inputmap

Name of mapping object to override.

Default: None

Output Arguments

map

Custom object that you can use to map data to root-level input port. To create a comma-separated list of variables to map from this object, use the `getInputString` function.

Examples

Empty Mapping Object

Create an empty custom mapping object.

```
map = getRootInportMap('Empty')
```

```
map =
```

```
1x0 InputMap array with properties:
```

```
Type  
DataSourceName  
Destination
```

Simple Mapping Object

Create a simple mapping object using a MATLAB time series object.

Create a time series object, `signalIn1`.

```

signalIn1 = timeseries((1:10)')

Common Properties:
    Name: 'unnamed'
    Time: [10x1 double]
    TimeInfo: [1x1 tsdata.timemetadata]
    Data: [10x1 double]
    DataInfo: [1x1 tsdata.datametadata]

```

Create a mapping object for the time series object for the model, `ex_minportsOnlyModel`.

```

modelFile = fullfile(matlabroot, 'help', 'toolbox', 'simulink', ...
    'examples', 'ex_minportsOnlyModel');
load_system(modelFile);
map = getRootInportMap('model', 'ex_minportsOnlyModel', ...
    'blockName', 'In1', 'signalname', 'signalIn1')

```

```
map =
```

```

InputMap with properties:
    Type: 'Inport'
    DataSourceName: 'signalIn1'
    Destination: [1x1 Simulink.iospecification.Destination]

```

Mapping Object with Vectors

Create a mapping object using vectors of block names and signal names for the model `ex_minportsOnlyModel`.

Create a mapping object of vectors.

```

modelFile = fullfile(matlabroot, 'help', 'toolbox', 'simulink', ...
    'examples', 'ex_minportsOnlyModel');
load_system(modelFile);
map = getRootInportMap('model', 'ex_minportsOnlyModel', ...
    'blockName', {'In1' 'In2'}, ...
    'signalname', {'signalIn1' 'signalIn2'})

```

```
map =
```

```
1x2 InputMap array with properties:
```

```
Type
DataSourceName
Destination
```

Overriding Maps

Create a mapping object that contains the signal *var2*, then override *var2* with *var1*.

Create a mapping object of vectors.

```
% Load the model and define variables
modelFile = fullfile(matlabroot, 'help', 'toolbox', 'simulink', ...
    'examples', 'ex_minportsOnlyModel');
load_system(modelFile);
modelValue = 'ex_minportsOnlyModel';
blockNameValue = 'In1';
signalNameValue = 'var2';
portType = 'Inport';

% Define var1 and override var2 with var1
signalNameToOverload = 'var1';
mapToOverload = getRootInportMap('model', modelValue, ...
    'blockName', blockNameValue, ...
    'signalName', signalNameToOverload)

mapToOverload =

    InputMap with properties:

        Type: 'Inport'
    DataSourceName: 'var1'
    Destination: [1x1 Simulink.iospecification.Destination]
```

- “Create Custom Mapping File Function”

Tips

- Load the model before running this function.
- If your custom mapping mode similar to an existing Simulink mapping mode, consider using the `getSlRootInportMap` function instead.

See Also

`getInputString` | `getSlRootInportMap`

Topics

“Create Custom Mapping File Function”

Introduced in R2012b

getSimulinkBlockHandle

Get block handle from block path

Syntax

```
handle = getSimulinkBlockHandle(path)
handle = getSimulinkBlockHandle(path,true)
```

Description

`handle = getSimulinkBlockHandle(path)` returns the numeric handle of the block specified by `path`, if it exists in a loaded model or library. Returns `-1` if the block is not found. Library links are resolved where necessary.

Use the numeric handle returned by `getSimulinkBlockHandle` to manipulate the block in subsequent calls to `get_param` or `set_param`. This approach is more efficient than making multiple calls to these functions using the full block path. Do not try to use the number of a handle alone (e.g., `5.007`) because you usually need to specify many more digits than MATLAB displays. Assign the handle to a variable and use that variable name to specify a block. The handle applies only to the current MATLAB session.

Use `getSimulinkBlockHandle` to check whether a block path is valid. This approach is more efficient than calling `get_param` inside a `try` statement.

`handle = getSimulinkBlockHandle(path,true)` attempts to load the model or library containing the specified block path, and then checks if the block exists. No error is returned if the model or library is not found. Any models or libraries loaded this way remain in memory even if the function does not find a block with the specified path.

Examples

Get the Handle of a Block

Get the handle of the `Pilot` block.

```
load_system('f14')
handle = getSimulinkBlockHandle('f14/Pilot')

handle =

    562.0004
```

You can use the handle in subsequent calls to `get_param` or `set_param`.

Load the Model and Get the Block Handle

Load the model `f14` if necessary (by specifying `true`), and get the handle of the `Pilot` block.

```
handle = getSimulinkBlockHandle('f14/Pilot',true)

handle =

    562.0004
```

You can use the handle in subsequent calls to `get_param` or `set_param`.

Check If a Model Contains a Specific Block

Check whether the model `f14` is loaded and contains a block named `Pilot`. Valid handles are always greater than zero. If the function does not find the block, it returns `-1`.

```
valid_block_path = getSimulinkBlockHandle('f14/Pilot') > 0

valid_block_path =

    0
```

The model contains the block but the model is not loaded, so this command returns `0` because it cannot find the block.

Using `getSimulinkBlockHandle` to check whether a block path is valid is more efficient than calling `get_param` inside a `try` statement.

Input Arguments

path — Block path name

character vector | cell array of character vectors

Block path name, specified as a character vector or a cell array of character vectors.

Example: `'f14/Pilot'`

Data Types: `char`

Output Arguments

handle — Numeric handle of a block

double | array of doubles

Numeric handle of a block, returned as a double or an array of doubles. Valid handles are always greater than zero. If the function does not find the block, it returns `-1`. If the `path` input is a cell array of character vectors, then the output is a numeric array of handles.

Data Types: `double`

See Also

`get_param` | `set_param`

Introduced in R2015a

getSlRootInportMap

Create custom object to map signals to root-level inports using Simulink mapping mode

Syntax

```
inputMap = getSlRootInportMap('model', modelname, 'MappingMode',
mappingmode, 'SignalName', signalname, 'SignalValue', signalvalue)
[inputMap, hasASignal] = getSlRootInportMap('model',
modelname, 'MappingMode', mappingmode, 'SignalName',
signalname, 'SignalValue', signalvalue)
```

```
inputMap = getSlRootInportMap('model',
modelname, 'MappingMode', 'Custom', 'CustomFunction',
customfunction, 'SignalName', signalname, 'SignalValue', signalvalue)
[inputMap, hasASignal] = getSlRootInportMap('model',
modelname, 'MappingMode', 'Custom', 'CustomFunction',
customfunction, 'SignalName', signalname, 'SignalValue', signalvalue)
```

Description

`inputMap = getSlRootInportMap('model', modelname, 'MappingMode', mappingmode, 'SignalName', signalname, 'SignalValue', signalvalue)` creates a root inport map using one of the Simulink mapping modes. Load the model before using this function. If you do not load the model first, the function loads the model to make the mapping and then closes the model afterwards. To create a comma-separated list of variables to map from this object, use the `getInputString` function.

`[inputMap, hasASignal] = getSlRootInportMap('model', modelname, 'MappingMode', mappingmode, 'SignalName', signalname, 'SignalValue', signalvalue)` returns a vector of logical values specifying whether or not the root inport map has a signal associated with it. To create a comma-separated list of variables to map from this object, use the `getInputString` function.

```
inputMap = getSlRootInportMap('model',
modelname, 'MappingMode', 'Custom', 'CustomFunction',
```

`customfunction, 'SignalName', signalname, 'SignalValue', signalvalue)` creates a root inport map using a custom mapping mode specified in `customfunction`. Load the model before using this function. If you do not load the model first, the function loads the model to make the mapping and then closes the model afterwards. To create a comma-separated list of variables to map from this object, use the `getInputString` function.

```
[inputMap, hasASignal] = getSIRootInportMap('model',  
modelname, 'MappingMode', 'Custom', 'CustomFunction',  
customfunction, 'SignalName', signalname, 'SignalValue', signalvalue)
```

returns a vector of logical values specifying whether or not the root inport map has a signal associated with it. To create a comma-separated list of variables to map from this object, use the `getInputString` function.

To map signals to root-level inports using custom mapping modes, you can use `getSIRootInport` with the Root Inport Mapper dialog box custom mapping capability.

Examples

Create inport map using Simulink mapping mode

Create a vector of inport maps using a built-in mapping mode.

```
Throttle = timeseries(ones(10,1)*10);  
Brake    = timeseries(zeros(10,1));  
inputMap = getSIRootInportMap('model', 'slexAutotransRootInportsExample', ...  
    'MappingMode', 'BlockName', ...  
    'SignalName', {'Throttle' 'Brake'}, ...  
    'SignalValue', {Throttle Brake});
```

Create inport map using custom function

Create a vector of inport maps using a custom function

```
port1    = timeseries(ones(10,1)*10);  
port2    = timeseries(zeros(10,1));  
inputMap = getSIRootInportMap('model', 'slexAutotransRootInportsExample', ...  
    'MappingMode', 'Custom', ...  
    'CustomFunction', 'slexCustomMappingMyCustomMap', ...
```

```
'SignalName',{'port1' 'port2'},...
'SignalValue',{port1 port2});
```

Input Arguments

modelName — Model name

character vector

Specify the model to associate with the root inport map.

Data Types: char

mappingmode — Simulink mapping mode

character vector

Specify the mapping mode to use with model name and data source. Possible values are:

'Index'	Assign sequential index numbers, starting at 1, to the data in the MAT-file, and map this data to the corresponding inport.
'BlockName'	Assign data to ports according to the name of the root-inport block. If the block name of a data element matches the name of a root-inport block, map the data to the corresponding port.
'SignalName'	Assign data to ports according to the name of the signal on the port. If the signal name of a data element matches the name of a signal at a port, map the data to the corresponding port.
'BlockPath'	Assign data to ports according to the block path of the root-inport block. If the block path of a data element matches the block path of a root-inport block, map the data to the corresponding port.
'Custom'	Apply mappings according to the definitions in a custom file.

Data Types: char

customfunction — Custom function file name

character vector

Specify name of file that implements a custom method to map signals to root-level ports. This function must be on the MATLAB path.

Data Types: char

signalname — signal name

scalar | cell array of character vectors

Specify the signal name(s) of the signal to associate with the root inport map.

Data Types: char | cell

signalvalue — signal value

scalar | cell arrays

Specify the values of the signals to map to the root inport map. For the list of supported data types for the values, see “Choose a Base Workspace and MAT-File Format”.

Output Arguments

inputMap — input map

scalar | vector

Mapping object that defines the mapping of input signals to root-level ports. To create a comma-separated list of variables to map from this object, use the `getInputString` function.

hasASignal — signal presence indicator

scalar | vector

A vector of logical values with the same length as `inputMap`. If the value is true the `inputMap` has a signal associated with it. If the value is false the `inputMap` does not have a signal associated with it and will use a ground value as an input

Data Types: logical

Tips

- Load the model before running this function.
- If your custom mapping mode is not similar to an existing Simulink mapping mode, consider using the `getRootInportMap` function instead.

See Also

`getInputString` | `getRootInportMap`

Topics

“Map Root Inport Signal Data”

Introduced in R2013b

getVariable

Get value of variable from workspace

Syntax

```
variableValue = getVariable(workspaceHandle, variableName)
variableValue = workspaceHandle.getVariable(variableName)
```

Description

`variableValue = getVariable(workspaceHandle, variableName)` returns the value of the variable. If the variable does not exist in the workspace, an error occurs.

If the value of the target variable is a handle to a handle object (such as `Simulink.Parameter`), `getVariable` returns a copy of the handle. Changes you make to the variable in the model workspace or to the returned variable (`variableValue`) affect both variables.

To return a deep copy of a handle object, use the `copy` method of the object. To modify a handle object that you store in a model workspace, it is a best practice to use both the `getVariable` and `assignin` methods (see “Modify Property Value of Handle Object” on page 2-337).

`variableValue = workspaceHandle.getVariable(variableName)` is an alternative syntax.

Input Arguments

workspaceHandle

Handle to the workspace containing the variable.

variableName

Name of the variable containing the value.

Output Arguments

variableValue

Value of the variable.

Examples

Get Value of Workspace Variable

Get the value of the workspace variable `K`, which is defined in model `mdl`.

```
wksp = get_param(mdl, 'ModelWorkspace')  
value = getVariable(wksp, 'K')
```

```
value =  
    5
```

Modify Property Value of Handle Object

Modify a property of the `Simulink.Parameter` object `K`, which is defined in model `mdl`. When you call `getVariable`, use the `copy` method because `Simulink.Parameter` is a handle class.

```
wksp = get_param(mdl, 'ModelWorkspace');  
value = copy(getVariable(wksp, 'K'));  
value.DataType = 'single';  
assignin(wksp, 'K', value);
```

See Also

`get_param`

Introduced in R2012a

hasVariable

Determine if variable exists in workspace

Syntax

```
variableExists = hasVariable(workspaceHandle, variableName)  
variableExists = workspaceHandle.hasVariable(variableName)
```

Description

variableExists = hasVariable(*workspaceHandle*, *variableName*) returns 1 if the variable exists in the workspace, and 0 if not.

variableExists = *workspaceHandle*.hasVariable(*variableName*) is an alternative syntax.

Input Arguments

workspaceHandle

Handle to the workspace.

variableName

Name of the variable.

Output Arguments

variableExists

Boolean value that indicates whether the variable exists in the workspace (1 if true and 0 if false).

Examples

Determine Existence of Variable

Determine if the variable K exists in the workspace for model mdl.

```
wksp = get_param(mdl, 'ModelWorkspace')
exists = hasVariable(wksp, 'K')

exists =
    1
```

See Also

`get_param`

Introduced in R2012a

hdllib

Display blocks that are compatible with HDL code generation

Syntax

```
hdllib
hdllib('off')
hdllib('html')
hdllib('librarymodel')
```

Description

`hdllib` displays the blocks that are supported for HDL code generation, and for which you have a license, in the Library Browser. To build models that are compatible with the HDL Coder software, use blocks from this Library Browser view.

If you close and reopen the Library Browser in the same MATLAB session, the Library Browser continues to show only the blocks supported for HDL code generation. To show all blocks, regardless of HDL code generation compatibility, at the command prompt, enter `hdllib('off')`.

`hdllib('off')` displays all the blocks for which you have a license in the Library Browser, regardless of HDL code generation compatibility.

`hdllib('html')` creates a library of blocks that are compatible with HDL code generation. It generates two additional HTML reports: a categorized list of blocks (`hdlblklist.html`) and a table of blocks and their HDL code generation parameters (`hdlsupported.html`).

To run `hdllib('html')`, you must have an HDL Coder license.

`hdllib('librarymodel')` displays blocks that are compatible with HDL code generation in the Library Browser. To build models that are compatible with the HDL Coder software, use blocks from this library.

The default library name is `hdlSupported`. After you generate the library, you can save it to a folder of your choice.

To keep the library current, you must regenerate it each time that you install a new software release.

To run `hdlLib('librarymodel')`, you must have an HDL Coder license.

Examples

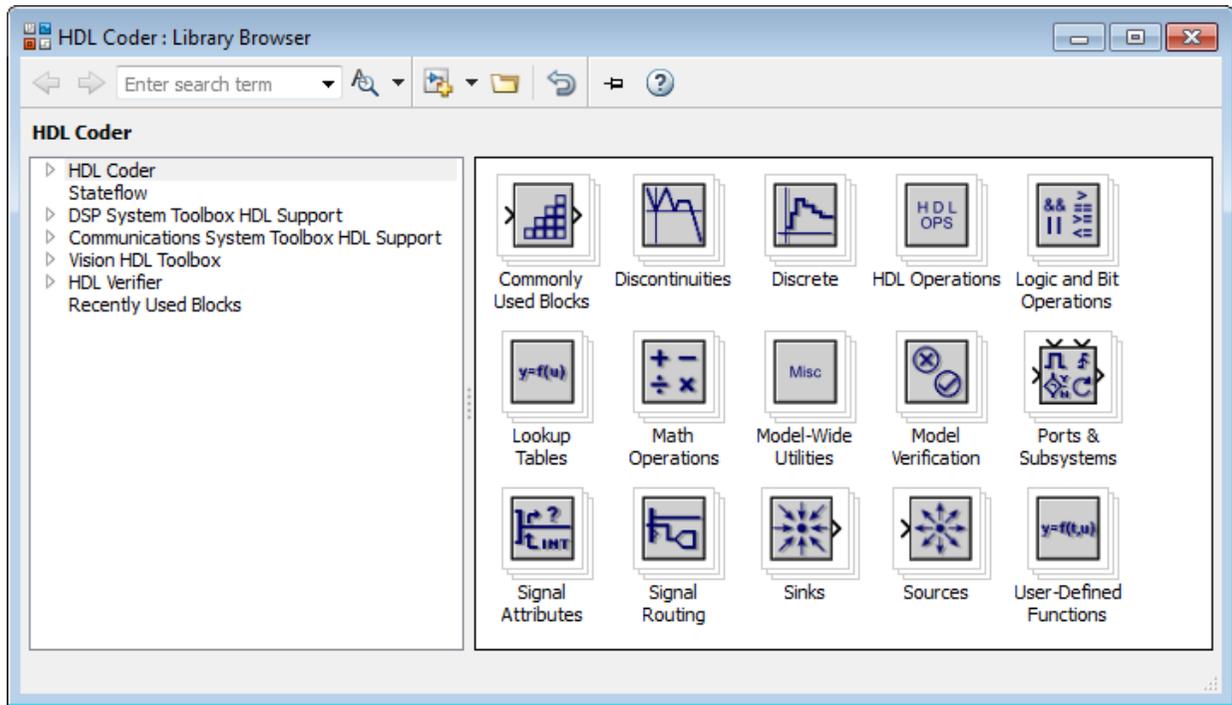
Display Supported Blocks in the Library Browser

To display blocks that are compatible with HDL code generation in the Library Browser:

```
hdlLib
```

```
### Generating view of HDL Coder compatible blocks in Library Browser.
```

```
### To restore the Library Browser to the default Simulink view, enter "hdlLib off".
```

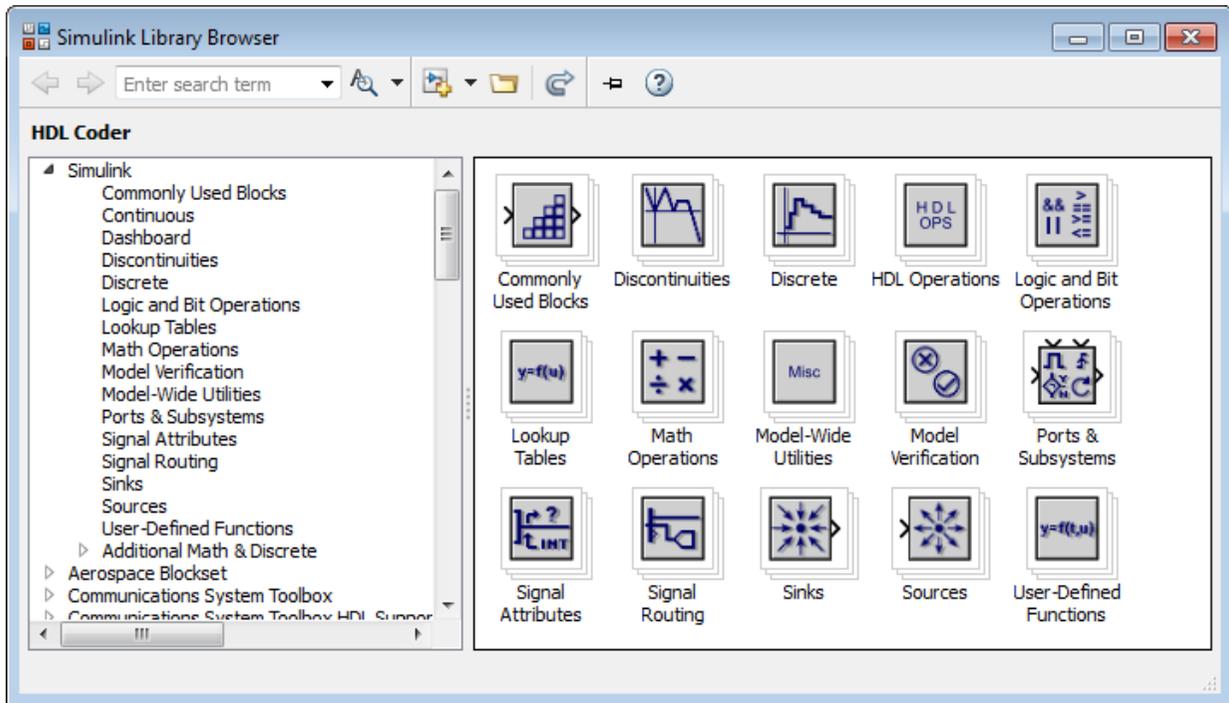


Display All Blocks in the Library Browser

To display all blocks in the Library Browser, regardless of HDL code generation compatibility:

```
hdlplib('off')
```

```
### Restoring Library Browser to default view; removing the HDL Coder compatibility fil
```



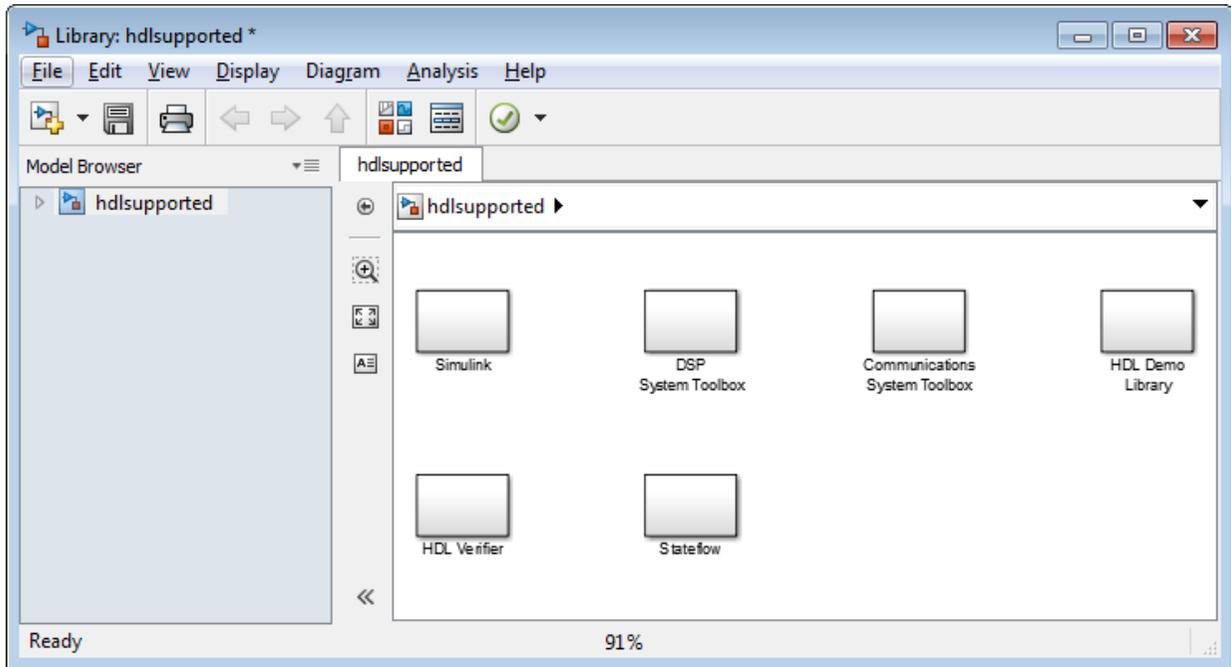
Create a Supported Blocks Library and HTML Reports

To create a library and HTML reports showing the blocks that are compatible with HDL code generation:

```
hdlLib('html')

### HDL supported block list hdlblklist.html
### HDL implementation list hdl-supported.html
```

The `hdl-supported` library opens. To view the reports, click the `hdlblklist.html` and `hdl-supported.html` links.

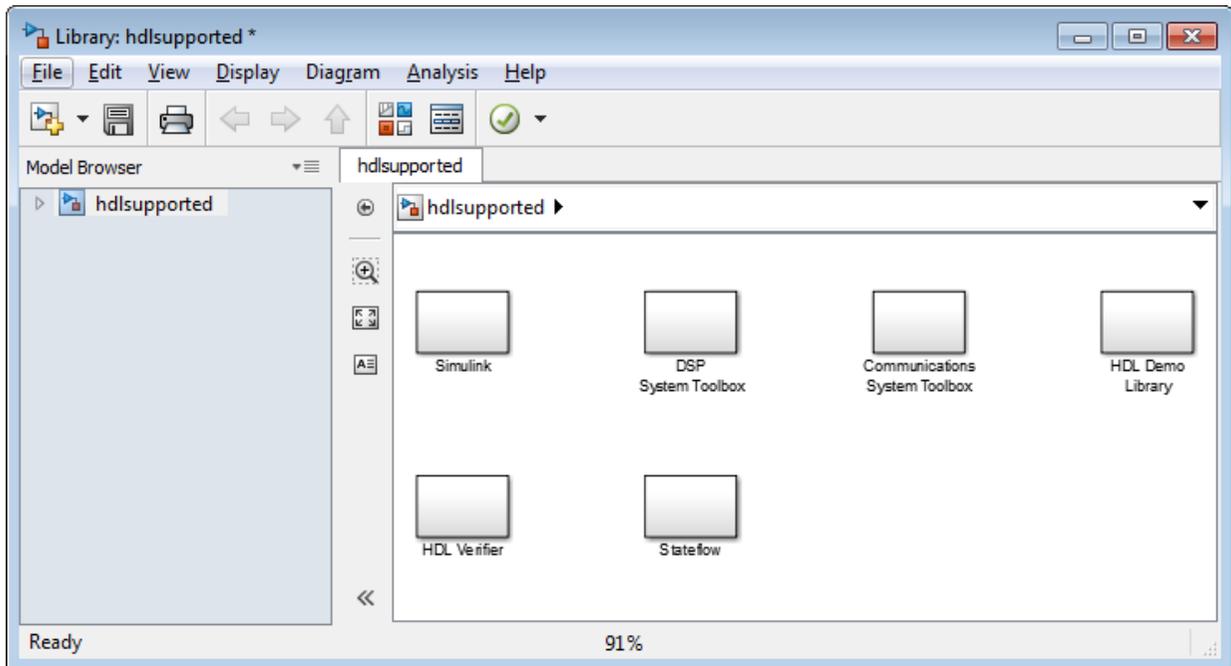


Create a Supported Blocks Library

To create a library that contains blocks that are compatible with HDL code generation:

```
hdlilib('librarymodel')
```

The `hdl_supported` block library opens.



- “Show Blocks Supported for HDL Code Generation” (HDL Coder)
- “View HDL-Specific Block Documentation” (HDL Coder)
- “Prepare Simulink Model For HDL Code Generation” (HDL Coder)

See Also

“Supported Blocks” (HDL Coder)

Topics

“Show Blocks Supported for HDL Code Generation” (HDL Coder)

“View HDL-Specific Block Documentation” (HDL Coder)

“Prepare Simulink Model For HDL Code Generation” (HDL Coder)

Introduced in R2006b

hilite_system

Highlight block, signal line, port, or annotation

Syntax

```
hilite_system(obj)  
hilite_system(obj, style)
```

Description

`hilite_system(obj)` highlights a block, line, port, or annotation in an open model using the default highlight style. Use `hilite_system` with a port to highlight the signal line attached to the port. Each use of `hilite_system` adds to the highlighting. Highlighting is not saved with the model.

`hilite_system(obj, style)` uses the specified highlighting style.

Input Arguments

obj — **Block, port, line, or annotation to highlight**

block path name | numeric handle | traceability tag

Block, port, line, or annotation to highlight, specified as:

- The full block path name
- A numeric handle for lines, ports, or annotations
- A traceability tag from the comments of Simulink Coder generated code.

Using a traceability tag requires a Simulink Coder license.

The format for a traceability tag is `<system>/block`, where `system` is either:

- Root

- A unique system number assigned by Simulink during code generation

Example: 'vdp/Mu', 'sldemo_fuelsys/fuel_rate_control/airflow_calc', '<Root>/Mu'

style — Highlighting style

'default' (default) | character vector

Highlighting style, specified as one of these values. You can customize the appearance of any of the styles. See “Customize a Highlighting Style” on page 2-349.

- 'default' — Default color scheme: red outline, yellow fill.
- 'none' — Clears the highlight.

To clear all highlighting, in the Simulink Editor, select **Display > Remove Highlighting**.

- 'debug' — Uses default color scheme.
- 'different' — Applies red outline, white fill.
- 'error' — Uses default color scheme.
- 'fade' — Applies gray outline, white fill.
- 'find' — Applies dark blue outline, blue fill.
- 'lineTrace' — Applies red outline, blue fill.
- 'unique' — Dark blue outline, white fill.
- 'user1', 'user2', 'user3', 'user4', 'user5' — Applies custom highlight: black outline, white fill by default (i.e., no highlight).

In addition, you can use these color schemes. The first word is the outline and the second is the fill color.

- 'orangeWhite'
- 'blackWhite'
- 'redWhite'
- 'blueWhite'
- 'greenWhite'

Examples

Highlight Block Using Default Highlight Style

Open the model `slexAircraftExample`.

```
slexAircraftExample
```

Highlight the Controller block. When you use the default highlight style, the block appears highlighted with a red outline and yellow fill.

```
hilite_system('slexAircraftExample/Controller')
```

Highlight a Block Using a Highlight Style

Open the model `vdp`.

```
vdp
```

Highlight the Mu block using the style 'fade'.

```
hilite_system('vdp/Mu', 'fade')
```

Use Block Highlighting to Trace Generated Code

If you have a Simulink Coder license, you can trace generated code to the corresponding source block in a model.

Open the model `f14`.

```
f14
```

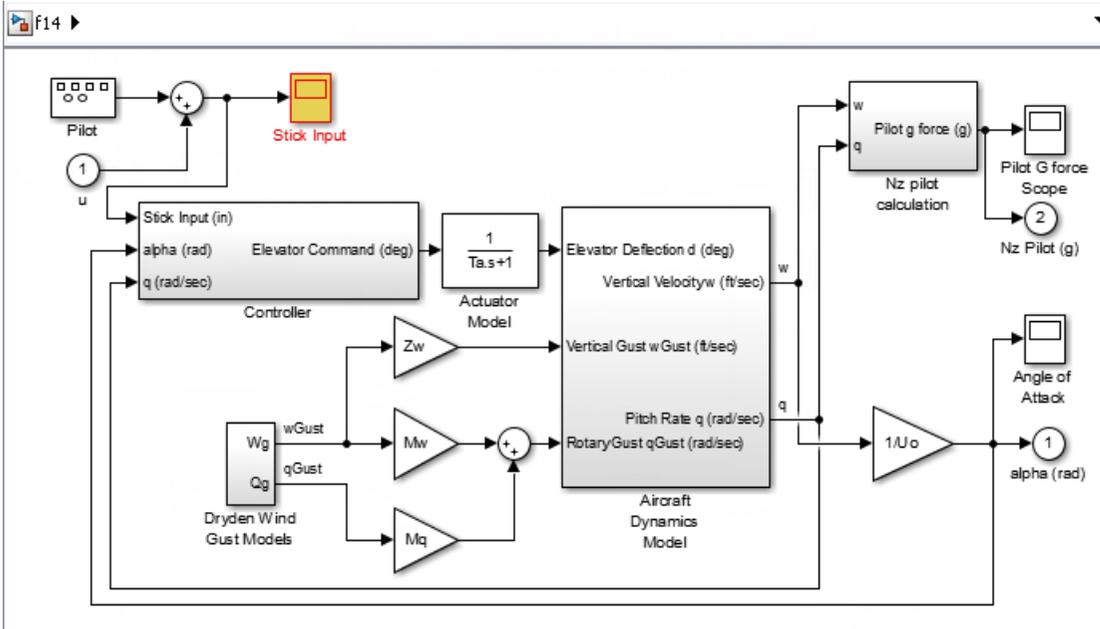
In the model configuration parameters, in the **Solver** pane, set **Type** to `Fixed-step`.

Generate code for the model using **Code > C/C++ Code > Build Model**.

In an editor or in the code generation report, open a generated source or header file. As you review lines of code, note traceability tags that correspond to code of interest.

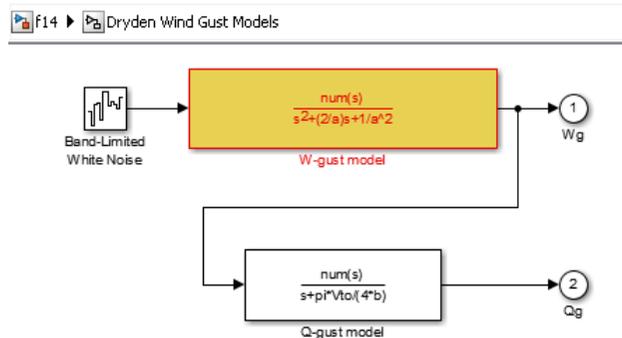
Highlight a block using a traceability tag.

```
hilite_system('<Root>/Stick Input')
```



Highlight a block in a subsystem.

```
hilite_system('<S3>/W-gust model')
```



Customize a Highlighting Style

You can customize a highlighting style by setting the `'HiliteAncestorsData'` parameter on the root-level model using `set_param` in this form:

```
set_param(0, 'HiliteAncestorsData', hiliteData)
```

Specify `hiliteData` as a structure array that has these fields:

- `'HiliteType'` — Highlighting style to customize, such as `'user1'`, `'debug'`, or `'error'`.
- `'ForegroundColor'` — Color for block fill.
- `'BackgroundColor'` — Color for block outline.

The supported values for `'ForegroundColor'` and `'BackgroundColor'` are:

- `'black'`
- `'white'`
- `'gray'`
- `'red'`
- `'orange'`
- `'yellow'`
- `'green'`
- `'darkGreen'`
- `'blue'`
- `'lightBlue'`
- `'cyan'`
- `'magenta'`

Define a highlight style for `'user1'`, and customize the style for `'debug'`.

```
set_param(0, 'HiliteAncestorsData', ...
          struct('HiliteType', 'user1', ...
                'ForegroundColor', 'darkGreen', ...
                'BackgroundColor', 'lightBlue'));
set_param(0, 'HiliteAncestorsData', ...
          struct('HiliteType', 'debug', ...
```

```
'ForegroundColor','red',...  
'BackgroundColor','black'));
```

Use the defined style to highlight a block.

```
f14  
hilite_system('f14/Controller/Alpha-sensor Low-pass Filter','user1')
```

See Also

`find_system` | `rtwtrace`

Introduced before R2006a

isLoaded

Determine if Simulink Project is loaded

Syntax

```
loaded = isLoaded(proj)
```

Description

`loaded = isLoaded(proj)` returns whether the project referenced by the project object `proj` is loaded.

Examples

Find Out if Project Is Loaded

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

Find out if the project is still loaded.

```
loaded = isLoaded(proj)
```

```
loaded =
```

```
1
```

Input Arguments

proj — Project

project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

Output Arguments

loaded — Loaded status

1 | 0

Project loaded status, returned as true (1) if the project is loaded.

Data Types: `logical`

See Also

Functions

`reload` | `simulinkproject`

Introduced in R2013a

legacy_code

Use Legacy Code Tool

Syntax

```
legacy_code('help')
specs = legacy_code('initialize')
legacy_code('sfcn_cmex_generate', specs)
legacy_code('compile', specs, compilerOptions)
legacy_code('generate_for_sim', specs, modelname)
legacy_code('slblock_generate', specs, modelname)
legacy_code('sfcn_tlc_generate', specs)
legacy_code('sfcn_makecfg_generate', specs)
legacy_code('rtwmakecfg_generate', specs)
legacy_code('backward_compatibility')
```

Description

The `legacy_code` function creates a MATLAB structure for registering the specification for existing C or C++ code and the S-function being generated. In addition, the function can generate, compile and link, and create a masked block for the specified S-function. Other options include generating

- A TLC file for simulation in Accelerator mode or code generation
- An `rtwmakecfg.m` file that you can customize to specify dependent source and header files that reside in a different directory than that of the generated S-function

`legacy_code('help')` displays instructions for using Legacy Code Tool.

`specs = legacy_code('initialize')` initializes the Legacy Code Tool data structure, `specs`, which registers characteristics of existing C or C++ code and properties of the S-function that the Legacy Code Tool generates.

`legacy_code('sfcn_cmex_generate', specs)` generates an S-function source file as specified by the Legacy Code Tool data structure, `specs`.

`legacy_code('compile', specs, compilerOptions)` compiles and links the S-function generated by the Legacy Code Tool based on the data structure, `specs`, and compiler options that you might specify. The compiler options must be supported by the `mex` function.

`legacy_code('generate_for_sim', specs, modelname)` generates, compiles, and links the S-function in a single step. If the `Options.useTlcWithAccel` field of the Legacy Code Tool data structure is set to logical 1 (`true`), the function also generates a TLC file for accelerated simulations.

`legacy_code('slblock_generate', specs, modelname)` generates a masked S-Function block for the S-function generated by the Legacy Code Tool based on the data structure, `specs`. The block appears in the Simulink model specified by `modelname`. If you omit `modelname`, the block appears in an empty model editor window.

`legacy_code('sfcn_tlc_generate', specs)` generates a TLC file for the S-function generated by the Legacy Code Tool based on the data structure, `specs`. This option is relevant if you want to:

- Force Accelerator mode in Simulink software to use the TLC inlining code of the generated S-function. See the description of the `ssSetOptions SimStruct` function and `SS_OPTION_USE_TLC_WITH_ACCELERATOR` S-function option for more information.
- Use Simulink Coder software to generate code from your Simulink model. For more information, see “Import Calls to External Code into Generated Code with Legacy Code Tool” (Simulink Coder).

`legacy_code('sfcn_makecfg_generate', specs)` generates an `sFunction_makecfg.m` file for the S-function generated by the Legacy Code Tool based on the data structure, `specs`. This option is relevant only if you use Simulink Coder software to generate code from your Simulink model. For more information, see “Use `makecfg` to Customize Generated Makefiles for S-Functions” (Simulink Coder) and “Import Calls to External Code into Generated Code with Legacy Code Tool” (Simulink Coder).

`legacy_code('rtwmakecfg_generate', specs)` generates an `rtwmakecfg.m` file for the S-function generated by the Legacy Code Tool based on the data structure, `specs`. This option is relevant only if you use Simulink Coder software to generate code from your Simulink model. For more information, see “Use `rtwmakecfg.m` API to Customize

Generated Makefiles” (Simulink Coder) and “Import Calls to External Code into Generated Code with Legacy Code Tool” (Simulink Coder).

`legacy_code('backward_compatibility')` automatically updates syntax for using Legacy Code Tool to the supported syntax described in this reference page and in “Integrate C Functions Using Legacy Code Tool”.

Input Arguments

specs

A structure with the following fields:

Name the S-function

`SFunctionName` (Required) — A character vector specifying a name for the S-function to be generated by the Legacy Code Tool.

Define Legacy Code Tool Function Specifications

- `InitializeConditionsFcnSpec` — A nonempty character vector specifying a reentrant function that the S-function calls to initialize and reset states. You must declare this function by using tokens that Simulink software can interpret as explained in “Declaring Legacy Code Tool Function Specifications”.
- `OutputFcnSpec` — A nonempty character vector specifying the function that the S-function calls at each time step. You must declare this function by using tokens that Simulink software can interpret as explained in “Declaring Legacy Code Tool Function Specifications”.
- `StartFcnSpec` — A character vector specifying the function that the S-function calls when it begins execution. This function can access S-function parameter arguments only. You must declare this function by using tokens that Simulink software can interpret as explained in “Declaring Legacy Code Tool Function Specifications”.
- `TerminateFcnSpec` — A character vector specifying the function that the S-function calls when it terminates execution. This function can access S-function parameter arguments only. You must declare this function by using tokens that Simulink software can interpret as explained in “Declaring Legacy Code Tool Function Specifications”.

Define Compilation Resources

- `HeaderFiles` — A cell array of character vectors specifying the file names of header files required for compilation.
- `SourceFiles` — A cell array of character vectors specifying source files required for compilation. You can specify the source files using absolute or relative path names.
- `HostLibFiles` — A cell array of character vectors specifying library files required for host compilation. You can specify the library files using absolute or relative path names.
- `TargetLibFiles` — A cell array of character vectors specifying library files required for target (that is, standalone) compilation. You can specify the library files using absolute or relative path names.
- `IncPaths` — A cell array of character vectors specifying directories containing header files. You can specify the directories using absolute or relative path names.
- `SrcPaths` — A cell array of character vectors specifying directories containing source files. You can specify the directories using absolute or relative path names.
- `LibPaths` — A cell array of character vectors specifying directories containing host and target library files. You can specify the directories using absolute or relative path names.

Specify a Sample Time

`SampleTime` — One of the following:

- `'inherited'` (default) — Sample time is inherited from the source block.
- `'parameterized'` — Sample time is represented as a tunable parameter. Generated code can access the parameter by calling MEX API functions, such as `mxGetPr` or `mxGetData`.
- `Fixed` — Sample time that you explicitly specify. For information on how to specify sample time, see “Specify Sample Time”.

If you specify this field, you must specify it last.

Define S-Function Options

`Options` — A structure that controls S-function options. The structure's fields include:

- `isMacro` — A logical value specifying whether the legacy code is a C macro. By default, the value is false (0).

- `isVolatile` — A logical value specifying the setting of the S-function `SS_OPTION_NONVOLATILE` option. By default, the value is `true` (1).
- `canBeCalledConditionally` — A logical value specifying the setting of the S-function `SS_OPTION_CAN_BE_CALLED_CONDITIONALLY` option. By default, the value is `true` (1).
- `useTlcWithAccel` — A logical value specifying the setting of the S-function `SS_OPTION_USE_TLC_WITH_ACCELERATOR` option. By default, the value is `true` (1).
- `language` — A character vector specifying either `'C'` or `'C++'` as the target language of the S-function that Legacy Code Tool will produce. By default, the value is `'C'`.

Note The Legacy Code Tool can interface with C++ functions, but not C++ objects. For a work around, see “Legacy Code Tool Limitations” in the Simulink documentation.

- `singleCPPMexFile` — A logical value that, if `true`, specifies that generated code:
 - Requires you to generate and manage an inlined S-function as only one file (`.cpp`) instead of two (`.c` and `.tlc`).
 - Maintains model code style (level of parentheses usage and preservation of operand order in expressions and condition expressions in if statements) as specified by model configuration parameters.

By default, the value is `false`.

Limitations You cannot set the `singleCPPMexFile` field to `true` if

- `Options.language='C++'`
- You use one of the following Simulink objects with the `IsAlias` property set to `true`:
 - `Simulink.Bus`
 - `Simulink.AliasType`
 - `Simulink.NumericType`

- The Legacy Code Tool function specification includes a `void*` or `void**` to represent scalar work data for a state argument
 - `HeaderFiles` field of the Legacy Code Tool structure specifies multiple header files
-
- `supportsMultipleExecInstances`— A logical value specifying whether to include a call to the `ssSupportsMultipleExecInstances` function. By default, the value is `false (0)`.
 - `convertNDMatrixToRowMajor`— A logical value specifying the automatic conversion of a matrix between a column-major format and a row-major format. The column-major format is used by MATLAB, Simulink, and the generated code. The row-major format is used by C. By default, the value is `false (0)`. If you currently specify the previous version of the option, `convert2DMatrixToRowMajor`, the function automatically specifies the new `convertNDMatrixToRowMajor` option.

Note This option does not support a 2-D matrix of complex data.

- `supportCoverage`— A logical value specifying whether the generated S-function must be compatible with Model Coverage. By default, the value is `false (0)`.
- `supportCoverageAndDesignVerifier`— A logical value specifying whether the generated S-function must be compatible with Model Coverage and Simulink Design Verifier™. By default, the value is `false (0)`.
- `outputsConditionallyWritten`— A logical value specifying whether the legacy code conditionally writes the output ports. If `true`, the generated S-function specifies that the memory associated with each output port cannot be overwritten and is global (`SS_NOT_REUSABLE_AND_GLOBAL`). If `false`, the memory associated with each output port is reusable and is local (`SS_REUSABLE_AND_LOCAL`). By default, the value is `false (0)`. For more information, see `ssSetOutputPortOptimOpts`.

modelName

The name of a Simulink model into which Legacy Code Tool is to insert the masked S-function block generated when you specify `legacy_code` with the action character vector `'slblock_generate'`. If you omit this argument, the block appears in an empty model editor window.

See Also

Topics

“Integrate C Functions Using Legacy Code Tool”

“Import Calls to External Code into Generated Code with Legacy Code Tool” (Simulink Coder)

Introduced in R2006b

libinfo

Get information about library blocks referenced by model

Syntax

```
libdata = libinfo('system')  
libdata = libinfo('system', constraint1, value1, ...)
```

Description

`libdata = libinfo('system')` returns information about library blocks referenced by `system` and all the systems underneath it.

`libdata = libinfo('system', constraint1, value1, ...)` restricts the search as indicated by the search constraint(s) `c1, v1, ...`

Input Arguments

system

The system to search recursively for library blocks.

constraint1, value1, ...

One or more pairs, each consisting of a search constraint followed by a constraint value. You can specify any of the search constraints that you can use with `find_system`.

Output Arguments

libdata

An array of structures that describes each library block referenced by `system`. Each structure has the following fields:

Block

Library

ReferenceBlock

LinkStatus

Path of the link to the library block

Name of the library containing the referenced block

Path of the library block

Value of the `LinkStatus` parameter for the link to the library block

See Also

`find_system`

Topics

“Custom Libraries and Linked Blocks”

Introduced before R2006a

linmod

Extract continuous-time linear state-space model around operating point

Syntax

```
argout = linmod('sys');
argout = linmod('sys', x, u);
argout = linmod('sys', x, u, para);
argout = linmod('sys', x, u, 'v5');
argout = linmod('sys', x, u, para, 'v5');
argout = linmod('sys', x, u, para, xpert, upert, 'v5');
```

Arguments

<i>sys</i>	Name of the Simulink system from which the linear model is extracted.
<i>x</i> and <i>u</i>	State (<i>x</i>) and the input (<i>u</i>) vectors. If specified, they set the operating point at which the linear model is extracted. When a model has model references using the Model block, you must use the Simulink structure format to specify <i>x</i> . To extract the <i>x</i> structure from the model, use the following command: <pre>x = Simulink.BlockDiagram.getInitialState('sys');</pre> <p>You can then change the operating point values within this structure by editing <code>x.signals.values</code>.</p> <p>If the state contains different data types (for example, 'double' and 'uint8'), then you cannot use a vector to specify this state. You must use a structure instead. In addition, you can only specify the state as a vector if the state data type is 'double'.</p>
<i>Ts</i>	Sample time of the discrete-time linearized model
'v5'	An optional argument that invokes the perturbation algorithm created prior to MATLAB 5.3. Invoking this optional argument is equivalent to calling <code>linmodv5</code> .

para

A three-element vector of optional arguments:

- para(1) — Perturbation value of delta, the value used to perform the perturbation of the states and the inputs of the model. This is valid for linearizations using the 'v5' flag. The default value is 1e-05.
- para(2) — Linearization time. For blocks that are functions of time, you can set this parameter with a nonnegative value that gives the time (τ) at which Simulink evaluates the blocks when linearizing a model. The default value is 0.
- para(3) — Set para(3)=1 to remove extra states associated with blocks that have no path from input to output. The default value is 0.

xpert and upert

The perturbation values used to perform the perturbation of all the states and inputs of the model. The default values are

```
xpert = para(1) + 1e-3*para(1)*abs(x)
upert = para(1) + 1e-3*para(1)*abs(u)
```

When a model has model references using the Model block, you must use the Simulink structure format to specify xpert. To extract the xpert structure, use the following command:

```
xpert = Simulink.BlockDiagram.getInitialState('sys');
```

You can then change the perturbation values within this structure by editing xpert.signals.values.

The perturbation input arguments are only available when invoking the perturbation algorithm created prior to MATLAB 5.3, either by calling linmodv5 or specifying the 'v5' input argument to linmod.

argout

`linmod`, `dlinmod`, and `linmod2` return state-space representations if you specify the output (left-hand) side of the equation as follows:

- `[A, B, C, D] = linmod('sys', x, u)` obtains the linearized model of `sys` around an operating point with the specified state variables `x` and the input `u`. If you omit `x` and `u`, the default values are zero.

`linmod` and `dlinmod` both also return a transfer function and MATLAB data structure representations of the linearized system, depending on how you specify the output (left-hand) side of the equation. Using `linmod` as an example:

- `[num, den] = linmod('sys', x, u)` returns the linearized model in transfer function form.
- `sys_struct = linmod('sys', x, u)` returns a structure that contains the linearized model, including state names, input and output names, and information about the operating point.

Description

`linmod` compute a linear state-space model by linearizing each block in a model individually.

`linmod` obtains linear models from systems of ordinary differential equations described as Simulink models. Inputs and outputs are denoted in Simulink block diagrams using Inport and Outport blocks.

The default algorithm uses preprogrammed analytic block Jacobians for most blocks which should result in more accurate linearization than numerical perturbation of block inputs and states. A list of blocks that have preprogrammed analytic Jacobians is available in the Simulink Control Design documentation along with a discussion of the block-by-block analytic algorithm for linearization.

The default algorithm also allows for special treatment of problematic blocks such as the Transport Delay and the Quantizer. See the mask dialog of these blocks for more information and options.

Notes

By default, the system time is set to zero. For systems that are dependent on time, you can set the variable `para` to a two-element vector, where the second element is used to set the value of t at which to obtain the linear model.

The ordering of the states from the nonlinear model to the linear model is maintained. For Simulink systems, a character vector variable that contains the block name associated with each state can be obtained using

```
[sizes,x0,xstring] = sys
```

where `xstring` is a vector of strings whose i th row is the block name associated with the i th state. Inputs and outputs are numbered sequentially on the diagram.

For single-input multi-output systems, you can convert to transfer function form using the routine `ss2tf` or to zero-pole form using `ss2zp`. You can also convert the linearized models to LTI objects using `ss`. This function produces an LTI object in state-space form that can be further converted to transfer function or zero-pole-gain form using `tf` or `zpk`.

The default algorithms in `linmod` handle Transport Delay blocks by replacing the linearization of the blocks with a Pade approximation. For the 'v5' algorithm, linearization of a model that contains Derivative or Transport Delay blocks can be troublesome. For more information, see “Linearizing Models”.

See Also

`dlinmod` | `linmod2` | `linmodv5`

Introduced in R2007a

linmod2

Extract continuous-time linear state-space model around operating point

Syntax

```
argout = linmod2('sys', x, u);  
argout = linmod2('sys', x, u, para);
```

Arguments

- | | |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>sys</i> | Name of the Simulink system from which the linear model is extracted. |
| <i>x, u</i> | State (<i>x</i>) and the input (<i>u</i>) vectors. If specified, they set the operating point at which the linear model is extracted. When a model has model references using the Model block, you must use the Simulink structure format to specify <i>x</i> . To extract the <i>x</i> structure from the model, use the following command:
<pre>x = Simulink.BlockDiagram.getInitialState('sys');</pre> <p>You can then change the operating point values within this structure by editing <code>x.signals.values</code>.</p> <p>If the state contains different data types (for example, 'double' and 'uint8'), then you cannot use a vector to specify this state. You must use a structure instead. In addition, you can only specify the state as a vector if the state data type is 'double'.</p> |

para

A three-element vector of optional arguments:

- `para(1)` — Perturbation value of delta, the value used to perform the perturbation of the states and the inputs of the model. This is valid for linearizations using the 'v5' flag. The default value is 1e-05.
- `para(2)` — Linearization time. For blocks that are functions of time, you can set this parameter with a nonnegative value that gives the time (τ) at which Simulink evaluates the blocks when linearizing a model. The default value is 0.
- `para(3)` — Set `para(3)=1` to remove extra states associated with blocks that have no path from input to output. The default value is 0.

argout

`linmod`, `dlinmod`, and `linmod2` return state-space representations if you specify the output (left-hand) side of the equation as follows:

- `[A, B, C, D] = linmod('sys', x, u)` obtains the linearized model of `sys` around an operating point with the specified state variables `x` and the input `u`. If you omit `x` and `u`, the default values are zero.

`linmod` and `dlinmod` both also return a transfer function and MATLAB data structure representations of the linearized system, depending on how you specify the output (left-hand) side of the equation. Using `linmod` as an example:

- `[num, den] = linmod('sys', x, u)` returns the linearized model in transfer function form.
- `sys_struct = linmod('sys', x, u)` returns a structure that contains the linearized model, including state names, input and output names, and information about the operating point.

Description

`linmod2` computes a linear state-space model by perturbing the model inputs and model states, and uses an advanced algorithm to reduce truncation error.

`linmod2` obtains linear models from systems of ordinary differential equations described as Simulink models. Inputs and outputs are denoted in Simulink block diagrams using Inport and Outport blocks.

Notes

By default, the system time is set to zero. For systems that are dependent on time, you can set the variable `para` to a two-element vector, where the second element is used to set the value of t at which to obtain the linear model.

The ordering of the states from the nonlinear model to the linear model is maintained. For Simulink systems, a character vector variable that contains the block name associated with each state can be obtained using

```
[sizes,x0,xstring] = sys
```

where `xstring` is a vector of strings whose i th row is the block name associated with the i th state. Inputs and outputs are numbered sequentially on the diagram.

For single-input multi-output systems, you can convert to transfer function form using the routine `ss2tf` or to zero-pole form using `ss2zp`. You can also convert the linearized models to LTI objects using `ss`. This function produces an LTI object in state-space form that can be further converted to transfer function or zero-pole-gain form using `tf` or `zpk`.

The default algorithms in `linmod` and `dlinmod` handle Transport Delay blocks by replacing the linearization of the blocks with a Pade approximation. For more information, see “Linearizing Models”.

See Also

`dlinmod` | `linmod` | `linmodv5`

Introduced in R2007a

linmodv5

Extract continuous-time linear state-space model around operating point

Syntax

```
argout = linmodv5('sys');  
argout = linmodv5('sys', x, u);  
argout = linmodv5('sys', x, u, para);  
argout = linmodv5('sys', x, u, para, xpert, upert);
```

Arguments

<i>sys</i>	Name of the Simulink system from which the linear model is extracted.
<i>x, u</i>	State (<i>x</i>) and the input (<i>u</i>) vectors. If specified, they set the operating point at which the linear model is extracted. When a model has model references using the Model block, you must use the Simulink structure format to specify <i>x</i> . To extract the <i>x</i> structure from the model, use the following command:

```
x = Simulink.BlockDiagram.getInitialState('sys');
```

You can then change the operating point values within this structure by editing `x.signals.values`.

If the state contains different data types (for example, 'double' and 'uint8'), then you cannot use a vector to specify this state. You must use a structure instead. In addition, you can only specify the state as a vector if the state data type is 'double'.

para

A three-element vector of optional arguments:

- *para*(1) — Perturbation value of delta, the value used to perform the perturbation of the states and the inputs of the model. This is valid for linearizations using the 'v5' flag. The default value is 1e-05.
- *para*(2) — Linearization time. For blocks that are functions of time, you can set this parameter with a nonnegative value that gives the time (τ) at which Simulink evaluates the blocks when linearizing a model. The default value is 0.
- *para*(3) — Set *para*(3)=1 to remove extra states associated with blocks that have no path from input to output. The default value is 0.

xpert, upert

The perturbation values used to perform the perturbation of all the states and inputs of the model. The default values are

```
xpert = para(1) + 1e-3*para(1)*abs(x)
upert = para(1) + 1e-3*para(1)*abs(u)
```

When a model has model references using the Model block, you must use the Simulink structure format to specify *xpert*. To extract the *xpert* structure, use the following command:

```
xpert = Simulink.BlockDiagram.getInitialState('sys');
```

You can then change the perturbation values within this structure by editing *xpert.signals.values*.

The perturbation input arguments are only available when invoking the perturbation algorithm created prior to MATLAB 5.3, either by calling `linmodv5` or specifying the 'v5' input argument to `linmod`.

argout

`linmod`, `dlinmod`, and `linmod2` return state-space representations if you specify the output (left-hand) side of the equation as follows:

- `[A, B, C, D] = linmod('sys', x, u)` obtains the linearized model of `sys` around an operating point with the specified state variables `x` and the input `u`. If you omit `x` and `u`, the default values are zero.

`linmod` and `dlinmod` both also return a transfer function and MATLAB data structure representations of the linearized system, depending on how you specify the output (left-hand) side of the equation. Using `linmod` as an example:

- `[num, den] = linmod('sys', x, u)` returns the linearized model in transfer function form.
- `sys_struct = linmod('sys', x, u)` returns a structure that contains the linearized model, including state names, input and output names, and information about the operating point.

Description

`linmodv5` computes a linear state space model using the full model perturbation algorithm created prior to MATLAB 5.3.

`linmodv5` obtains linear models from systems of ordinary differential equations described as Simulink models. Inputs and outputs are denoted in Simulink block diagrams using Inport and Outport blocks.

Notes

By default, the system time is set to zero. For systems that are dependent on time, you can set the variable `para` to a two-element vector, where the second element is used to set the value of t at which to obtain the linear model.

The ordering of the states from the nonlinear model to the linear model is maintained. For Simulink systems, a character vector variable that contains the block name associated with each state can be obtained using

```
[sizes,x0,xstring] = sys
```

where `xstring` is a vector of strings whose *i*th row is the block name associated with the *i*th state. Inputs and outputs are numbered sequentially on the diagram.

For single-input multi-output systems, you can convert to transfer function form using the routine `ss2tf` or to zero-pole form using `ss2zp`. You can also convert the linearized models to LTI objects using `ss`. This function produces an LTI object in state-space form that can be further converted to transfer function or zero-pole-gain form using `tf` or `zpk`.

The default algorithms in `linmod` and `dlinmod` handle Transport Delay blocks by replacing the linearization of the blocks with a Pade approximation. For the 'v5' algorithm, linearization of a model that contains Derivative or Transport Delay blocks can be troublesome. For more information, see “Linearizing Models”.

See Also

`dlinmod` | `linmod` | `linmod2`

Introduced in R2011b

load_system

Invisibly load Simulink model

Syntax

```
load_system('sys')
```

Description

`load_system('sys')` loads `sys`, where `sys` is the name of a Simulink model, into memory without making its model window visible.

You cannot use `load_system` to load MATLAB file models last saved in Simulink Version 1.3 (for example: `load_system mymodel.m`). If you have a MATLAB file model, you must upgrade to Simulink model file format as follows:

- 1 Execute the model as a function:

```
mymodel
```

- 2 Save the model as a Simulink model file:

```
save_system mymodel
```

Examples

The command

```
load_system('vdp')
```

loads the `vdp` sample model into memory.

See Also

`close_system` | `open_system`

Introduced before R2006a

loadIntoMemory

Load logged data into memory

Syntax

```
logs = loadIntoMemory(logs)
```

Description

`logs = loadIntoMemory(logs)` loads the data in `logs` into memory. Data is logged to a repository and brought into memory on an as-needed basis. When you want to work with all elements of a large set of logged data, use `loadIntoMemory` to bring all of the elements into memory at once. Loading all the data at once, rather than element by element, is much faster.

Examples

Load Logged Data into Memory

This example shows how to load a set of logged data into memory all at once, rather than element by element.

```
% Simulate model to generate logged data  
sim('sldemo_fuelsys')
```

The simulation logs all of the instrumented signals in the model to the `Simulink.SimulationData.Dataset` object `sldemo_fuelsys_output`. At the end of simulation, the signal data remains in the repository until used in the MATLAB workspace. When you work with small sets of data or only postprocess a subset, leaving signals in the repository improves performance. But when you have a large set of data and need to postprocess all of the signals, you should bring them all into memory at once.

```
% Load all logged signals into memory  
loadIntoMemory(sldemo_fuelsys_output);
```

All of the data in `sldemo_fuelsys_output` is now available for efficient postprocessing.

Input Arguments

logs — Data to load into memory

`'Simulink.SimulationData.Dataset'` | `'Simulink.SimulationOutput'`

Data to load into memory. The `loadIntoMemory` function can load `Simulink.SimulationData.Dataset` and `Simulink.SimulationOutput` data.

Example: `logargout`

Output Arguments

logs — Data

`'Simulink.SimulationData.Dataset'` | `'Simulink.SimulationOutput'`

Data loaded into memory.

See Also

`Simulink.SimulationData.Dataset` | `Simulink.SimulationOutput`

Introduced in R2017b

model

Execute particular phase of simulation of model

Syntax

```
[sys,x0,str,ts] = model([],[],[],'sizes');  
[sys,x0,str,ts] = model([],[],[],'compile');  
outputs = model(t,x,u,'outputs');  
derivs = model(t,x,u,'derivs');  
dstates = model(t,x,u,'update');  
model([],[],[],'term');
```

Description

The `model` command executes a specific phase of the simulation of a Simulink model whose name is `model`. The command's last argument (`flag`) specifies the phase of the simulation to be executed. See “Simulation Phases in Dynamic Systems” for a description of the steps that Simulink software uses to simulate a model.

This command ignores the effects of state transitions and conditional execution. Therefore, it is not suitable for models which have such logic. Use this command for models which can be represented as simple dynamic systems. Such systems should meet these requirements.

- All states in the model must be built-in non-bus data types. For a discussion on built-in data types, see “About Data Types in Simulink”.
- If you are using vector format to specify the state, this command can access only non-complex states of `double` data type.
- There is minimal amount of state logic (Stateflow, conditionally executed subsystems etc.)
- The models are not mixed-domain models. That is, most blocks in the model are built-in Simulink blocks and do not include user-written S-functions or blocks from other Sim* products.

For models which do not comply with these requirements, using this command can cause Simulink to produce results which can only be interpreted by further analyzing and simplifying the model.

Note The state variable x can be represented in structure as well as vector formats. The variable follows the limitations of the format in which it is specified.

This command is also not intended to be used to run a model step-by-step, for example, to debug a model. Use the Simulink debugger if you need to examine intermediate results to debug a model.

Arguments

<code>sys</code>	<p>Vector of model size data:</p> <ul style="list-style-type: none"> • <code>sys(1)</code> = number of continuous states • <code>sys(2)</code> = number of discrete states • <code>sys(3)</code> = number of outputs • <code>sys(4)</code> = number of inputs • <code>sys(5)</code> = reserved • <code>sys(6)</code> = direct-feedthrough flag (1 = yes, 0 = no) • <code>sys(7)</code> = number of sample times (= number of rows in <code>ts</code>)
<code>x0</code>	Vector containing the initial conditions of the system's states
<code>str</code>	Vector of names of the blocks associated with the model's states. The state names and initial conditions appear in the same order in <code>str</code> and <code>x0</code> , respectively.
<code>ts</code>	An m -by-2 matrix containing the sample time (period, offset) information
<code>outputs</code>	Outputs of the model at time step t .
<code>derivs</code>	Derivatives of the continuous states of the model at time t .

dstates	<p>States of the model at time t returned as either a structure or an array. Simulink returns a structure when the model has states and x is either empty (<code>[]</code>) or in structure format. Otherwise, Simulink returns an array.</p> <ul style="list-style-type: none"> • If the return type is a vector or array, Simulink returns real double discrete states only. • If the return type is a structure, Simulink returns a structure that contains both continuous and discrete states of built-in types only. Non-built-in types are omitted.
t	Time step, specified as real double in scalar format.
x	State vector, specified as real double in structure or vector format.
u	Inputs, specified as real double in vector format.
flag	<p>Specification of the simulation phase to be executed:</p> <ul style="list-style-type: none"> • 'sizes' executes the size computation phase of the simulation. This phase determines the sizes of the model's inputs, outputs, state vector, etc. • 'compile' executes the compilation phase of the simulation. The compilation phase propagates signal and sample time attributes. • 'update' computes the next values of the model's discrete states. • 'outputs' computes the outputs of the model's blocks at time t. • 'derivs' computes the derivatives of the model's continuous states at time step t. • 'term' causes Simulink software to terminate simulation of the model.

Examples

The following command executes the compilation phase of the `vdp` model that comes with Simulink software.

```
vdp([], [], [], 'compile')
```

The following command terminates the simulation initiated in the previous example.

```
vdp([], [], [], 'term')
```

Note Simulink does not let you close a model while it is compiling or simulating. For all phases except the `'sizes'` phase, before closing the model, you must invoke the model command with the `'term'` argument.

See Also

`sim`

Introduced in R2007a

modeladvisor

Open Model Advisor

Syntax

```
modeladvisor(model)
```

Description

`modeladvisor(model)` opens the Model Advisor for the model or subsystem specified by `model`. If the specified model or subsystem is not open, this command opens it.

Examples

Open Model Advisor for model

Open the Model Advisor for `vdp` example model:

```
modeladvisor('vdp')
```

Open Model Advisor for subsystem

Open the Model Advisor for the Aircraft Dynamics Model subsystem of the `f14` example model:

```
modeladvisor('f14/Aircraft Dynamics Model')
```

Open Model Advisor for currently selected model

Open the Model Advisor on the currently selected model:

```
modeladvisor (bdroot)
```

Open Model Advisor for currently selected subsystem

Open the Model Advisor on the currently selected subsystem:

```
modeladvisor (gcs)
```

Input Arguments

model — Model or subsystem name

character vector

Model or subsystem name or handle, specified as a character vector.

Data Types: `char`

See Also

“Run Model Checks”

Introduced before R2006a

new_system

Create Simulink model or library in memory

Syntax

```
h = new_system

h = new_system(name)
h = new_system(name, 'FromTemplate', template)
h = new_system(name, 'FromFile', file)
h = new_system( ___, 'ErrorIfShadowed')

h = new_system(name, 'Model')
h = new_system(name, 'Model', subsystem)
h = new_system(name, 'Library')
h = new_system( ___, 'ErrorIfShadowed')
```

Description

`h = new_system` creates a model named `untitled` (and then `untitled1`, `untitled2`, and so on) based on your default model template and returns the new model's numeric handle. Select your default model template on the Simulink start page or by using the `Simulink.defaultModelTemplate` function.

The `new_system` function does not open the new model. This function creates the model in memory. To save the model, use `save_system`, or open the model with `open_system` and then save it using the Simulink Editor.

`h = new_system(name)` creates a model based on your default model template and gives the new model the specified name. This function returns the new model's numeric handle. Select your default model template on the Simulink start page or by using the `Simulink.defaultModelTemplate` function.

If `name` is empty, the function creates a model named `untitled`, `untitled1`, `untitled2`, and so on.

The `new_system` function does not open the new model. This function creates the model in memory. To save the model, use `save_system`, or open the model with `open_system` and then save it using the Simulink Editor.

`h = new_system(name, 'FromTemplate', template)` creates the model based on the specified template.

`h = new_system(name, 'FromFile', file)` creates the model based on the specified model or template.

`h = new_system(____, 'ErrorIfShadowed')` creates the model or returns an error if another model, MATLAB file, or variable with the same name is on the MATLAB path or in the workspace. It uses any of the input arguments in the previous syntaxes.

`h = new_system(name, 'Model')` creates an empty model based on the Simulink default model and returns the new model's numeric handle. The Simulink default model is also known as the root block diagram and has the numeric handle 0. If `name` is empty, the function creates a model or library named `untitled`, `untitled1`, `untitled2`, and so on.

The `new_system` function does not open the new model. This function creates the model in memory. To save the model, use `save_system`, or open the model with `open_system` and then save it using the Simulink Editor.

`h = new_system(name, 'Model', subsystem)` creates a model based on the subsystem `subsystem` in a currently loaded model.

`h = new_system(name, 'Library')` creates an empty library that has the specified name and returns a numeric handle.

`h = new_system(____, 'ErrorIfShadowed')` returns an error if another model, MATLAB file, or variable with the same name is on the MATLAB path or in the workspace. This syntax uses any of the input arguments in the previous syntaxes.

Examples

Create a Model Based on Default Template

Create a model in memory called `untitled`.

```
h = new_system;
```

You can use `get_param` to get the name.

```
get_param(h, 'Name')
open_system(h)
open_system('untitled')
open_system(get_param(h, 'Name'))
```

```
ans =
```

```
untitled
```

Use the name, the handle, or `get_param` command as input to `open_system`. You can use any of these commands:

Create a Model Based on Named Template

Before you use this syntax, create a template. In the Simulink Editor, create the model you want to use as the template. Then select **File > Export Model to > Template**. For this example, name the template `mytemplate`.

By default, the template is on the MATLAB path, so if you change location, add the folder to the MATLAB path.

Create a model named `templateModel` based on the template `mytemplate`.

```
h = new_system('templateModel', 'FromTemplate', 'mytemplate');
```

```
Invoking template \\myuserdir\Documents\MATLAB\mytemplate.sltx
```

Create a Model Based on Another Model

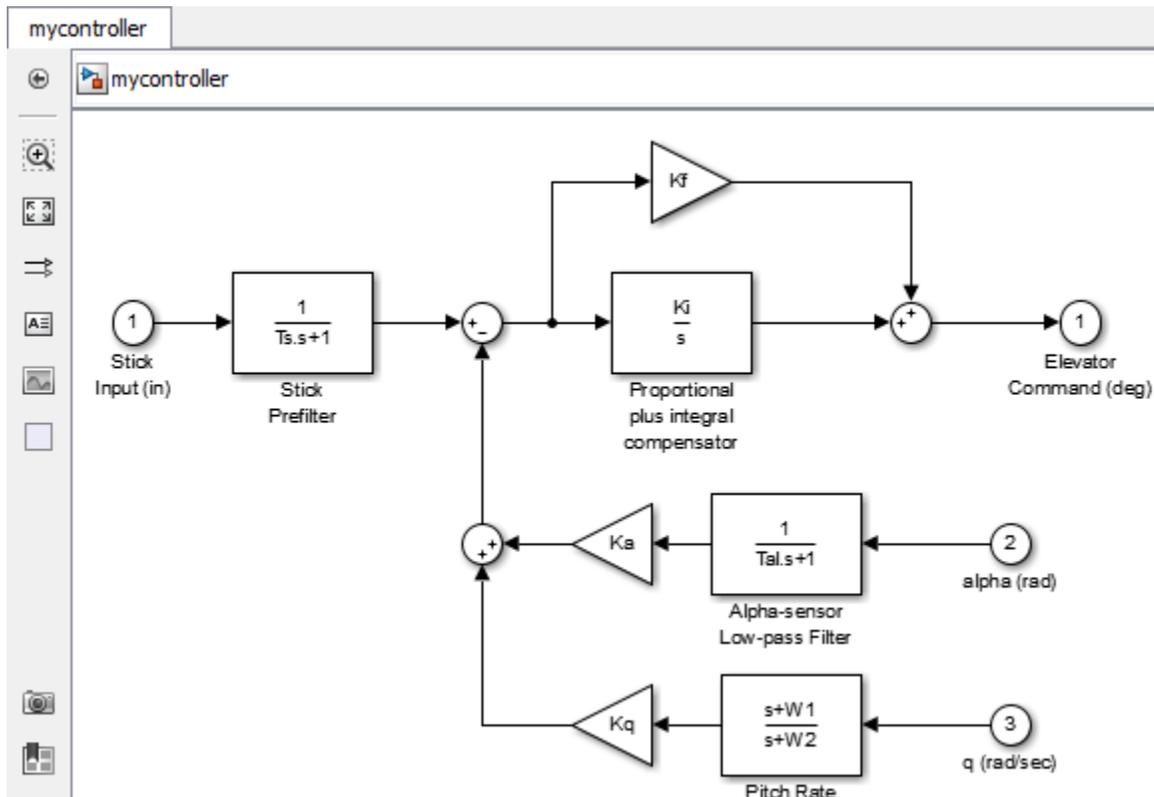
Create a model named `mynewmodel` based on `myoldmodel`, which is in the current folder.

```
h = new_system('mynewmodel', 'FromFile', 'myoldmodel.slx');
```

Create Model from Subsystem

Load the model f14. Create a model based on the Controller subsystem.

```
load_system('f14');
new_system('mycontroller', 'Model', 'f14/Controller');
open_system('mycontroller');
```



Create a Library

Create a library in memory and then open it.

```
new_system('mylib', 'Library')
open_system('mylib')
```

Ensure Model Name Is Unique

Create a variable with the name `myvar`.

```
myvar = 17
```

Try to create a model that uses the same name as the variable. When you use the `'ErrorIfShadowed'` option, the `new_system` function returns an error.

```
new_system('myvar2', 'Model', 'ErrorIfShadowed')
```

The model `'myvar2'` cannot be created because this name is shadowing another name on the path or in the workspace. Choose another name, or do not use the option `'ErrorIfShadowed'`.

Input Arguments

name — Name of new model or library

character vector

Name of new model or library, specified as a character vector that:

- Has 63 or fewer characters
- Is not a MATLAB keyword
- Is not `'simulink'`
- Is unique among model names, variables, and MATLAB files on the MATLAB path and in the workspace

Example: `'mymodel', 'mylibrary'`

subsys — Subsystem to base new model on

subsystem block path name

Subsystem to base the new model on, specified as the subsystem block path name in a currently open model.

Example: `'f14/Controller'`

template — Name of template to base new model on

character vector

Name of the template to base the new model on, specified as a character vector of the name of a template on the MATLAB path. Create a template in the Simulink Editor using **File > Export Model to > Template**.

Example: 'mytemplate', 'mytemplate.sltx'

file — Path name of model or template to base new model on

character vector

Path name of the model or template to base the new model on, specified as a character vector. You can use an .mdl, .slx, or .sltx file. Include the extension and use a full or relative path.

Example: 'Models/mymodel.slx', 'mytemplate.sltx', 'model.mdl'

See Also

Simulink.defaultModelTemplate | open_system | save_system

Introduced before R2006a

num2fixpt

Convert number to nearest value representable by specified fixed-point data type

Syntax

```
outValue = num2fixpt(OrigValue, FixPtDataType, FixPtScaling,  
                    RndMeth, DoSatur)
```

Description

`num2fixpt(OrigValue, FixPtDataType, FixPtScaling, RndMeth, DoSatur)` returns the result of converting `OrigValue` to the nearest value representable by the fixed-point data type `FixPtDataType`. Both `OrigValue` and `outValue` are of data type `double`. As illustrated in the example that follows, you can use `num2fixpt` to investigate quantization error that might result from converting a number to a fixed-point data type. The arguments of `num2fixpt` include:

<code>OrigValue</code>	Value to be converted to a fixed-point representation. Must be specified using a <code>double</code> data type.
<code>FixPtDataType</code>	The fixed-point data type used to convert <code>OrigValue</code> .
<code>FixPtScaling</code>	Scaling of the output in either <code>Slope</code> or <code>[Slope Bias]</code> format. If <code>FixPtDataType</code> does not specify a generalized fixed-point data type using the <code>sfix</code> or <code>ufix</code> command, <code>FixPtScaling</code> is ignored.
<code>RndMeth</code>	Rounding technique used if the fixed-point data type lacks the precision to represent <code>OrigValue</code> . If <code>FixPtDataType</code> specifies a floating-point data type using the <code>float</code> command, <code>RndMeth</code> is ignored. Valid values are <code>Zero</code> , <code>Nearest</code> , <code>Ceiling</code> , or <code>Floor</code> (the default).

`DoSatur` Indicates whether the output should be saturated to the minimum or maximum representable value upon underflow or overflow. If `FixPtDataType` specifies a floating-point data type using the `float` command, `DoSatur` is ignored. Valid values are `on` or `off` (the default).

Examples

Suppose you wish to investigate the quantization effect associated with representing the real-world value 9.875 as a signed, 8-bit fixed-point number. The command

```
num2fixpt(9.875, sfix(8), 2^-1)
```

```
ans =
```

```
9.500000000000000
```

reveals that a slope of 2^{-1} results in a quantization error of 0.375. The command

```
num2fixpt(9.875, sfix(8), 2^-2)
```

```
ans =
```

```
9.750000000000000
```

demonstrates that a slope of 2^{-2} reduces the quantization error to 0.125. But a slope of 2^{-3} , as used in the command

```
num2fixpt(9.875, sfix(8), 2^-3)
```

```
ans =
```

```
9.875000000000000
```

eliminates the quantization error entirely.

See Also

`fixptbestexp` | `fixptbestprec`

Introduced before R2006a

open_system

Open Simulink model, library, subsystem, or block dialog box

Syntax

```
open_system(obj)

open_system(sys, 'loadonly')

open_system(sbsys, 'window')
open_system(sbsys, 'tab')

open_system(blk, 'mask')
open_system(blk, 'force')
open_system(blk, 'parameter')
open_system(blk, 'OpenFcn')
```

Description

`open_system(obj)` opens the specified model, library, subsystem, or block. This is equivalent to double-clicking the model or library in the Current Folder Browser, or the subsystem or block in the Simulink Editor.

A model or library opens in a new window. For a subsystem or block within a model, the behavior depends on the type of block and its properties.

- Any `OpenFcn` callback parameter is evaluated.
- If there is no `OpenFcn` callback, and a mask is defined, the mask parameter dialog box opens.
- Without an `OpenFcn` callback or a mask parameter, Simulink opens the object.
 - A referenced model opens in a new window.
 - A subsystem opens in a new tab in the same window.
 - For blocks, the parameters dialog box for the block opens.

To open a specific subsystem or block, you must load the model or library containing it. Otherwise Simulink returns an error.

You can override the default behavior by supplying a second input argument.

`open_system(sys, 'loadonly')` loads the specified model or library without opening the Simulink Editor. This is equivalent to using `load_system`.

`open_system(sbsys, 'window')` opens the subsystem `sbsys` in a new Simulink Editor window. Before opening a specific subsystem or block, load the model or library containing it. Otherwise Simulink returns an error.

`open_system(sbsys, 'tab')` opens the subsystem in a new Simulink Editor tab in the same window. Before opening a specific subsystem or block, load the model or library containing it. Otherwise Simulink returns an error.

`open_system(blk, 'mask')` opens the mask dialog box of the block or subsystem specified by `blk`. Load the model or library containing `blk` before opening it.

`open_system(blk, 'force')` looks under the mask of a masked block or subsystem. It opens the dialog box of the block under the mask or opens a masked subsystems in a new Simulink Editor tab. This is equivalent to the **Look Under Mask** menu item. Before opening a specific subsystem or block, load the model or library containing it. Otherwise Simulink returns an error.

`open_system(blk, 'parameter')` opens the block parameter dialog box.

`open_system(blk, 'OpenFcn')` runs the block callback `OpenFcn`.

Examples

Open a Model

Open the `f14` model.

```
open_system('f14')
```

Load a Model Without Opening it

Load the f14 model.

```
open_system('f14','loadonly')
```

Open a Subsystem

Open the Controller subsystem of the f14 model.

```
load_system('f14')  
open_system('f14/Controller')
```

Open a Subsystem in New Tab in Existing Window

Open the f14 model and open the Controller subsystem in a new tab.

```
f14  
open_system('f14/Controller','tab')
```

Open a Subsystem in a Separate Window

Open a subsystem in its own Simulink Editor window.

```
open_system('f14')  
open_system('f14/Controller','window')
```

Open a Referenced Model

Open the model sldemo_mdhref_counter, which is referenced by the CounterA model block in sldemo_mdhref_basic.

```
open_system('sldemo_mdhref_basic')  
open_system('sldemo_mdhref_basic/CounterA')
```

The referenced model opens in its own Simulink Editor window.

Open Block Dialog Box

Open the block parameters dialog box for the first Gain block in the Controller subsystem.

```
load_system('f14')
open_system('f14/Controller/Gain')
```

Run Block Open Callback Function

Define an OpenFcn callback for a block and execute the block callback.

```
f14
set_param('f14/Pilot','OpenFcn','disp(''Hello World!'')')
open_system('f14/Pilot','OpenFcn')
```

The words `Hello World` appear on the MATLAB Command Prompt.

Open Masked Subsystem

Open the contents of the masked subsystem Vehicle in the model `sf_car`.

```
open_system('sf_car')
open_system('sf_car/Vehicle', 'force')
```

Open Multiple Systems with One Command

Create a cell array of two model names, `f14` and `vdp`. Open both models using `open_system` with the cell array name.

```
models = {'f14', 'vdp'}  
open_system(models)
```

Input Arguments

obj — Model, referenced model, library, subsystem, or block path

character vector

Model, referenced model, library, subsystem, or block path, specified as a character vector. If the model is not on the MATLAB path, specify the full path to the model file. Specify the block or subsystem using its full name, e.g., f14/Controller/Gain, on an opened or loaded model. On UNIX systems, the fully qualified path name of a model can start with a tilde (~), signifying your home directory.

Data Types: char

sys — Model or library path

character vector

The full name or path of a model or library, specified as a character vector.

Data Types: char

sbsys — Subsystem path

character vector

The full name or path of a subsystem in an open or loaded model, specified as a character vector.

Data Types: char

blk — Block or subsystem path

character vector

The full name or path of a block or subsystem in an open or loaded model, specified as a character vector.

Data Types: char

See Also

close_system | load_system | new_system | save_system

Introduced before R2006a

openDialog

Open configuration parameters dialog

Syntax

```
openDialog(configObj)
```

Arguments

configObj

A configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

Description

`openDialog` opens a configuration parameters dialog box. If *configObj* is a configuration set, the dialog box displays the configuration set. If *configObj* is a configuration reference, the dialog box displays the referenced configuration set, or generates an error if the reference does not specify a valid configuration set. If the dialog box is already open, its window becomes selected.

Examples

The following example opens a configuration parameters dialog box that shows the current parameters for the current model. The parameter values derive from the active configuration set or configuration reference (configuration object). The code is the same in either case; the only difference is which type of configuration object is currently active.

```
myConfigObj = getActiveConfigSet(gcs);  
openDialog(myConfigObj);
```

See Also

`attachConfigSet` | `attachConfigSetCopy` | `closeDialog` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `setActiveConfigSet`

Topics

“Manage a Configuration Set”

“Manage a Configuration Reference”

Introduced in R2006b

parsim

Simulate dynamic system in parallel or serial

Syntax

```
simOut = parsim(in)
simOut = parsim(in, 'ShowSimulationManager', 'on')
simOut = parsim(in, Name, Value, ...NameN, ValueN)
job = batch(@parsim, 1, {in}, 'Profile', 'ProfileName', 'Pool',
PoolSize)
```

Description

`simOut = parsim(in)` simulates a model using the inputs specified in the `SimulationInput` object, `in`. The `parsim` command uses an array of `SimulationInput` objects to run multiple simulations.

`simOut = parsim(in, 'ShowSimulationManager', 'on')` simulates a model in parallel using the inputs specified in the `SimulationInput` object and opens the Simulation Manager UI. For more information, see `Simulation Manager`.

`simOut = parsim(in, Name, Value, ...NameN, ValueN)` simulates a model in parallel using the inputs specified in the `SimulationInput` object and the options specified as the `Name, Value` pair.

`job = batch(@parsim, 1, {in}, 'Profile', 'ProfileName', 'Pool', PoolSize)` uses `parsim` with the `batch` command to run batch simulations.

The `parsim` command uses the Parallel Computing Toolbox™ license to run the simulations in parallel. `parsim` runs the simulations in serial if a parallel pool cannot be created and in the absence of Parallel Computing Toolbox .

Examples

Simulate Model in Parallel with `parsim`

Simulate the model, `cstr`, in parallel by sweeping over a variable. An array of `SimulationInput` objects is used to perform the sweep.

Specify sweep values.

```
FeedTempSweep = 250:10:300;
```

Create an array of `SimulationInput` objects.

```
for i = length(FeedTempSweep):-1:1;
in(i) = Simulink.SimulationInput('cstr');
in(i) = in(i).setVariable('FeedTemp0', FeedTempSweep(i));
end
```

Simulate the model in parallel.

```
out = parsim(in, 'ShowProgress', 'on')
```

Input Arguments

`in` — `Simulink.SimulationInput` object used to simulate the model
object, array

A `Simulink.SimulationInput` object or an array of `Simulink.SimulationInput` objects that is used to specify changes to the model for a simulation.

Example: `in = Simulink.SimulationInput('vdp')`

Name-Value Pair Arguments

Note All parameters passed to `parsim` command are unrelated to the parameters that are used with the `sim` command. Use the list of following input arguments to pass to the `parsim` command.

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` and `Value` must appear

inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'ShowProgress', 'on'`

ShowProgress — Shows the progress of the simulations

`'on'(default) | 'off'`

When `ShowProgress` is set to `'on'`, you can see the progress of the simulations in the command window. The progress is hidden when set to `'off'`.

Note When the progress is shown, a message 'Cleaning up parallel workers..' may be displayed before the completion of the last few simulations. This message does not depend on the completion of the simulations. Simulations "complete" when the outputs are fetched from the future.

SetupFcn — Function handle to run once per worker

function handle

You can specify a function handle to `'SetupFcn'` to run once per worker before the start of the simulations.

Example: `'SetupFcn', @() simulinkproject('OCRAex/OCRA_example.prj')`

Note When `buildRapidAcceleratorTarget` is used in the `SetupFcn` and the model has external inputs specified, either set `'LoadExternalInput'` to `'off'` or ensure that the specified external input is available on the workers to prevent compilation error.

CleanupFcn — Function handle to run once per worker after running simulations

function handle

You can specify a function handle to `'CleanupFcn'` to run once per worker after the simulations are completed.

AttachedFiles — Files to attach to pool

character vector or cell array of character vectors

Use `AttachedFiles` to attach files to pool, specified as a character vector or cell array of character vectors.

Example: `{ 'file1.m', 'file2.m' }`;

UseFastRestart — Uses fast restart

false (default) | true

When `UseFastRestart` is set to `true`, simulations run on the workers using fast restart.

Note When using `parsim`, use the `UseFastRestart` option and NOT the `FastRestart` option. See “Factors Affecting Fast Restart” for more information.

TransferBaseWorkspaceVariables — Transfers variables to the parallel workers

false (default) | true

When `TransferBaseWorkspaceVariables` is set to `true`, variables used in the model and defined in the base workspace are transferred to the parallel workers.

Note Use of `TransferBaseWorkspaceVariables` requires model compilation.

ShowSimulationManager — Starts the Simulation Manager App

'off' (default) | 'on'

When `ShowSimulationManager` is set to `'on'`, you can use the Simulation Manager App to monitor simulations.

Pool — Size of the parallel pool

numeric value

Pool determines the size of the parallel pool

Output Arguments

simOut — Simulation object containing logged simulation results

object

`Simulink.SimulationOutput` is an array of `Simulink.SimulationOutput` objects that contains all of the logged simulation results. The size of the array is equal to the size of the array of `Simulink.SimulationInput` objects.

All simulation outputs (logged time, states, and signals) are returned in a single `Simulink.SimulationOutput` object. You define the model time, states, and output that is logged using the **Data Import/Export** pane of the **Model Configuration Parameters** dialog box. You can log signals using blocks such as the To Workspace and Scope blocks. The **Signal & Scope Manager** can directly log signals.

See Also

`ExternalInput` | `PostSimFcn` | `PreSimFcn` | `Simulation Manager` | `Simulink.SimulationInput` | `applyToModel` | `setBlockParameter` | `setInitialState` | `setModelParameter` | `setVariable` | `validate`

Topics

“Run Multiple Simulations”

“Parallel Simulations Workflow”

Introduced in R2017a

performanceadvisor

Open Performance Advisor

Syntax

```
performanceadvisor(model)
```

Description

`performanceadvisor(model)` opens the Performance Advisor on the model or subsystem specified by `model`. If the specified model or subsystem is not open, this command opens it.

Input Arguments

model

A character vector specifying the name or handle to the model or subsystem.

Examples

Open Performance Advisor

Open Performance Advisor on the `vdp` example model.

```
performanceadvisor('vdp')
```

Performance Advisor opens the `vdp` model and opens Performance Advisor on the model.

- “Improve Simulation Performance Using Performance Advisor”
- “Perform a Quick Scan Diagnosis”

- “Improve vdp Model Performance”

Alternatives

In the Simulink Editor, select **Analysis > Performance Tools > Performance Advisor**.

See Also

Topics

“Improve Simulation Performance Using Performance Advisor”
“Perform a Quick Scan Diagnosis”
“Improve vdp Model Performance”
“Performance Advisor Window”

Introduced in R2013a

reload

Reload Simulink Project

Syntax

```
reload(proj)
```

Description

`reload(proj)` reloads the project. Use `reload` when you want to run the project startup shortcuts.

Examples

Reload Project

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

When you want to run the startup shortcuts again, reload the project.

```
reload(proj)
```

Input Arguments

proj — Project

project object

Project, specified as a project object already created with `simulinkproject` to manipulate a Simulink Project at the command line.

See Also

Functions

`isLoading` | `simulinkproject`

Introduced in R2013a

removeCategory

Remove Simulink Project category of labels

Syntax

```
removeCategory(proj, categoryName)
```

Description

`removeCategory(proj, categoryName)` removes a category of labels, `categoryName`, from the Simulink Project specified by `proj`.

Examples

Remove Category

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

Create a new category of labels.

```
createCategory(proj, 'Engineers', 'char');
```

Remove the new category of labels.

```
removeCategory(proj, 'Engineers');
```

A message appears warning you that you cannot undo the operation. Click **Continue**. You can configure warnings in the Preferences in the Simulink Project Tool.

Input Arguments

proj — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

categoryName — Name of category

character vector

Name of the category to remove, which exists in the project, specified as a character vector.

See Also

Functions

`createCategory` | `findCategory` | `simulinkproject`

Introduced in R2013a

removeFile

Remove file from Simulink Project

Syntax

```
removeFile(proj, file)
```

Description

`removeFile(proj, file)` removes a file from the project `proj`.

Examples

Remove File from Project

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

Remove a file.

```
removeFile(proj, 'models/AnalogControl.mdl')
```

Add the file back to the project.

```
addFile(proj, 'models/AnalogControl.mdl')
```

Input Arguments

proj — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

file — Path of file

character vector | file object

Path of the file to remove relative to the project root folder, including the file extension, specified as a character vector or a file object returned by `findFile`. The file must be in the project.

Example: `'models/myModelName.slx'`

See Also

Functions

`addFile` | `findFile` | `simulinkproject`

Introduced in R2013a

removeLabel

Remove label from Simulink Project

Syntax

```
removeLabel(category, labelName)  
removeLabel(file, categoryName, labelName)  
removeLabel(file, labelDefinition)
```

Description

`removeLabel(category, labelName)` removes the label from the specified category of labels in the currently loaded project.

`removeLabel(file, categoryName, labelName)` removes the specified label in the category `categoryName` from the file. Use this syntax to specify category and label by name.

`removeLabel(file, labelDefinition)` removes the specified label `labelDefinition` from the file. Before you can remove the label, you need to get the label from the file.Label property or by using `findLabel`.

Examples

Remove a Label

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

Examine the first existing category.

```
cat = proj.Categories(1)
```

```
cat =  
  
    Category with properties:  
  
        Name: 'Classification'  
        DataType: 'none'  
        LabelDefinitions: [1x8 slproject.LabelDefinition]
```

Define a new label in the category.

```
createLabel(cat, 'Future');
```

Remove the new label.

```
removeLabel(cat, 'Future');
```

Input Arguments

category — Category of labels

category object

Category of labels, specified as a category object. Get a category object from the `proj.Categories` property or by using `findCategory`.

labelName — Name of label

character vector

Name of the label to remove, specified as a character vector.

file — File to detach label from

file object

File to detach the label from, specified as a file object. You can get the file object by examining the project's `Files` property (`proj.Files`), or use `findFile` to find a file by name. The file must be within the root folder.

categoryName — Name of category that contains label

character vector

Name of the category that contains the label to remove, specified as a character vector.

labelDefinition — Label to detach

label definition object

Name of the label to detach, specified as a label definition object returned by the `file.Label` property or `findLabel`.

See Also

Functions

`addLabel` | `createLabel` | `findCategory` | `findLabel` | `simulinkproject`

Introduced in R2013a

replace_block

Replace blocks in Simulink model

Syntax

```
replBlks = replace_block(sys,current,new)
replBlks = replace_block(sys,Name,Value,new)
replBlks = replace_block( ____, 'noprompt')
```

Description

`replBlks = replace_block(sys,current,new)` replaces the blocks `current` in the model `sys` with blocks of type `new`. You can use a block from a Simulink library or from another model as the replacement block.

Load the model `sys` before using this function. The function prompts you to select the blocks you want to replace from a list of blocks that match the `current` argument.

Tip Save the model before using this command.

`replBlks = replace_block(sys,Name,Value,new)` replaces the blocks that match the block parameters specified by the `Name,Value` pair arguments. You can also use `find_system Name,Value` pairs to qualify the search for blocks to replace.

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

With the `replace_block` function, you can use block parameter and value pairs. For a list of all the block parameters, see “Common Block Properties” on page 6-109 and “Block-Specific Parameters” on page 6-128.

To specify additional information about the search for blocks to replace, you use `find_system` `Name, Value` pairs before the block parameters. For example, you can use `'CaseSensitive', 'off'` to make the search for blocks case insensitive or `'FollowLinks', 'on'` to follow links into library links. See `find_system` for that list of `Name, Value` pairs.

`replBlks = replace_block(____, 'noprompt')` replaces the blocks without prompting you to select them from a dialog box.

Examples

Replace Blocks in a Model

Replace blocks in the 'vdp' model.

Load the model 'vdp'.

```
load_system('vdp');
```

Replace Gain blocks with Integrator blocks.

```
RepNames = replace_block('vdp', 'Gain', 'Integrator');
```

A dialog box prompts you to select the blocks you want to replace.

With `vdp/Mu` selected in the dialog box, click **OK**.

Replace Scope blocks with To Workspace blocks.

```
RepNames = replace_block('vdp', 'Scope', 'simulink/Sinks/To Workspace');
```

A dialog box prompts you to select the blocks you want to replace.

With `vdp/Scope` selected in the dialog box, click **OK**.

Replace Blocks in a Subsystem Using Parameter Values

Replace blocks in the `Unlocked` subsystem of the `sldemo_clutch` model. Replace blocks whose `Gain` parameter is set to `bv`.

Load the model `sldemo_clutch`.

```
load_system('sldemo_clutch');
```

In the 'Unlocked' subsystem, replace blocks whose Gain value is `bv` with Integrator blocks.

```
replace_block('sldemo_clutch/Unlocked','Gain','bv','Integrator');
```

A dialog box prompts you to select the blocks to replace.

With `sldemo_clutch/Unlocked/VehicleDamping` selected in the dialog box, click **OK**.

Replace Blocks Without Dialog Box

Load the model `f14`.

```
load_system('f14')
```

Replace Gain blocks with Integrator blocks. The command returns the blocks it found to replace and replaces the blocks.

```
repl = replace_block('f14','Gain','Integrator','noprompt')
```

```
repl =
```

```
13x1 cell array

    {'f14/Aircraft...' }
    {'f14/Aircraft...' }
    {'f14/Aircraft...' }
    {'f14/Aircraft...' }
    {'f14/Controller/Gain' }
    {'f14/Controller/Gain2' }
    {'f14/Controller/Gain3' }
    {'f14/Gain' }
    {'f14/Gain1' }
    {'f14/Gain2' }
    {'f14/Gain5' }
    {'f14/Nz_pilot...' }
```

```
{'f14/Nz pilot...' }
```

Use `find_system` Pairs with `replace_block`

Select a block that is a library link. Follow the library links and replace Gain blocks with Integrator blocks within them.

```
replace_block(gcb, 'FollowLinks', 'on', 'BlockType', 'Gain', 'Integrator', 'noprompt')
```

Input Arguments

sys — Model or subsystem whose blocks to replace

character vector

Name of model whose blocks to replace, specified as a character vector. If you specify a model, the command replaces all blocks that match in the model. If you specify a subsystem, the command replaces blocks in that subsystem and below.

Example: `'vdp', 'sldemo_fuelsys/fuel_rate_control'`

current — Type of block to replace

BlockType value | MaskType value

Type of block to replace, specified as a BlockType or MaskType value. To find out the block type, select the block and, at the command prompt, enter:

```
get_param(gcb, 'BlockType')
```

For masked blocks, to find out the mask type, select the block and enter:

```
get_param(gcb, 'MaskType')
```

new — Block to replace current blocks

BlockType value | MaskType value | library path | block path name from a model

Block to replace the current block, specified in one of these forms:

- BlockType value of the replacement block. Specifying this value uses a library block as the replacement block.

- `MaskType` value of the replacement block. Specifying this value uses a library block as the replacement block.
- Library path of the replacement block, for example, 'simulink/Sinks/ToWorkspace'. Hover over the block in the library to see the library path.
- Block path name of a block from a different model, for example, 'vdp/Mu'. Use this value to reuse an instance of a block from another model in your model.

Output Arguments

rep1Blks — Blocks returned by the current argument

cell array of character vectors

Blocks returned by the `current` argument, returned as a cell array of character vectors. The function returns the values regardless of whether you complete the replacement.

See Also

`find_system` | `get_param`

Topics

“Common Block Properties” on page 6-109

“Block-Specific Parameters” on page 6-128

Introduced before R2006a

save_system

Save Simulink system

Syntax

```
save_system
save_system(sys)
save_system(sys, newsysname)
save_system(sys, newsysname.slx)
save_system(sys, newsysname, Name,Value)
save_system(sys, 'exported_file_name.xml', 'ExportToXML', true)
filename = save_system(sys)
```

Description

`save_system` saves the current top-level system. If the system has not previously been saved, `save_system` creates a new file in the current folder.

`save_system(sys)` saves the top-level system that you specify in `sys` to a file using the current system name. `sys` must be a system name with no file extension. The system must be loaded. `sys` can be a character vector, a cell array of character vectors, a numeric handle, or an array of numeric handles. If you specify any options they apply to all the systems that you save.

`save_system(sys, newsysname)` saves the top-level system that you specify to a file using the new system name `newsysname`. The system must be loaded. `newsysname` can be a system name, or a filename with file extension and optional path, or empty. If you do not specify a file extension (`.slx` or `.mdl`) then `save_system` uses the file format specified in your Simulink preferences.

`save_system(sys, newsysname.slx)` saves the top-level system `sys` to a new file `newsysname` in the SLX file format.

`save_system(sys, newsysname, Name,Value)` saves the system with additional options specified by one or more `Name,Value` pair arguments.

`save_system(sys, 'exported_file_name.xml', 'ExportToXML', true)` exports the system to a file in a simple XML format. Do not use `ExportToXML` with any other `save_system` options. The `ExportToXML` option warns and will be removed in a future release.

`filename = save_system(sys)` returns the fully-qualified file name of the file you saved.

`save_system` can save only entire systems. To save a subsystem, use the `Simulink.SubSystem.copyContentsToBlockDiagram` function to copy the subsystem contents to a new block diagram and then save it using `save_system`. See `Simulink.SubSystem.copyContentsToBlockDiagram`.

If you set the `UpdateHistory` property of the model to `UpdateHistoryWhenSave`, you see the following behavior:

- When you save interactively, you see a dialog prompting for a comment to include in the model history.
- When you save using `save_system`, you do not see a prompt for a comment. `save_system` reuses the previous comment, unless you set `'ModifiedComment'` before saving:

```
set_param(mymodel, 'ModifiedComment', mycomment)
```

Input Arguments

sys

Top-level system to save. `'sys'` must be a system name, not a file name, i.e., without a file extension.

The system must be open. `'sys'` can be a character vector, a cell array of character vectors, a numeric handle, or an array of numeric handles.

newsysname

New system name.

'*newsysname*' can be a system name, or a filename with file extension and optional path. If you do not specify a file extension (.slx or .mdl) then `save_system` uses the file format specified in your Simulink preferences.

'*newsysname*' can be empty ([]) , in which case the current name is used. You must specify a *newsysname* argument before any name-value pair arguments even when you don't want to name a new sys, and in that case you leave it empty.

If '*sys*' refers to more than one block diagram, '*newsysname*' must be a cell array of new names.

This command displays an error if you enter any of the following as the new system name:

- A MATLAB keyword
- 'simulink'
- More than 63 characters

Name-Value Pair Arguments

Specify optional comma-separated pairs of *Name*, *Value* arguments. *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as *Name1*, *Value1*, . . . , *NameN*, *ValueN*.

Example: `save_system(sys, newsysname, 'SaveModelWorkspace', true, 'BreakUserLinks', true, 'OverwriteIfChangedOnDisk', true)`

AllowPrompt —

false (default) | true

Logical value that indicates whether to display any output prompt or message in a dialog box or only messages at the command line. For example, prompts to make files writable, or messages about exported versions. If you want to allow prompts, then set to `true`. Also accepts `on` or `off`.

BreakAllLinks —

false (default) | true

Logical value that indicates whether the function replaces links to library blocks with copies of the library blocks in the saved file. The 'BreakAllLinks' option affects any linked block, including user-defined and Simulink library blocks. Also accepts `on` or `off`.

Note The 'BreakAllLinks' option can result in compatibility issues when upgrading to newer versions of Simulink software. For example:

- Any masks on top of library links to Simulink S-functions will not upgrade to the new version of the S-function.
- Any library links to masked subsystems in a Simulink library will not upgrade to the new subsystem behavior.
- Any broken links prevent the automatic library forwarding mechanism from upgrading the link.

If you have saved a model with broken links to builtin libraries, use the Upgrade Advisor to scan the model for out-of-date blocks and upgrade the Simulink blocks to their current versions.

BreakUserLinks —

`false` (default) | `true`

Logical value that indicates whether the function replaces links to user-defined library blocks with copies of the library blocks in the saved file. Also accepts `on` or `off`.

BreakToolboxLinks — Break all toolbox links

`false` (default) | `true` | `'on'` | `'off'`

Break all toolbox links, specified by a logical value that indicates whether the function replaces links to built-in MathWorks library blocks with copies of the library blocks in the saved file. The 'BreakToolboxLinks' option affects Simulink library blocks and blocks from any other libraries supplied with MathWorks toolboxes or blocksets.

ErrorIfShadowed —

`false` (default) | `true`

Logical value that indicates whether the function generates an error if the new name already exists on the MATLAB path or workspace. Also accepts `on` or `off`.

ExportToXML —

false (default) | true

Logical value that indicates whether the function exports the specified block diagram to a file in a simple XML format. Specify the full name of the file, including an extension. The block diagram in memory does not change and no callbacks execute. Do not use this option with any other `save_system` options. Also accepts `on` or `off`. The `ExportToXML` option warns and will be removed in a future release.

ExportToVersion — MATLAB release name

'R2012A' | 'R2014A_MDL' | 'R2016B_SLX' | ...

MATLAB release name, which specifies a previous Simulink version. `save_system` exports the system to a format that the specified previous Simulink version can load. You cannot export to your current version.

If the system contains functionality not supported by the specified Simulink software version, the command removes the functionality and replaces any unsupported blocks with empty masked subsystem blocks colored yellow. As a result, the converted system may generate different results.

To export to Release 2012a and later, you can specify model file format as SLX or MDL. If you do not specify a format, you export your default model file format.

You can also use `Simulink.exportToVersion`.

`SaveAsVersion` is a legacy option for this argument that is also supported.

These version names are not case sensitive:

If you use the Export to Previous Version dialog box instead of `save_system`, then the **Save as type** list supports 7 years of previous releases.

OverwriteIfChangedOnDisk —

false (default) | true

Logical value that indicates whether to overwrite the file on disk (`true`) even if it has been externally modified since the system was loaded. To save the model regardless of whether the file has been changed on disk, supply the `OverwriteIfChangedOnDisk` option with value `true`.

If the file has changed on disk since the model was loaded, `save_system` displays an error to prevent the changes on disk from being overwritten, unless you use the `OverwriteIfChangedOnDisk` option set to `true`. Also accepts `on` or `off`.

You can control whether `save_system` displays an error if the file has changed on disk. Select **Saving the model** under **Change Notification** in the **Model File** pane of the Simulink Preferences dialog box. This preference is on by default.

SaveDirtyReferencedModels —

`false` (default) | `true`

Logical value that indicates whether to save referenced models that contain unsaved changes. Also accepts `on` or `off`.

- If you specify `'true'`, then any unsaved referenced models are saved as well as your top model.
- If you have not specified `SaveDirtyReferencedModels`, or if you specify `'false'`, and there are unsaved referenced models, then you see an error, and no models are saved.

SaveModelWorkspace —

`false` (default) | `true`

Logical value that indicates whether the function saves the contents of the model workspace. The model workspace `DataSource` must be a MAT-file. If the data source is not a MAT-file, `save_system` does not save the workspace. See “Specify Source for Data in Model Workspace”. Also accepts `on` or `off`.

Output Arguments

filename

`save_system` returns the full name of the file that you saved, as a character vector. If you saved multiple files, the return value is a cell array of character vectors.

Examples

The following examples assume prerequisites such as: you have loaded a model and the folder where you want to save is writeable.

Save the current system.

```
save_system
```

Save the vdp system with the name vdp.

```
save_system('vdp')
```

Save the vdp system to a file with the name 'myvdp'. If you do not specify a file extension in the second argument (.slx or .mdl), then `save_system` uses the file format specified in your Simulink preferences.

```
save_system('vdp', 'myvdp')
```

Save the vdp system to another folder.

```
save_system('vdp', 'C:\TMP\vdp.slx')
```

Save an existing model `mymodel` to a different file and specifying the SLX file format:

```
save_system('mymodel', 'newsysname.slx')
```

Save the vdp system to a file with the name 'myvdp' and replace links to library blocks with copies of the library blocks in the saved file.

```
save_system('vdp','myvdp','BreakAllLinks', true)
```

Save the current model (with its current name), and break any library links in it:

```
save_system('mymodel', 'mymodel', 'BreakAllLinks', true)
```

or

```
save_system('mymodel', [], 'BreakAllLinks', true)
```

Save the current model with a new name, but display an error (instead of saving) if something with this name already exists on the MATLAB path:

```
save_system('mymodel', 'mynewmodel', 'ErrorIfShadowed', true)
```

Prevent saving the vdp system with a new name if something with this name already exists on the MATLAB path. In this case `save_system` displays an error (instead of saving) because 'max' is the name of a MATLAB function.

```
save_system('vdp', 'max', 'ErrorIfShadowed', true)
```

Export the vdp system to Simulink Version R2008a with the name 'myvdp'. It does not replace links to library blocks with copies of the library blocks.

```
save_system('vdp', 'myvdp', 'ExportToVersion', 'R2008a')
```

Save the current model with a new name, save the model workspace, break any library links, and overwrite if the file has changed on disk:

```
save_system('mymodel', 'mynewmodel', 'SaveModelWorkspace',  
true, 'BreakAllLinks', true, 'OverwriteIfChangedOnDisk', true)
```

Return the full path name of the file that you saved, as a character vector. If you saved multiple files, the return value is a cell array of character vectors.

```
filename = save_system('mymodel')
```

Return the full path name of a system saved to a new file.

```
filename = save_system('mymodel', 'newmodelname')
```

See Also

`Simulink.exportToVersion` | `close_system` | `new_system` | `open_system`

Topics

“Save a Model”

Introduced before R2006a

set_param

Set system and block parameter values

Syntax

```
set_param(Object, ParameterName, Value, ... ParameterNameN, ValueN)
```

Description

`set_param(Object, ParameterName, Value, ... ParameterNameN, ValueN)` sets the parameter to the specified value on the specified model or block object.

When you set multiple parameters on the same model or block, use a single `set_param` command with multiple pairs of `ParameterName, Value` arguments, rather than multiple `set_param` commands. This technique is efficient because using a single call requires evaluating parameters only once. If any parameter names or values are invalid, then the function doesn't set any parameters.

Tips:

- If you make multiple calls to `set_param` for the same block, then specifying the block using a numeric handle is more efficient than using the full block path. Use `getSimulinkBlockHandle` to get a block handle.
- If you use `matlab -nodisplay` to start a session, you cannot use `set_param` to run your simulation. The `-nodisplay` mode does not support simulation using `set_param`. Use the `sim` command instead.
- After you set parameters in the MATLAB workspace, to see the changes in a model, update the diagram.

```
set_param(model, 'SimulationCommand', 'Update')
```

For parameter names, see:

- “Model Parameters” on page 6-2

- “Block-Specific Parameters” on page 6-128
- “Common Block Properties” on page 6-109

Examples

Set Model Configuration Parameters for a Model

Open `vdp` and set the `Solver` and `StopTime` parameters.

```
vdp
set_param('vdp','Solver','ode15s','StopTime','3000')
```

Set Model Configuration Parameters for Current Model

Open a model and set the `Solver` and `StopTime` parameters. Use `bdroot` to get the current top-level model.

```
vdp
set_param(bdroot,'Solver','ode15s','StopTime','3000')
```

Set a Gain Block Parameter Value

Open `vdp` and set a `Gain` parameter value in the `Mu` block.

```
vdp
set_param('vdp/Mu','Gain','10')
```

Set Position of Block

Open `vdp` and set the position of the `Fcn` block.

```
vdp
set_param('vdp/Fcn','Position',[50 100 110 120])
```

Set Position of Block Using a Handle

Set the position of the Fcn block in the vdp model.

Use `getSimulinkBlockHandle` to load the vdp model if necessary (by specifying `true`), and get a handle to the Fcn block. If you make multiple calls to `set_param` for the same block, then using the block handle is more efficient than specifying the full block path as a character vector.

```
fcnblockhandle = getSimulinkBlockHandle('vdp/Fcn',true);
```

You can use the block handle in subsequent calls to `get_param` or `set_param`. If you examine the handle, you can see that it contains a double. Do not try to use the number of a handle alone (e.g., `5.007`) because you usually need to specify many more digits than MATLAB displays. Instead, assign the handle to a variable and use that variable name to specify a block.

Use the block handle with `set_param` to set the position.

```
set_param(fcnblockhandle,'Position',[50 100 110 120])
```

- “Associating User Data with Blocks”
- “Use MATLAB Commands to Change Workspace Data”
- “Control Simulations Programmatically”

Input Arguments

object — Name or handle of a model or block

character vector | handle

Handle or name of a model or block, specified as a numeric handle or a character vector. A numeric handle must be a scalar. You can also set parameters of lines and ports, but you must use numeric handles to specify them.

Tip If you make multiple calls to `set_param` for the same block, then specifying a block using a numeric handle is more efficient than using the full block path with `set_param`. Use `getSimulinkBlockHandle` to get a block handle. Do not try to use the number of a handle alone (e.g., `5.007`) because you usually need to specify many more digits than

MATLAB displays. Assign the handle to a variable and use that variable name to specify a block.

Example: `'vdp/Fcn'`

ParameterName — Model or block parameter name

character vector

Model or block parameter name, specified as the comma-separated pair consisting of the parameter name, specified as a character vector, and the value, specified in the format determined by the parameter type. Case is ignored for parameter names. Value are case sensitive. Values are often character vectors, but they can also be numeric, arrays, and other types. Many block parameter values are specified as character vectors, but two exceptions are these parameters: `Position`, specified as a vector, and `UserData`, which can be any data type.

Example: `'Solver', 'ode15s', 'StopTime', '3000'`

Example: `'SimulationCommand', 'start'`

Example: `'Position', [50 100 110 120]`

Data Types: `char`

See Also

`bdroot` | `gcb` | `gcs` | `getSimulinkBlockHandle` | `get_param` | `new_system` | `open_system`

Topics

“Associating User Data with Blocks”

“Use MATLAB Commands to Change Workspace Data”

“Control Simulations Programmatically”

“Model Parameters” on page 6-2

“Block-Specific Parameters” on page 6-128

“Common Block Properties” on page 6-109

Introduced before R2006a

setActiveConfigSet

Specify model's active configuration set or configuration reference

Syntax

```
setActiveConfigSet(model, configObjName)
```

Arguments

`model`

The name of an open model, or `gcs` to specify the current model

`configObjName`

The name of a configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

Description

`setActiveConfigSet` specifies the active configuration set or configuration reference (configuration object) of `model` to be the configuration object specified by `configObjName`. If no such configuration object is attached to the model, an error occurs. The previously active configuration object becomes inactive.

Examples

The following example makes `DevConfig` the active configuration object of the current model. The code is the same whether `DevConfig` is a configuration set or configuration reference.

```
setActiveConfigSet(gcs, 'DevConfig');
```

See Also

attachConfigSet | attachConfigSetCopy | closeDialog | detachConfigSet |
getActiveConfigSet | getConfigSet | getConfigSets | openDialog

Topics

“Manage a Configuration Set”

“Manage a Configuration Reference”

Introduced before R2006a

sfix

Create `Simulink.NumericType` object describing signed fixed-point data type

Syntax

```
a = sfix(WordLength)
```

Description

`sfix(WordLength)` returns a `Simulink.NumericType` object that describes a signed fixed-point number with the specified word length and unspecified scaling.

Note `sfix` is a legacy function. In new code, use `fixdt` instead. In existing code, replace `sfix(WordLength)` with `fixdt(1,WordLength)`.

Examples

Define a 16-bit signed fixed-point data type.

```
a = sfix(16)
```

```
a =
```

```
NumericType with properties:
```

```
    DataTypeMode: 'Fixed-point: unspecified scaling'  
    Signedness: 'Signed'  
    WordLength: 16  
    IsAlias: 0  
    DataScope: 'Auto'  
    HeaderFile: ''  
    Description: ''
```

See Also

`Simulink.NumericType` | `fixdt` | `float` | `sfrac` | `sint` | `ufix` | `ufrac` | `uint`

Introduced before R2006a

sfrac

Create `Simulink.NumericType` object describing signed fractional data type

Syntax

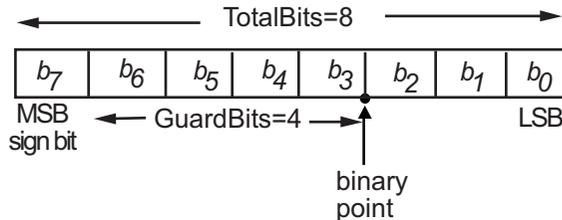
```
a = sfrac(WordLength)
a = sfrac(WordLength, GuardBits)
```

Description

`sfrac(WordLength)` returns a `Simulink.NumericType` object that describes the data type of a signed fractional data type with a word size given by `WordLength`.

`sfrac(WordLength, GuardBits)` returns a `Simulink.NumericType` object that describes the data type of a signed fractional number. The total word size is given by `WordLength` with `GuardBits` bits located to the left of the binary point.

The most significant (leftmost) bit is the sign bit. The default binary point for this data type is assumed to lie immediately to the right of the sign bit. If guard bits are specified, they lie to the left of the binary point and to right of the sign bit. For example, the structure for an 8-bit signed fractional data type with 4 guard bits is:



Note `sfrac` is a legacy function. In new code, use `fixdt` instead. In existing code, replace `sfrac(WordLength, GuardBits)` with `fixdt(1, WordLength, (WordLength-1-GuardBits))` and `sfrac(WordLength)` with `fixdt(1, WordLength, (WordLength-1))`.

Examples

Define an 8-bit signed fractional data type with 4 guard bits. Note that the range of this data type is $-2^4 = -16$ to $(1 - 2^{(1-8)}) \cdot 2^4 = 15.875$.

```
a = sfrac(8,4)
```

```
a =
```

```
NumericType with properties:
```

```
    DataTypeMode: 'Fixed-point: binary point scaling'  
        Signedness: 'Signed'  
        WordLength: 8  
    FractionLength: 3  
        IsAlias: 0  
        DataScope: 'Auto'  
        HeaderFile: ''  
        Description: ''
```

See Also

`Simulink.NumericType` | `fixdt` | `float` | `sfix` | `sint` | `ufix` | `ufrac` | `uint`

Introduced before R2006a

signalbuilder

Create and access Signal Builder blocks

Syntax

```
[time, data] = signalbuilder(block)
[time, data, signames] = signalbuilder(block)
[time, data, signames, groupnames] = signalbuilder(block)
block = signalbuilder([], 'create', time, data, signames,
groupnames)
block = signalbuilder(path, 'create', time, data, signames,
groupnames)
block = signalbuilder(path, 'create', time, data, signames,
groupnames, vis)
block = signalbuilder(path, 'create', time, data, signames,
groupnames, vis, pos)
block = signalbuilder(path, 'create', time, data, signames,
groupnames, vis, pos, {openui openmodel})
block = signalbuilder(block, 'append', time, data, signames,
groupnames)
block = signalbuilder(block, 'appendgroup', time, data, signames,
groupnames)
signalbuilder(block, 'append', ds)
signalbuilder(block, 'appendgroup', ds)
signalbuilder(block, 'append', [ds ds2])
signalbuilder(block, 'appendgroup', [ds ds2])
signalbuilder(block, 'appendsignal', time, data, signames)
signalbuilder(block, 'showsignal', signal, group)
signalbuilder(block, 'hidesignal', signal, group)
[time, data] = signalbuilder(block, 'get', signal, group)
ds=signalbuilder(block, 'get', group)
[ds1 ds2] =signalbuilder(block, 'get', group)
signalbuilder(block, 'set', signal, group, time, data)
signalbuilder(block, 'set', groupid, ds)
signalbuilder(block, 'set', groupid, [ds1 ds2])
index = signalbuilder(block, 'activegroup')
```

```
[index, activeGroupLabel]= signalbuilder(block, 'activegroup')
signalbuilder(block, 'activegroup', index)
signalbuilder(block, 'annotategroup', onoff)
signalbuilder(block, 'print', [])
signalbuilder(block, 'print', config, printArgs)
figh = signalbuilder(block, 'print', config, 'figure')
```

Description

Use the `signalbuilder` command to interact programmatically with Signal Builder blocks.

- “Creating and Accessing Signal Builder Blocks” on page 2-441
- “Adding New Groups” on page 2-443
- “Working with Signals” on page 2-444
- “Using Get/Set Methods for Specific Signals and Groups” on page 2-445
- “Querying, Labelling, and Setting the Active Group” on page 2-446
- “Enabling Current Group Display” on page 2-446
- “Printing Signal Groups” on page 2-447
- “Interpolating Missing Data Values” on page 2-448

Note When you use the `signalbuilder` command to interact with a Signal Builder block, the **Undo last edit** and **Redo last edit** buttons on the block dialog box are grayed out. You cannot undo the results of using the `signalbuilder` command.

Creating and Accessing Signal Builder Blocks

`[time, data] = signalbuilder(block)` returns the time (x -coordinate) and amplitude (y -coordinate) data of the Signal Builder block, `block`.

The output arguments, `time` and `data`, take different formats depending on the block configuration:

Configuration	Time/Data Format
1 signal, 1 group	Row vector of break points.

Configuration	Time/Data Format
>1 signal, 1 group	Column cell vector where each element corresponds to a separate signal and contains a row vector of points.
1 signal, >1 group	Row cell vector where each element corresponds to a separate group and contains a row vector of points.
>1 signal, >1 group	Cell matrix where each element (i, j) corresponds to signal i and group j.

`[time, data, signames] = signalbuilder(block)` returns the signal names, `signames`, in a character vector or a cell array of character vectors.

`[time, data, signames, groupnames] = signalbuilder(block)` returns the group names, `groupnames`, in a character vector or a cell array of character vectors.

`block = signalbuilder([], 'create', time, data, signames, groupnames)` creates a Signal Builder block in a new Simulink model using the specified values. The preceding table describes the allowable formats of `time` and `data`. If `data` is a cell array and `time` is a vector, the `time` values are duplicated for each element of `data`. Each vector in `time` and `data` must be the same length and have at least two elements. If `time` is a cell array, all elements in a column must have the same initial and final value. Signal names, `signames`, and group names, `groupnames`, can be omitted to use default values. The function returns the path to the new block, `block`. Always provide `time` and `data` when using the `create` command. These two parameters are always required.

`block = signalbuilder(path, 'create', time, data, signames, groupnames)` creates a new Signal Builder block at `path` using the specified values. If `path` is empty, the function creates a block in a new model, which has a default name. If `data` is a cell array and `time` is a vector, the `time` values are duplicated for each element of `data`. Each vector within `time` and `data` must be the same length and have at least two elements. If `time` is a cell array, all elements in a column must have the same initial and final value. Signal names, `signames`, and group names, `groupnames`, can be omitted to use default values. The function returns the path to the new block, `block`. Always provide `time` and `data` when using the `create` command. These two parameters are always required.

`block = signalbuilder(path,'create', time, data, signames, groupnames, vis)` creates a new Signal Builder block and sets the visible signals in each group based on the values of the matrix `vis`. This matrix must be the same size as the cell array, `data`. Always provide `time` and `data` when using the `create` command. These two parameters are always required. You cannot create Signal Builder blocks in which all signals are invisible. For example, if you set the `vis` parameter for all signals to 0, the first signal is still visible.

`block = signalbuilder(path,'create', time, data, signames, groupnames, vis, pos)` creates a new Signal Builder block and sets the block position to `pos`. Always provide `time` and `data` when using the `create` command. These two parameters are always required. You cannot create Signal Builder blocks in which all signals are invisible. For example, if you set the `vis` parameter for all signals to 0, the first signal is still visible.

If you create signals that are smaller than the display range or do not start from 0, the Signal Builder block extrapolates the undefined signal data. It does so by holding the final value.

`block = signalbuilder(path,'create', time, data, signames, groupnames, vis, pos, {openui openmodel})` creates a new Signal Builder block and opens or invisibly loads the model and Signal Builder block window.

- Set `openui` to 1 to open the Signal Builder block window when you create the Signal Builder block. Set `openui` to 0 keep this window closed upon block creation.
- Set `openmodel` to 1 to open the model when you create the Signal Builder block. Set `openmodel` to 0 to invisibly load the model upon block creation.

Adding New Groups

`block = signalbuilder(block, 'append', time, data, signames, groupnames)` or `block = signalbuilder(block, 'appendgroup', time, data, signames, groupnames)` appends new groups to the Signal Builder block, `block`. The `time` and `data` arguments must have the same number of signals as the existing block.

`signalbuilder(block, 'append', ds)` or `signalbuilder(block, 'appendgroup', ds)` appends one data set.

`signalbuilder(block, 'append', [ds ds2])` or `signalbuilder(block, 'appendgroup', [ds ds2])` appends N data sets.

Data sets must have the same number of elements as the signals in a signal group. Data set format limitations for the `set`, `append`, and `appendgroup` functions include:

- Elements must be MATLAB timeseries data.
- Timeseries data and/or time must not be empty.
- Timeseries data must be of type double.
- Timeseries data must be 1-D (scalar value at each time).

Note

- If you specify a value of ' ' or {} for *signames*, the function uses existing signal names for the new groups.
 - If you do not specify a value for *groupnames*, the function creates the new signal groups with the default group name pattern, `GROUP #n`.
-

Working with Signals

`signalbuilder(block, 'appendsignal', time, data, signames)` appends new signals to all signal groups in the Signal Builder block, *block*. You can append either the same signals to all groups, or append different signals to different groups. Regardless of which signals you append, append the same number of signals to all the groups. Append signals to all the groups in the block; you cannot append signals to a subset of the groups. Correspondingly, provide *time* and *data* arguments for either one group (append the same information to all groups) or different *time* and *data* arguments for different groups. To use default signal names, omit the signal names argument, *signames*.

`signalbuilder(block, 'showsignal', signal, group)` makes *signals* that are hidden from the Signal Builder block visible. By default, signals in the current active group are visible when created. You control the visibility of a signal at creation with the *vis* parameter. *signal* can be a unique signal name, a signal scalar index, or an array of signal indices. *group* is the list of one or more signal groups that contains the affected signals. *group* can be a unique group name, a scalar index, or an array of indices.

`signalbuilder(block, 'hidesignal', signal, group)` makes signals, *signal*, hidden from the Signal Builder block. By default, all signals are visible when created. *signal* can be a unique signal name, a signal scalar index, or an array of signal indices.

group is the list of one or more signal groups that contains the affected signals. *group* can be a unique group name, a scalar index, or an array of indices.

Note For the `showsignal` and `hidesignal` methods, if you do not specify a value for the `group` argument, `signalbuilder` applies the operation to all the signals and groups.

Using Get/Set Methods for Specific Signals and Groups

`[time, data] = signalbuilder(block, 'get', signal, group)` gets the time and data values for the specified signal(s) and group(s). The `signal` argument can be the name of a signal, a scalar index of a signal, or an array of signal indices. The `group` argument can be a group name, a scalar index, or an array of indices.

`ds=signalbuilder(block, 'get', group)` gets one data set for one requested signal Builder group.

`[ds1 ds2] =signalbuilder(block, 'get', group)` gets *N* data sets for *N* requested Signal Builder groups.

`signalbuilder(block, 'set', signal, group, time, data)` sets the time and data values for the specified signal(s) and group(s). Use empty values of *time* and *data* to remove groups and signals. To remove a signal group, you must also remove all the signals in that group in the same command.

`signalbuilder(block, 'set', groupid, ds)` sets one data set for the requested Signal Builder group. Specifying an empty data set deletes the groups specified in *groupid*.

`signalbuilder(block, 'set', groupid, [ds1 ds2])` sets *N* data sets for *N* requested groups.

Data sets must have the same number of elements as the signals in a signal group. Data set format limitations for the `set`, `append`, and `appendgroup` functions include:

- Elements must be MATLAB timeseries data.
- Timeseries data and/or time must not be empty.
- Timeseries data must be of type double.

- Timeseries data must be 1-D (scalar value at each time).

Note For the `set` method, if you do not specify a value for the `group` argument, `signalbuilder` applies the operation to all signals and groups.

When removing signals, you remove all signals from all groups. You cannot select a subset of groups from which to remove signals, unless you are also going to also remove that group.

Note The `signalbuilder` function does not allow you to alter and delete data in the same invocation. It also does not allow you to delete all the signals and groups from the application.

If you set signals that are smaller than the display range or do not start from 0, the Signal Builder block extrapolates the undefined signal data by holding the final value.

Querying, Labelling, and Setting the Active Group

`index = signalbuilder(block, 'activegroup')` gets the index of the active group.

`[index, activeGroupLabel]= signalbuilder(block, 'activegroup')` gets the label value of the active group.

`signalbuilder(block, 'activegroup', index)` sets the active group index to *index*.

Enabling Current Group Display

`signalbuilder(block, 'annotategroup', onoff)` controls the display of the current group name on the mask of the Signal Builder block.

<i>onoff</i> Value	Description
'on'	Default. Displays the current group name on the block mask.
'off'	Does not display the current group name on the block mask.

Printing Signal Groups

`signalbuilder(block, 'print', [])` prints the currently active signal group.

`signalbuilder(block, 'print', config, printArgs)` prints the currently active signal group or the signal group that `config` specifies. The argument `config` is a structure that allows you to customize the printed appearance of a signal group. The `config` structure may contain any of the following fields:

Field	Description	Example Value
<code>groupIndex</code>	Scalar specifying index of signal group to print	2
<code>timeRange</code>	Two-element vector specifying the time range to print (must not exceed the block's time range)	[3 6]
<code>visibleSignals</code>	Vector specifying index of signals to print	[1 2]
<code>yLimits</code>	Cell array specifying limits for each signal's <i>y</i> -axis	{[-1 1], [0 1]}
<code>extent</code>	Two-element vector of the form: [width, height] specifying the dimensions (in pixels) of the area in which to print the signals	[500 300]
<code>showTitle</code>	Logical value specifying whether to print a title; <code>true</code> (1) prints the title	false

Set up the structure with one or more of these fields before you print. For example, if you want to print just group 2 using a configuration structure, `configstruct`, set up the structure as follows. You do not need to specify any other fields.

```
configstruct.groupIndex=2
```

The optional argument `printArgs` allows you to configure print options (see `print` in the MATLAB Function Reference).

`figh = signalbuilder(block, 'print', config, 'figure')` prints the currently active signal group or the signal group that `config` specifies to a new hidden figure handle, `figh`.

Interpolating Missing Data Values

When specifying a periodic signal such as a Sine Wave, the `signalbuilder` function uses linear Lagrangian interpolation to compute data values for time steps that occur between time steps for which the `signalbuilder` function supplies data. When specifying periodic signals, specify them as a time vector that is defined as multiples of sample time, for example:

```
t = 0.2*[0:49]';
```

Examples

Example 1

Create a Signal Builder block in a new model editor window:

```
block = signalbuilder([], 'create', [0 5], {[2 2];[0 2]});
```

Get signal builder data from this block.

```
[time, data, signames, groupnames] = signalbuilder('untitled/Signal Builder')
```

```
time =
```

```
    [1x2 double]  
    [1x2 double]
```

```
data =
```

```
    [1x2 double]  
    [1x2 double]
```

```
signames =
```

```
    'Signal 1'    'Signal 2'
```

```
groupnames =
```

```
    'Group 1'
```

The Signal Builder block contains two signals in one group. Alter the second signal in the group:

```
signalbuilder(block, 'set', 2, 1, [0 5], [2 0])
```

To make this same change using the signal name and group name:

```
signalbuilder(block, 'set', 'Signal 2', 'Group 1', [0 5], [2 0])
```

Delete the first signal from the group:

```
signalbuilder(block, 'set', 1, 1, [], [])
```

Append the group with a new signal:

```
signalbuilder(block, 'append', [0 2.5 5], [0 2 0], 'Signal 2', 'Group 2');
```

Append another group with a new signal using appendgroup:

```
signalbuilder(block, 'appendgroup', [0 2.5 5], [0 2 0], 'Signal 2', 'Group 3');
```

Example 2

Create a Signal Builder block in a new model editor window:

```
block = signalbuilder([], 'create', [0 2], {[0 1],[1 0]});
```

The Signal Builder block has two groups, each of which contains a signal. To delete the second group, also delete its signal:

```
signalbuilder(block, 'set', 1, 2, [], [])
```

Example 3

Create a Signal Builder block in a new model editor window:

```
block = signalbuilder([], 'create', [0 1], ...
    {[0 0],[1 1];[1 0],[0 1];[1 1],[0 0]});
```

The Signal Builder block has two groups, each of which contains three signals.

Example 4

Create a Signal Builder block in a new model editor window:

```
block = signalbuilder([], 'create', {[0 10],[0 20]}, {[6 -6],...  
[2 5]});
```

The Signal Builder block has two groups. Each group contains one signal.

Append a new signal group to the existing block.

```
block = signalbuilder(block, 'append', [0 30], [10 -10]);
```

Append a new signal, sig3, to all groups.

```
signalbuilder(block, 'appendsignal', [0 30], [0 10], 'sig3');
```

Example 5

Create a Signal Builder block in a new model editor window:

```
time = [0 1];  
data = {[0 0],[1 1];[1 0],[0 1];[1 1],[0 0]};  
block = signalbuilder([], 'create', time, data);
```

The Signal Builder block has two groups. Each group contains three signals.

Delete the second group. To delete a signal group, also delete all the signals in the group.

```
signalbuilder(block, 'set', [1,2,3], 'Group 2', []);
```

Example 6

Create a Signal Builder block in a new model editor window:

```
block = signalbuilder([], 'create', [0 5], {[2 2];[0 2]});
```

The Signal Builder block has one group that contains two signals.

Hide the signal, Signal 1.

```
signalbuilder(block, 'hidesignal', 'Signal 1', 'Group 1')
```

Signal 1 is no longer visible in the Signal Builder block.

Make Signal 1 visible again.

```
signalbuilder(block, 'showsignal', 'Signal 1', 'Group 1')
```

Example 7

Create a Signal Builder block in a new model editor window:

```
block = signalbuilder([], 'create', [0 5], {[2 2] [0 2]});
```

The Signal Builder block has two groups, each with one signal.

Create a structure, `configstruct`, to customize the Signal Builder block that you want to print.

```
configstruct.groupIndex = 2;
configstruct.timeRange = [0 2];
configstruct.visibleSignals = 1;
configstruct.yLimits = {[0 1]};
configstruct.extent = [500 300];
configstruct.showTitle = true;
```

This sequence fills all the fields of the `configstruct` structure.

Print a view of the Signal Builder block to the default printer. The `configstruct` structure defines the view to print.

```
signalbuilder(block, 'print', configstruct)
```

Print with print options, for example `-dps`.

```
signalbuilder(block, 'print', configstruct, '-dps')
```

Print a view of the Signal Builder block as defined by the `configstruct` structure to a new hidden figure handle, `figH`.

```
figH = signalbuilder(block, 'print', configstruct, 'figure')
figure(figH)
```

Example 8

Create two Signal Builder blocks in new model editor windows:

```
block = signalbuilder([], 'create', [0 5], {[2 2];[0 2]});
block1 = signalbuilder('untitled/Signal Builder1', 'create', [1 2], {[1 2];[0 10]});
```

Get a data set for group 1 of `block`.

```
ds=signalbuilder(block, 'get', 1);
```

Get a data set for group 1 of *block1*.

```
ds1=signalbuilder(block1, 'get', 1);
```

Set the data set for group 1 of *block* to *ds1*.

```
signalbuilder(block, 'set', 1, ds1);
```

Append the original data set for group 1 of *block* (*ds*) to *block*.

```
signalbuilder(block, 'append', ds);
```

To create a third group in *block*, append *ds1* to the end of the groups in *block*.

```
signalbuilder(block, 'appendgroup', ds1);
```

See Also

Signal Builder

Topics

“Signal Groups”

Introduced in R2007a

signalEditor

Start Signal Editor

Syntax

```
signalEditor  
signalEditor(Name,Value)
```

Description

`signalEditor` starts Signal Editor without an associated model.

`signalEditor(Name,Value)` starts signal Editor using additional options specified by one or more name-value pair arguments.

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify the name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

- `Model` - Model name, specified as a character array, for which Signal Editor is to start. You can specify one model per call to the `signalEditor` function.

Note Load the model before starting Signal Editor for it.

- `DataSource` - Data set name, specified as a character array, to be edited. You can specify one data set file per call to the `signalEditor` function.

Note You can start multiple sessions of Signal Editor for the same model. However, you can associate a data set file with only one Signal Editor at a time. A data set file cannot have multiple Signal Editor sessions associated with it.

Examples

Start Signal Editor for a Model

Start the Signal Editor for the model, `slexAutotransRootInportsExample`.

Load the `slexAutotransRootInportsExample` model, then start Signal Editor for it.

```
open_system('slexAutotransRootInportsExample')
signalEditor('Model', 'slexAutotransRootInportsExample');
```

Start Signal Editor to Edit a Data Set File

Start Signal Editor to edit `myFile.mat`.

```
signalEditor('DataSource', 'myFile.mat');
```

Start Signal Editor and Data Set File for a Model

Load the `slexAutotransRootInportsExample` model, then start Signal Editor for the model, and edit `myFile.mat`.

```
open_system('slexAutotransRootInportsExample')
signalEditor('Model', 'slexAutotransRootInportsExample', 'DataSource', 'myFile.mat');
```

See Also

Topics

“Create and Edit Signal Data”

Introduced in R2017b

sim

Simulate dynamic system

Syntax

```
simOut = sim(model,Name,Value)
simOut = sim(model,ParameterStruct)
simOut = sim(model,ConfigSet)
simOut = sim(model)
simOut = sim(model,'ReturnWorkspaceOutputs','on')
simOut = sim(simIn)
simOut = sim(simIn,'ShowProgress',true)
```

Description

`simOut = sim(model,Name,Value)` simulates the specified model using parameter name-value pairs.

`simOut = sim(model,ParameterStruct)` simulates the specified model using the parameter values specified in the structure `ParameterStruct`.

`simOut = sim(model,ConfigSet)` simulates the specified model using the configuration settings specified in the model configuration set `ConfigSet`.

`simOut = sim(model)` simulates the specified model using existing model configuration parameters, and returns the result as either a `Simulink.SimulationOutput` object (single-output format) or as a time vector compatible with Simulink version R2009a or earlier.

To return simulation results using the single-output format (simulation object), select **Single simulation output** on the Data Import/Export pane of the Configuration Parameters dialog box. This selection overrides the `Dataset` format used for signal logging.

To return simulation results using the backward-compatible format (time vector), see “Backward-Compatible Syntax” on page 2-463.

`simOut = sim(model, 'ReturnWorkspaceOutputs', 'on')` simulates the specified model using existing model configuration parameters, and returns the result as a `Simulink.SimulationOutput` object (single-output format).

`simOut = sim(simIn)` simulates a model using the inputs specified in the `SimulationInput` object, `simIn`. The `sim` command is also used with an array of `SimulationInput` objects to run multiple simulations in a series.

`simOut = sim(simIn, 'ShowProgress', true)` simulates a model in a series using an array of `SimulationInput` objects and shows the progress of these simulations at the command line.

Examples

Simulate Model with `sim` Command-Line Options

Simulate the model, `vdp`, in Rapid Accelerator mode for an absolute tolerance of $1e-5$ and save the states in `xoutNew` and the output in `youtNew`.

Specify parameters as name-value pairs to the `sim` command:

```
simOut = sim('vdp', 'SimulationMode', 'rapid', 'AbsTol', '1e-5', ...
            'StopTime', '30', ...
            'ZeroCross', 'on', ...
            'SaveTime', 'on', 'TimeSaveName', 'tout', ...
            'SaveState', 'on', 'StateSaveName', 'xoutNew', ...
            'SaveOutput', 'on', 'OutputSaveName', 'youtNew', ...
            'SignalLogging', 'on', 'SignalLoggingName', 'logout')

### Building the rapid accelerator target for model: vdp
### Successfully built the rapid accelerator target for model: vdp

Simulink.SimulationOutput:
    tout: [86x1 double]
    xoutNew: [86x2 double]
    youtNew: [86x2 double]

SimulationMetadata: [1x1 Simulink.SimulationMetadata]
```

```
ErrorMessage: [0x0 char]
```

Simulate Model with `sim` Command-Line Options in Structure

Simulate the model, `vdp`, in Rapid Accelerator mode for an absolute tolerance of $1e-5$ and save the states in `xoutNew` and the output in `youtNew`.

Specify parameters using a name-value pairs structure `paramNameValStruct` for the `sim` command:

```
paramNameValStruct.SimulationMode = 'rapid';
paramNameValStruct.AbsTol         = '1e-5';
paramNameValStruct.SaveState      = 'on';
paramNameValStruct.StateSaveName  = 'xoutNew';
paramNameValStruct.SaveOutput     = 'on';
paramNameValStruct.OutputSaveName = 'youtNew';
simOut = sim('vdp',paramNameValStruct)

### Building the rapid accelerator target for model: vdp
### Successfully built the rapid accelerator target for model: vdp

Simulink.SimulationOutput:
    xoutNew: [65x2 double]
    youtNew: [65x2 double]

SimulationMetadata: [1x1 Simulink.SimulationMetadata]
ErrorMessage: [0x0 char]
```

Simulate Model with `sim` Command-Line Options in Configuration Set

Simulate the model, `vdp`, in Rapid Accelerator mode for an absolute tolerance of $1e-5$ and save the states in `xoutNew` and the output in `youtNew`.

Specify parameters as name-value pairs in configuration set `mdl_cs` for the `sim` command:

```
mdl = 'vdp';
load_system(mdl)
simMode = get_param(mdl, 'SimulationMode');
set_param(mdl, 'SimulationMode', 'rapid')
```

```
cs = getActiveConfigSet mdl;
mdl_cs = cs.copy;
set_param(mdl_cs, 'AbsTol', '1e-5', ...
          'SaveState', 'on', 'StateSaveName', 'xoutNew', ...
          'SaveOutput', 'on', 'OutputSaveName', 'youtNew')
simOut = sim(mdl, mdl_cs);
set_param(mdl, 'SimulationMode', simMode)

### Building the rapid accelerator target for model: vdp
### Successfully built the rapid accelerator target for model: vdp
```

Simulate Model with Default Parameter Settings

Simulate the model `vdp` using default model configuration parameters.

```
simOut = sim('vdp', 'ReturnWorkspaceOutputs', 'on')
```

Input Arguments

model — Model to simulate

character vector

Name of model to simulate, specified as a character vector.

Example: `simOut = sim('vdp')`

ParameterStruct — Structure containing parameter settings

structure

Structure with fields that are the names of the configuration parameters for the simulation. The corresponding values are the parameter values.

Example: `simOut = sim('vdp', paramNameValStruct)`

ConfigSet — Configuration set

object

The set of configuration parameters for a model.

Example: `simOut = sim('vdp', mdl_cs)`

simIn — SimulationInput object for a model

object

SimulationInput object created by specifying the model name. For more information, see `Simulink.SimulationInput`

```
Example: simIn = Simulink.SimulationInput('CSTR')
```

Use the `SimulationInput` object to specify Block Parameters, Model Parameters, Variables and External Inputs to a model to be simulated.

```
Example: simIn.addBlockParameter('CSTR/Feed Temperature', 'Value',  
'300'); simIn.addModelParameter('StartTime', '1');  
simIn.addVariable('FeedTemp0', 320)
```

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Solver', 'ode15s', 'TimeOut', 30` specifies that the model is simulated using the `ode15s` solver with a maximum simulation time of 30 seconds.

The `sim` command accepts all simulation parameters as name-value pair arguments. See “Model Parameters” on page 6-2 for a list of all simulation parameters.

In addition, the `sim` command accepts the following parameters.

CaptureErrors — Save errors to SimulationMetadata object

`off` (default) | `on`

By default, if an error occurs during simulation, the `sim` command stops and reports the error in the MATLAB Command Window. If you specify `'CaptureErrors', 'on'`, the `sim` command does not stop, but instead saves any errors to the `ErrorDiagnostic` structure within the `SimulationMetadata` object. The error message is saved in the `ErrorMessage` property of the `SimulationOutput` object.

This option is useful when running multiple simulations in a loop, so that one simulation error will not stop a script or function from continuing.

If you specify an array of input objects, the `sim` commands runs with `CaptureErrors` enabled. If an error occurs, the error messages are included in the

`Simulink.SimulationMetadata` object for the simulation, as well as simulation data up to the point of the failure.

This option is not available for simulation in SIL and PIL modes.

Example: `'CaptureErrors', 'on'`

ConcurrencyResolvingToFileSuffix — Append suffix to model name

character vector

(Rapid Accelerator mode only) Appends this suffix character vector to the filename of a model (before the file extension) if:

- The model contains a To File block.
- You call the `sim` command from `parfor`.

Example: `'ConcurrencyResolvingToFileSuffix', 'model'`

Debug — Run simulation in debug mode

off (default) | on | cmds

Starts the simulation in debug mode (see “Debugger Graphical User Interface” for more information). The value of this option can be a cell array of commands to be sent to the debugger after it starts.

Example: `'Debug', 'on'`

LoggingFileName — Specify name of MAT-file to log data

out.mat (default) | character vector

Use when you enable the `LoggingToFile` name-value pair for logging to persistent storage. Specify the destination MAT-file for data logging.

Tip Do not use a file name from one locale in a different locale.

Example: `'LoggingFileName', 'out.mat'`

LoggingToFile — Log simulation data to MAT-file

off (default) | on

Store logged data that uses `Dataset` format to persistent storage (MAT-file).

Use this feature when logging large amounts of data that can cause memory issues. For details, see “Log Data to Persistent Storage”.

Tip To avoid running out of memory when accessing stored data, you can use a reference to access the object stored in the MAT-file. Use a `Simulink.SimulationData.DatasetRef` object to access stored data by reference. Using this object loads signal logging and states data into the model workspace incrementally (signal by signal). Accessing data for other kinds of logging loads all of the data at once.

Example: `'LoggingToFile', 'on'`

RapidAcceleratorParameterSets — Pass run-time parameters directly to Rapid Accelerator simulations

structure

(Rapid Accelerator mode only) Returns structure that contains run-time parameters for running Rapid Accelerator simulations in `parfor`.

Example: `'RapidAcceleratorParameterSets', parameterSet(idx)`

RapidAcceleratorUpToDateCheck — Perform up-to-date check before simulation

on (default) | off

(Rapid Accelerator mode only) Enables/disables up-to-date check. If you set this value to 'off', Simulink does not perform an up-to-date check. It skips the start/stop callbacks in blocks. If you call the `sim` command from `parfor`, set this value to 'off'.

When you set this option to 'off', changes that you make to block parameter values in the model (for example, by using block dialog boxes, by using the `set_param` function, or by changing the values of MATLAB variables) do not affect the simulation. Instead, use `RapidAcceleratorParameterSets` to pass new parameter values directly to the simulation.

Example: `'RapidAcceleratorUpToDateCheck', 'off'`

SrcWorkspace — Workspace in which to evaluate MATLAB expressions

base (default) | current | parent

Specifies the workspace in which to evaluate MATLAB expressions defined in the model. Setting `SrcWorkspace` has no effect on a referenced model that executes in Accelerator

mode. Setting `SrcWorkspace` to `current` within a `parfor` loop causes a transparency violation.

Example: `'SrcWorkspace', 'current'`

TimeOut — Maximum simulation run time

positive scalar

Specify the time, in seconds, to allow the simulation to run. If you run your model for a period longer than the value of `TimeOut`, the software issues a warning and stops the simulation. `TimeOut` refers to the time spent for a simulation.

Example: `'TimeOut', 60`

Trace — Enables simulation tracing facilities

`minstep` | `siminfo` | `compile`

Enables simulation tracing facilities (specify one or more as a comma-separated list):

- `'minstep'` specifies that simulation stops when the solution changes so abruptly that the variable-step solvers cannot take a step and satisfy the error tolerances.
- `'siminfo'` provides a short summary of the simulation parameters in effect at the start of simulation.
- `'compile'` displays the compilation phases of a block diagram model.

By default, Simulink issues a warning message and continues the simulation.

Example: `'Trace', 'minstep', 'Trace', 'siminfo', 'Trace', 'compile'`

Output Arguments

simOut — Simulation object containing logged simulation results

object

`Simulink.SimulationOutput` object that contains all of the logged simulation results.

All simulation outputs (logged time, states, and signals) are returned in a single `Simulink.SimulationOutput` object. You define the model time, states, and output that is logged using the **Data Import/Export** pane of the **Model Configuration Parameters** dialog box. You can log signals using blocks such as the `To Workspace` and `Scope` blocks. The **Signal & Scope Manager** can directly log signals.

Note The output of the `sim` command always returns to `SimOut`, the single simulation output object. The simulation output object in turn, is returned to the workspace.

Definitions

Backward-Compatible Syntax

Starting with R2009b, the `sim` command was enhanced to provide greater compatibility with parallel computing. The improved single-output format saves all simulation results to a single object, simplifying the management of output variables.

For backward compatibility with R2009a or earlier releases, use the backward-compatible syntax:

```
[T,X,Y] = sim('model',Timespan, Options, UT)
[T,X,Y1,...,Yn] = sim('model',Timespan, Options, UT)
```

If you specify only the `model` argument, Simulink automatically saves the time, state, and output to the specified output arguments.

If you do not specify any output arguments, Simulink determines what data to log based on the settings for the **Configuration Parameters > Data Import/Export** pane. Simulink stores the simulation output either in the current workspace or in the variable `ans`, based on the setting for **Save simulation output as a single object** parameter.

Argument	Description
<i>T</i>	The time vector returned.
<i>X</i>	The state returned in matrix or structure format. The state matrix contains continuous states followed by discrete states.
<i>Y</i>	The output returned in matrix or structure format. For block diagram models, this variable contains all root-level blocks.
<i>Y1,...,Yn</i>	The outputs, which can only be specified for diagram models. Here, <i>n</i> must be the number of root-level blocks. Each output will be returned in the <i>Y1,...,Yn</i> variables.
' <i>model</i> '	The name of the model to simulate.

Argument	Description
<i>Timespan</i>	The timespan can be <code>TFinal</code> , <code>[TStart TFinal]</code> , or <code>[TStart OutputTimes TFinal]</code> . Output times are time points returned in <i>T</i> , but in general, <i>T</i> includes additional time points.
<i>Options</i>	Optional simulation parameters created in a structure by the <code>simset</code> command using name-value pairs.
<i>UT</i>	Optional external inputs. For supported expressions, see “Load Data to Root-Level Input Ports”.

Simulink requires only the `model` argument. Simulink takes all defaults from the block diagram, including unspecified options. If you specify any optional arguments, your specified settings override the settings in the block diagram.

Specifying an input argument of `sim` as the empty matrix, `[]`, causes Simulink to use the default for that argument.

This command simulates the Van der Pol equations, using the `vdp` model. The command uses all default parameters.

```
[t,x,y] = sim('vdp')
```

This command simulates the Van der Pol equations, using the parameter values associated with the `vdp` model, but defines a value for the `Refine` parameter.

```
[t,x,y] = sim('vdp', [], simset('Refine',2));
```

This command simulates the Van der Pol equations for 1,000 seconds, saving the last 100 rows of the return variables. The simulation outputs values for `t` and `y` only, but saves the final state vector in a variable called `xFinal`.

```
[t,x,y] = sim('vdp', 1000, simset('MaxRows', 100,
    'OutputVariables', 'ty', 'FinalStateName', 'xFinal'));
```

Tips

- Parameters specified using the `sim` command override the values defined in the **Model Configuration Parameters** dialog box. The software restores the original configuration values at the end of simulation.

- In the case of a model with a model reference block, the parameter specifications are applied to the top model.
- When simulating a model with infinite stop time, to stop the simulation, you must press **Ctrl+C**.
- To specify the time span for a simulation, you must specify the `StartTime` and `StopTime` parameters.
- To log the model time, states, or outputs, use the **Data Import/Export** pane of the Model Configuration Parameters dialog box.
- To log signals, either use a block such as the To Workspace block or the Scope block, or use the Signal and Scope Manager to log results directly.
- To get a list of simulation parameters for the model `vdp`, in the MATLAB Command Window, enter:

```
configSet = getActiveConfigSet('vdp')  
configSetNames = get_param(configSet, 'ObjectParameters')
```

This command lists several object parameters, including simulation parameters such as `'StopTime'`, `'SaveTime'`, `'SaveState'`, `'SaveOutput'`, and `'SignalLogging'`.

See Also

Functions

`Simulink.ConfigSet` | `Simulink.SimulationInput` |
`Simulink.SimulationOutput` | `parsim` | `sldebug`

Blocks

`Scope` | `To Workspace`

Topics

“Run Simulations Programmatically”
“Run Multiple Simulations”
“About Configuration Sets”
“Configuration Parameters Dialog Box Overview”
“Log Data to Persistent Storage”
“Model Parameters” on page 6-2

Introduced before R2006a

simplot

Redirects to the Simulation Data Inspector

Note `simplot` will be removed in a future release. Use the Simulation Data Inspector instead.

Syntax

```
simplot
```

Description

`simplot` redirects to the Simulation Data Inspector and returns empty handles. This function is no longer supported and has been replaced by the Simulation Data Inspector. Use the **Simulation Data Inspector** button in the Simulink Editor to capture simulation output in the Simulation Data Inspector. Programmatically, use the function `Simulink.sdi.view` instead.

See Also

`Simulink.sdi.view`

Simulation Manager

Monitor multiple simulations in one window

Description

Simulation Manager allows you to monitor the status of multiple simulations. Using this app, you can:

- View the progress of the simulations in a high-level grid view or a detailed list view.
- Find the errored simulations.
- Abort simulations.
- Select a simulation and view its specifications.
- View simulation results in the **Simulation Data Inspector**.

Open the Simulation Manager App

- `parsim` command – Set the argument `ShowSimulationManager` to `on` with Parallel Computing Toolbox. `Parsim` command uses a `SimulationInput` object to run simulations. For more information, see `Simulink.SimulationInput`.

```
out = parsim(in, 'ShowSimulationManager', 'on')
```

- `sim` command – Set the argument `ShowSimulationManager` to `on`.
- To reopen the Simulation Manager, use the command `openSimulationManager('modelName')`. This command lets you reopen the last running session.

Examples

Open Simulation Manager

Open the model `sldemo_suspn_3dof` and create a set of sweep values.

```
mdl = 'sldemo_suspn_3dof';  
open_system(mdl);  
Cf_sweep = Cf*(0.05:0.1:0.95);  
numSims = length(Cf_sweep);
```

Create an array of `Simulink.SimulationInput` objects to modify the block parameter **Road-Suspension Interaction** with the sweep values.

```
for i = numSims:-1:1  
    in(i) = Simulink.SimulationInput(mdl);  
    in(i) = setBlockParameter(in(i), [mdl '/Road-Suspension Interaction'], 'Cf', num2str(Cf_sweep(i)));  
end
```

Run multiple simulations and open the Simulation Manager to monitor them.

```
out = parsim(in, 'ShowSimulationManager', 'on')
```

In the absence of Parallel Computing Toolbox, the simulations run in serial.

Using Simulation Manager

Once you run the `parsim` command, the Simulation Manager UI opens up as follows:

The screenshot shows the Simulation Manager window with the following components:

- Toolbar:** Stop Job, Open Selected, Grid, List (selected), Simulation Details, Show Results.
- Navigation:** SIMULATIONS, DISPLAY, RESULTS.
- Current Job:** sldemo_suspn_3dof.
- Summary Table:**

Total Simulations	10
Elapsed Time	00:00:08
Number of Active Workers	6
Estimated Time Remaining	00:00:00
- Progress Legend:**
 - Errors/Aborted (0)
 - Completed (10)
 - Active (0)
 - Queued (0)
- Main Table:**

Run ID	Status	Progress	Elapsed Time	Machine
1	Completed	100%	00:00:03	ah-cdeshpan
2	Completed	100%	00:00:03	ah-cdeshpan
3	Completed	100%	00:00:03	ah-cdeshpan
4	Completed	100%	00:00:03	ah-cdeshpan
5	Completed	100%	00:00:03	ah-cdeshpan
6	Completed	100%	00:00:03	ah-cdeshpan
7	Completed	100%	00:00:02	ah-cdeshpan
8	Completed	100%	00:00:02	ah-cdeshpan
9	Completed	100%	00:00:02	ah-cdeshpan
10	Completed	100%	00:00:02	ah-cdeshpan
- Bottom Panel:** SIMULATION DETAILS.

The progress bar on top right is color coded based on the status of the simulations.

You can view all the multiple simulations in a detailed list view. This view gives you an option to add or delete columns to the view. Use the button,  to choose which columns to display. You can also sort the columns based on your preference.

SIMULATION MANAGER ?

Stop Job Open Selected Grid List Simulation Details Show Results

SIMULATIONS DISPLAY RESULTS

sldemo_susp3n_3dof

Total Simulations	10
Elapsed Time	00:02:01
Number of Active Workers	6
Estimated Time Remaining	00:00:00

■ Errors/Aborted (0) ■ Completed (10) ■ Active (0) ■ Queued (0)

Run ID	Status	Progress	Elapsed Time	BlockParam1_Cf
1	Completed	100%	00:00:06	125
2	Completed	100%	00:00:06	375
3	Completed	100%	00:00:06	625
4	Completed	100%	00:00:06	875
5	Completed	100%	00:00:06	1125
6	Completed	100%	00:00:06	1375
7	Completed	100%	00:00:02	1625
8	Completed	100%	00:00:02	1875
9	Completed	100%	00:00:02	2125
10	Completed	100%	00:00:02	2375

SIMULATION DETAILS

Run ID:	1	Parameters	Timing Info	Diagnostics						
Status:	Completed	<table border="1"> <thead> <tr> <th>Type</th> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>Block Parameter</td> <td>Cf</td> <td>125</td> </tr> </tbody> </table>			Type	Name	Value	Block Parameter	Cf	125
Type	Name	Value								
Block Parameter	Cf	125								
Progress:	100									
Elapsed Time:	00:00:06									

You can view more information about a particular run by clicking on it. When you click on a run, simulation details pop at the bottom of the window.

The screenshot shows the Simulation Manager application window. The title bar reads "Simulation Manager". The interface is divided into several sections:

- Toolbar:** Contains buttons for "Stop Job", "Open Selected", "Grid", "List", "Simulation Details", and "Show Results".
- Navigation:** Tabs for "SIMULATIONS", "DISPLAY", and "RESULTS".
- Summary Panel:**
 - Total Simulations: 10
 - Elapsed Time: 00:02:00 (with a green progress bar)
 - Number of Active Workers: 6
 - Estimated Time Remaining: 00:00:00
 - Legend: Errors/Aborted (0) [Red], Completed (10) [Green], Active (0) [Blue], Queued (0) [Grey]
- Simulation List Table:**

Run ID	Status	Progress	Elapsed Time	Machine
1	Completed	100%	00:00:06	ah-cdeshtan
2	Completed	100%	00:00:06	ah-cdeshtan
3	Completed	100%	00:00:06	ah-cdeshtan
4	Completed	100%	00:00:06	ah-cdeshtan
5	Completed	100%	00:00:06	ah-cdeshtan
6	Completed	100%	00:00:06	ah-cdeshtan
7	Completed	100%	00:00:03	ah-cdeshtan
8	Completed	100%	00:00:03	ah-cdeshtan
- Simulation Details Panel:**
 - Run ID: 4
 - Status: Completed
 - Progress: 100
 - Elapsed Time: 00:00:06
 - Parameters, Timing Info, Diagnostics tabs
 - Table:

Type	Name	Value
Block Parameter	Cf	875

To hide the details of the selected run, click the **Simulation Details** button,



Open Selected button, , allows you to open the model with the specifications of the selected run.

You can view the results of one or more runs in the Simulation Data Inspector by clicking the **Show Results** button, . Clicking on Show Results creates a Simulation Data Inspector run from the `Simulin.SimulationOutput` object and is displayed in Simulink Data Inspector. You can also return your results in the `Simulin.SimulationOutput` object.

To stop the job at the beginning of your simulations, you can use the **Stop Job** button, .

See Also

Functions

`ExternalInput` | `PostSimFcn` | `PreSimFcn` | `Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setInitialState` | `setModelParameter` | `setVariable` | `sim` | `validate`

Topics

“Parallel Simulations Workflow”
“Run Multiple Simulations”

simulink

Open Simulink Start Page

Syntax

```
simulink
```

Description

`simulink` opens the Simulink Start Page. Choose a model or project template, or browse the examples. See “Create and Open Models”.

The behavior of the `simulink` function changed in R2016a. Formerly it opened the Simulink Library Browser and loaded the Simulink block library. If you wish to preserve that behavior, use `sLibraryBrowser` instead to open the Library Browser, or `load_system simulink` to load the Simulink block library.

If you want to start Simulink without opening the Library Browser or Start Page, use the faster `start_simulink` instead.

See Also

```
sLibraryBrowser | start_simulink
```

Topics

“Create and Open Models”

“Modeling”

Introduced before R2006a

simulinkproject

Open Simulink Project and get project object

Syntax

```
simulinkproject
simulinkproject(projectPath)

proj = simulinkproject
proj = simulinkproject(projectPath)
```

Description

`simulinkproject` opens Simulink Project or brings focus to the tool if it is already open. After you open the tool, you can create new projects or access recent projects using the **Simulink Project** tab.

`simulinkproject(projectPath)` opens the Simulink project specified by any file or folder under the project root in `projectPath` and gives focus to Simulink Project.

`proj = simulinkproject` returns a project object `proj` you can use to manipulate the project at the command line. You need to get a project object before you can use any of the other project functions.

If you want to avoid giving focus to Simulink Project in your startup script, specify an output argument.

To avoid your startup script opening windows that take focus away from the MATLAB Desktop, use `start_simulink` instead of the `simulink` function, and use `simulinkproject` with an output argument instead of `uiopen`.

`proj = simulinkproject(projectPath)` opens the Simulink project specified by `projectPath` and returns a project object.

Examples

Open Simulink Project Tool

Open the Simulink Project Tool.

```
simulinkproject
```

Open a Simulink Project

Specify either the `.prj` file path or the folder that contains your `.SimulinkProject` folder and `.prj` file. The project opens and brings focus to Simulink Project.

```
simulinkproject('C:/projects/project1/')
```

Open a Simulink Project and Get a Project Object

Open a specified project and get a project object to manipulate the project at the command line. To avoid your startup script opening windows that take focus away from the MATLAB Desktop, use `start_simulink` instead of the `simulink` function, and use `simulinkproject` with an output argument instead of `uiopen`. If you use `uiopen(myproject.prj)` this calls `simulinkproject` with no output argument and gives focus to Simulink Project.

```
start_simulink  
proj = simulinkproject('C:/projects/project1/myproject.prj')
```

Get Airframe Example Project

Open the Airframe project and create a project object to manipulate and explore the project at the command line.

```
sldemo_slproject_airframe  
proj = simulinkproject  
  
proj =
```

ProjectManager with properties:

```
        Name: 'Simulink Project Airframe Example'  
    Information: [1x1 slproject.Information]  
    Dependencies: [1x1 slproject.Dependencies]  
        Categories: [1x1 slproject.Category]  
            Files: [1x31 slproject.ProjectFile]  
        Shortcuts: [1x7 slproject.Shortcut]  
    ProjectPath: [1x7 slproject.PathFolder]  
ProjectReferences: [1x0 slproject.ProjectReference]  
    StartupFiles: [1x0 slproject.StartupFile]  
    ShutdownFiles: [1x0 slproject.ShutdownFile]  
    RootFolder: 'C:\slexamples\airframe11'
```

Find Project Commands

Find out what you can do with your project.

methods(proj)

Methods for class slproject.ProjectManager:

```
addFile  
addFolderIncludingChildFiles  
addPath  
addReference  
addShortcut  
addShutdownFile  
addStartupFile  
close  
createCategory  
export  
findCategory  
findFile  
isLoading  
listModifiedFiles  
listRequiredFiles  
refreshSourceControl  
reload  
removeCategory  
removeFile  
removePath
```

```
removeReference  
removeShortcut  
removeShutdownFile  
RemoveStartupFile
```

Examine Project Properties Programmatically

After you get a project object using the `simulinkproject` function, you can examine project properties.

Examine the project files.

```
files = proj.Files  
  
files =  
  
    1x31 ProjectFile array with properties:  
  
        Path  
        Labels  
        Revision  
        SourceControlStatus
```

Use indexing to access files in this list. The following command gets file number 14. Each file has properties describing its path, attached labels, and source control information.

```
proj.Files(15)  
  
ans =  
  
    ProjectFile with properties:  
  
        Path: 'C:\slexamples\airframe24\models\DigitalControl.slx'  
        Labels: [1x1 slproject.Label]  
        Revision: '2'  
        SourceControlStatus: Unmodified
```

Examine the labels of the file.

```
proj.Files(15).Labels  
  
ans =
```

```

Label with properties:

File: 'C:\slexamples\airframe24\models\DigitalControl.slx'
  DataType: 'none'
  Data: []
  Name: 'Design'
  CategoryName: 'Classification'

```

Get a particular file by name.

```

myfile = findFile(proj, 'models/AnalogControl.mdl')

myfile =

ProjectFile with properties:

Path: 'C:\slexamples\airframe24\models\AnalogControl.mdl'
  Labels: [1x1 slproject.Label]
  Revision: '2'
  SourceControlStatus: Unmodified

```

Find out what you can do with the file.

```

methods(myfile)

Methods for class slproject.ProjectFile:

addLabel      findLabel      removeLabel

```

Update the file dependencies.

```

update(proj.Dependencies)

```

The project runs a dependency analysis to update the known dependencies between project files.

For more information on working with project files, including modified files and dependencies, see “Automate Simulink Project Tasks Using Scripts”.

- “Automate Simulink Project Tasks Using Scripts”
- “Create a New Project From a Folder”
- “Open Recent Projects”
- “Retrieve a Working Copy of a Project from Source Control”

Input Arguments

projectPath — Full path to project file or folder
character vector

Full path to project .prj file, or the path to the project root folder, or any subfolder or file under your project root, specified as a character vector.

Example: 'C:/projects/project1/myProject.prj'

Example: 'C:/projects/project1/'

Output Arguments

proj — Project
project object

Project, returned as a project object. Use the project object to manipulate the currently open Simulink project at the command line.

Properties of `proj` output argument.

Project Property	Description
Categories	Categories of project labels
Dependencies	Dependencies between project files in a MATLAB digraph object
Files	Paths and names of project files
Information	Information about the project such as the description, source control integration, repository location, and whether it is a top-level project
Name	Project name
ProjectPath	Folders that the project puts on the MATLAB path
ProjectReferences	Folders that contain referenced projects. Contains read-only project objects for referenced projects.

Project Property	Description
RootFolder	Full path to project root folder
Shortcuts	An array of the shortcuts in this project
ShutdownFiles	An array of the shutdown files in this project
StartupFiles	An array of the startup files in this project

Tips

Alternatively, you can use `slproject.loadProject` to load a project, and `slproject.getCurrentProjects` to get a project object. Use `simulinkproject` to open projects and explore projects interactively. Use `slproject.getCurrentProjects` and `slproject.loadProject` for project automation scripts.

See Also

Functions

`addFile` | `addFolderIncludingChildFiles` | `addLabel` | `addPath` | `addReference` | `addShortcut` | `createCategory` | `findFile` | `findLabel` | `listModifiedFiles` | `refreshSourceControl` | `removeFile` | `slproject.create` | `slproject.getCurrentProjects` | `slproject.loadProject` | `start_simulink`

Topics

“Automate Simulink Project Tasks Using Scripts”
 “Create a New Project From a Folder”
 “Open Recent Projects”
 “Retrieve a Working Copy of a Project from Source Control”
 “What Are Simulink Projects?”

Introduced in R2012a

Simulink.architecture.add

Add tasks or triggers to selected architecture of model

Syntax

```
Simulink.architecture.add(Type, Object)
```

Description

`Simulink.architecture.add(Type, Object)` adds the new task or trigger `Object` of the specified `Type` to a model.

Examples

Add periodic trigger

Add a task, `MyTask1`, to the software node `MulticoreProcessor` of the selected architecture of the `slexMulticoreExample` model.

```
slexMulticoreExample;  
Simulink.architecture.add('Task', 'slexMulticoreExample/MulticoreProcessor/Core2/MyTask1');
```

Input Arguments

Type — Object type

'PeriodicTrigger' | 'AperiodicTrigger' | 'Task'

Object type that identifies the kind of trigger or task to add, , specified as a 'PeriodicTrigger', 'AperiodicTrigger', or 'Task'.

- 'PeriodicTrigger'

Adds a periodic trigger to the architecture. Set the properties of the trigger with the `Simulink.architecture.set_param` function.

- 'AperiodicTrigger'

Adds an aperiodic trigger to the architecture. Set the properties of the trigger with the `Simulink.architecture.set_param` function.

- 'Task'

Adds a task to the architecture. Set the properties of the task with the `Simulink.architecture.set_param` function.

Object — Trigger or task object identifier

character vector

Trigger or task object identifier to add to architecture, specified as a character vector.

Example: 'slxMulticoreExample/MulticoreProcessor/Core2/MyTask1'

Data Types: char

See Also

`Simulink.architecture.delete` | `Simulink.architecture.find_system` | `Simulink.architecture.get_param` | `Simulink.architecture.importAndSelect` | `Simulink.architecture.profile` | `Simulink.architecture.register` | `Simulink.architecture.set_param`

Introduced in R2014a

Simulink.architecture.config

Create or convert configuration for concurrent execution

Syntax

```
Simulink.architecture.config(model, 'Convert')  
Simulink.architecture.config(model, 'Add')  
Simulink.architecture.config(model, 'OpenDialog')
```

Description

`Simulink.architecture.config(model, 'Convert')` converts the active configuration set in the specified model to one for concurrent execution.

`Simulink.architecture.config(model, 'Add')` adds and activates a new configuration set for concurrent execution.

`Simulink.architecture.config(model, 'OpenDialog')` opens the Concurrent Execution dialog box for a model configuration.

Examples

Convert existing configuration set

Convert existing configuration set for concurrent execution in the model `vdp`.

```
vdp;  
Simulink.architecture.config('vdp', 'Convert');
```

Add new configuration set

Add a new configuration set (copied from the existing configuration set) for concurrent execution in the model `vdp`.

```
vdp;  
Simulink.architecture.config('vdp', 'Add');
```

Open Concurrent Execution dialog box

Open the Concurrent Execution dialog box in the model `slexMulticoreExample`.

```
slexMulticoreExample;  
Simulink.architecture.config('slexMulticoreExample', 'OpenDialog');
```

Input Arguments

model — Model name

character vector

Model name whose configuration set you want to convert or add to, specified as a character vector.

Example:

Data Types: `char`

See Also

`Simulink.architecture.add` | `Simulink.architecture.profile` |
`Simulink.architecture.set_param`

Introduced in R2014a

Simulink.architecture.delete

Delete triggers and tasks from selected architecture of model

Syntax

```
Simulink.architecture.delete(Object)
```

Description

`Simulink.architecture.delete(Object)` deletes the specified object trigger or task.

Examples

Delete task Plant

Delete the task Task3 from the Core2 periodic trigger of the MulticoreProcessor software node of the selected architecture of the `slexMulticoreExample` model.

```
slexMulticoreExample  
Simulink.architecture.delete('slexMulticoreExample/MulticoreProcessor/Core2/Task3')
```

Input Arguments

object — Object to delete, specified as a character vector

character vector

Object to be deleted. Possible objects are:

- Periodic trigger

Note You cannot delete the last periodic trigger. The software node must contain at least one periodic trigger.

- Aperiodic trigger
- Task

Example: [bdroot '/MulticoreProcessor/Core2/Task3']

Data Types: char

See Also

Simulink.architecture.add | Simulink.architecture.find_system |
Simulink.architecture.get_param |
Simulink.architecture.importAndSelect | Simulink.architecture.profile
| Simulink.architecture.register

Introduced in R2014a

Simulink.architecture.find_system

Find objects under architecture object

Syntax

```
object = Simulink.architecture.find_system(RootObject)
```

```
object = Simulink.architecture.find_system(RootObject, ParamName,  
ParamValue)
```

Description

`object = Simulink.architecture.find_system(RootObject)` looks for all objects under `RootObject`.

`object = Simulink.architecture.find_system(RootObject, ParamName, ParamValue)` returns the object in `RootObject` whose parameter `ParamName` has the value `ParamValue`. Parameter name and value character vectors are case-sensitive.

Examples

Look for all objects

To find all the objects in `slexMulticoreExample`:

```
slexMulticoreExample  
t = Simulink.architecture.find_system('slexMulticoreExample')  
  
t =  
  
    'slexMulticoreExample'  
    'slexMulticoreExample/MulticoreProcessor'  
    'slexMulticoreExample/MulticoreProcessor/Core1'  
    'slexMulticoreExample/MulticoreProcessor/Core1/Task1'
```

```
'slexMulticoreExample/MulticoreProcessor/Core1/Task2'
'slexMulticoreExample/MulticoreProcessor/Core2'
'slexMulticoreExample/MulticoreProcessor/Core2/Task3'
'slexMulticoreExample/MulticoreProcessor/Core2/Task4'
```

Look for all tasks

To find all the tasks in `slexMulticoreExample`:

```
slexMulticoreExample
t = Simulink.architecture.find_system('slexMulticoreExample','Type','Task')

t =

    'slexMulticoreExample/MulticoreProcessor/Core1/Task1'
    'slexMulticoreExample/MulticoreProcessor/Core1/Task2'
    'slexMulticoreExample/MulticoreProcessor/Core2/Task3'
    'slexMulticoreExample/MulticoreProcessor/Core2/Task4'
```

Input Arguments

RootObject — Object to search

character vector

Object to search for parameter value, specified as a character vector giving the object full path name. Possible objects are:

- Model
- Software node
- Hardware node
- Periodic trigger
- Aperiodic trigger
- Task

Example: `'slexMulticoreExample'`

ParamName — Name of parameter to find

character vector | scalar | vector

Name of the parameter to find. Possible values are:

- 'Name'
- 'Type'
- 'ClockFrequency'
- 'Color'
- 'Period'
- 'EventHandlerType'
- 'SignalNumber'
- 'EventName'

Example: 'EventName'

ParamValue — Parameter value to find

character vector | scalar | vector

Parameter value to find, specified as a character vector, a scalar, or a vector.

Example: 'ERTDefaultEvent'

See Also

Simulink.architecture.add | Simulink.architecture.delete |
Simulink.architecture.importAndSelect | Simulink.architecture.profile
| Simulink.architecture.register | Simulink.architecture.set_param

Introduced in R2014a

Simulink.architecture.get_param

Get configuration parameters of architecture objects

Syntax

```
ParamValue = Simulink.architecture.get_param(Object, ParamName)
```

Description

`ParamValue = Simulink.architecture.get_param(Object, ParamName)` returns the value of the specified parameter for the object, `Object`. `ParamName` is case-sensitive.

Examples

Get period

Get the period of task `Task3` of trigger `Core2` of software node `MulticoreProcessor` of the selected architecture for the model `slexMulticoreExample`.

```
slexMulticoreExample;  
p = Simulink.architecture.get_param('slexMulticoreExample/MulticoreProcessor/Core2/Task3', 'Period')
```

```
p =
```

```
0.2
```

Input Arguments

Object — Object whose parameter value to return

character vector

Object whose parameter value to return, specified as a character vector giving the object full path name. Possible objects are:

- Software node
- Hardware node
- Periodic trigger
- Aperiodic trigger
- Task

ParamName — Parameter whose value to return

character vector

Name of a parameter for which `Simulink.architecture.get_param` returns a value.

The following are the possible `ParamName` values:

For a model:

- 'ArchitectureName'
- 'Type'

For a software node:

- 'Name'
- 'Type'

For a hardware node

- 'Name'
- 'ClockFrequency'
- 'Color'
- 'Type'

For a periodic trigger:

- 'Name'
- 'Period'
- 'Color'
- 'Type'

For an aperiodic trigger:

- 'Name'
- 'Color'
- 'EventHandlerType'
- 'SignalNumber'
- 'EventName'
- 'Type'

For a task:

- 'Name'
- 'Period'
- 'Color'
- 'Type'

See Also

`Simulink.architecture.add` | `Simulink.architecture.delete` |
`Simulink.architecture.find_system` |
`Simulink.architecture.importAndSelect` | `Simulink.architecture.profile`
| `Simulink.architecture.register` | `Simulink.architecture.set_param`

Introduced in R2014a

Simulink.architecture.importAndSelect

Import and select target architecture for concurrent execution environment for model

Syntax

```
Simulink.architecture.importAndSelect(model, Architecture)
```

```
Simulink.architecture.importAndSelect(model,  
CustomArchitectureDescriptionFile)
```

Description

`Simulink.architecture.importAndSelect(model, Architecture)` imports and selects the built-in target architecture for the concurrent execution environment for the model.

Importing and selecting target architectures requires that the associated support packages or hardware is installed on your computer.

```
Simulink.architecture.importAndSelect(model,  
CustomArchitectureDescriptionFile)
```

 imports and selects the architecture from an XML-based architecture description file.

Importing and selecting target architectures requires that the associated support packages or hardware is installed on your computer.

Examples

Import and select a different architecture

Import and select the sample architecture to the model `slexMulticoreExample`.

```
slexMulticoreExample
Simulink.architecture.importAndSelect('slexMulticoreExample','Sample Architecture')
```

Import and select a custom architecture

Import and select the custom architecture defined in the XML file `custom_arch.xml`. This example requires you to create a `custom_arch.xml` file first.

```
slexMulticoreExample
Simulink.architecture.importAndSelect('slexMulticoreExample','custom_arch.xml')
```

Input Arguments

model — Model

character vector

Model to import architecture to, specified as a character vector.

Data Types: `char`

Architecture — Target architecture name

character vector

Target architecture name to import into the concurrent execution environment for the model. Possible target names are:

Property	Description
'Multicore'	Single CPU with multiple cores
'Sample Architecture'	Example architecture consisting of single CPU with multiple cores and two FPGAs. You can use this architecture to model for concurrent execution.
'Simulink Real-Time'	Simulink Real-Time™ target
'Xilinx Zynq ZC702 evaluation kit'	Xilinx® Zynq® ZC702 evaluation kit target
'Xilinx Zynq ZC706 evaluation kit'	Xilinx Zynq ZC706 evaluation kit target

Property	Description
'Xilinx Zynq Zedboard'	Xilinx Zynq ZedBoard™ target

Data Types: char

CustomArchitectureDescriptionFile — Custom target architecture file

XML file

Custom target architecture file name, in XML format, that describes a custom target for the concurrent execution environment for the model, specified as a character vector giving the XML file name.

Example: `custom_arch.xml`

See Also

`Simulink.architecture.add` | `Simulink.architecture.delete` |
`Simulink.architecture.find_system` | `Simulink.architecture.profile` |
`Simulink.architecture.register` | `Simulink.architecture.set_param`

Topics

“Define a Custom Architecture File”

Introduced in R2014a

Simulink.architecture.profile

Generate profile report for model configured for concurrent execution

Syntax

```
Simulink.architecture.profile(model)  
Simulink.architecture.profile(model,numSamples)
```

Description

`Simulink.architecture.profile(model)` generates a profile report for a model configured for concurrent execution. Subsequent calls to the command for the same model name overwrite the existing profile report.

`Simulink.architecture.profile(model,numSamples)` specifies the number of samples to generate a profile report.

Examples

Generate profile report

Generate profile report for the model `slexMulticoreExample`.

```
slexMulticoreExample;  
Simulink.architecture.profile('slexMulticoreExample');
```

The command creates the file `slexMulticoreExample_ProfileReport.html` in the current folder and opens it.

Generate profile report for 120 time steps

Generate profile report for the model `slexMulticoreExample` with data for 120 time steps.

```
slexMulticoreExample;  
Simulink.architecture.profile('slexMulticoreExample',120);
```

The command creates the file `slexMulticoreExample_ProfileReport.html` in the current folder.

Input Arguments

model — Model to profile

character vector

Model to profile, specified as a character vector. Specify a model that is configured for concurrent execution.

Data Types: `char`

numSamples — Number of time steps

100 (default) | real, positive integer

Number of time steps, specified as a real, positive integer. This value determines the number of steps to collect data for in the profiled model.

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

See Also

`Simulink.architecture.add` | `Simulink.architecture.delete` |
`Simulink.architecture.find_system` | `Simulink.architecture.get_param` |
`Simulink.architecture.importAndSelect` | `Simulink.architecture.register`
| `Simulink.architecture.set_param`

Topics

“Profile and Evaluate on a Desktop”

Introduced in R2014a

Simulink.architecture.register

Add custom target architecture to concurrent execution target architecture selector

Syntax

```
Simulink.architecture.register(CustomArchFile)
```

Description

`Simulink.architecture.register(CustomArchFile)` adds an XML-format custom target architecture file `CustomArchFile` to the concurrent execution target architecture selector. To access this selector, click the **Select** button on the Concurrent Execution pane of the Concurrent Execution dialog box.

Examples

Add custom target architecture

Add custom target architecture defined in the XML file `custom_arch.xml` to the concurrent execution target architecture selector. This example requires you to create a `custom_arch.xml` first.

```
slexMulticoreExample  
Simulink.architecture.register('custom_arch.xml')
```

Input Arguments

CustomArchFile — Custom target architecture file

XML file

Custom target architecture file that describes a custom target for concurrent execution, specified as an XML file.

See Also

`Simulink.architecture.add` | `Simulink.architecture.delete` |
`Simulink.architecture.find_system` |
`Simulink.architecture.importAndSelect` | `Simulink.architecture.profile`
| `Simulink.architecture.set_param`

Introduced in R2014a

Simulink.architecture.set_param

Set architecture object properties

Syntax

```
Simulink.architecture.set_param(Object, ParamName, ParamValue)
```

Description

`Simulink.architecture.set_param(Object, ParamName, ParamValue)` sets the specified parameter of `Object` to the specified value. Parameter name and value character vectors are case sensitive.

Examples

Set software node name

Set the software node name from `MulticoreProcessor` to `MyCPUNewName`.

```
slexMulticoreExample
Simulink.architecture.set_param([bdroot '/MulticoreProcessor'], 'Name', 'MyCPUNewName');
```

Change Periodic

Set `Core2` trigger period to `.01`.

```
slexMulticoreExample  
Simulink.architecture.set_param([bdroot '/MyCPUNewName/Core2'],'Period','.01')
```

Input Arguments

Object — Object whose parameter value to set

character vector

Object whose parameter value to set, specified as a character vector giving the object full path name. Possible objects are:

- Software node
- Hardware node
- Periodic trigger
- Aperiodic trigger
- Task

ParamName — Name of the parameter to set

character vector

Name of parameter whose value to set.

These are the possible parameters whose values you can set for each of the object types:

For software node:

- 'Name' — Name of the software node.

For hardware node:

- 'Name' — Name of the hardware node.
- 'ClockFrequency' — Frequency of the hardware node clock.
- 'Color' — Color of the trigger icon, specified as an RGB triplet (vector).

For a periodic trigger:

- 'Name' — Name of the trigger.
- 'Period' — Period of the trigger.

- `'Color'` — Color of the trigger icon, specified as an RGB triplet (vector).

For an aperiodic trigger:

- `'Name'` — Name of the trigger.
- `'Color'` — Color of the trigger icon, specified as an RGB triplet (vector).
- `'EventHandlerType'` — Trigger source for the interrupt-driven task. Possible values:
 - `'Event (Windows)'`
 - `'Posix Signal (Linux/VxWorks 6.x)'`
- `'SignalNumber'` — Signal number for the trigger. You can set this value only if `EventHandlerType` is set to `Event (Windows)`.
- `'EventName'` — Event name for the trigger. You can set this value only if `EventHandlerType` is set to `Posix Signal (Linux/VxWorks 6.x)`.

For task:

- `'Name'` — Name of the task.
- `'Period'` — Period of the task.
- `'Color'` — Color of the task icon, specified as an RGB triplet (vector).

Data Types: `char`

ParamValue — Value to set the parameter to

character vector | vector

Value to set the parameter to, specified as a character vector, scalar, or vector. The possible values depend on the parameter.

Example: `'MyCPUNewName'`

See Also

`Simulink.architecture.add` | `Simulink.architecture.delete` |
`Simulink.architecture.find_system` | `Simulink.architecture.get_param` |
`Simulink.architecture.importAndSelect` | `Simulink.architecture.profile`
| `Simulink.architecture.register`

Introduced in R2014a

Simulink.Block.getSampleTimes

Return sample time information for a block

Syntax

```
ts = Simulink.Block.getSampleTimes(block)
```

Input Arguments

block

Full name or handle of a Simulink block

Output Arguments

ts

The command returns *ts* which is a 1×*n* array of Simulink.SampleTime objects associated with the model passed to Simulink.Block.getSampleTimes. Here *n* is the number of sample times associated with the block. The format of the returns is:

```
1xn Simulink.SampleTime
Package: Simulink
value: [1x2 double]
Description: [char string]
ColorRGBValue: [1x3 double]
Annotation: [char string]
OwnerBlock: [char string]
ComponentSampleTimes: [1x2 struct]
Methods
```

- *value* — A two-element array of doubles that contains the sample time period and offset
- *Description* — A character vector that describes the sample time type
- *ColorRGBValue* — A 1x3 array of doubles that contains the red, green and blue (RGB) values of the sample time color

- `Annotation` — A character vector that represents the annotation of a specific sample time (e.g., 'D1')
- `OwnerBlock` — For asynchronous and variable sample times, a character vector containing the full path to the block that controls the sample time. For all other types of sample times, an empty character vector.
- `ComponentSampleTimes` — A structure array of elements of the same type as `Simulink.BlockDiagram.getSampleTimes` if the sample time is an async union or if the sample time is hybrid and the component sample times are available.

Description

`ts = Simulink.Block.getSampleTimes(block)` performs an update diagram and then returns the sample times of the block connected to the input argument `mdl/signal`. This method performs an update diagram to ensure that the sample time information returned is up-to-date. If the model is already in the compiled state via a call to the model API, then an update diagram is not necessary.

Using this method allows you to access all information in the Sample Time Legend programmatically.

See Also

`Simulink.BlockDiagram.getSampleTimes`

Introduced in R2009a

Simulink.BlockDiagram.addBusToVector

Convert virtual bus signals into vector signals by adding Bus to Vector blocks

Syntax

```
[destBlocks,busToVectorBlocks,ignoredBlocks] =  
Simulink.BlockDiagram.addBusToVector(model)  
[destBlocks,busToVectorBlocks,ignoredBlocks] =  
Simulink.BlockDiagram.addBusToVector(model,includeLibs)  
[destBlocks,busToVectorBlocks,ignoredBlocks] =  
Simulink.BlockDiagram.addBusToVector(model,includeLibs,reportOnly)  
[destBlocks,busToVectorBlocks,ignoredBlocks] =  
Simulink.BlockDiagram.addBusToVector(model,includeLibs,reportOnly,  
strictOnly)
```

Description

[destBlocks,busToVectorBlocks,ignoredBlocks] =
Simulink.BlockDiagram.addBusToVector(model) searches a model, excluding any library blocks, for bus signals used implicitly as vectors, and returns the results of the search.

[destBlocks,busToVectorBlocks,ignoredBlocks] =
Simulink.BlockDiagram.addBusToVector(model,includeLibs) searches a model, and if includeLibs is true, includes in the search library blocks for bus signals used implicitly as vectors.

[destBlocks,busToVectorBlocks,ignoredBlocks] =
Simulink.BlockDiagram.addBusToVector(model,includeLibs,reportOnly) searches a model, and if reportOnly is set to false, then the function inserts a Bus to Vector block into each bus that is used as a vector in any block that it searches. The insertion replaces the implicit use of a bus as a vector with an explicit conversion of the bus to a vector. The source and destination blocks of the signal do not change.

If Simulink.BlockDiagram.addBusToVector adds Bus to Vector blocks to the model or any library, the function changes the saved copy of the diagram.

If `Simulink.BlockDiagram.addBusToVector` changes a library block, the change affects every instance of that block in every model that uses the library.

`[destBlocks, busToVectorBlocks, ignoredBlocks] = Simulink.BlockDiagram.addBusToVector(model, includeLibs, reportOnly, strictOnly)` searches a model, and if `strictOnly` is true, the function checks for input bus signals used implicitly as vectors that are fed into one of these blocks. These blocks cannot take virtual bus signals, but they can accept nonvirtual bus signals.

- Delay
- Selector
- Assignment
- Vector Concatenate
- Reshape
- Permut Dimensions

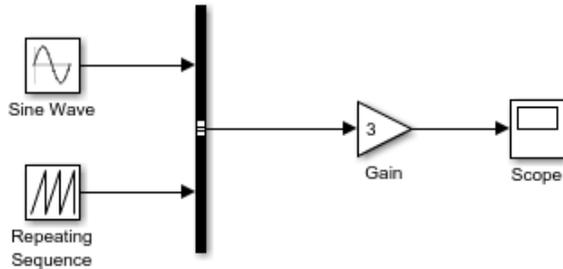
Examples

Insert Bus to Vector Block to Handle Bus Used as Vector

The `ex_bus_to_mux_ng` model simulates correctly, but the input to the Gain block is a bus, while the output is a vector. Therefore, the Gain block uses a bus signal as a vector.

Open the model.

```
open_system(fullfile(matlabroot, 'examples', 'simulink', ...  
'ex_bus_to_mux_ng'))
```



Insert a Bus to Vector block to convert the bus input signal for the Gain block to a vector signal because the Gain block can accept only non-bus signals.

```
[blocks,busToVectors] = Simulink.BlockDiagram.addBusToVector(...
'ex_bus_to_mux_ng',true,false)

### Processing block diagram 'ex_bus_to_mux_ng'
### Number of blocks left that are connected to a bus being used as a vector: 1
### Successfully inserted Bus to Vector Blocks in model. Preparing to save model and/or libraries
### To eliminate modeling errors in the future, please enable strict bus modeling by setting the 'Bus signal
### Done processing block diagram 'ex_bus_to_mux_ng'

blocks =

    struct with fields:

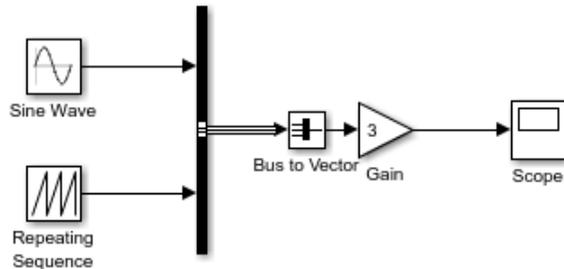
        BlockPath: 'ex_bus_to_mux_ng/Gain'
        InputPort: 1
        LibPath: ''

busToVectors =

    cell

    'ex_bus_to_mux_ng/Bus to Vector'
```

The Gain block no longer implicitly converts the bus to a vector. The inserted Bus to Vector block performs the conversion explicitly. The results of simulation are the same for both models. The Bus to Vector block is virtual and does not affect simulation results, code generation, or performance.



Input Arguments

model — Model name or handle

character vector

Model name or handle, specified as a character vector.

includeLibs — Search library blocks

false (default) | true

Search library blocks, specified as false or true.

- false — Search only the blocks in the model.
- true — Search library blocks for bus signals used implicitly as vectors.

Specify as the second argument.

Data Types: logical

reportOnly — Report results without changing model

true (default) | false

Choice to report results without changing the model, specified as false or true.

- false — Update the model by inserting Bus to Vector blocks for bus signals that are implicitly used as vectors.
- true — Report search results, but do not change the model.

Specify as the third argument (you must also specify the `model` and `includeLibs` arguments).

Data Types: `logical`

`strictOnly` — Check input bus signals used implicitly as vectors that feed blocks that can accept nonvirtual, but not virtual, bus signals

`false` (default) | `true`

Check input bus signals used implicitly as vectors that feed blocks that can accept nonvirtual, but not virtual, bus signals, specified as `false` or `true`. If `strictOnly` is `true`, the function checks for input bus signals used implicitly as vectors that are fed into one of these blocks. These blocks cannot take virtual bus signals, but they can accept nonvirtual bus signals.

- Delay
- Selector
- Assignment
- Vector Concatenate
- Reshape
- Permut Dimensions

Specify as the fourth argument (you must also specify the `model` and `includeLibs` arguments).

Data Types: `logical`

Output Arguments

`destBlocks` — Blocks connected to buses but that treat buses as vectors

array of structures

Blocks connected to buses that treat buses as vectors, returned as an array of structures. Each structure in the array contains these fields:

- `BlockPath` — Character vector specifying the path to the block to which the bus connects.
- `InputPort` — Integer specifying the input port to which the bus connects.

- `LibPath` — If the block is a library block instance, and `includeLibs` is `true`, the field value is the path to the source library block. Otherwise, `LibPath` is empty (`[]`).

busToVectorBlocks — Bus to Vector blocks added by function

cell array

Bus to Vector blocks added by the function, specified as a cell array. If `reportOnly` is set to `false`, the cell array contains the paths to each Bus to Vector block that the function added to replace buses used as vectors. Otherwise, `busToVectorBlocks` is empty (`[]`).

ignoredBlocks — Cases where function cannot insert Bus to Vector block

array of structures

Cases where function cannot insert Bus to Vector block, specified as an array of structures. Each structure in the array contains these fields:

- `BlockPath` — Character vector specifying the path to the block to which the bus connects.
- `InputPort` — Integer specifying the input port to which the bus connects.

These cases occur when a Bus to Vector cannot be inserted because the input virtual bus signal consists of elements with mixed attributes.

Tips

- Before you execute this function:
 - 1 Ensure that the model compiles without error.
 - 2 Save the model.
- Back up the model and any libraries before calling the function with `reportOnly` set to `false`.
- To preview the effects of the change on blocks in all models, call `Simulink.BlockDiagram.addBusToVector` with `includeLibs` set to `true` and `reportOnly` set to `true`. Then, examine the information returned in the `destBlocks` output argument.

See Also

Bus to Vector

Introduced in R2007a

Simulink.BlockDiagram.buildRapidAcceleratorTarget

Build Rapid Accelerator target for model and return run-time parameter set

Syntax

```
rtp = Simulink.BlockDiagram.buildRapidAcceleratorTarget (mdl)
```

Description

`rtp = Simulink.BlockDiagram.buildRapidAcceleratorTarget (mdl)` builds a Rapid Accelerator target for model, `mdl`, and returns run-time parameter set, `rtp`.

Input Arguments

`mdl`

Name or handle of a Simulink model

Output Arguments

`rtp`

Run-time parameter set that contains two elements:

Element	Description				
<code>modelChecksum</code>	1x4 vector that encodes the structure of the model.				
<code>parameters</code>	A structure of the tunable parameters in the model. This structure contains the following fields.				
	<table border="1"> <thead> <tr> <th>Field</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td></td> <td></td> </tr> </tbody> </table>	Field	Description		
Field	Description				

Element	Description		
	dataTypeName	The data type name, for example, double.	
	dataTypeId	Internal data type identifier for use by Simulink Coder.	
	complex	Complex type or real type specification. Value is 0 if real, 1 if complex.	
	dtTransIdx	Internal data type identifier for use by Simulink Coder.	
	values	All values associated with this entry in the <code>parameters</code> substructure.	
	map	Mapping structure information that correlates the values to the model tunable parameters. This structure contains the following fields.	
		Field	Description
		Identifier	Tunable parameter name.
		ValueIndices	Start and end indices into the values field, [<code>startIdx</code> , <code>endIdx</code>].
	Dimensions	Dimension of this tunable parameter (matrices are generally stored in column-major format).	

Examples

Build Rapid Accelerator Target for Model

In the MATLAB Command Window, type:

```
rtp = Simulink.BlockDiagram.buildRapidAcceleratorTarget('f14')

### Building the rapid accelerator target for model: f14
### Successfully built the rapid accelerator target for model: f14

rtp =
```

```
modelChecksum: [2.6812e+09 2.7198e+09 589261472 4.0180e+09]  
parameters: [1x1 struct]
```

See Also

Topics

“How Acceleration Modes Work”

“Choosing a Simulation Mode”

“Design Your Model for Effective Acceleration”

Introduced in R2012b

Simulink.BlockDiagram.copyContentsToSubsystem

Copy contents of block diagram to empty subsystem

Syntax

```
Simulink.BlockDiagram.copyContentsToSubsystem(bdiag, subsys)
```

Description

`Simulink.BlockDiagram.copyContentsToSubsystem(bdiag, subsys)` copies the contents of the block diagram *bdiag* to the subsystem *subsys*. The block diagram and subsystem must have already been loaded. The subsystem cannot be part of the block diagram.

The function affects only blocks, lines, and annotations; it does not affect nongraphical information such as configuration sets. You can use this function to convert a referenced model derived from an atomic subsystem into an atomic subsystem that is equivalent to the original subsystem.

This function cannot be used if the destination subsystem contains any blocks or signals. Other types of information can exist in the destination subsystem and are not affected by the function. Use `Simulink.SubSystem.deleteContents` if necessary to empty the subsystem before using `Simulink.BlockDiagram.copyContentsToSubsystem`.

Input Arguments

bdiag

Block diagram name or handle

subsys

Subsystem name or handle

Examples

Copy the contents of `vdp` to an empty subsystem named `vdp_subsystem` that is in the model named `new_model_with_vdp`:

```
open_system('vdp');
new_system('new_model_with_vdp')
open_system('new_model_with_vdp');
add_block('built-in/Subsystem', 'new_model_with_vdp/vdp_subsystem')
Simulink.BlockDiagram.copyContentsToSubsystem...
('vdp', 'new_model_with_vdp/vdp_subsystem')
```

See Also

[Simulink.BlockDiagram.deleteContents](#) |
[Simulink.SubSystem.convertToModelReference](#) |
[Simulink.SubSystem.copyContentsToBlockDiagram](#) |
[Simulink.SubSystem.deleteContents](#)

Topics

“Systems and Subsystems”
“Create a Subsystem”

Introduced in R2007a

Simulink.BlockDiagram.createSubsystem

Create subsystem containing specified set of blocks

Syntax

```
Simulink.BlockDiagram.createSubsystem(blocks)  
Simulink.BlockDiagram.createSubsystem()
```

Description

`Simulink.BlockDiagram.createSubsystem(blocks)` creates a new subsystem and moves the specified blocks into the subsystem. All of the specified blocks must originally reside at the top level of the model or in the same existing subsystem within the model.

If any of the blocks have unconnected input ports, the command creates input port blocks for each unconnected input port in the subsystem and connects the input port block to the unconnected input port. The command similarly creates and connects output port blocks for unconnected output ports on the specified blocks. If any of the specified blocks is an input port, the command creates an input port block in the parent system and connects it to the corresponding input port of the newly created subsystem. The command similarly creates and connects output port blocks for each of the specified blocks that is an output port block.

`Simulink.BlockDiagram.createSubsystem()` creates a new subsystem in the currently selected model and moves the currently selected blocks in the current model to the new subsystem.

Input Arguments

blocks

An array of block handles

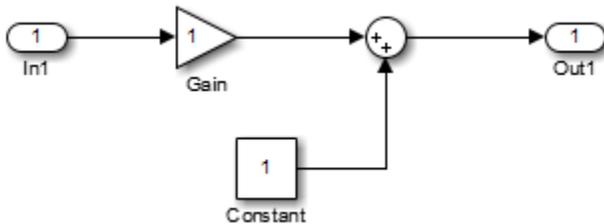
Default: []

Examples

This function converts the contents of a model or subsystem into a subsystem.

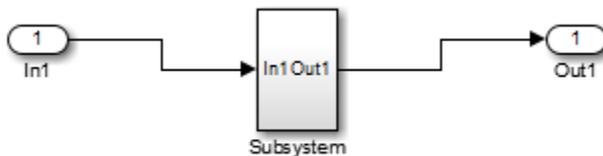
```
function convert2subsys(sys)
    blocks = find_system(sys, 'SearchDepth', 1);
    bh = [];
    for i = 2:length(blocks)
        bh = [bh get_param(blocks{i}, 'handle')];
    end
    Simulink.BlockDiagram.createSubsystem(bh);
end
```

For example, suppose you create this model and save it as `initial_model.slx`.



Execute this code to convert the model to create a subsystem:

```
convert2subsys('initial_model');
```



See Also

[Simulink.BlockDiagram.copyContentsToSubsystem](#) |
[Simulink.BlockDiagram.deleteContents](#) |

`Simulink.SubSystem.convertToModelReference |`
`Simulink.SubSystem.copyContentsToBlockDiagram`

Topics

“Systems and Subsystems”

“Create a Subsystem”

Introduced in R2009a

Simulink.BlockDiagram.deleteContents

Delete contents of block diagram

Syntax

```
Simulink.BlockDiagram.deleteContents(bdiag)
```

Description

`Simulink.BlockDiagram.deleteContents(bdiag)` deletes the contents of the block diagram *bdiag*. The function affects only blocks, lines, and annotations. The block diagram must have already been loaded.

Input Arguments

bdiag

Block diagram name or handle

Examples

Delete the graphical content of an open block diagram named `f14`, including all subsystems:

```
Simulink.BlockDiagram.deleteContents('f14');
```

See Also

```
Simulink.BlockDiagram.copyContentsToSubsystem |  
Simulink.SubSystem.convertToModelReference |  
Simulink.SubSystem.copyContentsToBlockDiagram |  
Simulink.SubSystem.deleteContents
```

Topics

“Modeling”

“Create a Subsystem”

Introduced in R2007a

Simulink.BlockDiagram.expandSubsystem

Expand subsystem contents to containing model level

Syntax

```
Simulink.BlockDiagram.expandSubsystem(block)
```

Description

`Simulink.BlockDiagram.expandSubsystem(block)` expands the contents of the subsystem for the specified Subsystem block. Subsystem expansion involves moving the contents of a virtual subsystem into the system that contains that subsystem.

You can expand virtual subsystems that are not masked, linked, or commented. For details, see “Subsystems That You Can Expand”.

Input Arguments

block

A character vector that specifies one of the following:

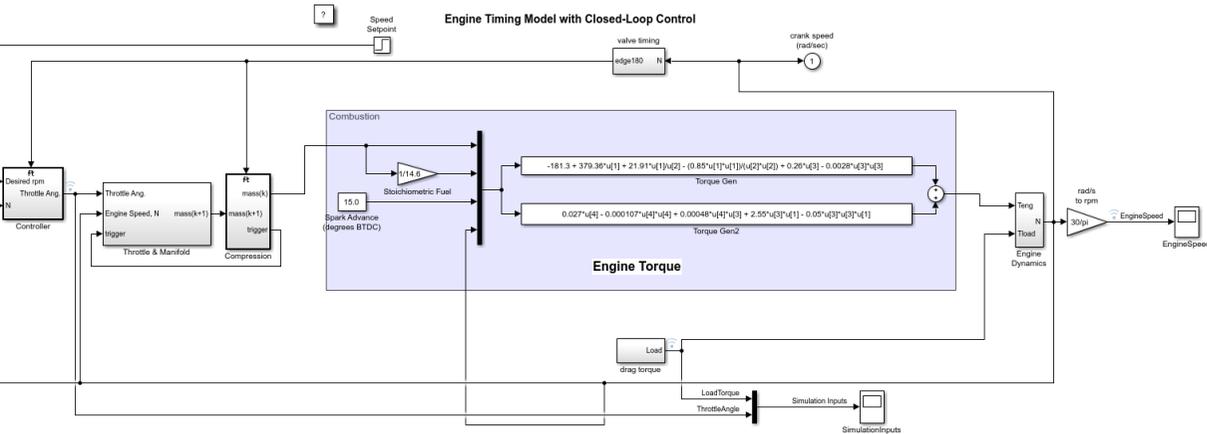
- The path to a subsystem block in a loaded model.
- The block handle of a subsystem block in a loaded model.
- `gcb` (the currently selected block)

Examples

The following function expands the Combustion subsystem.

```
open_system('sldemo_enginewc')  
Simulink.BlockDiagram.expandSubsystem('sldemo_enginewc/Combustion')
```

The blocks and signals that were in the Combustion subsystem become part of the top-level model that contained the Combustion subsystem, replacing that Subsystem block.



See Also

[Simulink.BlockDiagram.copyContentsToSubsystem](#) |
[Simulink.BlockDiagram.createSubsystem](#) |
[Simulink.BlockDiagram.deleteContents](#) |
[Simulink.SubSystem.convertToModelReference](#) |
[Simulink.SubSystem.copyContentsToBlockDiagram](#)

Topics

[“Expand Subsystem Contents”](#)
[“Systems and Subsystems”](#)

Introduced in R2014a

Simulink.BlockDiagram.getChecksum

Return checksum of model

Syntax

```
[checksum, details] = Simulink.BlockDiagram.getChecksum('model')
```

Description

[*checksum*, *details*] = Simulink.BlockDiagram.getChecksum('model') returns the checksum of the specified model. Simulink software computes the checksum based on attributes of the model and the blocks the model contains.

One use of this command is to determine why the Accelerator mode in Simulink software regenerates code. For an example, see `slAccelDemoWhyRebuild`.

Note Simulink.BlockDiagram.getChecksum compiles the specified model, if the model is not already in a compiled state.

This command accepts the argument *model*, which is the full name or handle of the model for which you are returning checksum data.

This command returns the following output:

- *checksum* — Array of four 32-bit integers that represents the model's 128-bit checksum.
- *details* — Structure of the form

```
ContentsChecksum: [1x1 struct]
InterfaceChecksum: [1x1 struct]
ContentsChecksumItems: [nx1 struct]
InterfaceChecksumItems: [mx1 struct]
```

- ContentsChecksum — Structure of the following form that represents a checksum that provides information about all blocks in the model.

Value: [4x1 uint32]
MarkedUnique: [bool]

- Value — Array of four 32-bit integers that represents the model's 128-bit checksum.
- MarkedUnique — True if any blocks in the model have a property that prevents code reuse.
- InterfaceChecksum — Structure of the following form that represents a checksum that provides information about the model.

Value: [4x1 uint32]
MarkedUnique: [bool]

- Value — Array of four 32-bit integers that represents the model's 128-bit checksum.
- MarkedUnique — Always true. Present for consistency with ContentsChecksum structure.
- ContentsChecksumItems and InterfaceChecksumItems — Structure arrays of the following form that contain information that Simulink software uses to compute the checksum for ContentsChecksum and InterfaceChecksum, respectively:

Handle: [char array]
Identifier: [char array]
Value: [type]

- Handle — Object for which Simulink software added an item to the checksum. For a block, the handle is a full block path. For a block port, the handle is the full block path and a character vector that identifies the port.
- Identifier — Descriptor of the item Simulink software added to the checksum. If the item is a documented parameter, the identifier is the parameter name.
- Value — Value of the item Simulink software added to the checksum. If the item is a parameter, Value is the value returned by

`get_param(handle, identifier)`

`Simulink.BlockDiagram.getChecksum` returns a checksum that depends on why and how you compiled the model. This function also compiles the model if it is not in a compiled state. The model compiles for:

- Simulation— if the simulation mode is Accelerator or you have not installed Simulink Coder
- Code generation— in all other cases

To compile the model before calling `Simulink.BlockDiagram.getChecksum`, use this command:

```
modelName([], [], [], 'compile')
```

Note The checksum that `Simulink.BlockDiagram.getChecksum` returns can vary from the checksum returned if you first compile the model at the command line (using the `model` command) before running `Simulink.BlockDiagram.getChecksum`.

Tip

The structural checksum reflects changes to the model that can affect the simulation results, including:

- Changing the solver type, for example from Variable-step to Fixed-step
- Adding or deleting blocks or connections between blocks
- Changing the values of nontunable block parameters, for example, the **Seed** parameter of the Random Number block
- Changing the number of inputs or outputs of blocks, even if the connectivity is vectorized
- Changing the number of states or the initial states in the model
- Selecting a different function in the Trigonometric Function block
- Changing signs used in a Sum block
- Adding a Target Language Compiler (TLC) file to inline an S-function

Examples of model changes that do not affect the structural checksum include:

- Changing the position of a block
- Changing the position of a line
- Resizing a block
- Adding, removing, or changing a model annotation

See Also

`Simulink.SubSystem.getChecksum` | `Simulink.getFileChecksum`

Introduced in R2006b

Simulink.BlockDiagram.getInitialState

Return initial state data of block diagram

Syntax

```
x0 = Simulink.BlockDiagram.getInitialState('model')
```

Description

`x0 = Simulink.BlockDiagram.getInitialState('model')` returns the initial state data of the block diagram specified by the input argument *model*. You can use this initial state data as the initial state for simulating a model or to provide an initial state condition to the linearization commands. To specify the initial state for a simulation, use the `LoadInitialState` model argument or the **Data Import/Export > Initial state** configuration parameter.

To specify the format for the initial state data, use the `SaveFormat` model argument. The default format is 'Dataset'. Other formats 'Array', 'Structure', and 'StructureWithTime'. Alternatively, you can set the initial state format using the **Data Import/Export > Format** configuration parameter.

- If format is 'Dataset', then the `Simulink.BlockDiagram.getInitialState` function returns a `Simulink.SimulationData.Dataset` object.
- For other format settings, the function returns a structure of the form:

```
time: 0  
signals: [1xn struct]
```

where *n* is the number of states contained in the model, including any models referenced by Model blocks. The `signals` field is a structure of the form:

```
values: [1xm double]  
dimensions: [1x1 double]  
label: [char array]  
blockName: [char array]
```

```
inReferencedModel: [bool]  
sampleTime: [1x2 double]
```

- `values` — Numeric array of length `m`, where `m` is the number of states in the signal
- `dimensions` — Length of the values vector
- `label` — Indication of whether the state is continuous (CSTATE) or discrete. If the state is discrete:

The name of the discrete state is shown for S-function blocks.

The name of the discrete state is shown for those built-in blocks that assign their own names to discrete states.

DSTATE is used in all other cases.

- `blockName` — Full path to block associated with this state
- `inReferencedModel` — Indication of whether the state originates in a model referenced by a Model block (1) or in the top model (0)
- `sampleTime` — Array containing the sample time and offset of the block that owns the state

Using this function to return the initial state data simplifies specifying initial state values for models with multiple states. Each state is associated with the full path to its parent block.

See Also

`linmod`

Topics

“Initial state”
“Format”

Introduced in R2006b

Simulink.BlockDiagram.getSampleTimes

Return all sample times associated with model

Syntax

```
ts = Simulink.BlockDiagram.getSampleTimes('model')
```

Description

`ts = Simulink.BlockDiagram.getSampleTimes('model')` performs an update diagram and then returns the sample times associated with the block diagram `model`. The update diagram ensures that the sample time information returned is up to date. If the model is already in the compiled state via a call to the model API, then an update diagram is not necessary.

Using this method allows you to access all information in the Sample Time Legend programmatically.

Input Arguments

model

Name or handle of a Simulink model

Output Arguments

`ts`

The command returns a `1xn` array of `Simulink.SampleTime` objects associated with the model passed to `Simulink.BlockDiagram.getSampleTimes`. Here `n` is the number of sample times associated with the block diagram. The format of the returns is as follows:

```
1xn Simulink.SampleTime  
Package: Simulink
```

```
value: [1x2 double]
Description: [char string]
ColorRGBValue: [1x3 double]
Annotation: [char string]
OwnerBlock: [char string]
ComponentSampleTimes: [1x2 struct]
Methods
```

- `value` — A two-element array of doubles that contains the sample time period and offset.
- `Description` — A character vector that describes the sample time type.
- `ColorRGBValue` — A 1x3 array of doubles that contains the red, green, and blue (RGB) values of the sample time color.
- `Annotation` — A character vector that represents the annotation of a specific sample time (e.g., 'D1').
- `OwnerBlock` — For asynchronous and variable sample times, a character vector containing the full path to the block that controls the sample time. For all other types of sample times, an empty character vector.
- `ComponentSampleTimes` — A structure array of elements of the same type as `Simulink.BlockDiagram.getSampleTimes` if the sample time is an async union or if the sample time is hybrid and the component sample times are available.

See Also

`Simulink.Block.getSampleTimes`

Introduced in R2009a

Simulink.BlockDiagram.loadActiveConfigSet

Package: Simulink.BlockDiagram

Load, associate, and activate configuration set with model

Syntax

```
Simulink.BlockDiagram.loadActiveConfigSet(model, filename)
```

Description

`Simulink.BlockDiagram.loadActiveConfigSet(model, filename)` loads a configuration set, associates it with a model, and makes it the active configuration set. `model` is the name or handle of a model. `filename` is the name of the file (.m or .mat) that creates or contains a configuration set object to load. If you do not provide a file extension, it defaults to .m. If the file name is the same as a model name on the MATLAB path, the software cannot determine which file contains the configuration set object and displays an error message.

Examples

Save the configuration set from the `sldemo_counters` model to `my_config_set.m`.

```
% Open the sldemo_counters model
sldemo_counters
% Save the active configuration set to my_config_set.m
Simulink.BlockDiagram.saveActiveConfigSet('sldemo_counters', 'my_config_set.m')
```

Load the configuration set from `my_config_set.m`, associate it with the `vdp` model, and make it the active configuration set.

```
% Open the vdp model
vdp
% Load the configuration set from my_config_set.m, making it the active
% configuration set for vdp.
Simulink.BlockDiagram.loadActiveConfigSet('vdp', 'my_config_set.m')
```

Tips

- If you load a configuration set with the same name as the active configuration set, the software overwrites the active configuration set.
- If you load a configuration set with the same name as an inactive configuration set associated with the model, the software detaches the inactive configuration from the model.
- If you load a configuration set object that contains an invalid custom target, the software sets the “**System target file**” (**Simulink Coder**) parameter to `ert.tlc`.

See Also

`Simulink.BlockDiagram.saveActiveConfigSet` | `Simulink.ConfigSet` | `attachConfigSet` | `attachConfigSetCopy` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `setActiveConfigSet`

Topics

“Load a Saved Configuration Set”

Introduced in R2010b

Simulink.BlockDiagram.propagateConfigSet

Propagate top model configuration reference to referenced models

Syntax

```
[isPropagated, convertedModels] =  
Simulink.BlockDiagram.propagateConfigSet(model)  
[isPropagated, convertedModels] =  
Simulink.BlockDiagram.propagateConfigSet(model, 'include',  
refModels)  
[isPropagated, convertedModels] =  
Simulink.BlockDiagram.propagateConfigSet(model, 'exclude',  
refModels)  
handle = Simulink.BlockDiagram.propagateConfigSet(model, 'gui')
```

Description

`[isPropagated, convertedModels] = Simulink.BlockDiagram.propagateConfigSet(model)` propagates the configuration reference for `model` to all referenced models. Execute the function from a writable folder.

`[isPropagated, convertedModels] = Simulink.BlockDiagram.propagateConfigSet(model, 'include', refModels)` propagates the configuration reference for `model` to the models in the `refModels` list. Execute the function from a writable folder.

`[isPropagated, convertedModels] = Simulink.BlockDiagram.propagateConfigSet(model, 'exclude', refModels)` propagates the configuration reference for `model` to all referenced models in the hierarchy except for the models in the `refModels` list. Execute the function from a writable folder.

`handle = Simulink.BlockDiagram.propagateConfigSet(model, 'gui')` opens the **Configuration Reference Propagation to Referenced Models** dialog box.

Examples

Propagate a Configuration Reference to All Referenced Models

```
[isPropagated,convertedModels] = ...  
Simulink.BlockDiagram.propagateConfigSet('sldemo_mdhref_depgraph')
```

Propagate a Configuration Reference to Listed Referenced Models

```
[isPropagated,convertedModels] = ...  
Simulink.BlockDiagram.propagateConfigSet(...  
'sldemo_mdhref_depgraph','include',...  
{'sldemo_mdhref_heater','sldemo_mdhref_house'})
```

Propagate a Configuration Reference to Referenced Models with Exclusions

```
[isPropagated,convertedModels] = ...  
Simulink.BlockDiagram.propagateConfigSet(...  
'sldemo_mdhref_depgraph','exclude',...  
{'sldemo_mdhref_heater','sldemo_mdhref_house'})
```

Open the Configuration Reference Propagation to Referenced Models Dialog Box for a Model

```
Simulink.BlockDiagram.propagateConfigSet(...  
'sldemo_mdhref_depgraph','gui')
```

- “Share a Configuration Across Referenced Models”
- “Manage a Configuration Reference”

Input Arguments

model — Top model

character vector

Top model with configuration reference to propagate, specified as a character vector.

Example: 'mdl'

refModels — Referenced models

cell array of character vectors

List of referenced models to be included or excluded in propagation, specified as a cell array of character vectors.

Example: {'mdl1','mdl2','mdl3'}

Output Arguments

isPropagated — Success of propagation

false (default) | true

Indication of whether configuration reference propagation is successful, specified as a Boolean.

convertedModels — Converted models

cell array of character vectors

List of converted model names, specified as a cell array of character vectors.

handle — Handle to dialog box

handle

Handle to the **Configuration Reference Propagation to Referenced Models** dialog box. Returned when you specify the 'gui' argument to the function.

See Also

Simulink.BlockDiagram.restoreConfigSet

Topics

“Share a Configuration Across Referenced Models”

“Manage a Configuration Reference”

Introduced in R2012b

Simulink.BlockDiagram.restoreConfigSet

Restore model configuration for converted models

Syntax

```
[isRestored, restoredModels] =  
Simulink.BlockDiagram.restoreConfigSet(model)
```

Description

```
[isRestored, restoredModels] =  
Simulink.BlockDiagram.restoreConfigSet(model) restores the model  
configuration for all converted models after propagating a configuration reference from a  
top model to the referenced models. Execute the function from a writable folder.
```

Examples

Restore the Model Configuration for Converted Models

```
[isRestored, restoredModels] = ...  
Simulink.BlockDiagram.restoreConfigSet('sldemo_mdlref_depgraph');
```

- “Share a Configuration Across Referenced Models”
- “Manage a Configuration Reference”

Input Arguments

model — Top model

character vector

Name of top model, specified as a character vector.

Example: 'mdl'

Output Arguments

isRestored — Success of restoration

false (default) | true

Indication of whether configuration reference propagation is successful, specified as a Boolean.

restoredModels — Restored models

cell array of character vectors

List of restored model names, specified as a cell array of character vectors.

See Also

`Simulink.BlockDiagram.propagateConfigSet`

Topics

“Share a Configuration Across Referenced Models”

“Manage a Configuration Reference”

Introduced in R2012b

Simulink.BlockDiagram.saveActiveConfigSet

Package: Simulink.BlockDiagram

Save active configuration set of model

Syntax

```
Simulink.BlockDiagram.saveActiveConfigSet(model, filename)
```

Description

`Simulink.BlockDiagram.saveActiveConfigSet(model, filename)` saves the active configuration set of a model to a `.m` or `.mat` file. `model` is the name or handle of the model. `filename` is the name of the file to save the model configuration set. If you specify a `.m` extension, the file contains a function that creates a configuration set object. If you specify a `.mat` extension, the file contains a configuration set object. If you do not provide a file extension, the active configuration set is saved to a file with a `.m` extension. Do not specify `filename` to be the same as a model name; otherwise the software cannot determine which file contains the configuration set object when loading the file.

Examples

Save the configuration set from the `sldemo_counters` model to `my_config_set.m`.

```
% Open the sldemo_counters model
sldemo_counters
% Save the active configuration set to my_config_set.m
Simulink.BlockDiagram.saveActiveConfigSet('sldemo_counters', 'my_config_set.m')
```

See Also

`Simulink.BlockDiagram.loadActiveConfigSet` | `Simulink.ConfigSet` | `attachConfigSet` | `attachConfigSetCopy` | `detachConfigSet` | `getActiveConfigSet` | `getConfigSet` | `getConfigSets` | `setActiveConfigSet`

Topics

“Save a Configuration Set”

Introduced in R2010b

Simulink.Bus.cellToObject

Convert cell array containing bus information to bus objects

Syntax

```
Simulink.Bus.cellToObject (busCells)
```

Description

`Simulink.Bus.cellToObject (busCells)` creates a set of bus objects in the MATLAB base workspace from a cell array of bus information.

Examples

Create Bus Objects from Cell Array of Bus Information

Create a cell array of cell arrays of bus information, and use that cell array to generate a bus object in the base workspace.

Create a cell array of bus information.

```
busCell = { ...  
    { ...  
        'myBusObj', ...  
        'MyHeader.h', ...  
        'My description', ...  
        'Exported', ...  
        '-1', ...  
        {'a',1,'double', [0.2 0],'real','Frame'}; ...  
        {'b',1,'double', [0.2 0],'real','Sample'}}, ...  
    }, ...  
};
```

Generate a bus object in the base workspace from the cell array.

```
Simulink.Bus.cellToObject (busCell)
```

- “When to Use Bus Objects”
- “Create Bus Objects Programmatically”
- “Create Bus Objects with the Bus Editor”
- “Save and Import Bus Objects”

Input Arguments

busCells — Bus object information

cell array of cell arrays

Bus object information, specified as a cell array of cell arrays. Each subordinate cell array contains bus object information, using these data fields:

- Bus name
- Header file
- Description
- Data scope
- Alignment
- Elements

The `elements` field is an array containing this data for each element:

- Element name
- Dimensions
- Data type
- Sample time
- Complexity
- Dimensions mode
- Minimum
- Maximum
- Units
- Description

Tips

The inverse function is `Simulink.Bus.objectToCell`.

See Also

`Simulink.Bus` | `Simulink.Bus.objectToCell` | `Simulink.BusElement`

Topics

“When to Use Bus Objects”

“Create Bus Objects Programmatically”

“Create Bus Objects with the Bus Editor”

“Save and Import Bus Objects”

Introduced before R2006a

Simulink.Bus.createMATLABStruct

Create MATLAB structures using same hierarchy and attributes as bus signals

Syntax

```
structFromBus = Simulink.Bus.createMATLABStruct (busSource)
structFromBus = Simulink.Bus.createMATLABStruct (busSource,
partialValues)
structFromBus = Simulink.Bus.createMATLABStruct (busSource,
partialValues,dims)

structsForBuses = Simulink.Bus.createMATLABStruct (portHandles)
structsForBuses = Simulink.Bus.createMATLABStruct (portHandles,
partialStructures)
structsForBuses = Simulink.Bus.createMATLABStruct (busObjectNames)
```

Description

`structFromBus = Simulink.Bus.createMATLABStruct (busSource)` creates a MATLAB structure that has the same hierarchy and attributes (such as type and dimension) as the bus specified in `busSource`. The resulting structure uses the ground values of the bus signal.

`structFromBus = Simulink.Bus.createMATLABStruct (busSource, partialValues)` creates a structure that uses specified values of `partialValues` instead of the corresponding ground values of the bus signal.

`structFromBus = Simulink.Bus.createMATLABStruct (busSource, partialValues,dims)` creates a structure that has the specified dimensions. To create a structure for an array of buses, use `dims`.

`structsForBuses = Simulink.Bus.createMATLABStruct (portHandles)` creates a cell array of structures for bus signal ports, specified with port handles. The resulting cell array of structures uses ground values. Use this syntax to create initialization structures for multiple bus ports. This syntax improves performance compared to using separate `Simulink.Bus.createMATLABStruct` calls to create the structures.

`structsForBuses = Simulink.Bus.createMATLABStruct(portHandles, partialStructures)` creates a cell array of structures that uses the specified values of `partialStructures` instead of the ground values.

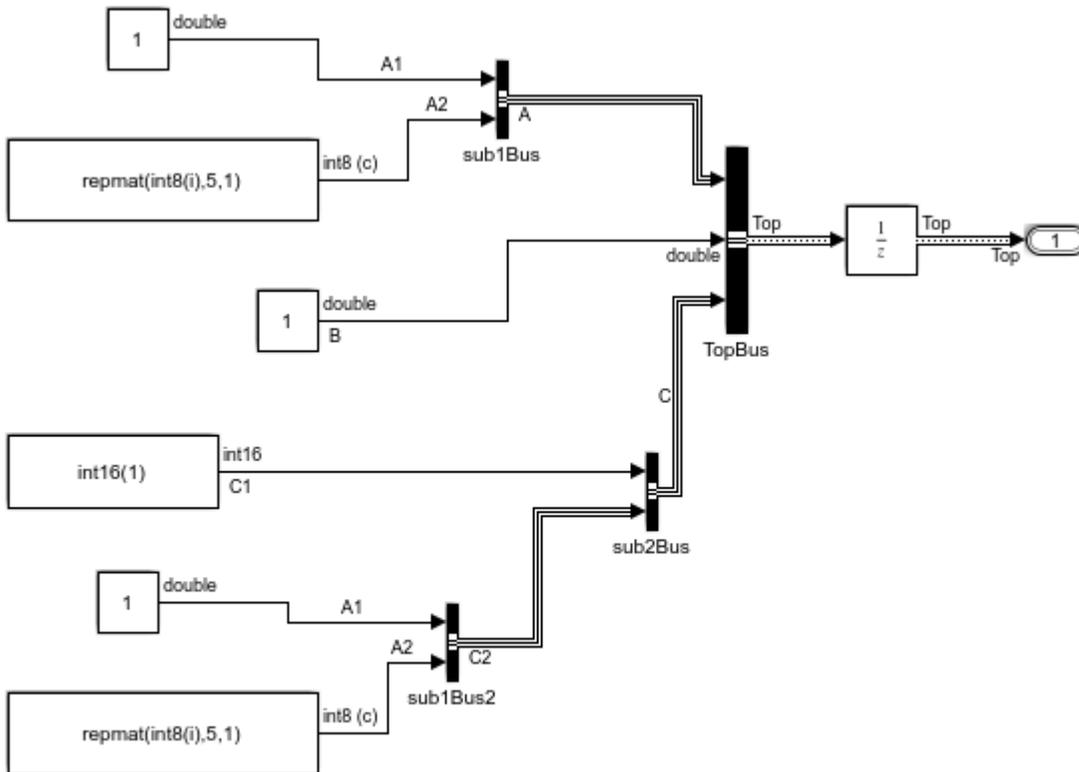
`structsForBuses = Simulink.Bus.createMATLABStruct(busObjectNames)` creates a cell array of structures based on the specified bus objects.

Examples

MATLAB Structure from Bus Object

Open a Simulink® model and simulate it.

```
model = fullfile(matlabroot, 'examples', 'simulink', 'busic_example');
open_system(model);
sim('busic_example')
```



Copyright 2004-2010 The MathWorks, Inc.

Create a MATLAB® structure using the bus object `Top`, which the `basic_example` model loads.

```
mStruct = Simulink.Bus.createMATLABStruct('Top')
```

```
mStruct =
```

```
struct with fields:
```

```
A: [1x1 struct]
B: 0
```

```
C: [1x1 struct]
```

Set a value for the field of the `mStruct` structure that corresponds to bus element A1 of bus A.

```
mStruct.A.A1 = 3;  
mStruct.A
```

```
ans =
```

```
struct with fields:
```

```
A1: 3  
A2: [5x1 int8]
```

Simulink sets the other fields in the structure to the ground values of the corresponding bus elements.

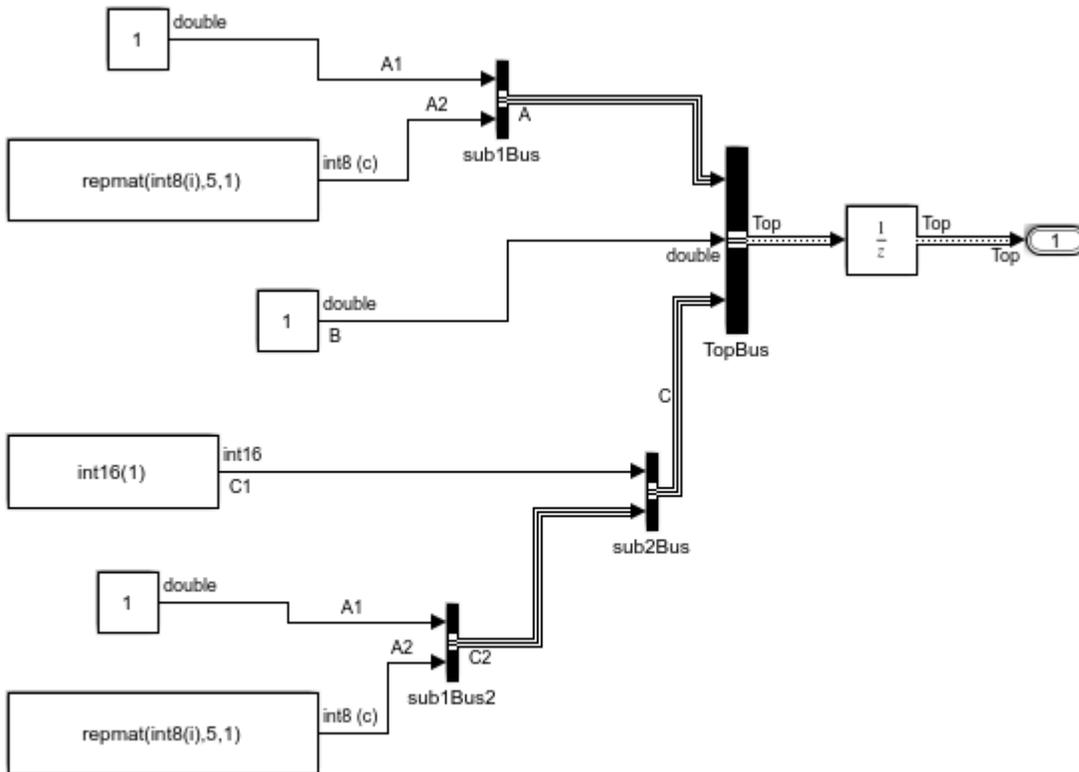
You can use `mStruct` as the initial condition structure for the Unit Delay block.

Initialize Signal Elements That Use a Data Type Other than double

Create a MATLAB® structure for a bus whose signal elements use a data type other than `double`. Use a partial structure to specify initialization values for a subset of the elements. When you create the partial structure, match the data types of the fields with the data types of the corresponding elements.

Open a Simulink® model.

```
model = fullfile(matlabroot, 'examples', 'simulink', 'busic_example');  
open_system(model);  
sim('busic_example')
```



Copyright 2004-2010 The MathWorks, Inc.

The C1 signal element that the Constant5 block produces uses the data type `int16`.

Find the port handle for the Bus Creator block port that produces the `Top` bus signal.

```
ph = get_param('busic_example/TopBus','PortHandles');
```

Create a partial structure that specifies values for a subset of the elements in the bus signal created by the `TopBus` block. To set the value of the `C.C1` field, use a typed expression. Match the data type in the expression with the data type of the signal element in the model (`int16`).

```
PartialstructForK = struct('B',3,'C',struct('C1',int16(5)));
```

Create a full structure by using the port handle (ph) for the TopBus block. Override the ground values for the C.C1 and B elements.

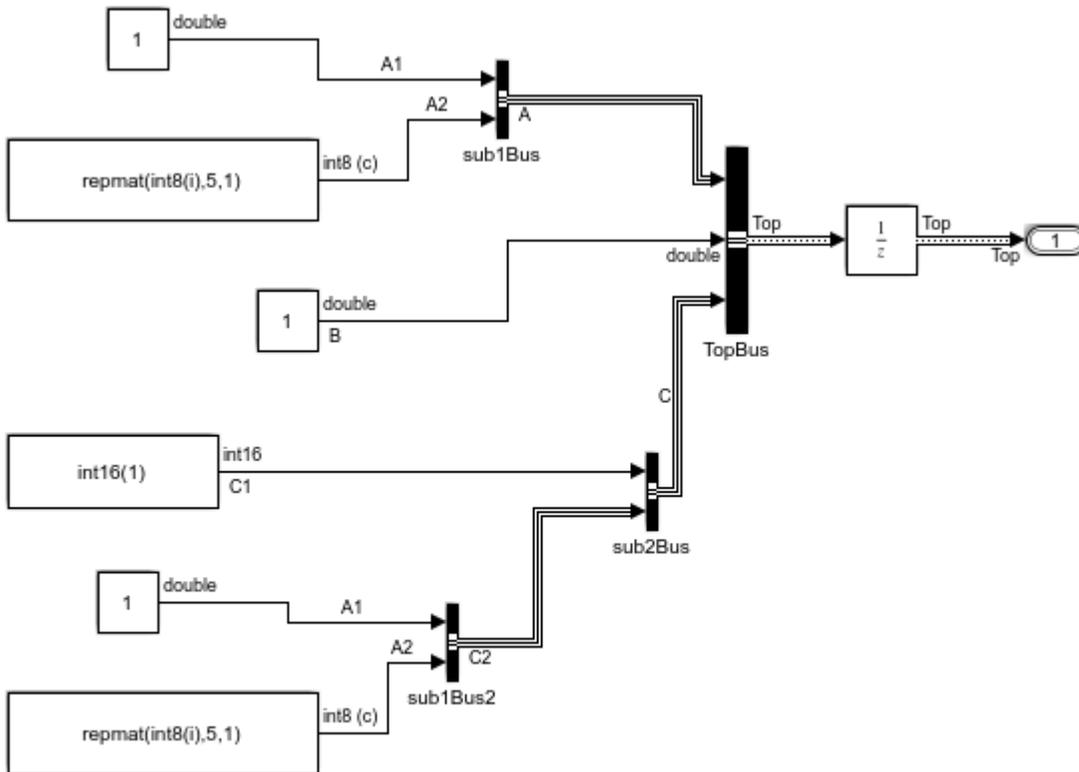
```
outPort = ph.Outport;  
mStruct = Simulink.Bus.createMATLABStruct(outPort,PartialstructForK);
```

The field C.C1 in the output structure continues to use the data type int16.

MATLAB Structure with Specified Dimensions

Open a Simulink® model and simulate it.

```
model = fullfile(matlabroot, 'examples', 'simulink', 'basic_example');  
open_system(model);  
sim('basic_example')
```



Copyright 2004-2010 The MathWorks, Inc.

Create a partial structure, which is a MATLAB® structure that specifies values for a subset of bus elements for the bus signal created by the TopBus block.

```
PartialStructForK = struct('A',struct('A1',4),'B',3)
```

```
PartialStructForK =  
struct with fields:  
A: [1x1 struct]
```

```
B: 3
```

Create a MATLAB structure using the bus object `Top` (which the `basic_example` model loads), a partial structure, and dimensions for the resulting structure.

```
structFromBus = Simulink.Bus.createMATLABStruct...  
    ('Top',PartialStructForK,[2 3])
```

```
structFromBus =
```

```
2x3 struct array with fields:
```

```
A  
B  
C
```

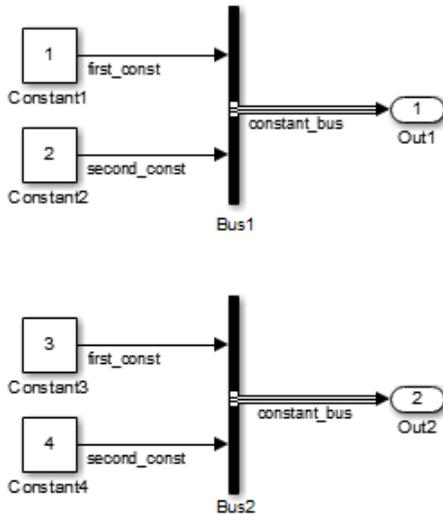
Close the system.

```
close_system('basic_example')
```

Cell Array of MATLAB Structures

Open a Simulink model and simulate it.

```
open_system(docpath(fullfile(docroot,'toolbox','simulink',...  
    'examples','ex_two_outports_create_struct')))  
sim('ex_two_outports_create_struct')
```



Find the port handles for the Bus Creator blocks Bus1 and Bus2.

```
ph_1 = get_param...
    ('ex_two_outputs_create_struct/Bus Creator', 'PortHandles')
ph_2 = get_param...
    ('ex_two_outputs_create_struct/Bus Creator1', 'PortHandles')
```

Create a MATLAB structure using an array of port handles.

```
mStruct = Simulink.Bus.createMATLABStruct...
    ([ph_1.Outport ph_2.Outport])
```

```
mStruct =
    [1x1 struct]
    [1x1 struct]
```

Close the system.

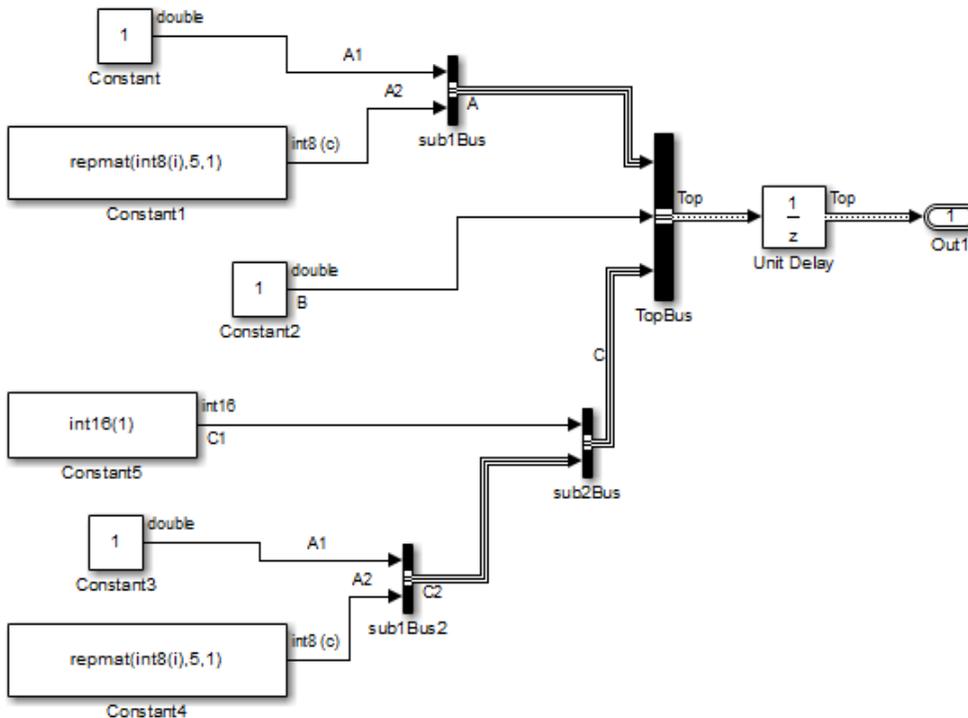
```
close_system('ex_two_outputs_create_struct')
```

MATLAB Structure from Bus Port and Partial Structure

Create a MATLAB structure based on a port that connects to a bus signal. Use a partial structure to specify values for a subset of bus elements of the bus signal that connects to the port.

Open a Simulink model.

```
run([docroot '/toolbox/simulink/ug/examples/signals/basic_example.mdl']);
sim('basic_example')
```



Find the port handle for the Bus Creator block port that produces the Top bus signal. The Outport handle is the handle that you need.

```
ph = get_param('basic_example/TopBus', 'PortHandles')
```

```
ph =
```

```

    Inport: [143.0013 144.0013 145.0013]
    Outport: 34.0013
    Enable: []
    Trigger: []
    State: []
    LConn: []
    RConn: []
    Ifaction: []

```

Create a partial structure, which is a MATLAB structure that specifies values for a subset of bus elements for the bus signal created by the TopBus block.

```
PartialstructForK = struct('A',struct('A1',4),'B',3)
```

```
PartialstructForK =
```

```

    A: [1x1 struct]
    B: 3

```

Bus elements represented by structure fields `Top.B` and `Top.A` are at the same nesting level in the bus. You can use this partial structure to override the ground values for the B and A bus signal elements.

When you create a structure from a bus object or from a bus port, you can use a partial structure as an optional argument.

Create a MATLAB structure by using the port handle (`ph`) for the TopBus block. Override the ground values for the A.A1 and B bus elements.

```
outPort = ph.Outport;
mStruct = Simulink.Bus.createMATLABStruct(outPort,PartialstructForK)
```

```
mStruct =
```

```

    A: [1x1 struct]
    B: 3
    C: [1x1 struct]

```

Input Arguments

busSource — Source representing a bus signal

a `Simulink.Bus` object | port handle

Source representing a bus signal to use for creating a MATLAB structure, specified as the name of a bus object or port handle.

- If you use the `dims` argument, then for `busSource`, use a bus object.
- For an array of buses signal, you cannot use a port handle.
- If you use a bus object name, then the bus object must be in the MATLAB base workspace. The data type for the bus object name is `char`.
- If you use a port handle, then the model must compile successfully before you use the `createMATLABStruct` method. The data type for the port handle is a `double`.

Example:

```
structFromBus = Simulink.Bus.createMATLABStruct('myBusObject');  
structForPortHandle = Simulink.Bus.createMATLABStruct(port_handle_1);
```

partialValues — Values for a subset of leaf nodes of the resulting structure

partial structure | []

Values for a subset of leaf nodes of the resulting structure, specified as a partial structure or empty array. Each field that you specify in a partial structure must match the data attributes of the corresponding bus element exactly. For details, see “Create Partial Structures for Initialization”.

Use an empty matrix [] when you use the `dims` argument and want to use ground values for all of the nodes in the resulting structure.

Data Types: `struct`

dims — Dimensions of the resulting structure

vector

Dimensions of the resulting structure, specified as a vector.

Each dimension element must be an integer that is greater than or equal to 1. If you specify `partialValues`, then each dimension element in `dims` must be greater than or equal to its corresponding dimension element in the partial structure.

To use ground values, use an empty matrix ([]) for `partialValues`.

Data Types: `double`

portHandles — Handles of bus signal ports

array

Handles of bus signal ports, specified as an array. If you use the `partialStructures` argument, then the number of port handles that you specify in `portHandles` must be the same as the number of partial structures.

Data Types: `double`**partialStructures** — Partial structures

cell array

Partial structures specified as a cell array. The number of port handles that you specify in `portHandles` must be the same as the number of partial structures.

Data Types: `cell`**busObjectNames** — Bus object names

cell array

Bus object names, specified as a cell array.

Data Types: `cell`

Output Arguments

structFromBus — Bus signal hierarchy and attributes

MATLAB structure

Bus signal hierarchy and attributes, returned as a MATLAB structure.

The dimensions of `structFromBus` depend on the input arguments:

- If you specify only `busSource`, then the dimension is 1.
- If you also specify `partialValues`, then the dimensions match the dimensions of `partialValues`.
- If you specify the `dims` argument, then the dimensions match the dimensions of `dims`.

structsForBuses — Structures having the same hierarchy and attributes as bus signals

cell array

Structures having the same hierarchy and attributes as bus signals, returned as a cell array of structures of data with same hierarchy and attributes as a bus signals that you specify with an array of port handles. The cell array of structures uses ground values of the bus signals.

The dimensions of `StructsForBuses` depend on the input arguments:

- If you specify only `portHandles`, then the dimension is 1.
- If you also specify `partialStructures`, then the dimensions match the dimensions of `partialStructures`.

Tips

- If you use the `Simulink.Bus.createMATLABStruct` function repeatedly for the same model (for example, in a loop in a script), you can improve performance by avoiding multiple model compilations. For improved speed, put the model in compile before using the function multiple times. For example, to put the `vdp` model in compile, use this command:

```
[sys,x0,str,ts] = vdp([],[],[],'compile')
```

After you create the MATLAB structure, terminate the compile. For example:

```
vdp([],[],[],'term')
```

- You can use the Bus Editor to invoke the `Simulink.Bus.createMATLABStruct` function. Use one of these approaches:
 - Select the **File > Create a MATLAB structure** menu item.
 - Select the bus object for which you want to create a full MATLAB structure. Then, in the toolbar, click the **Create a MATLAB structure** button (.

You can then edit the MATLAB structure in the MATLAB Editor and evaluate the code to create or update the values in this structure.

- You can use the `Simulink.Bus.createMATLABStruct` function to specify the initial value of the output of a referenced model. For details, see the “Referenced Model: Setting Initial Value for Bus Output” section of the Detailed Workflow for Managing Data with Model Reference example.

See Also

“Specify Initial Conditions for Bus Signals” | “Composite Signals” | Bus to Vector | Bus Assignment | Bus Creator | Simulink.Bus | Simulink.Bus.cellToObject | Simulink.Bus.createObject | Simulink.Bus.objectToCell | Simulink.Bus.save | Simulink.BusElement | Simulink.SimulationData.createStructOfTimeseries |

Introduced in R2010a

Simulink.Bus.createObject

Create bus objects from blocks or MATLAB structures

Syntax

```
busInfo = Simulink.Bus.createObject(model,blocks)
busInfo = Simulink.Bus.createObject(struct)
busInfo = Simulink.Bus.createObject( ____,file)
busInfo = Simulink.Bus.createObject( ____,format)
```

Description

`busInfo = Simulink.Bus.createObject(model,blocks)` creates bus objects (instances of `Simulink.Bus` class in the MATLAB base workspace) for specified blocks, and returns information about the objects that it created.

`busInfo = Simulink.Bus.createObject(struct)` creates bus objects in the MATLAB workspace from a MATLAB structure of `timeseries` objects or a numeric MATLAB structure.

`busInfo = Simulink.Bus.createObject(____,file)` saves the bus objects in a MATLAB file that contains a cell array of cell arrays. Each subordinate cell array represents a bus object and contains this data:

- Bus name
- Header file
- Description
- Data scope
- Alignment
- Elements

The `elements` field is an array containing this data for each element:

- Element name

- Dimensions
- Data type
- Sample time
- Complexity
- Dimensions mode
- Minimum
- Maximum
- Units
- Description

`busInfo = Simulink.Bus.createObject(____, format)` saves the bus objects in a file that contains either a cell array of bus information, or the bus objects themselves.

Examples

Use Bus Creator Blocks to Create a Bus Object

Create a bus object from the Bus Creator block called Bus Creator2.

```
open_system('busdemo')
bus2Info = Simulink.Bus.createObject...
('busdemo','busdemo/Bus Creator2')
close_system('busdemo')
```

Create a bus object from two Bus Creator blocks, using block handles to specify the blocks. Assign the block handles to variables and use the variables in a vector to specify the blocks used for creating the bus object.

Open the model.

```
clear;
open_system('busdemo')
```

In the Simulink Editor, select the Bus Creator2 block. In MATLAB, assign the block handle to a variable.

```
bc2 = gcbh;
```

In the Simulink Editor, select the Bus Creator block. In MATLAB, assign the block handle to a variable.

```
bc1 = gcbh;
```

Use the block handle variables in a vector to create a bus object.

```
bus3Info = Simulink.Bus.createObject...  
( 'busdemo', [bc2 bc1], 'busdemo_busobject' )  
close_system('busdemo')
```

Create Bus Objects from Cell Array of Bus Information

Create a cell array of cell arrays of bus information, and use that cell array to generate a bus object in the base workspace.

Create a cell array of bus information.

```
busCell = { ...  
    { ...  
        'myBusObj', ...  
        'MyHeader.h', ...  
        'My description', ...  
        'Exported', ...  
        '-1', ...  
        {{ 'a', 1, 'double', [0.2 0], 'real', 'Frame' }; ...  
          { 'b', 1, 'double', [0.2 0], 'real', 'Sample' }}, ...  
        }, ...  
    };
```

Generate a bus object in the base workspace from the cell array.

```
Simulink.Bus.cellToObject(busCell)
```

Input Arguments

model — Model name or handle

character vector

Model name or handle, specified as a character vector.

blocks — Blocks to create bus objects for

character vector | cell array of block pathnames | vector of block names

Blocks to create bus object for, specified as a character vector, cell array of block pathnames, or vector of block names.

- For just one block, specify the full path name of the block.
- For multiple blocks, specify either a cell array of block pathnames or a vector of block names.
- If you specify a Bus Creator block that is at the highest level of a bus hierarchy, the function creates bus objects for all buses in the hierarchy.

struct — Structure used to create bus objectsstructure of MATLAB `timeseries` objects | numeric structure

Structure used to create bus objects, specified as a structure of MATLAB `timeseries` objects or a numeric structure.

file — File to save bus objects in

character vector

File to save bus objects in, specified as a character vector. The file name must be unique. If you omit this argument, the function saves the created bus objects in a cell array, not in a file.

format — Format for storing bus objects in file

'cell' (default) | 'object'

Format for storing bus objects in file, specified as either 'cell' or 'object'. The cell format is more compact, but the cell format is easier to read.

Output Arguments

busInfo — Bus information for specified blocks

structure array

Bus information for specified blocks, returned as a structure array. Each element of the structure array corresponds to one block and contains these fields:

- `block` – Handle of the block
- `busName` – Name of the bus object associated with the block

Tips

If you specify a model name, the model must compile successfully before you use the `Simulink.Bus.createObject` command.

See Also

[Bus Creator](#) | [Simulink.Bus](#) | [Simulink.Bus.cellToObject](#) |
[Simulink.Bus.createMATLABStruct](#) | [Simulink.Bus.objectToCell](#) |
[Simulink.BusElement](#)

Topics

- “When to Use Bus Objects”
- “Create Bus Objects Programmatically”
- “Create Bus Objects with the Bus Editor”
- “Save and Import Bus Objects”

Introduced before R2006a

Simulink.Bus.objectToCell

Use bus objects to create cell array containing bus information

Syntax

```
busCells = Simulink.Bus.objectToCell(busNames)
```

Description

`busCells = Simulink.Bus.objectToCell(busNames)` inputs a cell array of names of bus objects that are in the MATLAB base workspace. The function returns a cell array of cell arrays in which each subordinate cell array contains the bus information defined by one of the bus objects. The order of the elements in the output cell array corresponds to the order of the names in the input cell array.

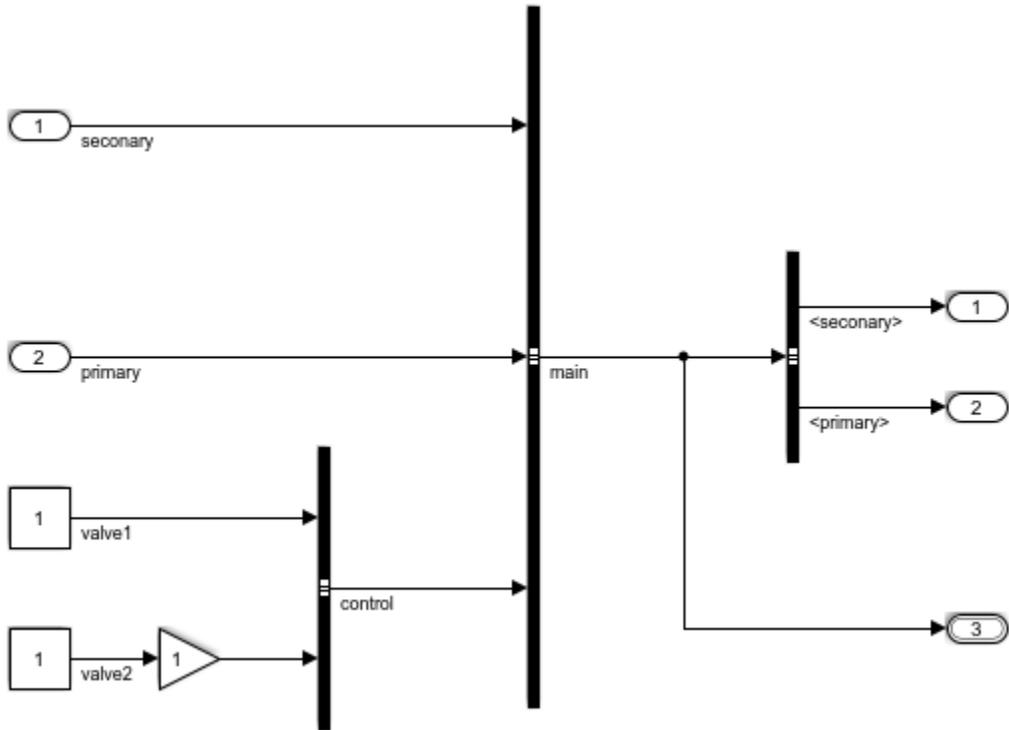
Examples

Create Cell Array Containing Bus Object Information

Use the `Simulink.Bus.objectToCell` function to create a cell array of information about bus objects in the base workspace.

Open a model that has two bus objects defined, CONTROL and MAIN.

```
open_system(fullfile(matlabroot, 'examples', 'simulink', ...  
    'ex_bus_object_tutorial_using_objects'))
```



Create cell array of information about the CONTROL bus object.

```
busCells = Simulink.Bus.objectToCell({'CONTROL'});
busCells{1}
```

```
ans =
```

```
1x6 cell array
```

```
    {'CONTROL'}    {0x0 char}    {0x0 char}    {'Auto'}    {'-1'}    {2x1 cell}
```

- “Create Bus Objects Programmatically”
- “When to Use Bus Objects”
- “Create Bus Objects with the Bus Editor”

- “Modify Bus Objects”
- “Save and Import Bus Objects”

Input Arguments

busNames — Bus objects in base workspace for which to create cell arrays of bus object information

cell array of names of bus objects, specified as character vectors

Bus objects in the base workspace for which to create cell arrays of bus object information, specified as a cell array of names of bus objects. Specify the bus object names as character vectors. If `busNames` is empty, the function converts all bus objects in the base workspace.

Output Arguments

busCells — Bus object information

cell array of cell arrays

Bus object information, specified as a cell array of cell arrays. Each subordinate cell array contains bus object information, using these data fields:

- Bus name
- Header file
- Description
- Data scope
- Alignment
- Elements

The `elements` field is an array containing this data for each element:

- Element name
- Dimensions
- Data type
- Sample time

- Complexity
- Dimensions mode
- Minimum
- Maximum
- Units
- Description

Tips

The inverse function is `Simulink.Bus.cellToObject`.

See Also

`Simulink.Bus` | `Simulink.Bus.cellToObject` |
`Simulink.Bus.createMATLABStruct` | `Simulink.Bus.createObject` |
`Simulink.Bus.save` | `Simulink.BusElement`

Topics

- “Create Bus Objects Programmatically”
- “When to Use Bus Objects”
- “Create Bus Objects with the Bus Editor”
- “Modify Bus Objects”
- “Save and Import Bus Objects”

Introduced in R2007a

Simulink.Bus.save

Save bus objects in MATLAB file

Syntax

```
Simulink.Bus.save(fileName)
Simulink.Bus.save(fileName,format)
Simulink.Bus.save(fileName,format,busNames)
```

Description

`Simulink.Bus.save(fileName)` saves all bus objects (instances of `Simulink.Bus` class) that are in the MATLAB base workspace in a MATLAB file that contains a cell array of cell arrays. Each subordinate cell array represents a bus object and contains this data:

- Bus name
- Header file
- Description
- Data scope
- Alignment
- Elements

The `elements` field is an array containing this data for each element:

- Element name
- Dimensions
- Data type
- Sample time
- Complexity
- Dimensions mode

- Minimum
- Maximum
- Units
- Description

`Simulink.Bus.save(fileName, format)` saves the bus objects in a MATLAB file that contains either a cell array of bus information or the bus objects themselves.

`Simulink.Bus.save(fileName, format, busNames)` saves only those bus objects whose names appear in `busNames`.

Examples

Save a Bus Object

Use the `Simulink.Bus.save` function to save a bus object.

Define a cell array of bus object information.

```
busCell = { ...
    { ...
        'myBusObj', ...
        'MyHeader.h', ...
        'My description', ...
        'Exported', ...
        '-1', ...
        {'a',1,'double',[0.2 0],'real','Frame'}; ...
        {'b',1,'double',[0.2 0],'real','Sample'}}; ...
    }, ...
};
```

Create `myBusObj` bus object from the cell array.

```
Simulink.Bus.cellToObject(busCell);
```

Save the bus object in the `BusCellFile1` file, in cell format.

```
fileName = 'BusCellFile1';
Simulink.Bus.save(fileName);
```

Save the bus object in bus format.

```
Simulink.Bus.save('BusObjFile','object');
```

Save myBusObj in cell format in BusCellFile2.m.

```
Simulink.Bus.save('BusCellFile2','cell',{'myBusObj'});
```

- “When to Use Bus Objects”
- “Create Bus Objects Programmatically”
- “Create Bus Objects with the Bus Editor”
- “Save and Import Bus Objects”

Input Arguments

fileName — File in which to store bus objects

character vector

File in which to store bus objects, specified as a character vector.

format — Format for storing bus objects in file

'cell' (default) | 'object'

Format for storing bus objects in file, specified as either 'cell' or 'object'. The cell format is more compact, but the object format is easier to read.

busNames — Bus objects to save

cell array of bus objects

Bus objects to save, specified as a cell array of bus objects. Only the specified bus objects in the base workspace are saved.

Tips

Executing a MATLAB file created by `Simulink.Bus.save` in cell array format calls `Simulink.Bus.cellToObject` to recreate the bus objects and returns the new bus objects in the cell array. To suppress the creation of bus objects, specify the optional argument 'false' when you execute the MATLAB file.

See Also

Bus Creator | Simulink.Bus | Simulink.Bus.cellToObject |
Simulink.Bus.createMATLABStruct | Simulink.Bus.objectToCell |
Simulink.BusElement

Topics

“When to Use Bus Objects”
“Create Bus Objects Programmatically”
“Create Bus Objects with the Bus Editor”
“Save and Import Bus Objects”

Introduced before R2006a

Simulink.createFromTemplate

Create model or project from template

Syntax

```
Simulink.createFromTemplate(templatename)
h = Simulink.createFromTemplate(templatename)
h = Simulink.createFromTemplate(templatename,Name,Value)
```

Description

`Simulink.createFromTemplate(templatename)` creates a model or a project from the template file specified by `templatename`.

`h = Simulink.createFromTemplate(templatename)` creates a model or a project from the template file and returns `h`, either a numeric model handle or a `simulinkproject` object.

`h = Simulink.createFromTemplate(templatename,Name,Value)` specifies additional options as one or more `Name, Value` pair arguments.

Examples

Create a Model From a Template

```
Simulink.createFromTemplate('simple_simulation.sltx')
```

Create a Project From a Template and Get the Handle

Create a project from a template, specify the name and root folder, and return the handle to the new project (a `simulinkproject` object) for manipulating it programmatically.

```
proj = Simulink.createFromTemplate('code_generation_example.sltx', 'Name', 'myProject', 'E
```

- “Create and Open Models”
- “Create a Template from a Model”
- “Using Templates to Create Standard Project Settings”

Input Arguments

templatename — Template file name

character vector

Template file name, specified as a character vector. If the template is not on the MATLAB path, specify the fully-qualified path to the template file and `*.sltx` extension.

Example:

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

Folder — Project root folder

character vector

Project root folder, if creating a new project, specified as a character vector.

Data Types: `char`

Name — New model or project name

character vector

New model or project name, specified as a character vector.

Data Types: `char`

Output Arguments

h — Handle

numeric handle | `simulinkproject`

Handle to the new model project, returned either as a numeric model handle or a `simulinkproject` object.

See Also

`Simulink.defaultModelTemplate` | `Simulink.exportToTemplate` |
`Simulink.findTemplates`

Topics

“Create and Open Models”

“Create a Template from a Model”

“Using Templates to Create Standard Project Settings”

Introduced in R2016a

Simulink.data.assigninGlobal

Modify variable values in context of Simulink model

Syntax

```
Simulink.data.assigninGlobal(modelName, varName, varValue)
```

Description

`Simulink.data.assigninGlobal(modelName, varName, varValue)` assigns the value `varValue` to the variable or data dictionary entry `varName` in the context of the Simulink model `modelName`. `assigninGlobal` creates the variable or data dictionary entry if it does not already exist. The function operates in the Design Data section of the data dictionary that is linked to the target model or in the MATLAB base workspace if the target model is not linked to any data dictionary.

If the target model is linked to a data dictionary that references other dictionaries, `assigninGlobal` searches for `varName` in the entire dictionary hierarchy. If `assigninGlobal` does not find a matching entry, the function creates an entry in the dictionary that is linked to the target model.

Examples

Modify Variable in Model With or Without Data Dictionary

Create a variable `myNewVariable` with value `237` in the context of the Simulink model `vdp.slx`, which is not linked to any data dictionary. `myNewVariable` appears as a variable in the MATLAB base workspace.

```
Simulink.data.assigninGlobal('vdp', 'myNewVariable', 237)
```

Create a variable `myNewEntry` with value `true` in the context of the Simulink model `sldemo_fuelsys_dd_controller.slx`, which is linked to the data dictionary

sldemo_fuelsys_dd_controller.sldd. The entry myNewEntry appears in the Design Data section of the dictionary.

```
Simulink.data.assigninGlobal('sldemo_fuelsys_dd_controller',...
    'myNewEntry',true)
```

Confirm the addition of myNewEntry to the data dictionary sldemo_fuelsys_dd_controller.sldd by viewing the dictionary in Model Explorer.

```
myDictionaryObj = Simulink.data.dictionary.open(...
    'sldemo_fuelsys_dd_controller.sldd');
show(myDictionaryObj)
```

- “Store Data in Dictionary Programmatically”

Input Arguments

modelName — Name of target Simulink model

character vector

Name of target Simulink model, specified as a character vector.

Example: 'myTestModel'

Data Types: char

varName — Name of target variable or data dictionary entry

character vector

Name of target variable or data dictionary entry, specified as a character vector.

Example: 'myTargetVariable'

Data Types: char

varValue — Value to assign to variable or data dictionary entry

MATLAB expression

Value to assign to variable or data dictionary entry, specified as a MATLAB expression that returns any valid data type or data dictionary content.

Example: 27.5

Example: myBaseWorkspaceVariable

Example: `Simulink.Parameter`

Tips

- `assignInGlobal` helps you transition Simulink models to using data dictionaries. You can use the function to assign values to model variables before and after linking a model to a data dictionary.

See Also

`Simulink.data.dictionary.open` | `Simulink.data.evalInGlobal` |
`Simulink.data.existsInGlobal`

Topics

“Store Data in Dictionary Programmatically”

“What Is a Data Dictionary?”

“Considerations before Migrating to Data Dictionary”

Introduced in R2015a

Simulink.data.dictionary.cleanupWorkerCache

Restore defaults after parallel simulation with data dictionary

Syntax

```
Simulink.data.dictionary.cleanupWorkerCache
```

Description

`Simulink.data.dictionary.cleanupWorkerCache` restores default settings after you have finished parallel simulation of a model that is linked to a data dictionary. Use this function in a `spmd` block, after you finish parallel simulation using `parfor` blocks, to restore default settings that were altered by the `Simulink.data.dictionary.setupWorkerCache` function.

During parallel simulation of a model that is linked to a data dictionary, you can allow each worker to access and modify the data in the dictionary independently of other workers. The function `Simulink.data.dictionary.setupWorkerCache` grants each worker a unique dictionary cache to allow independent access to the data, and the function `Simulink.data.dictionary.cleanupWorkerCache` restores cache settings to their default values.

You must have a Parallel Computing Toolbox license to perform parallel simulation using a `parfor` block.

Examples

Sweep Variant Control Using Parallel Simulation

To use parallel simulation to sweep a variant control (a `Simulink.Parameter` object whose value influences the variant condition of a `Simulink.Variant` object) that you store in a data dictionary, use this code as a template. Change the names and values of the model, data dictionary, and variant control to match your application.

To sweep block parameter values or the values of workspace variables that you use to set block parameters, use `Simulink.SimulationInput` objects instead of the programmatic interface to the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values”.

You must have a Parallel Computing Toolbox license to perform parallel simulation.

```
% For convenience, define names of model and data dictionary
model = 'mySweepMdl';
dd = 'mySweepDD.sldd';

% Define the sweeping values for the variant control
CtrlValues = [1 2 3 4];

% Grant each worker in the parallel pool an independent data dictionary
% so they can use the data without interference
spmd
    Simulink.data.dictionary.setupWorkerCache
end

% Determine the number of times to simulate
numberOfSims = length(CtrlValues);

% Prepare a nondistributed array to contain simulation output
simOut = cell(1,numberOfSims);

parfor index = 1:numberOfSims
    % Create objects to interact with dictionary data
    % You must create these objects for every iteration of the parfor-loop
    dictObj = Simulink.data.dictionary.open(dd);
    sectObj = getSection(dictObj,'Design Data');
    entryObj = getEntry(sectObj,'MODE');
    % Suppose MODE is a Simulink.Parameter object stored in the data dictionary

    % Modify the value of MODE
    temp = getValue(entryObj);
    temp.Value = CtrlValues(index);
    setValue(entryObj,temp);

    % Simulate and store simulation output in the nondistributed array
    simOut{index} = sim(model);

    % Each worker must discard all changes to the data dictionary and
    % close the dictionary when finished with an iteration of the parfor-loop
```

```
        discardChanges(dictObj);
        close(dictObj);
end

% Restore default settings that were changed by the function
% Simulink.data.dictionary.setupWorkerCache
% Prior to calling cleanupWorkerCache, close the model

spsmd
    bdclose(model)
    Simulink.data.dictionary.cleanupWorkerCache
end
```

Note If data dictionaries are open, you cannot use the command `Simulink.data.dictionary.cleanupWorkerCache`. To identify open data dictionaries, use `Simulink.data.dictionary.getOpenDictionaryPaths`.

See Also

`Simulink.data.dictionary.closeAll` |
`Simulink.data.dictionary.getOpenDictionaryPaths` |
`Simulink.data.dictionary.setupWorkerCache` | `parfor` | `spsmd`

Topics

“Store Data in Dictionary Programmatically”
“What Is a Data Dictionary?”
“Run Code on Parallel Pools” (Parallel Computing Toolbox)

Introduced in R2015a

Simulink.data.dictionary.closeAll

Close all connections to all open data dictionaries

Syntax

```
Simulink.data.dictionary.closeAll
Simulink.data.dictionary.closeAll(dictFileName)
Simulink.data.dictionary.closeAll( ____, unsavedAction)
```

Description

`Simulink.data.dictionary.closeAll` attempts to close all connections to all data dictionaries that are open. For example, if you create objects, such as `Simulink.data.Dictionary`, that refer to a dictionary, that dictionary is open.

Some commands and functions, such as `Simulink.data.dictionary.cleanupWorkerCache`, cannot operate when dictionaries are open. It is a best practice to close each connection individually by using functions and methods such as the `close` method of a `Simulink.data.Dictionary` object. To find dictionaries that are open, use `Simulink.data.dictionary.getOpenDictionaryPaths`. However, you can use this function to close all connections to all dictionaries.

You can also use this function to close dictionaries in a shutdown script that is part of a Simulink Project.

`Simulink.data.dictionary.closeAll(dictFileName)` closes all connections to the dictionary named `dictFileName`. If you open multiple dictionaries that use this file name (for example, if the dictionaries have different file paths), the function closes all connections to all of the dictionaries.

You cannot specify `dictFileName` as a full file path such as `'C:\temp\myDict.sldd'`.

`Simulink.data.dictionary.closeAll(____, unsavedAction)` closes all connections to the target dictionaries by discarding or saving unsaved changes. You can choose whether to save or discard all changes to all of the target dictionaries.

Examples

Close All Connections to All Open Dictionaries

Discard any unsaved changes. All of the entries in the dictionaries revert to the last saved state.

```
Simulink.data.dictionary.closeAll('-discard')
```

Close All Connections to Single Data Dictionary

Open multiple connections to a data dictionary, make a change, and close all of the connections by discarding the unsaved change.

At the command prompt, open a data dictionary by creating a `Simulink.data.Dictionary` object that refers to the dictionary.

```
dictObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd.sldd');
```

Display the dictionary in the Model Explorer

```
show(dictObj)
```

You now have two connections to this dictionary: The `Simulink.data.Dictionary` object and the Model Explorer.

Make a change to the dictionary by adding an entry.

```
dDataSectObj = getSection(dictObj, 'Design Data');  
addEntry(dDataSectObj, 'myEntry', 5.2);
```

The `Simulink.data.dictionary.Section` object `dDataSectObj` is a third connection to the dictionary.

Close the connections to the dictionary. Discard the unsaved change.

```
Simulink.data.dictionary.closeAll('sldemo_fuelsys_dd.sldd', '-discard')
```

The dictionary no longer appears as a node in the **Model Hierarchy** pane of the Model Explorer. The `Simulink.data.Dictionary` object `dictObj` is disconnected from the

dictionary. You cannot interact with the dictionary by using the `Simulink.data.dictionary.Section` object `dDataSectObj`.

Clear the objects that referred to the dictionary.

```
clear dictObj dDataSectObj
```

- “Store Data in Dictionary Programmatically”

Input Arguments

dictFileName — File name of target data dictionary or dictionaries

character vector

File name of target data dictionary or dictionaries, specified as a character vector. Use the file extension `sldd`.

Example: `'myDict.sldd'`

Data Types: `char`

unsavedAction — Action for unsaved changes

`'-discard'` | `'-save'`

Action for unsaved changes, specified as `'-discard'` (to discard changes) or `'-save'` (to save changes).

Tips

A data dictionary is open if any of these conditions are true:

- The dictionary appears as a node in the **Model Hierarchy** pane of the Model Explorer. To close this connection to the dictionary, right-click the node in Model Explorer and select **Close**. Alternatively, use the `hide` method of a `Simulink.data.Dictionary` object.
- You created an object of any of these classes that refer to the dictionary:
 - `Simulink.data.Dictionary`

- `Simulink.data.dictionary.Section`
- `Simulink.data.dictionary.Entry`

To close these connections to the dictionary, use the `close` method of the `Simulink.data.Dictionary` object or clear the object. Clear the `Simulink.data.dictionary.Section` and `Simulink.data.dictionary.Entry` objects.

- A model that is linked to the dictionary is open. To close this connection to the dictionary, close the model.

See Also

`Simulink.data.Dictionary` | `Simulink.data.dictionary.cleanupWorkerCache` | `Simulink.data.dictionary.getOpenDictionaryPaths` | `Simulink.data.dictionary.setupWorkerCache`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2016a

Simulink.data.dictionary.create

Create new data dictionary and create `Simulink.data.Dictionary` object

Syntax

```
dictionaryObj = Simulink.data.dictionary.create(dictionaryFile)
```

Description

`dictionaryObj = Simulink.data.dictionary.create(dictionaryFile)` creates a data dictionary file in your current working folder or in a file path you can specify in `dictionaryFile`. The function returns a `Simulink.data.Dictionary` object representing the new data dictionary.

Examples

Create New Data Dictionary and Data Dictionary Object

Create a data dictionary `myNewDictionary.sldd` in your current working folder and a `Simulink.data.Dictionary` object representing the new data dictionary. Assign the object to the variable `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.create('myNewDictionary.sldd')
```

```
myDictionaryObj =
```

```
data dictionary with properties:
```

```
    DataSourcees: {0x1 cell}
    HasUnsavedChanges: 0
    NumberOfEntries: 0
```

- “Store Data in Dictionary Programmatically”

Input Arguments

dictionaryFile — Name of new data dictionary

character vector

Name of new data dictionary, specified as a character vector containing the file name and, optionally, path of the dictionary to create. If you do not specify a path, `Simulink.data.dictionary.create` creates the new data dictionary file in your working MATLAB folder. `Simulink.data.dictionary.create` also supports file paths specified relative to your working folder.

Example: `'myDictionary.slidd'`

Example: `'C:\Users\jsmith\myDictionary.slidd'`

Example: `'..\myOtherDictionary.slidd'`

Data Types: `char`

Output Arguments

dictionaryObj — Newly created data dictionary

`Simulink.data.Dictionary` object

Newly created data dictionary, returned as a `Simulink.data.Dictionary` object.

Alternatives

You can use the Simulink Editor to create a data dictionary and link it to a model. See “Migrate Single Model to Use Dictionary” for more information.

See Also

`Simulink.data.Dictionary` | `Simulink.data.dictionary.open`

Topics

“Store Data in Dictionary Programmatically”

“What Is a Data Dictionary?”

Introduced in R2015a

Simulink.data.dictionary.getOpenDictionaryPaths

Return file names and paths of open data dictionaries

Syntax

```
openDDs = Simulink.data.dictionary.getOpenDictionaryPaths  
openDDs = Simulink.data.dictionary.getOpenDictionaryPaths(  
dictFileName)
```

Description

`openDDs = Simulink.data.dictionary.getOpenDictionaryPaths` returns the file names and paths of all data dictionaries that are open. For example, a data dictionary is open if you create objects, such as `Simulink.data.Dictionary`, that refer to the dictionary. If you open two or more dictionaries that have the same file name but different file paths, this function returns multiple file paths.

Before executing commands and functions that cannot operate when dictionaries are open, use this function to identify open dictionaries so that you can close them. For example, when you run parallel simulations as described in “Sweep Variant Control Using Parallel Simulation”, this function helps you identify open dictionaries before executing the command `Simulink.data.dictionary.cleanupWorkerCache`.

`openDDs = Simulink.data.dictionary.getOpenDictionaryPaths(dictFileName)` returns the file paths of data dictionaries that have the file name `dictFileName`. If you open two or more dictionaries that have the same file name but different file paths, you can use this syntax to return all of the file paths.

Examples

Identify and Close All Open Data Dictionaries

Open, identify, and close a data dictionary. After you close the connections to the dictionary, you can use commands and functions, such as

`Simulink.data.dictionary.cleanupWorkerCache`, that cannot operate when dictionaries are open.

At the command prompt, open a data dictionary by creating a `Simulink.data.Dictionary` object that refers to the dictionary.

```
dictObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd.sldd');
```

Display the dictionary in the Model Explorer

```
show(dictObj)
```

Identify all of the dictionaries that are open.

```
openDDs = Simulink.data.dictionary.getOpenDictionaryPaths;
```

The file path of the dictionary that you opened, `sldemo_fuelsys_dd.sldd`, appears in the cell array of character vectors `openDDs`.

Close the connection from the Model Explorer to the dictionary.

```
hide(dictObj)
```

The dictionary no longer appears as a node in the **Model Hierarchy** pane of the Model Explorer.

Close the connection from the `Simulink.data.Dictionary` object to the dictionary.

```
close(dictObj)
clear dictObj
```

- “Store Data in Dictionary Programmatically”

Input Arguments

dictFileName — File name of target data dictionary or dictionaries

character vector

File name of target data dictionary or dictionaries, specified as a character vector. Use the file extension `sldd`.

Example: `'myDict.sldd'`

Data Types: char

Output Arguments

openDDs — File names and paths of open data dictionaries

cell array of character vectors

File names and paths of open data dictionaries, returned as a cell array of character vectors.

Tips

A data dictionary is open if any of these conditions are true:

- The dictionary appears as a node in the **Model Hierarchy** pane of the Model Explorer. To close this connection to the dictionary, right-click the node in Model Explorer and select **Close**. Alternatively, use the `hide` method of a `Simulink.data.Dictionary` object.
- You created an object of any of these classes that refer to the dictionary:
 - `Simulink.data.Dictionary`
 - `Simulink.data.dictionary.Section`
 - `Simulink.data.dictionary.Entry`

To close these connections to the dictionary, use the `close` method of the `Simulink.data.Dictionary` object or clear the object. Clear the `Simulink.data.dictionary.Section` and `Simulink.data.dictionary.Entry` objects.

- A model that is linked to the dictionary is open. To close this connection to the dictionary, close the model.

See Also

`Simulink.data.Dictionary` | `Simulink.data.dictionary.cleanupWorkerCache` | `Simulink.data.dictionary.closeAll` | `Simulink.data.dictionary.setupWorkerCache`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2016a

Simulink.data.dictionary.open

Open data dictionary for editing

Syntax

```
dictionaryObj = Simulink.data.dictionary.open(dictionaryFile)
```

Description

`dictionaryObj = Simulink.data.dictionary.open(dictionaryFile)` opens the specified data dictionary and returns a `Simulink.data.Dictionary` object representing an existing data dictionary identified by its file name and, optionally, file path with `dictionaryFile`.

Make sure any dictionaries referenced by the target dictionary are on the MATLAB path.

Examples

Open Existing Data Dictionary

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd')
```

```
myDictionaryObj =
```

```
Dictionary with properties:
```

```
    DataSourcees: {'myRefDictionary_ex_API.sldd'}  
    HasUnsavedChanges: 0  
    NumberOfEntries: 4
```

- “Store Data in Dictionary Programmatically”

Input Arguments

dictionaryFile — Target data dictionary

character vector

Target data dictionary, specified as a character vector containing the file name and, optionally, path of the dictionary. If you do not specify a path, `Simulink.data.dictionary.open` searches the MATLAB path for the specified file. `Simulink.data.dictionary.open` also supports paths specified relative to the MATLAB working folder.

Example: `'myDictionary_ex_API.sldd'`

Example: `'C:\Users\jsmith\myDictionary_ex_API.sldd'`

Example: `'..\myOtherDictionary.sldd'`

Data Types: `char`

See Also

`Simulink.data.Dictionary` | `Simulink.data.dictionary.create` | `show`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

Simulink.data.dictionary.setupWorkerCache

Enable parallel simulation with data dictionary

Syntax

```
Simulink.data.dictionary.setupWorkerCache
```

Description

`Simulink.data.dictionary.setupWorkerCache` prepares the workers in a parallel pool for simulating a model that is linked to a data dictionary. Use this function in a `spmd` block, prior to starting a `parfor` block, to provide the workers in a parallel pool a way to safely interact with a single data dictionary.

During parallel simulation of a model that is linked to a data dictionary, you can allow each worker to access and modify the data in the dictionary independently of other workers. `Simulink.data.dictionary.setupWorkerCache` temporarily provides each worker in the pool with its own data dictionary cache, allowing the workers to use the data in the dictionary without permanently changing it.

You must have a Parallel Computing Toolbox license to perform parallel simulation using a `parfor` block.

Examples

Sweep Variant Control Using Parallel Simulation

To use parallel simulation to sweep a variant control (a `Simulink.Parameter` object whose value influences the variant condition of a `Simulink.Variant` object) that you store in a data dictionary, use this code as a template. Change the names and values of the model, data dictionary, and variant control to match your application.

To sweep block parameter values or the values of workspace variables that you use to set block parameters, use `Simulink.SimulationInput` objects instead of the

programmatic interface to the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values”.

You must have a Parallel Computing Toolbox license to perform parallel simulation.

```
% For convenience, define names of model and data dictionary
model = 'mySweepMdl';
dd = 'mySweepDD.slidd';

% Define the sweeping values for the variant control
CtrlValues = [1 2 3 4];

% Grant each worker in the parallel pool an independent data dictionary
% so they can use the data without interference
spmd
    Simulink.data.dictionary.setupWorkerCache
end

% Determine the number of times to simulate
numberOfSims = length(CtrlValues);

% Prepare a nondistributed array to contain simulation output
simOut = cell(1,numberOfSims);

parfor index = 1:numberOfSims
    % Create objects to interact with dictionary data
    % You must create these objects for every iteration of the parfor-loop
    dictObj = Simulink.data.dictionary.open(dd);
    sectObj = getSection(dictObj, 'Design Data');
    entryObj = getEntry(sectObj, 'MODE');
    % Suppose MODE is a Simulink.Parameter object stored in the data dictionary

    % Modify the value of MODE
    temp = getValue(entryObj);
    temp.Value = CtrlValues(index);
    setValue(entryObj, temp);

    % Simulate and store simulation output in the nondistributed array
    simOut{index} = sim(model);

    % Each worker must discard all changes to the data dictionary and
    % close the dictionary when finished with an iteration of the parfor-loop
    discardChanges(dictObj);
    close(dictObj);
end
```

```
end

% Restore default settings that were changed by the function
% Simulink.data.dictionary.setupWorkerCache
% Prior to calling cleanupWorkerCache, close the model

spmd
    bdclose(model)
    Simulink.data.dictionary.cleanupWorkerCache
end
```

Note If data dictionaries are open, you cannot use the command `Simulink.data.dictionary.cleanupWorkerCache`. To identify open data dictionaries, use `Simulink.data.dictionary.getOpenDictionaryPaths`.

See Also

`Simulink.data.dictionary.cleanupWorkerCache` | `parfor` | `spmd`

Topics

“Store Data in Dictionary Programmatically”

“What Is a Data Dictionary?”

“Run Code on Parallel Pools” (Parallel Computing Toolbox)

Introduced in R2015a

Simulink.data.evalinGlobal

Evaluate MATLAB expression in context of Simulink model

Syntax

```
returnValue = Simulink.data.evalinGlobal(modelName,expression)
```

Description

```
returnValue = Simulink.data.evalinGlobal(modelName,expression)
```

evaluates the MATLAB expression `expression` in the context of the Simulink model `modelName` and returns the values returned by `expression`. `evalinGlobal` evaluates `expression` in the Design Data section of the data dictionary that is linked to the target model or in the MATLAB base workspace if the target model is not linked to any data dictionary.

Examples

Evaluate MATLAB Expression in Model With or Without Data Dictionary

Evaluate the MATLAB expression `myNewVariable = 237;` in the context of the model `vdp`, which is not linked to any data dictionary. `myNewVariable` appears as a variable in the MATLAB base workspace.

```
Simulink.data.evalinGlobal('vdp','myNewVariable = 237;')
```

Evaluate the MATLAB expression `myNewEntry = true;` in the context of the model `sldemo_fuelsys_dd_controller`, which is linked to the data dictionary `sldemo_fuelsys_dd_controller.sldd`. `myNewEntry` appears as an entry in the Design Data section of the dictionary.

```
Simulink.data.evalinGlobal('sldemo_fuelsys_dd_controller',...  
'myNewEntry = true;')
```

Confirm the creation of the entry `myNewEntry` in the data dictionary `sldemo_fuelsys_dd_controller.sldd` by viewing the dictionary in Model Explorer.

```
myDictionaryObj = Simulink.data.dictionary.open(...
'sldemo_fuelsys_dd_controller.sldd');
show(myDictionaryObj)
```

- “Store Data in Dictionary Programmatically”

Input Arguments

modelName — Name of target Simulink model

character vector

Name of target Simulink model, specified as a character vector.

Example: 'myTestModel'

Data Types: char

expression — MATLAB expression to evaluate

character vector

MATLAB expression to evaluate, specified as a character vector.

Example: 'a = 5.3'

Example: 'whos'

Example: 'CurrentSpeed.Value = 290.73'

Data Types: char

Output Arguments

returnValue — Value returned by specified expression

valid entry or variable value

Value returned by the specified MATLAB expression.

Tips

- `evalinGlobal` helps you transition Simulink models to the use of data dictionaries. You can use the function to manipulate model variables before and after linking a model to a data dictionary.

See Also

`Simulink.data.assigninGlobal` | `Simulink.data.existsInGlobal` | `evalin`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

Simulink.data.existsInGlobal

Check existence of variable in context of Simulink model

Syntax

```
varExists = Simulink.data.existsInGlobal(modelName,varName)
```

Description

`varExists = Simulink.data.existsInGlobal(modelName,varName)` returns an indication of the existence of a variable or data dictionary entry `varName` in the context of the Simulink model `modelName`. `Simulink.data.existsInGlobal` searches the Design Data section of the data dictionary that is linked to the target model or the MATLAB base workspace if the target model is not linked to any data dictionary.

Examples

Determine Existence of Variable in Model With or Without Data Dictionary

Determine the existence of a variable `PressVect` in the context of the Simulink model `vdp.slx`, which is not linked to any data dictionary.

```
Simulink.data.existsInGlobal('vdp','PressVect')
```

```
ans =
```

```
0
```

Because `vdp.slx` is not linked to any data dictionary, `existsInGlobal` searches only in the MATLAB base workspace for `PressVect`.

Determine the existence of a variable `PressVect` in the context of the Simulink model `sldemo_fuelsys_dd_controller.slx`, which is linked to the data dictionary `sldemo_fuelsys_dd_controller.sldd`.

```
Simulink.data.existsInGlobal('sldemo_fuelsys_dd_controller','PressVect')  
  
ans =  
  
    1
```

Because `sldemo_fuelsys_dd_controller.slx` is linked to the data dictionary `sldemo_fuelsys_dd_controller.sldd`, `existsInGlobal` searches for `PressVect` only in the Design Data section of the dictionary.

- “Store Data in Dictionary Programmatically”

Input Arguments

modelName — Name of target Simulink model

character vector

Name of target Simulink model, specified as a character vector.

Example: `'myTestModel'`

Data Types: `char`

varName — Name of target variable or data dictionary entry

character vector

Name of target variable or data dictionary entry, specified as a character vector.

Example: `'myTargetVariable'`

Data Types: `char`

Output Arguments

varExists — Indication of existence of target variable or data dictionary entry

1 | 0

Indication of existence of target variable or data dictionary entry, returned as 1 to indicate existence or 0 to indicate absence.

Tips

- `existsInGlobal` helps you transition Simulink models to the use of data dictionaries. You can use the function to find model variables before and after linking a model to a data dictionary.

Alternatives

You can use Model Explorer to search a data dictionary or any workspace for entries or variables.

See Also

`Simulink.data.assignInGlobal` | `Simulink.data.evalInGlobal` | `exist`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

Simulink.data.getEnumTypeInfo

Get information about enumerated data type

Syntax

```
information = Simulink.data.getEnumTypeInfo(enumTypeName,  
infoRequest)
```

Description

```
information = Simulink.data.getEnumTypeInfo(enumTypeName,  
infoRequest) returns information about an enumerated data type enumTypeName.
```

Use this function only to return information about an enumerated data type. To customize an enumerated data type, for example, by specifying a default enumeration member or by controlling the scope of the type definition in generated code, see “Customize Simulink Enumeration”.

Examples

Return Default Value of Enumerated Data Type

Get the default enumeration member of an enumerated data type `LEDcolor`. Suppose `LEDcolor` defines two enumeration members, `GREEN` and `RED`, and uses `GREEN` as the default member.

```
Simulink.data.getEnumTypeInfo('LEDcolor','DefaultValue')  
  
ans =  
  
    GREEN
```

Get Scope of Enumerated Data Type Definition in Generated Code

For an enumerated data type `LEDcolor`, find out if generated code exports or imports the definition of the type to or from a header file.

```
Simulink.data.getEnumTypeInfo('LEDcolor','DataScope')
Simulink.data.getEnumTypeInfo('LEDcolor','HeaderFile')
```

```
ans =
```

```
Auto
```

```
ans =
```

```
''
```

Because `DataScope` is `'Auto'` and `HeaderFile` is empty, generated code defines the enumerated data type `LEDcolor` in the header file `model_types.h` where `model` is the name of the model used to generate code.

- “Customize Simulink Enumeration”

Input Arguments

enumTypeName — Name of target enumerated data type

character vector

Name of the target enumerated data type, specified as a character vector.

Example: `'myFirstEnumType'`

Data Types: `char`

infoRequest — Information to return

valid character vector

Information to return, specified as one of the character vector options in the table.

Specified value	Information returned	Example return value
'DefaultValue'	The default enumeration member, returned as an instance of the enumerated data type.	enumMember1
'Description'	The custom description of this data type, returned as a character vector. Returns an empty character vector if a description was not specified for the type.	'My first enum type.'
'HeaderFile'	The name of the custom header file that defines the data type in generated code, returned as a character vector. Returns an empty character vector if a header file was not specified for the type.	'myEnumType.h'
'DataScope'	Indication whether generated code imports or exports the definition of the data type. A return value of 'Auto' indicates generated code defines the type in the header file <i>model_types.h</i> or imports the definition from the header file identified by <i>HeaderFile</i> . A return value of 'Exported' or 'Imported' indicates generated code exports or imports the definition to or from the header file identified by <i>HeaderFile</i> .	'Exported'
'StorageType'	The integer data type used by generated code to store the numeric values of the enumeration members, returned as a character vector. Returns 'int' if you did not specify a storage type for the enumerated type, in which case generated code uses the native integer type of the hardware target.	'int32'
'AddClassNameToEnumNames'	Indication whether generated code prefixes the names of enumeration members with the name of the data type. Returned as true or false.	true

See Also

`Simulink.defineIntEnumType`

Topics

“Customize Simulink Enumeration”

“Simulink Enumerations”

Introduced in R2014b

Simulink.defineIntEnumType

Define enumerated data type

Syntax

```
Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues)
Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues,
'Description', ClassDesc)
Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues,
'DefaultValue', DefValue)
Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues,
'DataScope', ScopeSelection)
Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues,
'HeaderFile', FileName)
Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues,
'AddClassNameToEnumNames', Flag)
Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues,
'StorageType', DataType)
```

Description

`Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues)` defines an enumeration named *ClassName* with enumeration values specified with *CellofEnums* and underlying numeric values specified by *IntValues*.

`Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues, 'Description', ClassDesc)` defines the enumeration with a description (character vector).

`Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues, 'DefaultValue', DefValue)` defines a default value for the enumeration, which is one of the character vectors you specify for *CellofEnums*.

`Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues, 'DataScope', ScopeSelection)` specifies whether the data type definition should be imported from, or exported to, a header file during code generation.

`Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues, 'HeaderFile', FileName)` specifies the name of a header file containing the enumeration class definition for use in code generated from a model.

`Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues, 'AddClassNameToEnumNames', Flag)` specifies whether the code generator applies the class name as a prefix to the enumeration values that you specify for *CellofEnums*. For *Flag*, specify `true` or `false`. For example, if you specify `true`, the code generator would use `BasicColors.Red` instead of `Red` to represent an enumerated value.

`Simulink.defineIntEnumType(ClassName, CellofEnums, IntValues, 'StorageType', DataType)` specifies the data type used to store the enumerations' underlying integer values in code generated from a model.

Input Arguments

ClassName

The name of the enumerated data type.

CellofEnums

A cell array of character vectors that defines the enumerations for the data type.

IntValues

An array of numeric values that correspond to enumerations of the data type.

'Description', ClassDesc

Specifies a character vector that describes the enumeration data type.

'DefaultValue', DefValue

Specifies the default enumeration value.

'HeaderFile', FileName

Specifies a character vector naming the header file that is to contain the data type definition.

By default, the generated `#include` directive uses the preprocessor delimiter `"` instead of `<` and `>`. To generate the directive `#include <myTypes.h>`, specify `FileName` as `'<myTypes.h>'`.

'DataScope', 'Auto' | 'Exported' | 'Imported'

Specifies whether the data type definition should be imported from, or exported to, a header file during code generation.

Value	Action
Auto (default)	<p>If no value is specified for <code>Headerfile</code>, export the type definition to <code>model_types.h</code>, where <code>model</code> is the model name.</p> <p>If a value is specified for <code>Headerfile</code>, import the data type definition from the specified header file.</p>
Exported	<p>Export the data type definition to a header file.</p> <p>If no value is specified for <code>Headerfile</code>, the header file name defaults to <code>type.h</code>, where <code>type</code> is the data type name.</p>
Imported	<p>Import the data type definition from a header file.</p> <p>If no value is specified for <code>Headerfile</code>, the header file name defaults to <code>type.h</code>, where <code>type</code> is the data type name.</p>

'AddClassNameToEnumNames', Flag

A logical flag that specifies whether code generator applies the class name as a prefix to the enumerations.

'StorageType', DataType

Specifies a character vector that identifies the data type used to store the enumerations' underlying integer values in generated code. The following data types are supported: `'int8'`, `'int16'`, `'int32'`, `'uint8'`, or `'uint16'`.

Examples

Assume an external data dictionary includes the following enumeration:

```
BasicColors.Red(0), BasicColors.Yellow(1), BasicColors.Blue(2)
```

Import the enumeration class definition into the MATLAB workspace while specifying `int16` as the underlying integer data type for generated code:

```
Simulink.defineIntEnumType('BasicColors', ...  
    {'Red', 'Yellow', 'Blue'}, ...  
    [0;1;2], ...  
    'Description', 'Basic colors', ...  
    'DefaultValue', 'Blue', ...  
    'HeaderFile', 'mybasiccolors.h', ...  
    'DataScope', 'Exported', ...  
    'AddClassNameToEnumNames', true, ...  
    'StorageType', 'int16');
```

See Also

enumeration

Topics

“Import Enumerations Defined Externally to MATLAB”

“Define Simulink Enumerations”

Introduced in R2010b

Simulink.defaultModelTemplate

Set or get default model template

Syntax

```
Simulink.defaultModelTemplate(templatename)  
templatepath = Simulink.defaultModelTemplate
```

Description

`Simulink.defaultModelTemplate(templatename)` sets the template file specified by `templatename` as the default model template to use for new models. This setting is persistent between Simulink sessions.

`templatepath = Simulink.defaultModelTemplate` gets the full path to the current default model template.

Examples

Set the default model template

```
Simulink.defaultModelTemplate('simple_simulation.sltx')
```

Get the default model template

```
mydefaulttemplate = Simulink.defaultModelTemplate
```

Clear and restore the default model template

Use `set_param` to set a root block diagram parameter. This clears the default template so that new models will inherit this property of the root block diagram, and warns.

```
set_param(0, 'StopTime', '99');
```

Restore the default template.

```
Simulink.defaultModelTemplate('$restore');
```

- “Create and Open Models”
- “Create a Template from a Model”
- “Using Templates to Create Standard Project Settings”

Input Arguments

templatename — Template file name

character vector

Template file name, specified as a character vector. If the template is not on the MATLAB path, specify the fully-qualified path to the template file and *.sltx extension.

Example: \\Home\username\Documents\MATLAB\template.sltx

Data Types: char

Output Arguments

templatepath — Template path

character vector

Template path, specified as a character vector, showing the full path to the current default model template.

See Also

[Simulink.createFromTemplate](#) | [Simulink.exportToTemplate](#) |
[Simulink.findTemplates](#) | [new_system](#)

Topics

“Create and Open Models”

“Create a Template from a Model”

“Using Templates to Create Standard Project Settings”

Introduced in R2016b

Simulink.exportToTemplate

Create template from model or project

Syntax

```
templatefile = Simulink.exportToTemplate(obj,templatename)
templatefile = Simulink.exportToTemplate(obj,templatename,
Name,Value)
```

Description

`templatefile = Simulink.exportToTemplate(obj,templatename)` creates a template file (*templatename.sltx*) from a model or project specified by `obj`.

If you have project templates created in R2014a or earlier (.zip files), use `Simulink.exportToTemplate` to upgrade them to .sltx files, then you can use them in the start page.

`templatefile = Simulink.exportToTemplate(obj,templatename, Name,Value)` specifies additional template options as one or more Name, Value pair arguments.

Examples

Create a Template From a Model

Open the `vdp` model and create a template from it.

```
vdp
myvdptemplate = Simulink.exportToTemplate(bdroot,'vdptemplate')
```

Create a Template From a Model and Specify Description

Open the vdp model and create a template from it, specifying a description.

```
vdp  
myvdptemplate = Simulink.exportToTemplate (bdroot, 'vdptemplate', 'Description', 'Use this
```

- “Create and Open Models”
- “Create a Template from a Model”
- “Using Templates to Create Standard Project Settings”

Input Arguments

obj — Model, library, or project

character vector | numeric handle | slproject.ProjectManager

Model, library, or project, specified by name or numeric handle, or a slproject.ProjectManager object returned by the simulinkproject function.

Data Types: double | char

templatename — Template file name

character vector

Template file name, specified as a character vector that can optionally include the fully-qualified path to a template file and *.sltx extension.

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: Title, 'My Project Template'

Group — Group of template

character vector

Group of template, specified as a character vector. On the Start Page, templates are shown under group headings.

Example: 'Simscape'

Data Types: char

Author — Author of template

character vector

Author of template, specified as a character vector.

Data Types: char

Description — Description of template

character vector

Description of template, specified as a character vector.

Data Types: char

ThumbnailFile — Thumbnail image file name

character vector

Thumbnail image file name, specified as a character vector.

Data Types: char

Title — Title of model or project template

character vector

Title of template, specified as a character vector. On the Start Page, the templates titles are shown on the tiles. The title can be different from the file name, and you can use any characters in the title. The default value is the name of the model or project.

Example: 'My Project Template'

Data Types: char

Output Arguments

templatefile — Template file

character vector

Template file, returned as *templatename.sltx* file.

See Also

`Simulink.createFromTemplate` | `Simulink.defaultModelTemplate` |
`Simulink.findTemplates`

Topics

“Create and Open Models”

“Create a Template from a Model”

“Using Templates to Create Standard Project Settings”

Introduced in R2016a

Simulink.exportToVersion

Export model or library for use in previous version of Simulink

Syntax

```
exported_file = Simulink.exportToVersion(modelname,target_filename,
version)
exported_file = Simulink.exportToVersion(modelname,target_filename,
version,Name,Value)
```

Description

`exported_file = Simulink.exportToVersion(modelname,target_filename,version)` exports the model or library `modelname` to a file named `target_filename` in a format that the specified previous Simulink version can load.

If the system contains functionality not supported by the specified Simulink software version, the command removes the functionality and replaces any unsupported blocks with empty masked subsystem blocks colored yellow. As a result, the converted system may generate different results.

The `save_system ExportToVersion` option is a legacy option for this functionality that is also supported.

`exported_file = Simulink.exportToVersion(modelname,target_filename,version,Name,Value)` specifies additional options as one or more Name, Value pair arguments.

Examples

Export a Model to a Previous Version

Get the current top-level system and export it.

```
Simulink.exportToVersion(bdroot, 'mymodel.slx', 'R2014b');
```

Export a Model to a Previous Version and Break Links

Get the current top-level system and export it, replacing links to library blocks with copies of the library blocks in the saved file.

```
Simulink.exportToVersion(bdroot, 'mymodel.slx', 'R2014b', 'BreakUserLinks', true);
```

- “Save a Model”

Input Arguments

modelName — Model to export

character vector

Model to export, specified as a character vector, without any file extension. The model must be loaded and unmodified. The target file must not be the same as the model file.

Data Types: char

target_filename — Exported file name

character vector

Exported file name, specified as a character vector. The target file must not be the same as the model file.

Example: 'mymodel.slx'

Data Types: char

version — MATLAB release name

'R2012A' | 'R2014A_MDL' | 'R2016B_SLX' | ...

MATLAB release name, specified as a character vector, which specifies a previous Simulink version. `Simulink.exportToVersion` exports the system to a format that the specified previous Simulink version can load. You cannot export to your current version. These version names are not case sensitive.

To export to Release 2012a and later, you can specify model file format as SLX or MDL using the suffix `_MDL` or `_SLX`. If you do not specify a format, you export your default model file format.

If you use the Export to Previous Version dialog box instead of `Simulink.exportToVersion`, then the **Save as type** list supports 7 years of previous releases.

Example: 'R2015B'

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

AllowPrompt — Allow prompt or message dialog box

false (default) | true | 'on' | 'off'

Allow prompt or message dialog box, specified by a logical value that indicates whether to display any output prompt or message in a dialog box or only messages at the command line. For example, prompts to make files writable, or messages about exported versions. If you want to allow prompts, then set to `true`, or `on`.

BreakUserLinks — Break user-defined links

false (default) | true | 'on' | 'off'

Break user-defined links, specified by a logical value that indicates whether the function replaces links to user-defined library blocks with copies of the library blocks in the saved file.

BreakToolboxLinks — Break all toolbox links

false (default) | true | 'on' | 'off'

Break all toolbox links, specified by a logical value that indicates whether the function replaces links to built-in MathWorks library blocks with copies of the library blocks in

the saved file. The 'BreakToolboxLinks' option affects Simulink library blocks and blocks from any other libraries supplied with MathWorks toolboxes or blocksets.

Note The 'BreakToolboxLinks' option can result in compatibility issues when upgrading to newer versions of Simulink software. For example:

- Any masks on top of library links to Simulink S-functions will not upgrade to the new version of the S-function.
- Any library links to masked subsystems in a Simulink library will not upgrade to the new subsystem behavior.
- Any broken links prevent the automatic library forwarding mechanism from upgrading the link.

If you have saved a model with broken links to built-in libraries, use the Upgrade Advisor to scan the model for out-of-date blocks and upgrade the Simulink blocks to their current versions.

Output Arguments

exported_file — Exported file

character vector

Exported file, returned in the format that the specified previous Simulink version can load.

See Also

`save_system`

Topics

“Save a Model”

Introduced in R2016a

Simulink.findTemplates

Find model or project templates with specified properties

Syntax

```
filename = Simulink.findTemplates(templatename)
filename = Simulink.findTemplates(templatename,Name,Value)
[filename,info] = Simulink.findTemplates(templatename)
```

Description

`filename = Simulink.findTemplates(templatename)` returns the names and `TemplateInfo` objects for all matching templates that include `templatename`.

`filename = Simulink.findTemplates(templatename,Name,Value)` also specifies additional template properties as one or more `Name, Value` pair arguments.

`[filename,info] = Simulink.findTemplates(templatename)` returns the names and `TemplateInfo` objects for all matching templates.

Examples

Find a Particular Template

Get the full path to the default model template.

```
filename = Simulink.findTemplates('factory_default_model');
```

Find All Templates With Specified Folders or Authors

Get all templates inside folders called `work`.

```
filename = Simulink.findTemplates('work/')
```

Get all templates for which the `Author` property includes the character vector `Smith`.

```
filename = Simulink.findTemplates('*', 'Author', 'Smith')
```

Find All DSP Templates and Get TemplateInfo Objects

Get the paths to all DSP model templates, and `sltemplate.TemplateInfo` objects for each of them.

```
[filename,info] = Simulink.findTemplates('dsp*', 'Type', 'Model');
```

- “Create and Open Models”
- “Create a Template from a Model”
- “Using Templates to Create Standard Project Settings”

Input Arguments

templatename — Template name

character vector

Template name, specified as a character vector containing a portion of a file name, which can contain the wildcard asterisk character “*”.

Example:

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

You can specify regular expressions for any of the `Value` character vectors, e.g., including the wildcard asterisk character “*”.

Example: 'Author', '*son'

Type — Model, library, or project

'Model' | 'Library' | 'Project'

Model, library, or project template type, specified as a character vector for model, library, or project.

Example: 'Simscape'

Data Types: char

Title — Title of template

character vector

Title of template, specified as a character vector.

Example: 'Simscape'

Data Types: char

Group — Group of template

character vector

Group of template, specified as a character vector. On the Start Page, templates are shown under group headings.

Example: 'Simscape'

Data Types: char

Author — Author of template

character vector

Author of template, specified as a character vector.

Data Types: char

Description — Description of template

character vector

Description of template, specified as a character vector.

Data Types: char

Output Arguments

filename — Template name

character vector | cell array of character vectors

Template names of matching templates, returned as character vectors.

info — Template information

template info objects | array of template info objects

Template information of matching templates, returned as `sltemplate.TemplateInfo` objects.

See Also

`Simulink.createFromTemplate` | `Simulink.exportToTemplate`

Topics

“Create and Open Models”

“Create a Template from a Model”

“Using Templates to Create Standard Project Settings”

Introduced in R2016a

Simulink.findVars

Analyze relationship between variables and blocks in models

Syntax

```
[variables] = Simulink.findVars(context)
[variables] = Simulink.findVars(context,variablefilter)
[variables] = Simulink.findVars( ____,Name,Value)
```

Description

`[variables] = Simulink.findVars(context)` finds and returns variables that are used in the blocks and models specified by `context`, including subsystems and referenced models. The function returns an empty vector if `context` does not use any variables.

`[variables] = Simulink.findVars(context,variablefilter)` finds only the variables or enumerated types that are specified by `variablefilter`. For example, use this syntax to determine where a variable is used in a model.

`[variables] = Simulink.findVars(____,Name,Value)` finds variables with additional options specified by one or more `Name,Value` pair arguments. For example, you can search for unused variables. You can also search for enumerated data types that are used in `context`, in addition to variables.

Examples

Variables in Use in a Model

Find variables used by `MyModel`.

```
variables = Simulink.findVars('MyModel');
```

Specific Variable in Use in a Model

Find all uses of the base workspace variable `k` by `MyModel`. Use the cached results to avoid compiling `MyModel`.

```
variables = Simulink.findVars('MyModel', 'Name', 'k',  
'SearchMethod', 'cached', 'SourceType', 'base workspace');
```

Regular Expression Matching

Find all uses of a variable whose name matches the regular expression `^trans`.

```
variables = Simulink.findVars('MyModel', 'Regexp', 'on',  
'Name', '^trans');
```

Variables Common to Two Models

Given two models, find the variables used by the first model, the second, and both

```
model1Vars = Simulink.findVars('model1');  
model2Vars = Simulink.findVars('model2');  
commonVars = intersect(model1Vars, model2Vars);
```

Variables Not Used in a Model

Find the variables that are defined in the model workspace of `MyModel` but that are not used by the model.

```
unusedVars = Simulink.findVars('MyModel', 'FindUsedVars', 'off',  
'SourceType', 'model workspace');
```

Specific Variable Not Used in a Model

Determine if the base workspace variable `k` is not used by `MyModel`.

```
varObj = Simulink.VariableUsage('k', 'base workspace');
unusedVar = Simulink.findVars('MyModel', varObj,
    'FindUsedVars', 'off');
```

Variables Used by a Block

Find the variables that are used by the block `Gain1` in `MyModel`.

```
variables = Simulink.findVars('MyModel',
    'Users', 'MyModel/Gain1');
```

Variables Used in a Model Reference Hierarchy

Find the variables that are used in a model reference hierarchy. Begin the search with the model `MyNestedModel`, and search the entire hierarchy below `MyNestedModel`.

```
variables = Simulink.findVars('MyNestedModel', 'SearchReferencedModels', 'on');
```

Variables and Enumerated Types Used in a Model

Find variables and enumerated types that are used in `MyModel`.

```
varsAndEnumTypes = Simulink.findVars('MyModel', 'IncludeEnumTypes', 'on');
```

- “Search Using Model Explorer”

Input Arguments

context — Models and blocks to search

character vector | cell array of character vectors

Models and blocks to search, specified as a character vector or a cell array of character vectors. You can specify `context` in one of the following ways:

- The name of a model. For example, (`'vdp'`) specifies the model `vdp.slx`.
- The name or path of a block or masked block. For example, (`'vdp/Gain1'`) specifies a block named `Gain1` at the root level of the model `vdp.slx`.

- A cell array of model or block names.

Data Types: `char` | `cell`

variablefilter — Specific variables to find

array of `Simulink.VariableUsage` objects

Specific variables to find, specified as an array of `Simulink.VariableUsage` objects. Each `Simulink.VariableUsage` object identifies a variable to find.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'FindUsedVars', 'off'`

FindUsedVars — Find variables that are used or not used

`'on'` (default) | `'off'`

Flag to find variables that are explicitly used or not used, specified as the comma-separated pair consisting of `'FindUsedVars'` and `'on'` or `'off'`. If you specify `FindUsedVars` as `'off'`, the function finds variables that are not used in context but that are defined in the workspace specified by `SourceType`.

Example: `'FindUsedVars', 'off'`

IncludeEnumTypes — Find enumerated types that are used

`'off'` (default) | `'on'`

Flag to find enumerated data types that are used, specified as the comma-separated pair consisting of `'IncludeEnumTypes'` and `'on'` or `'off'`. The function finds enumerated types that are used explicitly in context as well as types that define variables that are used in context.

If you specify `SourceType` as `'base workspace'`, `'model workspace'`, or `'mask workspace'`, the function does not report enumerated types because those sources cannot define enumerated types.

You cannot find unused enumerated types by specifying `FindUsedVars` as `'off'`.

Example: `'IncludeEnumTypes', 'on'`

RegExp — Enable regular expression matching

`'off'` (default) | `'on'`

Flag to enable regular expression matching for input arguments, specified as the comma-separated pair consisting of `'RegExp'` and `'on'`. You can match only input arguments that have character vector values.

Example: `'RegExp', 'on'`

SearchMethod — Compile status

`'compiled'` (default) | `'cached'`

Compile status, specified as the comma-separated pair consisting of `'SearchMethod'` and one of these values:

- `'compiled'` — Return up-to-date results by compiling every model in the search context before search.
- `'cached'` — Return quicker results by using results cached during the previous compile.

Example: `'SearchMethod', 'compiled'`

SearchReferencedModels — Enable search in referenced models

`'off'` (default) | `'on'`

Flag to enable search in referenced models, specified as the comma-separated pair consisting of `'SearchReferencedModels'` and `'on'`.

Example: `'SearchReferencedModels', 'on'`

Name — Name of a variable or enumerated type to search for character vector

Name of a variable or enumerated data type to search for, specified as the comma-separated pair consisting of `'Name'` and a character vector.

Example: `'Name', 'trans'`

Data Types: `char`

SourceType — Workspace or source defining the variables or enumerated types

character vector

Workspace or source defining the variables, specified as the comma-separated pair of 'SourceType' and one of these options:

- 'base workspace'
- 'model workspace'
- 'mask workspace'
- 'data dictionary'

The function filters results for variables that are defined in the specified source.

Example: 'SourceType', 'base workspace'

If you search for enumerated data types by specifying 'IncludeEnumTypes' as 'on', 'SourceType' represents the way an enumerated type is defined. You can specify one of these options:

- 'MATLAB file'
- 'dynamic class'
- 'data dictionary'

The function filters results for enumerated types that are defined in the specified source.

Example: 'SourceType', 'MATLAB file'

If you do not specify SourceType, the function does not filter results by source.

Users — Name of block to search for variables

character vector

Name of specific block to search for variables, specified as the comma-separated pair consisting of 'Users' and a character vector.

To search a set of specific blocks, enable regular expression matching by specifying RegExp as 'on' and use regular expressions in the character vector. For example, you can specify 'Users', 'MyModel/Gain*' to search all blocks in MyModel whose names begin with Gain.

Example: 'Users', 'MyModel/Gain1'

Example: 'Users', 'MyModel/mySubsystem/Gain2'

Example: 'Users', 'MyModel/Gain*'

Limitations

`Simulink.findVars` does not work with these constructs:

- MATLAB code in scripts and initialization and callback functions
- Libraries and blocks in libraries
- Variables in MATLAB Function blocks, except for input arguments

However, `Simulink.findVars` can find enumerated types anywhere they are used in MATLAB Function blocks.

- Calls directly to MATLAB from the Stateflow action language
- S-functions that use data type variables registered using `ssRegisterDataType`

To make the variables searchable, use `ssRegisterTypeFromNamedObject` instead.

- Variables referenced by machine-parented data in Stateflow

`Simulink.findVars` discovers variable usage in inactive subsystem variants only if you select **Analyze all choices during update diagram and generate preprocessor conditionals** in the Variant Subsystem block dialog box. If you do not select this check box, the function does not discover variable usage in inactive variants.

See Also

`Simulink.VariableUsage` | `find_system` | `intersect`

Topics

“Search Using Model Explorer”

“Model Exploration”

“Variables”

Introduced in R2010a

Simulink.getFileChecksum

Checksum of file

Syntax

```
checksum = Simulink.getFileChecksum(filename)
```

Description

`checksum = Simulink.getFileChecksum(filename)` returns the checksum of the specified file, using the MD5 checksum algorithm. Use the checksum to see if the file has changed compared to a previous checksum. You can use checksums as part of an audit trail.

Use `Simulink.getFileChecksum` to get a checksum for any file. If the file contents do not change from one checksum to the next, the checksum from `Simulink.getFileChecksum` stays the same. Otherwise, the checksum is different with each change to the file contents.

For functional information on a model, use `Simulink.BlockDiagram.getChecksum` instead. `Simulink.BlockDiagram.getChecksum` looks at the functional aspect of the model. If the functional aspect doesn't change, then `Simulink.BlockDiagram.getChecksum` returns the same checksum.

For example, if you moved a block, the file contents are different (measured by `Simulink.getFileChecksum`) but the function of the model is unchanged (measured by `Simulink.BlockDiagram.getChecksum`).

Examples

Get Checksum of a File

Use `fullfile` to specify a full path to a file and get the checksum.

```
filechecksum = Simulink.getFileChecksum(fullfile(matlabroot, 'toolbox', ...  
'matlab', 'demos', 'gatlin.mat'));
```

Input Arguments

filename — File name to get checksum for
file of any type

File name to get checksum for, with file extension and optional full path. Use `fullfile` to specify a full path to a file, or use the form 'C:\Work\filename.mat'.

Example: 'lengthofline.m'

Data Types: char

Output Arguments

checksum — Checksum value
character vector

Checksum value in a 32-character vector.

See Also

`Simulink.BlockDiagram.getChecksum` | `Simulink.SubSystem.getChecksum`

Introduced in R2014b

Simulink.getSuppressedDiagnostics

Return `Simulink.SuppressedDiagnostic` objects associated with a block, subsystem, or model

Syntax

```
suppressed_diagnostics = Simulink.getSuppressedDiagnostics(source)
```

Description

`suppressed_diagnostics = Simulink.getSuppressedDiagnostics(source)` returns an array of `Simulink.SuppressedDiagnostic` objects that are associated with the specified source.

Examples

Get All `Simulink.SuppressedDiagnostic` Objects on a Specific Block

Using the model from “Suppress Diagnostic Messages Programmatically”, get all suppressed diagnostics associated with a specified block.

Use the `Simulink.suppressDiagnostic` function to suppress the parameter precision loss warning thrown by the Constant block, one.

```
Simulink.suppressDiagnostic('Suppressor_CLI_Demo/one', ...  
    'SimulinkFixedPoint:util:fxpParameterPrecisionLoss');
```

Get the `Simulink.SuppressedDiagnostic` objects associated with the block.

```
suppressed_diagnostic = Simulink.getSuppressedDiagnostics('Suppressor_CLI_Demo/one')  
suppressed_diagnostic =  
    SuppressedDiagnostic with properties:
```

```
Source: 'Suppressor_CLI_Demo/one'  
Id: 'SimulinkFixedPoint:util:fxpParameterPrecis...'  
LastModifiedBy: ''  
Comments: ''  
LastModified: '2016-Jul-04 14:12:24'
```

- “Suppress Diagnostic Messages Programmatically”

Input Arguments

source — System, block, or model object throwing warning

model | subsystem | block path | block handle

The source of the diagnostic, specified as a model, subsystem, block path, block handle, cell array of block paths, or cell array of block handles.

To get the block path, use the `gcb` function.

To get the block handle, use the `getSimulinkBlockHandle` function.

Data Types: `char` | `cell`

Output Arguments

suppressed_diagnostics — Suppressed diagnostics

array

Suppressed diagnostics, returned as an array of `Simulink.SuppressedDiagnostic` objects.

See Also

`Simulink.SuppressedDiagnostic` | `Simulink.SuppressedDiagnostic.restore`
| `Simulink.getSuppressedDiagnostics` | `Simulink.restoreDiagnostic` |
`Simulink.suppressDiagnostic`

Topics

“Suppress Diagnostic Messages Programmatically”

Introduced in R2016b

Simulink.importExternalCTypes

Generate Simulink representations of custom data types defined by C or C++ code

Syntax

```
importInfo = Simulink.importExternalCTypes(headerFiles)
importInfo = Simulink.importExternalCTypes(modelName)
importInfo = Simulink.importExternalCTypes( ____, Name, Value)
```

Description

`importInfo = Simulink.importExternalCTypes(headerFiles)` parses the C or C++ header files (.h or .hpp) identified by `headerFiles` for `typedef`, `struct`, and `enum` type definitions, and generates Simulink representations of the types. The output, `importInfo`, identifies the successfully and unsuccessfully imported types.

You can use the Simulink representations to:

- Reuse your existing algorithmic C code and, through simulation, test its interaction with your Simulink control algorithm. For an example that shows how to use the Legacy Code Tool, see “Integrate C Function Whose Arguments Are Pointers to Structures”.
- Generate code (Simulink Coder) that reuses the types and data that your existing code defines. You can then integrate and compile the generated and existing code into a single application. For an example, see “Exchange Structured and Enumerated Data Between Generated and External Code” (Embedded Coder).
- Create and organize data (signals, parameters, and states) in a model by using standard data types that your organization defines in C code.
 - To create structures of signals in Simulink, use nonvirtual buses. See “Getting Started with Buses”.
 - To create structures of parameters, use MATLAB structures and `Simulink.Parameter` objects. See “Organize Related Block Parameter Definitions in Structures”.

- To create enumerated data, see “Use Enumerated Data in Simulink Models”.
- To match a primitive `typedef` statement, use a `Simulink.AliasType` object to set parameter and signal data types in a model.

By default, the function:

- Imports an enumerated type by generating a script file that derives an enumeration class from `Simulink.IntEnumType`, as described in “Define Simulink Enumerations”. If necessary, you can then edit the class definition to customize it (for example, by implementing the `addClassNameToEnumNames` method).
- Imports a structure type by generating a `Simulink.Bus` object in the base workspace.
- Imports a primitive `typedef` statement by generating a `Simulink.AliasType` object in the base workspace.
- Interprets generic C data types, such as `int` or `short`, according to the word lengths of your host computer. For example, for most modern machines, `int` has a 32-bit word length, so the function represents an `int` structure field as a bus element that uses the Simulink data type `int32`.

To override this default behavior, identify your target hardware board by using the `HardwareImplementation` pair argument.

For additional information about default behavior, see “Tips” on page 2-662.

`importInfo = Simulink.importExternalCTypes(modelName)` generates Simulink representations of custom C data types by analyzing a model that you identify with `modelName`. When you use the **Simulation Target** configuration parameters in a model to identify header files for inclusion during simulation, use this syntax to import types for the purpose of simulating the model on your host computer. The function interprets generic C data types according to the word lengths of your host computer.

When you use this syntax, do not use pair arguments, such as `HardwareImplementation`, that can conflict with the configuration parameters of the target model. When you use such pair arguments with this syntax, the function generates a warning.

`importInfo = Simulink.importExternalCTypes(____, Name, Value)` specifies additional options using one or more name-value pair arguments. You can use this syntax to:

- Specify the names of types to import by using the `Names` pair argument.
- Control the way that Simulink stores the imported types, for example, by generating the types in a Simulink data dictionary. Use the `MATFile` and `DataDictionary` pair arguments.
- Control the way that the function interprets generic C data types. Use the `HardwareImplementation` pair argument.
- Maintain synchrony between the C-code definitions and the Simulink representations by attempting to import the updated C-code definitions again. You can choose whether to overwrite the existing Simulink representations. Use the `Overwrite` and `Verbose` pair arguments.

Examples

Import Simple Structure and Enumerated Types

This example shows how to generate Simulink representations of a C structure type (`struct`) and an enumerated (`enum`) data type from a header file.

In your current folder, create the file `ex_cc_simpleTypes.h`.

```
typedef enum {
    PWR_LOSS = 0,           /* Default value */
    OVERSPD,
    PRESS_LOW,
} fault_T;

typedef struct {
    double coeff;
    double init;
} params_T;
```

Generate Simulink representations of the types by calling `Simulink.importExternalCTypes`.

```
Simulink.importExternalCTypes('ex_cc_simpleTypes.h');
```

The function creates a `Simulink.Bus` object, `params_T`, in the base workspace. To inspect the properties of the object, open the Bus Editor.

```
buseditor
```

Each bus element uses a name and a data type (double) that match the corresponding structure field in `ex_cc_simpleTypes.h`.

In your current folder, inspect the generated file, `fault_T.m`, which defines the enumerated type `fault_T` as an enumeration class.

You can use the bus object and the enumeration class to set signal and parameter data types in Simulink models.

Import Structure Type Whose Fields Use Custom Data Types

This example shows how to generate a Simulink representation of a structure type whose fields use custom data types (typedef).

Create the file `ex_integer_aliases.h` in your current folder.

```
typedef int sint_32;
typedef unsigned short uint_16;
```

Create the file `ex_cc_struct_alias.h` in your current folder.

```
#include "ex_integer_aliases.h"

typedef struct {
    sint_32 accum;
    uint_16 index;
} my_ints_T;
```

Import the structure type into Simulink as a `Simulink.Bus` object in the base workspace. Import the typedef statements as `Simulink.AliasType` objects.

```
Simulink.importExternalCTypes('ex_cc_struct_alias.h');
```

Inspect the data types of the bus elements in the bus object. For example, inspect the `DataType` property of the first bus element, which corresponds to the structure field `accum`.

```

my_ints_T.Elements(1)

ans =

  BusElement with properties:

      Name: 'accum'
  Complexity: 'real'
  Dimensions: 1
      DataType: 'sint_32'
          Min: []
          Max: []
  DimensionsMode: 'Fixed'
      SampleTime: -1
          Unit: ''
  Description: ''

```

The `Simulink.importExternalCTypes` function uses the generated `Simulink.AliasType` objects to set the data types of the bus elements.

Inspect the `Simulink.AliasType` objects in the base workspace. For example, the object named `sint_32` corresponds to one of the `typedef` statements in `ex_integer_aliases.h`.

```

sint_32

sint_32 =

  AliasType with properties:

  Description: ''
  DataScope: 'Imported'
  HeaderFile: 'ex_integer_aliases.h'
  BaseType: 'int32'

```

For most host computers (which the function targets by default), the word length of `int` is 32 bits and the word length of `unsigned short` is 16 bits. The function maps `int` and `unsigned short` to the Simulink types `int32` and `uint16`.

If you have Embedded Coder, the code that you generate from the model can use `sint_32` and `uint_16` instead of the standard data type names, `int32_T` and `uint16_T`.

Store Imported Types in Data Dictionary

This example shows how to store the imported data types in a Simulink data dictionary. A data dictionary stores data specifications (such as for signals and block parameter values), data types, and other design data for one or more Simulink models.

In your current folder, create the file `ex_cc_simpleTypes.h`.

```
typedef enum {
    PWR_LOSS = 0,           /* Default value */
    OVERSPD,
    PRESS_LOW,
} fault_T;

typedef struct {
    double coeff;
    double init;
} params_T;
```

Create a subfolder called `myDictionaries`.

```
mkdir('myDictionaries')
```

Generate Simulink representations of the types by calling `Simulink.importExternalCTypes`. Permanently store the type definitions by creating a new data dictionary, `ex_cc_myTypes.sldd`, in the new subfolder.

```
Simulink.importExternalCTypes('ex_cc_simpleTypes.h', ...
    'DataDictionary', 'ex_cc_myTypes.sldd', ...
    'OutputDir', 'myDictionaries');
```

To inspect the contents of the dictionary, set your current folder to `myDictionaries` and double-click the dictionary file.

To use the Simulink representations in the dictionary, you must link a model or models to the dictionary. See “Migrate Models to Use Simulink Data Dictionary”.

Import Only Specified Types

This example shows how to generate Simulink representations only for enumerated and structure data types that you identify by name.

In your current folder, create the file `ex_cc_manySimpleTypes.h`. The file defines three structure types: `params_T`, `signals_T`, and `states_T`.

```
typedef struct {
    double coeff;
    double init;
} params_T;

typedef struct {
    double flow_rate;
    double steam_press;
} signals_T;

typedef struct {
    double accum;
    double error;
} states_T;
```

Generate Simulink representations only for `params_T` and `signals_T`.

```
Simulink.importExternalCTypes('ex_cc_manySimpleTypes.h', ...
    'Names', {'params_T', 'signals_T'});
```

The `Simulink.Bus` objects, `params_T` and `signals_T`, appear in the base workspace.

Import Types for 16-Bit Hardware

By default, `Simulink.importExternalCTypes` represents an enumerated data type by creating an enumeration class that derives from the built-in class `Simulink.IntEnumType`. When you simulate or generate code from a model that uses the generated class, configuration parameters that you select for the model (for example,

on the **Hardware Implementation** pane) determine the specific integer length that `Simulink.IntEnumType` and the enumeration class employ.

By default, the function interprets generic, primitive C data types, such as `short` and `int`, according to the word lengths of your host computer. For example, to represent an `int` structure field, the function typically applies the 32-bit data type `int32` to the corresponding bus element. When you want to simulate and generate code for hardware other than your host computer, use the `HardwareImplementation` pair argument to identify the target hardware and, by extension, the word lengths of the hardware.

This example shows how to import data types from code that you intend to use on 16-bit hardware. For this board, `int` has a 16-bit length, and each item of enumerated data (`enum`) consumes 16 bits.

In your current folder, create the file `ex_cc_intTypes.h`.

```
typedef enum {
    PWR_LOSS = 0,           /* Default value */
    OVERSPD,
    PRESS_LOW,
} fault_T;

typedef struct {
    int coeff;
    int init;
} params_T;
```

The code defines an enumerated data type and a structure type whose fields use the generic C data type `int`.

To generate an accurate Simulink representation of the structure type, first open an existing model or create a new model. For this example, create a new model named `ex_hdwImpl_16bit`.

In the new model, set **Configuration Parameters > Hardware Implementation > Device vendor** to `Atmel`. Set **Device type** to `AVR`.

Alternatively, at the command prompt, use these commands to create and configure the model:

```
new_system('ex_hdwImpl_16bit','Model');
set_param('ex_hdwImpl_16bit','ProdHWDeviceType','Atmel->AVR')
```

Generate Simulink representations of the types. To specify the word lengths of the target 16-bit hardware, extract the model configuration parameters (which include the **Hardware Implementation** settings) as a `Simulink.ConfigSet` object.

```
configSet = getActiveConfigSet('ex_hdwImpl_16bit');
Simulink.importExternalCTypes('ex_cc_intTypes.h','HardwareImplementation',configSet);
```

The `Simulink.Bus` object `params_T` appears in the base workspace. The bus elements, such as `coeff`, use the Simulink data type `int16`.

```
params_T.Elements(1)

ans =

    BusElement with properties:

        Name: 'coeff'
    Complexity: 'real'
    Dimensions: 1
        DataType: 'int16'
            Min: []
            Max: []
    DimensionsMode: 'Fixed'
        SampleTime: -1
            Unit: ''
        Description: ''
```

In your current folder, the file `fault_T.m` defines the enumeration class `fault_T`. The class derives from `Simulink.IntEnumType`, so you must use model configuration parameters to identify the target hardware and, by extension, the correct native integer length.

```
type('fault_T.m')

classdef fault_T < Simulink.IntEnumType
    % MATLAB enumeration class definition generated from template

    enumeration
        PWR_LOSS(0),
        OVERSPD(1),
        PRESS_LOW(2)
```

```
end

methods (Static)

function defaultValue = getDefaultValue()
    % GETDEFAULTVALUE Returns the default enumerated value.
    % If this method is not defined, the first enumeration is used.
    defaultValue = fault_T.PWR_LOSS;
end

function dScope = getDataScope()
    % GETDATASCOPE Specifies whether the data type definition should be imported
    % or exported to, a header file during code generation.
    dScope = 'Imported';
end

function desc = getDescription()
    % GETDESCRIPTION Returns a description of the enumeration.
    desc = '';
end

function headerFile = getHeaderFile()
    % GETHEADERFILE Specifies the name of a header file.
    headerFile = 'ex_cc_intTypes.h';
end

function flag = addClassNameToEnumNames()
    % ADDCLASSNAMETOENUMNAMES Indicate whether code generator applies the class
    % name to the enumeration.
    flag = false;
end

end

end
```

Import Structure Type Whose Fields Use 16-Bit Fixed-Point Data Types

Create the file `ex_cc_fixpt_struct.h` in your current folder.

```
typedef struct {
```

```

int coeff; /* Word length 16,
           binary fraction length 7 */

int init; /* Word length 16,
           binary fraction length 3 */

} params_T;

```

The file defines a structure type whose fields use fixed-point data types. For example, the structure stores the field `coeff` in a signed, 16-bit integer data type. A binary fraction length of 7 relates the stored integer value to the real-world value.

Suppose that this code operates on 16-bit hardware (such that the generic C data type `int` has a 16-bit word length). To generate a Simulink representation of the type, first create a `coder.HardwareImplementation` object that identifies the hardware.

```

hdw = coder.HardwareImplementation;
hdw.ProdHWDeviceType = 'Atmel->AVR';

```

Generate a Simulink representation of the structure type.

```

Simulink.importExternalCTypes('ex_cc_fixpt_struct.h', ...
    'HardwareImplementation', hdw);

```

The `Simulink.Bus` object, `params_T`, appears in the base workspace. Each bus element, such as `coeff`, uses the data type `int16`.

```
params_T.Elements(1)
```

```
ans =
```

```
BusElement with properties:
```

```

    Name: 'coeff'
Complexity: 'real'
Dimensions: 1
    DataType: 'int16'
        Min: []
        Max: []
DimensionsMode: 'Fixed'
    SampleTime: -1
        Unit: ''
    Description: ''

```

`Simulink.importExternalCTypes` cannot infer the fixed-point scaling (binary fraction length) from the C code. You must manually specify the data types of the bus elements. To specify the data types at the command prompt, use the `fixdt` function.

```
params_T.Elements(1).DataType = 'fixdt(1,16,7)';  
params_T.Elements(2).DataType = 'fixdt(1,16,3)';
```

To specify the data types interactively (by using the Data Type Assistant), use the Bus Editor.

```
buseditor
```

Manually Synchronize Simulink Representations with C-Code Definitions

This example shows how to maintain the Simulink representations of C data types whose definitions you modify during the life of a modeling project.

Import Custom C Types

Create the file `ex_cc_myTypes_rec.h` in your current folder. The file defines a custom structure type.

```
typedef struct {  
    double flow;  
    double pres;  
    double tqe;  
} sigStructType;
```

Generate a `Simulink.Bus` object that represents the type.

```
Simulink.importExternalCTypes('ex_cc_myTypes_rec.h');
```

Modify Type Definition in C Code

In `ex_cc_myTypes_rec.h`, add a field named `spd` to `sigStructType`.

In the same file, create a new structure type, `stateStructType`.

```
typedef struct {  
    double flow;
```

```

        double pres;
        double tqe;
        double spd;
    } sigStructType;

typedef struct {
    double err;
    double read;
    double write;
} stateStructType;

```

Attempt to Import Types Again

Attempt to generate bus objects that represent the types.

```
importInfo = Simulink.importExternalCTypes('ex_cc_myTypes_rec.h');
```

The function generates warnings at the command prompt. Instead of relying on the warnings, you can inspect the output, `importInfo`, to determine whether the function failed to import any types.

```
importInfo.failedToImport.Bus
```

```
ans =

    1×1 cell array

    {'sigStructType'}
```

The function did not import `sigStructType`. The corresponding bus object in the base workspace still has only three bus elements. To determine the reason that the function did not import `sigStructType`, inspect the `report` field of `importInfo`.

Import `sigStructType` again. This time, overwrite the existing bus object.

```
importInfo = Simulink.importExternalCTypes('ex_cc_myTypes_rec.h', ...
    'Names', importInfo.failedToImport.Bus, 'Overwrite', 'on');
```

When you overwrite existing Simulink representations, any customizations that you made to the Simulink representations (such as the application of fixed-point data types to bus elements) are overwritten.

- “Data Types Supported by Simulink”
- “Data Types for Bus Signals”
- “Use Enumerated Data in Simulink Models”
- “Control Data Type Names in Generated Code” (Embedded Coder)
- “Control Signal Data Types”
- “Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” (Simulink Coder)

Input Arguments

headerFiles — Names and paths of header files to parse

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

Names and paths of header files to parse, specified as a character vector, cell array of character vectors, string, or string array. Include the `.h` or `.hpp` file extension.

If you use a hierarchy of included (`#include`) header files to define your types, when you specify `HeaderFiles`, you need to identify only the entry-point files. The function parses the included files as well as the identified entry-point files. If the included files are not in the same folder as the corresponding entry-point file, use the `IncludeDirs` pair argument to identify the additional folders.

Example: `'myHeader.h'`

Example: `{ 'thisHeader.hpp', 'thatHeader.hpp' }`

Data Types: `char` | `cell` | `string`

modelName — Name of loaded Simulink model for which to import types

character vector | string scalar

Name of a loaded Simulink model for which to import types, specified as a character vector or string scalar. A model is loaded if, for example, you open the model or use the `load_system` function. When you use this argument, the function:

- Searches the model configuration parameters for custom header files and parses those header files for data types to import. Only the configuration parameters on the **Simulation Target** pane affect this search.

For example, if in the model you set **Configuration Parameters > Simulation Target > Insert custom C code in generated > Header file** to `#include "myTypes.h"`, the function parses `myTypes.h` for types to import.

- Interprets generic C data types such as `int` or `short` according to the word lengths of your host computer. Do not use the `HardwareImplementation` pair argument to override this interpretation.

Example: `'myModel'`

Data Types: `char | string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

```
Simulink.importExternalCTypes('myHdr.h', 'DataDictionary', 'myDictionary.sldd')
```

MATFile — Name and path of MAT-file to create for storing generated objects

`''` (empty) (default) | character vector | string scalar

Name and, optionally, path of the MAT-file to create for storing generated `Simulink.Bus` and `Simulink.AliasType` objects, specified as a character vector or string. If you do not use `MATFile`, by default, the function generates the objects in the base workspace.

The function does not generate enumeration definitions in MAT-files.

If you import some `struct` types and primitive `typedef` statements by using `MATFile` and later import some of the same types again by using `MATFile`, the function entirely replaces the old MAT-file with a new one. The function discards any changes that you made to the contents of the old MAT-file.

You cannot use the `MATFile` and `DataDictionary` pair arguments simultaneously.

Example: `'myMat.mat'`

Example: `'myMat'`

Example: `fullfile('subfolder','myMat')`

Data Types: `char` | `string`

DataDictionary — Name and path of Simulink data dictionary to use or create for storing types

`'` (default) | character vector | string scalar

Name and, optionally, path of the Simulink data dictionary to use or create for storing generated enumerations and objects, specified as a character vector or string. When you use this pair argument, the function imports enumerated types as `Simulink.data.dictionary.EnumTypeDefinition` objects, and stores those objects (as well as `Simulink.Bus` objects and `Simulink.AliasType` objects) in the target dictionary.

For information about data dictionaries, see “What Is a Data Dictionary?”.

You can optionally specify a `.sldd` extension.

You cannot use the `DataDictionary` and `MATFile` pair arguments simultaneously.

Example: `'myDict.slidd'`

Example: `'myDict'`

Example: `fullfile('subfolder','myDict.slidd')`

Data Types: `char` | `string`

Names — Names of types to import

`'` (default) | character vector | cell array of character vectors | string scalar | string array

Names of types to import, specified as a character vector, cell array of character vectors, string, or string array. By default, if you do not use `Names`, the function attempts to import all of the custom types that the identified header files define.

To match multiple type names with a single character vector, use an asterisk (*).

Example: `'myEnumType'`

Example: `{ 'myEnumType', 'myStructType' }`

Example: `'my*Type'`

Data Types: `char` | `cell` | `string`

Defines — Compiler options to define macros that influence type definitions

`'` (default) | character vector | string scalar

Compiler options to define macros that influence C type definitions, specified as a character vector, or string scalar. For example, a macro influences a type definition if you enclose the definition in an `#ifdef` block that checks whether the macro is defined.

Use `Defines` to specify macro definitions that you otherwise define through compiler options such as `-D`.

Example: `'SIGSTRUCT=1'`

Example: `'SIGSTRUCT=1 ENUM=1'`

Data Types: `char` | `string`

UnDefines — Compiler options to delete macros that influence type definitions

`'` (default) | character vector | string scalar

Compiler options to delete macros that influence C type definitions, specified as a character vector or string scalar. For example, a macro influences a type definition if you enclose the definition in an `#ifdef` block that checks whether the macro is defined.

Use `UnDefines` to specify macro deletions that you otherwise define through compiler options such as `-U`.

Example: `'SIGSTRUCT'`

Example: `'SIGSTRUCT ENUM'`

Data Types: `char` | `string`

IncludeDirs — Folders that contain subordinate, included header files

`'` (default) | character vector | cell array of character vectors | string scalar | string array

Folders that contain subordinate, included (`#include`) header files, specified as a character vector, cell array of character vectors, string, or string array. Use this pair

argument to enable the function to locate and parse additional header files on which the primary header files (which you specify with the `headerFiles` argument) depend.

If you use the `modelName` syntax instead of the `headerFiles` syntax, in the target model, you can use the **Simulation Target** configuration parameters to specify include paths. In that case, you do not need to use the `IncludeDirs` pair argument.

Example: `'myHeaders'`

Example: `fullfile('myProject','myHeaders')`

Example:

```
{fullfile('myProject','myHeaders'),fullfile('myProject','myOtherHeaders')}
```

Data Types: `char` | `cell` | `string`

OutputDir — Folder for storing generated files

`'` (default) | character vector | string scalar

Folder for storing generated files, specified as a character vector or string. The function places generated files, such as `classdef` script files and data dictionary files, in this folder.

The folder that you specify must exist before you use the function.

Example: `'myDictionaries'`

Example: `fullfile('myProject','myDictionaries')`

Data Types: `char` | `string`

HardwareImplementation — Word lengths for interpreting generic, primitive C data types

`'` (default) | `Simulink.ConfigSet` object | `coder.HardwareImplementation` object

Word lengths for interpreting generic, primitive C data types, specified as a `Simulink.ConfigSet` or `coder.HardwareImplementation` object.

- To use a `Simulink.ConfigSet` object, you can extract a configuration set from a model by using functions such as `getConfigSet` and `getActiveConfigSet`. This technique enables you to use the Configuration Parameters dialog box to identify your target hardware (through the **Hardware Implementation** configuration parameters).

- To use a `coder.HardwareImplementation` object (which you create and configure programmatically), specify properties of the object, such as `ProdHWDeviceType`, to identify your target hardware. The object then sets other properties, such as `ProdBitPerInt`, that reflect the native integer size of the hardware.

The function inspects the object to determine which Simulink integer data types to employ when interpreting generic C data types such as `int`. For example, if you create a `coder.HardwareImplementation` object to identify 16-bit hardware and then use the function to import a structure type whose fields use the C data type `int`, the function generates a bus object whose bus elements use the Simulink data type `int16`. The function uses the production hardware settings, not the test hardware settings.

For more information about hardware implementation settings for Simulink models, see “Configure Run-Time Environment Options” (Simulink Coder).

Overwrite — Specification to overwrite existing Simulink representations

`'off'` (default) | `'on'`

Specification to overwrite existing Simulink representations, specified as `'on'` or `'off'`. If an imported type already has a representation in Simulink:

- If you specify `'off'` or if you do not specify `Overwrite`, the function does not import the type. In the output argument, `importInfo`, the `failedToImport` field identifies the type.
- If you specify `'on'`, the function overwrites the existing Simulink representation.

If you use the function to import some types into the base workspace or a data dictionary and later customize the generated Simulink representations, when you use the function again and set `Overwrite` to `'on'`, the function does not preserve your customizations. These customizations can include:

- In an enumeration class definition, implementing extra methods or modifying the generated methods such as `getDataScope` (see “Customize Simulink Enumeration”).
- Modifying the properties of a generated `Simulink.Bus` or `Simulink.AliasType` object (for example, manually setting the data types of bus elements to a fixed-point data type).

Verbose — Specification to generate messages for successful import operations

`'off'` (default) | `'on'`

Specification to generate messages for successful import operations, specified as 'on' or 'off'.

- If you specify 'off' or if you do not specify `Verbose`, the function imports types silently. Messages do not appear in the Command Window unless the function cannot import a type.
- If you specify 'on', the function generates a message in the Command Window for each operation during the import process.

Output Arguments

importInfo — Information about types that were imported and not imported

structure

Information about types that were imported and not imported, returned as a structure with these fields.

report — Descriptions of types that were imported and not imported

character vector

Descriptions of types that were imported and not imported, returned as a character vector. Inspect the value of this field to determine the reason that the function could not import a type.

failedToImport — Types that were not imported

structure

Types that were not imported, returned as a structure with these fields.

Field Name	Field Value	Purpose
Bus	Cell array of character vectors	Names of structure (struct) types that were not imported.
Enum	Cell array of character vectors	Names of enumerated types (enum) that were not imported.

Field Name	Field Value	Purpose
AliasType	Cell array of character vectors	Names of primitive typedef statements that were not imported.

importedTypes — Types that were successfully imported
structure

Types that were successfully imported, returned as a structure with these fields.

Field Name	Field Value	Purpose
Bus	Cell array of character vectors	Names of structure (struct) types that were imported. The generated Simulink.Bus objects use these names.
Enum	Cell array of character vectors	Names of enumerated types (enum) that were imported. The generated enumeration classes or Simulink.data.dictionary.EnumTypeDefinition objects use these names.
AliasType	Cell array of character vectors	Names of primitive typedef statements that were imported. The generated Simulink.AliasType objects use these names.

Limitations

- The function does not support:
 - C data types that do not correspond to a type that Simulink supports. For example, Simulink does not recognize an equivalent for long double. For

information about data types that Simulink supports, see “Data Types Supported by Simulink”.

- Pointer types, such as a structure that defines a field whose value is a pointer or a `typedef` statement whose base type is a pointer type.
- Structures that define a field whose value has more than one dimension.

If a field value is a 1-D array, the function creates a bus element that represents a vector, not a matrix.

- Unions.
- If a structure field represents fixed-point data, or if a `typedef` statement maps to a fixed-point base type, the function sets the data type of the corresponding bus element or `Simulink.AliasType` object to the relevant Simulink integer type (such as `int16`). The importer cannot determine the fixed-point scaling by parsing the C code. After using the function, you must manually specify the data type of the bus element or the base type of the `Simulink.AliasType` object by using the `fixdt` function.

Tips

By default:

- For an imported enumeration, because the Simulink enumeration class derives from `Simulink.IntEnumType`, when you simulate or generate code from a model, the enumeration uses the integer size that is native to your target hardware. You specify the characteristics of your target hardware by using model configuration parameters such as **Production device vendor and type** and **Native word size in production hardware**.
- For an imported structure type:
 - The function imports a structure field as numerically complex only if the field uses one of the corresponding Simulink Coder structure types as the data type. For example, if a structure field in your external code uses the data type `cint8_T`, the function imports the field as a bus element (`Simulink.BusElement` object) whose data type is `int8` and whose `Complexity` property is set to `'complex'`.
 - For nested structures, the function generates a bus object for each unique structure type.
- For an imported structure type or enumeration, if your external code uses a `typedef` statement to name the type, the name of the generated bus object or Simulink

enumeration class matches the `typedef` name. If your code does not use a `typedef` statement, the name of the object or class is `struct_type` or `enum_type` where `type` is the tag name of the type. If you do not specify a tag name or apply a `typedef` name, Simulink generates an arbitrary name for the object or class.

- The function configures the generated Simulink representations as imported for the purposes of simulation and code generation. For example, for bus objects, the function sets the `DataScope` property to 'Imported' and the `HeaderFile` property to the name of your external header file. To simulate or generate code from a model that uses one of these Simulink representations, you must make your header file available to the model.

When you specify files for `Simulink.importExternalCTypes` to use or generate, for example, by using the `DataDictionary` pair argument:

- If the existing files to use are in your current folder or on the MATLAB path, you do not need to specify a file path. You can specify the file name by itself.
- To control the folder location of generated files, you can specify paths as well as file names. You can also use the `OutputDir` pair argument.

See Also

`Simulink.AliasType` | `Simulink.Bus` | `enumeration`

Topics

“Data Types Supported by Simulink”

“Data Types for Bus Signals”

“Use Enumerated Data in Simulink Models”

“Control Data Type Names in Generated Code” (Embedded Coder)

“Control Signal Data Types”

“Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” (Simulink Coder)

Introduced in R2017a

Simulink.ModelDataLogs.convertToDataset

Convert logging data from `Simulink.ModelDataLogs` format to `Simulink.SimulationData.Dataset` format

Syntax

```
convertedDataset =  
sourceModelDataLogsObject.convertToDataset(convertedDatasetName)
```

Description

Note The `ModelDataLogs` class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy `ModelDataLogs` API”.

```
convertedDataset =  
sourceModelDataLogsObject.convertToDataset(convertedDatasetName)  
converts the sourceModelDataLogsObject to a  
Simulink.SimulationData.Dataset object. The name of the converted object is based  
on convertedDatasetName.
```

The resulting `Simulink.SimulationData.Dataset` object is a flat list. This list has one element for each `Simulink.Timeseries` or `Simulink.TsArray` object in the `Simulink.ModelDataLogs` object.

Limitations

Source of Simulink.ModelDataLogs Logged Data	Conversion Limitation
Referenced model	<p>Loads all ancestors of the referenced model not previously loaded. If any ancestor model does not appear on the MATLAB path, the conversion fails.</p> <p>If the model has changed, or the model ancestors have changed, after Simulink logged the data, the conversion can fail. For example, adding, deleting, or renaming a block after logging can cause conversion failure.</p>
Variant model or subsystem	The current active variant must be the same one that was active when Simulink logged the data. Otherwise, the conversion fails.
Frame signal	The conversion fails.
Mux block	The conversion produces a different <code>Simulink.SimulationData.Dataset</code> object as the dataset than Simulink creates when you simulate the model using the Dataset format for the logged data.
Stateflow chart	Not supported.

Input Arguments

sourceModelDataLogsObject

A `Simulink.ModelDataLogs` object that you want to convert to a `Simulink.SimulationData.Dataset` object.

convertedDatasetName

Name of the dataset that the conversion process creates.

Output Arguments

convertedDataset

The `Simulink.SimulationDataset` object that the `Simulink.ModelDataLogs.convertToDataset` function creates.

For details about the converted dataset, see `Simulink.SimulationData.Dataset`.

Example

In releases before R2016a, you could log signals using `ModelDataLogs` format. If you have a MAT-file with signal logging data that uses the `ModelDataLogs` format, here is how you can convert that data to `Dataset` format. This example assumes that the model that generated the logging data had the **Configuration Parameters > Data Import/Export > Signal logging** name set to `logout`.

- 1 Load the MAT-file.
- 2 Convert `logout` to a dataset called `myModel_dataset`. (The elements information will be different for your data.)

```
dataset = logout.convertToDataset('myModel_Dataset')

dataset =
  Simulink.SimulationData.Dataset
  Package: Simulink.SimulationData

Characteristics:
      Name: 'myModel_Dataset'
  Total Elements: 2

Elements:
  1: 'x1'
  2: 'x2'
```

- Use `get` or `getElement` to access elements by index or name.
- Use `addElement` or `setElement` to add or modify elements.

Methods, Superclasses

See Also

`Simulink.ModelDataLogs` | `Simulink.SimulationData.Dataset` |
`Simulink.SimulationData.updateDatasetFormatLogging`

Topics

“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”

Introduced in R2011a

Simulink.restoreDiagnostic

Restore diagnostic warnings to a specific block, subsystem, or model

Syntax

```
Simulink.restoreDiagnostic(source)
Simulink.restoreDiagnostic(source, message_id)
Simulink.restoreDiagnostic(diagnostic)
Simulink.restoreDiagnostic(system, 'FindAll', 'on')
```

Description

`Simulink.restoreDiagnostic(source)` restores all suppressed diagnostics associated with the blocks specified by `source`.

`Simulink.restoreDiagnostic(source, message_id)` restores all instances of `message_id` on the blocks specified by `source`.

`Simulink.restoreDiagnostic(diagnostic)` restores the suppressed diagnostics associated with MSLDiagnostic object `diagnostic`.

`Simulink.restoreDiagnostic(system, 'FindAll', 'on')` restores all suppressed diagnostics associated with the system specified by `system`.

Examples

Restore All Diagnostics for a Specified Block

Using the model from “Suppress Diagnostic Messages Programmatically”, restore all suppressed diagnostics on a specified block.

Create a cell array of message identifiers. Use the `Simulink.suppressDiagnostic` function to suppress these multiple diagnostics from the Constant block, one.

```
diags = {'SimulinkFixedPoint:util:fxpParameterPrecisionLoss', ...
        'SimulinkFixedPoint:util:fxpParameterUnderflow'};
Simulink.suppressDiagnostic('Suppressor_CLI_Demo/one', diags);
```

Remove all suppressions and restore diagnostics to the block.

```
Simulink.restoreDiagnostic('Suppressor_CLI_Demo/one');
```

Restore a Diagnostic for a Specified Block

Using the model from “Suppress Diagnostic Messages Programmatically”, restore a suppressed diagnostic on a specified block.

Use the `Simulink.suppressDiagnostic` function to suppress the parameter precision loss warning thrown by the Constant block, `one`.

```
Simulink.suppressDiagnostic('Suppressor_CLI_Demo/one', ...
    'SimulinkFixedPoint:util:fxpParameterPrecisionLoss');
```

Remove the suppression and restore diagnostics to the block.

```
Simulink.restoreDiagnostic('Suppressor_CLI_Demo/one', ...
    'SimulinkFixedPoint:util:fxpParameterPrecisionLoss');
```

Restore All Diagnostics for a Specified System

Using the model from “Suppress Diagnostic Messages Programmatically”, restore all suppressed diagnostics on the specified subsystem.

To restore all diagnostics from a system, use `'FindAll'`, `'on'` to search within the system hierarchy. Specify the system or system handle within which to search.

```
Simulink.restoreDiagnostic('Suppressor_CLI_Demo/Convert', ...
    'FindAll', 'On');
```

- “Suppress Diagnostic Messages Programmatically”

Input Arguments

source — Block or model object throwing diagnostic

block path | block handle

The source of the diagnostic, specified as a block path, block handle, cell array of block paths, or cell array of block handles.

To get the block path, use the `gcb` function.

To get the block handle, use the `getSimulinkBlockHandle` function.

Data Types: `char` | `cell`

message_id — message identifier of diagnostic

message identifier | cell array of message identifiers

Message identifier of the diagnostic, specified as a character vector or a cell array of character vectors. You can find the message identifier of warnings and errors thrown during simulation by accessing the `ExecutionInfo` property of the `Simulink.SimulationMetadata` object associated with a simulation. You can also use the `lastwarn` function.

Data Types: `char` | `cell`

system — Name of subsystem

character vector

The subsystem name, or handle, specified as a character vector.

Data Types: `char`

diagnostic — Diagnostic object

`MSLDiagnostic` object

Diagnostic specified as an `MSLDiagnostic` object. Access the `MSLDiagnostic` object through the `ExecutionInfo` property of the `Simulink.SimulationMetadata` object.

Data Types: `struct`

See Also

`Simulink.SuppressedDiagnostic` | `Simulink.SuppressedDiagnostic.restore`
| `Simulink.getSuppressedDiagnostics` | `Simulink.suppressDiagnostic`

Topics

“Suppress Diagnostic Messages Programmatically”

Introduced in R2016b

getBlockSimState

Class: Simulink.SimState.ModelSimState

Package: Simulink.SimState

Access SimState of individual Stateflow Chart, MATLAB Function, or S-function block

Syntax

```
blockSimState = getBlockSimState(x, 'blockpath')
```

Description

blockSimState = getBlockSimState(*x*, '*blockpath*') returns the SimState of the block specified as *blockpath*. *blockpath* must be either a Stateflow Chart, MATLAB Function, or S-function block. For other types of blocks, see the `loggedStates` property of the Simulink.SimState.ModelSimState class.

Input Arguments

x

The *x* argument is a Simulink.SimState.ModelSimState object.

blockpath

The path to the block for which you are requesting the SimState values.

Output Arguments

blockSimState

The SimState of the block specified.

Examples

```
chartState = getBlockSimState(x, 'mymodel/chart')
```

See Also

`Simulink.SimState.ModelState.setBlockSimState`

setBlockSimState

Class: Simulink.SimState.ModelSimState

Package: Simulink.SimState

Set SimState of individual Stateflow Chart, MATLAB Function, or S-function block

Syntax

```
setBlockSimState(x, 'blockpath', blockSimState)
```

Description

`setBlockSimState(x, 'blockpath', blockSimState)` sets the SimState of the block specified as `blockpath`. `blockpath` must be either a Stateflow Chart, MATLAB Function, or S-function block. For other types of blocks, see the `loggedStates` property of the `Simulink.SimState.ModelSimState` class.

Input Arguments

x

The argument `x` is a `Simulink.SimState.ModelSimState` object.

blockpath

The path to the block for which you are setting the SimState values

blockSimState

The SimState of the block specified.

Examples

```
newObj = setBlockSimState(obj, 'mymodel/chart', newChartState);
```

See Also

`Simulink.SimState.ModelState.getBlockSimState`

Simulink.saveVars

Save workspace variables and their values in MATLAB code format

Syntax

Note `Simulink.saveVars` is not recommended. Use `matlab.io.saveVariablesToScript` instead.

```
Simulink.saveVars(filename)
Simulink.saveVars(filename, VarNames)
Simulink.saveVars(filename, '-regexp', RegExps)
Simulink.saveVars(filename, Specifications, UpdateOption)
Simulink.saveVars(filename, Specifications, Configuration)
Simulink.saveVars(filename, Specifications, MatlabVer)
[r1, r2] = Simulink.saveVars(filename, Specifications)
```

Description

`Simulink.saveVars(filename)` saves all variables in the current workspace for which MATLAB code can be generated to a MATLAB file named `filename.m`. If MATLAB code cannot be generated for a variable, the variable is saved into a companion MAT-file named `filename.mat`, and a warning is generated. If either file already exists, it is overwritten. The `filename` cannot match the name of any variable in the current workspace, and can optionally include the suffix `.m`. Using `Simulink.saveVars` has no effect on the contents of any workspace.

Executing the MATLAB file restores the variables saved in the file to the current workspace. If a companion MAT-file exists, code in the MATLAB file loads the MAT-file, restoring its variables also. When both a MATLAB file and a MAT-file exist, do not load the MATLAB file unless the MAT file is available, or an error will occur. Do not load a MAT-file directly, or incomplete data restoration will result. No warning occurs if loading a file overwrites any existing variables.

You can edit a MATLAB file that `Simulink.saveVars` creates. You can insert comments between or within the MATLAB code sections for saved variables. However, if you later use `Simulink.saveVars` to update or append to the file, only comments between MATLAB code sections will be preserved. Internal comments should therefore be used only in files that you do not expect to change any further.

You must not edit the header section in the MATLAB file, which comprises the first five comment lines. Simulink does not check that a manually edited MATLAB file is syntactically correct. MathWorks recommends not editing any MATLAB code in the file. You cannot edit a MAT-file and should never attempt to do so.

`Simulink.saveVars(filename, VarNames)` saves only the variables specified in *VarNames*, which is a comma-separated list of variable names. You can use the wildcard character `*` to save all variables that match a pattern. The `*` matches one or more characters, including non-alphanumeric characters.

`Simulink.saveVars(filename, '-regexp', RegExps)` saves only variables whose names match one of the regular expressions in *RegExps*, which is a comma-separated list of expressions. See “Regular Expressions” (MATLAB) for more information. A call to the function can specify both *VarNames* and `-regexprs RegExps`, in that order and comma-separated.

`Simulink.saveVars(filename, Specifications, UpdateOption)` saves the variables described by *Specifications* (which represents the variable specifications in any of the above syntaxes) as directed by *UpdateOption*, which can be any one of the following:

- `'-create'` — Create a new MATLAB file (and MAT-file if needed) as directed by the *Specifications*. If either file already exists, it is overwritten. This is the default behavior.
- `'-update'` — Update the existing MATLAB file (and MAT-file if needed) specified by *filename* by changing only variables that match the *Specifications* and already exist in any files. The order of the variables in files is preserved. Comments within MATLAB code sections are not preserved.
- `'-append'` — Update the existing MATLAB file (and MAT-file if needed) specified by *filename* by:
 - Updating variables that match the *Specifications* and already exist in the file or files, preserving the existing order in the file or files. Comments within MATLAB code sections are not preserved.

- Appending variables that match the *Specifications* and do not exist in the file or files by appending the variables to the file or files. These new sections initially have no comments.

`Simulink.saveVars(filename, Specifications, Configuration)` saves the variables described by *Specifications* (which represents the variable specifications in any of the above syntaxes) according to the specified *Configuration*. The *Configuration* can contain any or all of the following options, in any order, separated by commas if more than one appears:

- `'-maxnumel'` *MaxNum* — Limits the number of elements saved for an array to *MaxNum*, which must be an integer between 1 and 10000. For a character array, the upper limit is set to twice the value that you specify with *MaxNum*. If an array is larger than *MaxNum*, the whole array appears in the MAT-file rather than the MATLAB file, generating a warning. Default: 1000
- `'-maxlevels'` *MaxLevels* limits the number of levels of hierarchy saved for a structure or cell array to *MaxLevels*, which must be an integer between 1 and 200. If a structure or cell array is deeper than *MaxLevels*, the whole entity appears in the MAT-file rather than the MATLAB file, generating a warning. Default: 20
- `'-textwidth'` *TextWidth* sets the text wrap width in the MATLAB file to *TextWidth*, which must be an integer between 32 and 256. Default: 76
- `'-2dslice'` — Sets two dimensions for 2-D slices that represent n-D (where n is greater than 2) char, logic, or numeric array data. `Simulink.saveVars` uses the first two dimensions of the n-D array to specify the size of the 2-D slice, unless you supply two positive integer arguments after the `-2dslice` option. If you specify two integer arguments:
 - The two integers must be positive.
 - The two integers must be less than or equal to the number of dimensions of the n-D array.
 - The second integer must be greater than the first.

`Simulink.saveVars(filename, Specifications, MatlabVer)` acts as described by *Specifications* (which represents the specifications after *filename* in any of the above syntaxes) saving any MAT-file that it creates in the format required by the MATLAB version specified by *MatlabVer*. Possible values:

- `'-v7.3'` — 7.3 or later

- '-v7.0' — 7.0 or later
- '-v6' — Version 6 or later
- '-v4' — Any MATLAB version

`[r1, r2] = Simulink.saveVars(filename, Specifications)` acts as described by *Specifications* (which represents the specifications after *filename* in any of the above syntaxes) and reports what variables it has saved:

- *r1* — A cell array of character vectors. The character vectors name all variables (if any) that were saved to a MATLAB file.
- *r2* — A cell array of character vectors. The character vectors name all variables (if any) that were saved to a MAT-file.

Input Arguments

filename

The name of the file or names of the files that the function creates or updates. The *filename* cannot match the name of any variable in the current workspace. The *filename* can have the suffix `.m`, but the function ignores it.

VarNames

A variable or sequence of comma-separated variables. The function saves only the specified variables to the output file. You can use the wildcard character `*` to save all variables that match a pattern. The `*` matches one or more characters, including non-alphanumeric characters.

'-regexp', RegExps

After the keyword, a regular expression or sequence of comma-separated regular expressions. The function saves to the output file only those variables whose names match one of the expressions. See “Regular Expressions” (MATLAB) for more information. A call to the function can specify both *VarNames* and `-regexprs RegExps`, in that order and comma-separated.

UpdateOption

Any of three keywords that control the action of the function. The possible values are:

- `'-create'` — Create a new MATLAB file (and MAT-file if needed) as directed by the *Specifications*.
- `'-update'` — Update the existing MATLAB file (and MAT-file if needed) specified by *filename* by changing only variables that match the *Specifications* and already exist in the file or files. The order of the variables in the file or files is preserved.
- `'-append'` — Update the existing MATLAB file (and MAT-file if needed) specified by *filename* by:
 - Updating variables that match the *Specifications* and already exist in the file or files, preserving the existing order in the file or files.
 - Appending variables that match the *Specifications* and do not exist in the file or files by appending the variables that match the *Specifications* to the file or files.

Default: `'-create'`

Configuration

Any or all of the following options, in any order, separated by commas if more than one appears:

- `'-maxnumel'` *MaxNum* — Limits the number of elements saved for an array to *MaxNum*, which must be an integer between 0 and 10000. If an array is larger than that, the whole array appears in the MAT-file rather than the MATLAB script file, generating a warning. Default: 1000
- `'-maxlevels'` *MaxLevels* — Limits the number of levels saved for a structure or cell array to *MaxLevels*, which must be an integer between 0 and 200. If a structure or cell array is deeper than that, the whole entity appears in the MAT-file rather than the MATLAB script file, generating a warning. Default: 20
- `'-textwidth'` *TextWidth* — Sets the text wrap width in the MATLAB script file to *TextWidth*, which must be an integer between 32 and 256. Default: 76
- `'-2dslice'` — Sets two dimensions for 2-D slices that represent n-D (where n is greater than 2) arrays of char, logic, or numeric data. Using the `'-2dslice'` option produces more readable generated code that is consistent with how MATLAB displays n-D array data.

`Simulink.saveVars` uses the first two dimensions of the n-D array to specify the size of the 2-D slice, unless you supply two positive integer arguments after the `-2dslice` option. If you specify two integer arguments:

- The two integers must be positive.
- The two integers must be less than or equal to the number of dimensions of the n-D array.
- The second integer must be greater than the first.

Note You can use the MATLAB Preferences to change the defaults for the `-maxnumel`, `-maxlevels`, `'-2dslice'`, and `-textwidth` configuration options. In the **Workspace** pane, use the options in the **Saving variables as MATLAB script files** group.

MatlabVer

Specifies the MATLAB version whose syntax will be used by any MAT-file saved by the function.

- `'-v7.3'` — 7.3 or later
- `'-v7.0'` — 7.0 or later
- `'-v6'` — Version 6 or later
- `'-v4'` — Any MATLAB version

Default: `'-v7.3'`

Output Arguments

r1

A list of the names of all variables (if any) that were saved to a MATLAB file.

r2

A list of the names of all variables (if any) that were saved to a MAT-file.

Examples

Define some base workspace variables, then save them all to a new MATLAB file named `MyVars.m` using the default values for all input arguments except the `filename`.

```
a = 1;
b = 2.5;
c = 'A string';
d = {a, b, c};
Simulink.saveVars('MyVars');
```

Define additional base workspace variables, then append them to the existing file `MyVars.m` without changing the values previously saved in the file:

```
K = Simulink.Parameter;
MyType = fixdt (1,16,3);
Simulink.saveVars('MyVars', '-append', 'K', 'MyType');
```

Update the variables `V1` and `V2` with their values in a MATLAB file, or for any whose value cannot be converted to MATLAB code, in a MAT-file. The file must already exist. Any array with more than 10 elements will be saved to a MAT-file that can be loaded on any version of MATLAB. The return argument `r1` lists the names of any variables saved to a MATLAB file; `r2` lists any saved to a MAT-file.

```
[r1, r2] = Simulink.saveVars('MyFile', 'V1', 'V2', '-update',
'-maxnumel', 10, '-v4');
```

Specify a 2-D slice for the output of the `my3Dtable` 3-D array. Specify that the 2-D slice expands along the first and third dimensions:

```
my3DTable = zeros(3, 4, 2, 'single');
Simulink.saveVars('mfile.m', 'my3DTable', '-2dslice', 1, 3);
```

The generated MATLAB code is:

```
my3DTable = zeros(3, 4, 2, 'single');
my3DTable (:,1,:) = single ( ...
    [1 13;
     5 17;
     9 21]);
my3DTable (:,2,:) = single( ...
    [2 14;
     6 18;
    10 22]);
my3DTable (:,3,:) = single( ...
    [3 15;
     7 19;
    11 23]);
my3DTable (:,4,:) = single( ...
```

```
[4 16;  
 8 20;  
12 24]);
```

Limitations

The `Simulink.saveVars` function:

- Does not preserve shared references.
- Ignores dynamic properties of objects.
- Saves the following to the MAT-file although they could appear in the MATLAB file:
 - `Simulink.ConfigSet` objects with custom target components.
(Use the `Simulink.ConfigSet` method `saveAs` instead.)
 - `Simulink.Timeseries` and `Simulink.ModelDataLogs` objects.

If you save many variables, the generated MATLAB file can contain many lines of code and take a long time to execute. To avoid the long execution time, consider these alternatives:

- Permanently store variables in a data dictionary instead of using `Simulink.saveVars`. A data dictionary also provides more tools for managing variables. See “Determine Where to Store Variables and Objects for Simulink Models”.
- Save variables in a MAT-file by using the `save` function.

Tips

- If you do not need to save variables in an easily-understood form, see the `save` function.
- If you need to save only bus objects, use the `Simulink.Bus.save` function.
- If you need to save only a configuration set, use the `Simulink.ConfigSet.saveAs` method.

See Also

`Simulink.Bus.save` | `Simulink.Bus.save` | `Simulink.ConfigSet` |
`matlab.io.saveVariablesToScript` | `save`

Introduced in R2010a

Simulink.sdi.addToRun

Add data to existing run

Syntax

```
signalIDs = Simulink.sdi.addToRun(runID, 'base', varName)
signalIDs = Simulink.sdi.addToRun(runID, 'model', modelNameOrHandle)
signalIDs = Simulink.sdi.addToRun(runID, 'vars', var)
signalIDs = Simulink.sdi.addToRun(runID, 'namevalue', dataName,
dataValue)
```

Description

`signalIDs = Simulink.sdi.addToRun(runID, 'base', varName)` adds data, `varName`, from the base workspace to an existing run, specified by `runID`.

`signalIDs = Simulink.sdi.addToRun(runID, 'model', modelNameOrHandle)` adds model simulation data, specified on the **Data Import/Export** pane of the Configuration Parameters dialog box, to an existing run, specified by `runID`. Open the model before you use this syntax.

`signalIDs = Simulink.sdi.addToRun(runID, 'vars', var)` adds data stored as variables, `var`, from the calling workspace to an existing run, specified by `runID`.

`signalIDs = Simulink.sdi.addToRun(runID, 'namevalue', dataName, dataValue)` adds simulation data `dataValue`, to an existing run, specified by `runID`, and lets you specify a name, `dataName`, for the data in the run.

Examples

Add Simulation Data from Base Workspace

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput', 'on', 'SaveFormat', ...
```

```
        'StructureWithTime', 'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('My Run')

% Add simulation output from the base workspace
Simulink.sdi.addToRun(runID, 'base', {'simOut'});

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

Add Simulation Data As Specified in a Model

```
% Open the model
sldemo_absbrake;
```

Click **Run** to simulate the model.

```
% Create a Data Inspector run
runID = Simulink.sdi.createRun('My Run');
Simulink.sdi.addToRun(runID, 'model', 'sldemo_absbrake');

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

Add Simulation Data by Passing Variables Directly to `simulink.sdi.addToRun`

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput', 'on', 'SaveFormat', ...
            'StructureWithTime', 'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('My Run');
Simulink.sdi.addToRun(runID, 'vars', simOut);

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

Add Simulation Data and Name the Data

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
            'SaveFormat','StructureWithTime', ...
            'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('My Run');
% Name simulation output passed to Simulink.sdi.addToRun
Simulink.sdi.addToRun(runID,'namevalue',{'MyData'},{simOut});

% See the results in Simulation Data Inspector
Simulink.sdi.view;
```

- “Inspect and Compare Data Programmatically”

Input Arguments

runID — Unique run identifier

integer

Unique number identifying a run in the Simulation Data Inspector, specified as an integer.

varName — Base workspace data

cell array

The names of variables in the base workspace, specified as a cell array of character vectors.

Example: {'simOut'}

modelNameOrHandle — Model name

character vector

The model name, or a model handle, specified as a character vector.

Example: 'sldemo_absbrake'

var — Variable data

variable

Data stored as variables. These variables are assumed to be in the calling workspace.

Example: `simOut`

dataName — Signal data name

cell array

Name of the data in the run, specified as a cell array.

Example: `{ 'MyData' }`

dataValue — Signal data values

cell array

Values of the signal data, specified as a cell array.

Example: `{ simOut }`

Output Arguments

signalIDs — Unique signal identifier

array

Unique signal identifier, returned as an array of integers where each element is a unique ID for a signal added to the run.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.createRun` | `Simulink.sdi.view`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.cleanupWorkerResources

Clean up worker repositories

Syntax

```
Simulink.sdi.cleanupWorkerResources
```

Description

`Simulink.sdi.cleanupWorkerResources` removes redundant data from each Parallel Computing Toolbox worker file used by the Simulation Data Inspector. Call this function while worker pools are running. The Simulation Data Inspector automatically cleans up repository files when you close the worker pool.

Examples

Access Data from a Parallel Simulation

This example executes parallel simulations of the model `slexAircraftExample` with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector API.

Generate Data with Parallel Simulations

Create a vector of filter parameter values to use a different value in each simulation. Then, run a parallel simulation, selecting a different filter value for each worker.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport(1)

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];
```

```
parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Change the filter time constant and simulate
    modelWorkspace = get_param('slexAircraftExample', 'modelworkspace');
    modelWorkspace.assignin('Ts', Ts_val)
    sim('slexAircraftExample')

    % Create a worker run for each simulation
    workerRun(index) = Simulink.sdi.WorkerRun.getLatest
end
```

Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

```
ds(7) = Simulink.SimulationData.Dataset;
```

```
for a = 1:7
    ds(a) = workerRun(a).getDataset;
end
ds(1)
```

```
ans =
```

```
Simulink.SimulationData.Dataset '' with 2 elements
```

		Name	BlockPath
1	[1x1 Signal]	alpha, rad	...rcraftExample/Aircraft Dynamics Model
2	[1x1 Signal]	Stick	slexAircraftExample/Pilot

- Use braces { } to access, modify, or add elements using index.

Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the `getDatasetRef` method to reference the data associated with the `WorkerRun`.

```
for b = 1:7
    datasetRef(b) = workerRun(b).getDatasetRef;
end
datasetRef(1)
```

```
ans =
```

```
DatasetRef with properties:
```

```
    Name: 'Run 6: slexAircraftExample'
      Run: [1x1 Simulink.sdi.Run]
numElements: 2
```

Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local `Run` objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```
for c = 1:7
    Runs(c) = workerRun(c).getLocalRun;
    Ts_val_str = num2str(Ts_vals(c));
    desc = strcat('Ts = ', Ts_val_str);
    Runs(c).Description = desc;
    Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);
end
```

Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

```
Simulink.sdi.cleanupWorkerResources
```

See Also

`Simulink.sdi.WorkerRun` | `Simulink.sdi.isPCTSupportEnabled`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

Simulink.sdi.clear

Clear all data from the Simulation Data Inspector

Syntax

```
Simulink.sdi.clear
```

Description

`Simulink.sdi.clear` clears all run data from the Simulation Data Inspector.

Examples

Remove All Runs from the Simulation Data Inspector

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
            'SaveFormat','StructureWithTime');
```

```
% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('First Run','base',{ 'simOut' });
```

```
Simulink.sdi.clear;
```

```
% The number of runs is now zero
runCount = Simulink.sdi.getRunCount()
```

- “Inspect and Compare Data Programmatically”

See Also

`Simulink.sdi.clearPreferences` | `Simulink.sdi.close`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.clearPreferences

Clear Simulation Data Inspector preference changes

Syntax

```
Simulink.sdi.clearPreferences
```

Description

`Simulink.sdi.clearPreferences` reverts all Simulation Data Inspector preferences to their default values.

Examples

Restore All Simulation Data Inspector Preferences to Default Values

You can restore default values to all Simulation Data Inspector preferences programmatically with `Simulink.sdi.clearPreferences`.

```
Simulink.sdi.clearPreferences
```

Modify Run Naming Rule Then Restore Default

This example shows how to use the Simulation Data Inspector API to modify the Simulation Data Inspector run naming rule, check a run's name, restore default preferences, and check the run naming rule.

```
% Load model
load_system('sldemo_fuelsys')

% Modify run naming rule
Simulink.sdi.setRunNamingRule('<model_name> Run <run_index>')
```

```
% Simulate system
sim('sldemo_fuelsys')

% Check run name
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
run = Simulink.sdi.getRun(runID);
run.name

ans =

    'sldemo_fuelsys Run 11'

% Clear preferences to reset the run naming rule
Simulink.sdi.clearPreferences

% Check run naming rule
Simulink.sdi.getRunNamingRule

ans =

    'Run <run_index>: <model_name>'
```

- “Inspect and Compare Data Programmatically”
- “Organize Your Simulation Data Inspector Workspace”

See Also

[Simulink.sdi.setMarkersOn](#) | [Simulink.sdi.setRunNamingRule](#) | [Simulink.sdi.setRunOverwrite](#) | [Simulink.sdi.setSubPlotLayout](#) | [Simulink.sdi.setTableGrouping](#)

Topics

“Inspect and Compare Data Programmatically”
“Organize Your Simulation Data Inspector Workspace”

Introduced in R2017a

Simulink.sdi.close

Package: Simulink.sdi

Close the Simulation Data Inspector

Syntax

```
Simulink.sdi.close  
Simulink.sdi.close('filename')
```

Description

`Simulink.sdi.close` closes the Simulation Data Inspector.

`Simulink.sdi.close('filename')` closes the Simulation Data Inspector and saves the data in the file, `filename`.

Examples

Close the Simulation Data Inspector from the Command Line

You can close the Simulation Data Inspector from the MATLAB command line when you have finished inspecting and analyzing your data.

```
Simulink.sdi.close
```

Create a Run and View the Data

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.

Create Data for the Run

This example creates `timeseries` objects for a sine and a cosine. To visualize your data, the Simulation Data Inspector requires at least a time vector that corresponds with your data.

```
% Generate timeseries data
time = linspace(0, 20, 100);

sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals, time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals, time);
cos_ts.Name = 'Cosine, T=8';
```

Create a Simulation Data Inspector Run and Add Your Data

To give the Simulation Data Inspector access to your data, use the `create` method and create a run. This example modifies some of the run's properties to help identify the data. You can easily view run and signal properties with the Simulation Data Inspector.

```
% Create a run
run = Simulink.sdi.Run.create;
run.Name = 'Sinusoids';
run.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
run.add('vars', sine_ts, cos_ts);
```

Plot Your Data Using the `Simulink.sdi.Signal` Object

The `getSignalByIndex` method returns a `Simulink.sdi.Signal` object that can be used to plot the signal in the Simulation Data Inspector. You can also programmatically control aspects of the plot's appearance, such as the color and style of the line representing the signal. This example customizes the subplot layout and signal characteristics.

```
% Get signal, modify its properties, and change Checked property to true
sine_sig = run.getSignalByIndex(1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';
sine_sig.Checked = true;
```

```
% Add another subplot for the cosine signal
Simulink.sdi.setSubPlotLayout(2, 1);

% Plot the cosine signal and customize its appearance
cos_sig = run.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.plotOnSubPlot(2, 1, true);

% View the signal in the Simulation Data Inspector
Simulink.sdi.view
```

Close the Simulation Data Inspector and Save Your Data

```
Simulink.sdi.close('sinusoids.mat')
```

- “Inspect and Compare Data Programmatically”
- “Save and Share Simulation Data Inspector Data and Views”

See Also

```
Simulink.sdi.clear | Simulink.sdi.clearPreferences | Simulink.sdi.save  
| Simulink.sdi.view
```

Topics

- “Inspect and Compare Data Programmatically”
- “Save and Share Simulation Data Inspector Data and Views”

Introduced in R2013b

Simulink.sdi.compareRuns

Package: Simulink.sdi

Compare the data in two simulation runs

Syntax

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

Description

`diffResult = Simulink.sdi.compareRuns(runID1, runID2)` compares the data in the runs corresponding to `runID1` and `runID2`, returning the result in the `Simulink.sdi.DiffRunResult` object, `diffResult`.

`diffResult = Simulink.sdi.compareRuns(runID1, runID2, alignmentMethods)` compares the simulation runs corresponding to `runID1` and `runID2` using the specified alignment criteria and returns the comparison result in the `Simulink.sdi.DiffRunResult` object, `diffResult`. For more information about how the Simulation Data Inspector aligns signals for comparison, see “How the Simulation Data Inspector Compares Data”.

Examples

Compare Simulation Data Inspector Runs Programmatically

This example shows how to compare runs of simulation data and access the comparison results with the Simulation Data Inspector API.

Generate Runs of Simulation Data

Simulate the model to create runs of simulation data to analyze with the Simulation Data Inspector API.

```
% Open model
load_system('ex_sldemo_absbrake')

% Set the desired slipratio to 0.24 and simulate
set_param('ex_sldemo_absbrake/Desired relative slip','Value','0.24')
sim('ex_sldemo_absbrake');

% Change the desired slip ratio to 0.25 and simulate
set_param('ex_sldemo_absbrake/Desired relative slip','Value','0.25')
sim('ex_sldemo_absbrake');
```

Compare Runs

Get the run IDs for the runs you just created with the `Simulink.sdi.getAllRunIDs` function.

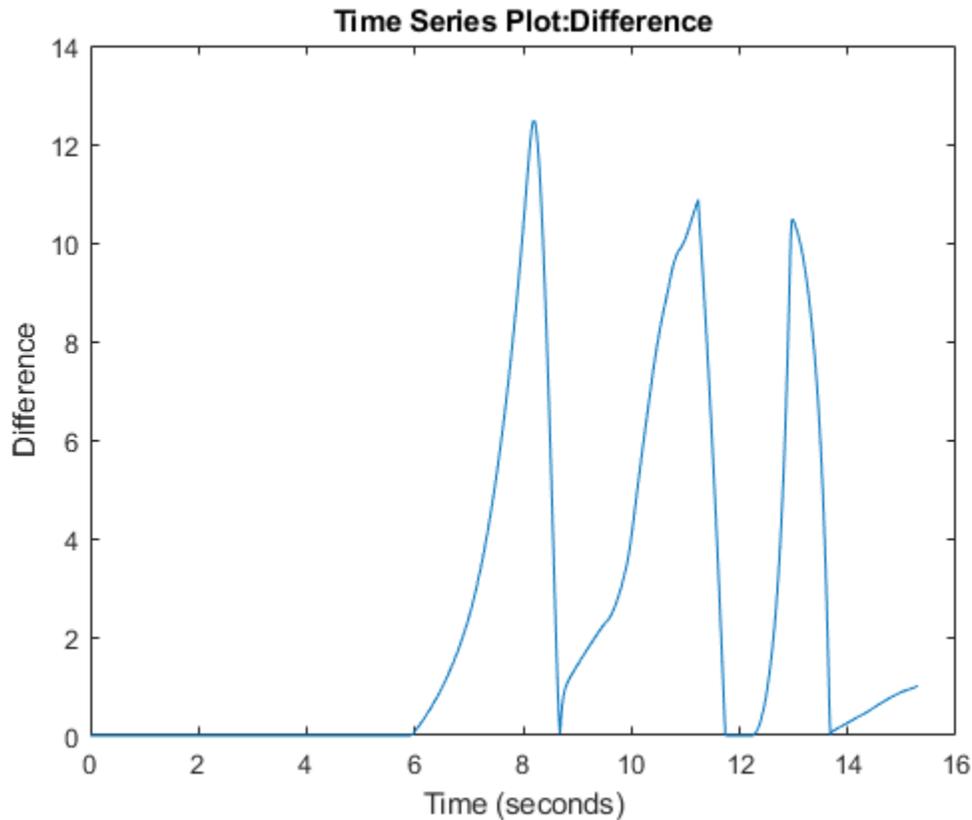
```
% Get run IDs for last two runs
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);

% Compare runs
runResult = Simulink.sdi.compareRuns(runID1, runID2);
```

Create a Plot of a Comparison Result

Use the `Simulink.sdi.DiffRunResult` object you created in the previous step with `Simulink.sdi.compareRuns` to access the data for the `Ww` signal result to plot it in a figure.

```
% Plot the |Ww| signal difference
signalResult_Ww = runResult.getResultByIndex(1);
figure(1)
plot(signalResult_Ww.Diff)
```



Analyze Simulation Data with Signal Tolerances

You can change tolerance values on a signal-by-signal basis to evaluate the effect of a model parameter change. This example uses the `slexAircraftExample` model and the Simulation Data Inspector to evaluate the effect of changing the time constant for the low-pass filter following the control input.

```
% Load example model
load_system('slexAircraftExample')

% Mark the alpha, rad signal for streaming
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'alpha')
```

```
% Simulate system
sim('slexAircraftExample')

% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample', 'modelworkspace');
modelWorkspace.assignin('Ts', 0.2)

% Simulate again
sim('slexAircraftExample')

% Get run data
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);

% Compare runs
diffRun1 = Simulink.sdi.compareRuns(runID1, runID2);

% Get signal result
sigResult1 = diffRun1.getResultByIndex(1);

% Check whether signals matched
sigResult1.match

ans =

    logical

     0

% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = run1.getSignalIDByIndex(1);
sig1 = Simulink.sdi.getSignal(sigID1);

% Change absolute tolerance to 0.2
sig1.absTol = 0.2;

% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1, runID2);
sigResult2 = diffRun2.getResultByIndex(1);
```

```
% Check the result
sigResult2.match

ans =

    logical

     1
```

Compare Runs with Alignment Criteria

This example shows how to compare runs using your desired criteria for aligning signals between runs.

Generate Runs to Compare

This example simulates the `slexAircraftExample` model with two different values of `Ts` to generate two simulation runs for comparison.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput', 'on', 'SaveFormat', ...
            'Dataset', 'ReturnWorkspaceOutputs', 'on');

% Create a run from the simulation output
runID1 = Simulink.sdi.createRun('My Run', 'namevalue', {'simOut'}, ...
                               {simOut});

% Get workspace and change workspace variable Ts
modelWorkspace = get_param('slexAircraftExample', 'modelworkspace');
modelWorkspace.assignin('Ts', 0.2);

% Simulate again
simOut = sim('slexAircraftExample', 'SaveOutput', 'on', 'SaveFormat', ...
            'Dataset', 'ReturnWorkspaceOutputs', 'on');

% Create another run and get signal IDs
runID2 = Simulink.sdi.createRun('New Run', 'namevalue', {'simOut'}, ...
                               {simOut});
```

Define Alignment Criteria for the Comparison

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.

```
% Define the alignment criteria for the comparison to align signals by
% name, then by block path, then by SID
alignMethods = [Simulink.sdi.AlignType.SignalName
                Simulink.sdi.AlignType.BlockPath
                Simulink.sdi.AlignType.SID];
```

Compare the Runs with the Specified Alignment Criteria

Compare the simulation data in your two runs, with the alignment criteria you specified.

```
% Compare the runs
diffResults = Simulink.sdi.compareRuns(runID1 ,runID2, alignMethods);
```

Check the Comparison Results for the Aligned Signals

You can use the `getResultByIndex` method to access the aligned signals and the results of the comparison for each signal in the runs you compared.

```
% Check the number of comparisons in the result
numComparisons = diffResults.count;

% Iterate through each element and display results in command window
for k = 1:numComparisons

    resultAtIdx = diffResults.getResultByIndex(k);

    % Get signal IDs for each comparison result
    sigID1 = resultAtIdx.signalID1;
    sigID2 = resultAtIdx.signalID2;

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

    % Display whether signals match
    displayStr = 'Signals with IDs %s and %s %s \n';
    if resultAtIdx.match
        fprintf(displayStr, sig1.Name, sig2.Name, 'match');
    else
        fprintf(displayStr, sig1.Name, sig2.Name, 'do not match');
```

end

end

```

Signals with IDs Actuator Model and Actuator Model do not match
Signals with IDs Integrate qdot and Integrate qdot do not match
Signals with IDs Integrate and Integrate do not match
Signals with IDs Alpha-sensor Low-pass Filter and Alpha-sensor Low-pass Filter do not match
Signals with IDs Pitch Rate Lead Filter and Pitch Rate Lead Filter do not match
Signals with IDs Proportional plus integral compensator and Proportional plus integral compensator do not match
Signals with IDs Stick Prefilter and Stick Prefilter do not match
Signals with IDs Q-gust model and Q-gust model do not match
Signals with IDs W-gust model[1] and W-gust model[1] do not match
Signals with IDs W-gust model[2] and W-gust model[2] do not match
Signals with IDs alpha, rad and alpha, rad do not match

```

- “Inspect and Compare Data Programmatically”
- “Compare Simulation Data”
- “How the Simulation Data Inspector Compares Data”

Input Arguments

runID1 — Baseline run identifier

integer

Numerical identification for the **Baseline** run in the comparison.

runID2 — Compare to run identifier

integer

Numerical identification for the **Compare to** run in the comparison.

alignmentMethods — Signal alignment methods

array

Array specifying alignment algorithms to use to pair signals with the runs for comparison. The Simulation Data Inspector aligns signals first by the first element in the array, then by the second element in the array, and so on.

Value	Aligns By
Simulink.sdi.AlignType.BlockPath	Path to the signal's source block

Value	Aligns By
<code>Simulink.sdi.AlignType.SID</code>	Simulink identifier “Locate Diagram Components Using Simulink Identifiers”
<code>Simulink.sdi.AlignType.SignalName</code>	Signal name
<code>Simulink.sdi.AlignType.DataSource</code>	Path of the variable in the MATLAB workspace

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

Output Arguments

diffResult — Comparison results

'Simulink.sdi.diffRunResult'

`Simulink.sdi.DiffRunResult` object that provides access to comparison results.

See Also

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

Topics

“Inspect and Compare Data Programmatically”

“Compare Simulation Data”

“How the Simulation Data Inspector Compares Data”

Introduced in R2011b

Simulink.sdi.compareSignals

Compare data from two signals

Syntax

```
diff = Simulink.sdi.compareSignals(signalID1,signalID2)
```

Description

`diff = Simulink.sdi.compareSignals(signalID1,signalID2)` compares two signals and returns the result in a `Simulink.sdi.DiffSignalsResult` object.

Examples

Compare Two Signals

Call `Simulink.sdi.createRun` to get signal IDs for a simulation run in the Simulation Data Inspector. The function `Simulink.sdi.compareSignals` returns a `Simulink.sdi.DiffSignalResult` object containing the result of the comparison.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
            'SaveFormat','StructureWithTime',...
            'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run and get signal IDs
[~,~,signalIDs] = Simulink.sdi.createRun('My Run','namevalue',{'MyData'},{simOut});

sig1 = signalIDs(1);
sig2 = signalIDs(2);

% Compare two signals, which returns results in
% instance of a Simulink.sdi.DiffSignalResult object
diff = Simulink.sdi.compareSignals(sig1,sig2);
```

```
% Find if the signal data match
match = diff.match;

% Get the tolerance used in Simulink.sdi.compareSignals
tolerance = diff.tol;
```

Compare Signals From Two Different Runs

```
% Load the model 'slexAircraftExample'
load_system('slexAircraftExample');

% Configure model "slexAircraftExample" for logging
set_param('slexAircraftExample','SolverType','Fixed-Step','SaveOutput','on',...
          'SaveFormat','StructureWithTime','ReturnWorkspaceOutputs',...
          'on');

% CD to temporary directory and build
cd(tempdir);
rtwbuild('slexAircraftExample');

% Run the executable
if ispc
    system('slexAircraftExample');
elseif unix
    system('./slexAircraftExample');
end

% Create a Data Inspector run using slexAircraftExample.mat created in the current
% directory
[~,~,signalIDs] = Simulink.sdi.createRun('My Run','file','slexAircraftExample.mat');

% Get first signal id to compare
sig1 = signalIDs(1);

% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
            'SaveFormat','StructureWithTime',...
            'ReturnWorkspaceOutputs','on');

% Create a Data Inspector run and get signal IDs
[~,~,signalIDs] = Simulink.sdi.createRun('My Run','namevalue',...;
```

```

                                {'MyData'}, {simOut});

% Get second signal id to compare
sig2 = signalIDs(1);

% compare two signals
result = Simulink.sdi.compareSignals(sig1, sig2);
if result.match
    disp('****The signals match****');
else
    disp('****The signals did not match****');
end

% Plot results in a figure
plot(result.tol, 'Color', 'r');
hold on;
plot(result.diff, 'Color', 'g');
legend('Tolerance', 'Difference');

```

- “Inspect and Compare Data Programmatically”

Input Arguments

signalID1 — Unique signal identifier

integer

Signal ID, a unique number identifying the first signal for comparison.

signalID2 — Unique signal identifier

integer

Signal ID, a unique number identifying the second signal for comparison.

Output Arguments

diff — Comparison difference data

object

Simulink.sdi.DiffSignalResult object containing the results of the comparison.

See Also

`Simulink.sdi.DiffSignalResult` | `Simulink.sdi.createRun`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.copyRun

Create a copy of a run

Syntax

```
runIDcopy = Simulink.sdi.copyRun(runID)
[runIDcopy,runIndex] = Simulink.sdi.copyRun(runID)
[runIDcopy,runIndex,signalIDs] = Simulink.sdi.copyRun(runID)
```

Description

`runIDcopy = Simulink.sdi.copyRun(runID)` copies the run associated with `runID` and returns a run ID, `runIDcopy`, associated with the new run. The new run contains all of the simulation output data and metadata from the original run.

`[runIDcopy,runIndex] = Simulink.sdi.copyRun(runID)` copies the run associated with `runID` and returns the run ID, `runIDcopy`, and the `runIndex` for the new run.

`[runIDcopy,runIndex,signalIDs] = Simulink.sdi.copyRun(runID)` copies the run associated with `runID` and returns the run ID, run index, and array of new signal IDs, `signalIDs`, for signals in the new run.

Examples

Copy Simulink.sdi.Run Object Representing a Run In the Simulation Data Inspector

```
% Configure model 'slexAircraftExample' for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
            'SaveFormat','StructureWithTime',...
            'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
```

```
runID = Simulink.sdi.createRun('First Run', 'base', {'simOut'});  
  
[newRunID, runIndex, signalIDs] = Simulink.sdi.copyRun(runID);  
  
% See the results in Simulation Data Inspector  
Simulink.sdi.view;
```

- “Inspect and Compare Data Programmatically”

Input Arguments

runID — Unique run identifier

integer

Run ID, a unique number identifying a run in the Simulation Data Inspector, specified as an integer.

Output Arguments

runIDcopy — Unique run identifier

integer

The unique number identifying the copied run, returned as an integer.

runIndex — Simulation run index

integer

Number representing the new index to the list of runs currently in the Simulation Data Inspector, returned as an integer.

signalIDs — Unique signal identifiers

array

Vector of numbers, where each element is a unique ID for a signal in this run. The signal IDs are different in the new run.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.view`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.copyRunViewSettings

Copy signal line style and color specifications from one run to another run

Syntax

```
alignedSignals = Simulink.sdi.copyRunViewSettings(sourceRun,  
destinationRun,moveCheckboxes)
```

Description

`alignedSignals = Simulink.sdi.copyRunViewSettings(sourceRun, destinationRun,moveCheckboxes)` copies the signal line style and color from one run to another run. The function returns an array of signal identifiers that align between runs where the settings were copied.

Examples

Copy Signal Line Style and Color

```
% Configure model 'slexAircraftExample' for logging and simulate  
simOut = sim('slexAircraftExample','SaveOutput','on',...  
            'SaveFormat','StructureWithTime',...  
            'ReturnWorkspaceOutputs','on');  
  
% Create the first Simulation Data Inspector run  
runID_1 = Simulink.sdi.createRun('First Run','base',{'simOut'});  
  
% Simulate the model again  
simOut = sim('slexAircraftExample','SaveOutput','on',...  
            'SaveFormat','StructureWithTime',...  
            'ReturnWorkspaceOutputs','on');  
  
% Create second run in the Simulation Data Inspector  
runID_2 = Simulink.sdi.createRun('Second Run','base',{'simOut'});
```

```
% Copy the signal settings from first run to the second run  
alignedSignals = Simulink.sdi.copyRunViewSettings(runID_1,runID_2,true);
```

- “Inspect and Compare Data Programmatically”

Input Arguments

sourceRun — Source run identifier

integer or object

A run ID, specified as an integer, or a `Simulink.sdi.Run` object, specified as an object, for the source run.

destinationRun — Destination run identifier

integer or object

A run ID, specified as an integer, or a `Simulink.sdi.Run` object, specified as an object, for the destination run.

moveCheckboxes — Move plotted signals

`false` (default) | `true`

Indicate to move the check boxes in the **Runs** pane, which plot signals, from the source run to the destination run, specified as `true` or `false`.

Output Arguments

alignedSignals — Aligned signals between runs

integer array

Signals that match between runs where the settings were copied, returned as an array of signal identifiers. See “Signal Alignment in the Simulation Data Inspector” for more information on signal alignment.

See Also

`Simulink.sdi.copyRun`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2016a

Simulink.sdi.createRun

Create a run in the Simulation Data Inspector

Syntax

```
runID = Simulink.sdi.createRun
runID = Simulink.sdi.createRun(var)
runID = Simulink.sdi.createRun(runName)
runID = Simulink.sdi.createRun(runName, 'base', varName)
runID = Simulink.sdi.createRun(runName, 'model', modelNameOrHandle)
runID = Simulink.sdi.createRun(runName, 'vars', var)
runID = Simulink.sdi.createRun(runName, 'namevalue', dataName,
dataValue)
runID = Simulink.sdi.createRun(runName, 'file', fileName)

[runID, runIndex] = Simulink.sdi.createRun( ___ )
[runID, runIndex, signalIDs] = Simulink.sdi.createRun( ___ )
```

Description

`runID = Simulink.sdi.createRun` creates an empty unnamed run in the Simulation Data Inspector and returns the corresponding run ID.

`runID = Simulink.sdi.createRun(var)` creates a run in the Simulation Data Inspector with data `var` from the base workspace.

`runID = Simulink.sdi.createRun(runName)` creates an empty run named `runName` in the Simulation Data Inspector and returns the corresponding run ID.

`runID = Simulink.sdi.createRun(runName, 'base', varName)` creates a run in the Simulation Data Inspector with data, `varName`, from the base workspace.

`runID = Simulink.sdi.createRun(runName, 'model', modelNameOrHandle)` creates a run in the Simulation Data Inspector with model simulation output data, as specified on the **Data Import/Export** pane of the Configuration Parameters dialog box.

`modelNameOrHandle` is a character vector specifying the model name of a model handle. Open the model before you use this syntax.

`runID = Simulink.sdi.createRun(runName, 'vars', var)` creates a run in the Simulation Data Inspector with data stored in variables, `var`. These variables must be in the calling workspace.

`runID = Simulink.sdi.createRun(runName, 'namevalue', dataName, dataValue)` creates a run in the Simulation Data Inspector from simulation data `dataValue`. `dataName` specifies a name for the data.

`runID = Simulink.sdi.createRun(runName, 'file', fileName)` creates a run in the Simulation Data Inspector with data from a MAT-file, `fileName`.

`[runID, runIndex] = Simulink.sdi.createRun(____)` creates a run in the Simulation Data Inspector and returns the run ID and the run index. Use this option with any of the input argument combinations in the previous syntaxes.

`[runID, runIndex, signalIDs] = Simulink.sdi.createRun(____)` creates a run in the Simulation Data Inspector and returns the run ID, the run index, and the signal IDs. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Create Empty Run With No Name

```
runID = Simulink.sdi.createRun;
```

Create Run Directly From Simulation Output

```
% Configure the model sllexAircraftExample for logging and simulate
simOut = sim('sllexAircraftExample', 'SaveOutput', 'on', ...
            'SaveFormat', 'StructureWithTime', ...
            'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run from the simulation
% output data in the base workspace
```

```

Simulink.sdi.createRun(simOut);

% Open the Simulation Data Inspector tool to view the data
Simulink.sdi.view;

```

Create Run From Simulation Output In the Base Workspace

```

% Configure the model slexAircraftExample for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput', 'on', ...
            'SaveFormat', 'StructureWithTime', ...
            'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run from the simulation
% output data in the base workspace
Simulink.sdi.createRun('My Run', 'base', {'simOut'});

% Open the Simulation Data Inspector tool to view the data
Simulink.sdi.view;

```

Create Run Using Simulation Output From a Model

The model must be open to use this function signature.

```

% Open the model sldemo_absbrake
sldemo_absbrake;

```

Click **Run** to simulate the model. The model is already configured for signal logging.

```

% Create a Simulation Data Inspector run named 'My Run' using
% simulation output data from the model
Simulink.sdi.createRun('My Run', 'model', 'sldemo_absbrake');

% Open the Simulation Data Inspector tool to view the data
Simulink.sdi.view;

```

Create Run Using Passed Variables

```

% Configure the model slexAircraftExample for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput', 'on', ...

```

```
        'SaveFormat', 'StructureWithTime', ...
        'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run named 'My Run' using
% simulation output data from the model
Simulink.sdi.createRun('My Run', 'vars', simOut);

% Open the Simulation Data Inspector tool to view the data
Simulink.sdi.view;
```

Create Run and Include Name of Simulation Data

```
% Configure the model slexAircraftExample for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput', 'on', ...
            'SaveFormat', 'StructureWithTime', ...
            'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run named 'My Run' using
% simulation output data from the model
Simulink.sdi.createRun('My Run', 'namevalue', {'MyData'}, {simOut});

% Open the Simulation Data Inspector tool to view the data
Simulink.sdi.view;
```

Create Run Using MAT-File Data

This example includes data from a code generation build and requires Simulink Coder.

```
% Load the model slexAircraftExample
load_system('slexAircraftExample');

% Configure the model for logging and simulate
set_param('slexAircraftExample', 'SolverType', 'Fixed-Step', ...
        'SaveOutput', 'on', ...
        'SaveFormat', 'StructureWithTime', ...
        'ReturnWorkspaceOutputs', 'on');

% Build the model to a temporary directory
cd(tempdir);
rtwbuild('slexAircraftExample');
```

```

% Run the executable
if ispc
    system('slexAircraftExample');
elseif unix
    system('./slexAircraftExample');
end

% A MAT-file is generated in the current directory

% Create a Simulation Data Inspector run using the data in the MAT-file
Simulink.sdi.createRun('My Run','file','slexAircraftExample.mat');

% Open the Simulation Data Inspector tool to view the data
Simulink.sdi.view;

```

A run named, My Run, appears in the Simulation Data Inspector.

- “Inspect and Compare Data Programmatically”

Input Arguments

runName — Run name

character vector

Name of the run as it appears in the Simulation Data Inspector, specified as a character vector.

varName — Base workspace data

cell array

The names of variables in the base workspace, specified as a cell array of character vectors.

Example: {'simOut'}

modelNameOrHandle — Model name

character vector

The model name, or a model handle, specified as a character vector.

Example: 'sldemo_absbrake'

var — Variable data

variable

Data stored as variables. These variables are assumed to be in the calling workspace.

Example: `simOut`

dataName — Signal data name

cell array

Name of the data in the run, specified as a cell array.

Example: `{ 'MyData' }`

dataValue — Signal data values

cell array

Values of the signal data, specified as a cell array.

Example: `{ simOut }`

fileName — Simulation data file name

character vector

The file name and path of a MAT-file containing simulation data, specified as a character vector.

Example: `'slexAircraftExample.mat'`

Output Arguments

runID — Unique run identifier

integer

Unique number identifying a run in the Simulation Data Inspector, returned as an integer.

runIndex — Simulation run index

integer

Number representing an index to the list of runs currently in the Simulation Data Inspector, returned as an integer.

signalIDs — Unique signal identifiers

array

Vector of numbers, where each element is a unique ID for a signal in a run.

Tips

- Before calling `Simulink.sdi.createRun` with either 'base' or 'model' as an input argument, you must configure the model for logging and simulate the model.
- When you create and add a run, the Simulation Data Inspector maintains a list of these runs. The first run in the list is given a `runIndex` of 1. If you delete a run from the Simulation Data Inspector, the subsequent runs move up the list and each `runIndex` changes. However, the run IDs remain the same.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.deleteRun` | `Simulink.sdi.getRun`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.createRunOrAddToStreamedRun

Create a single run for all simulation outputs

Syntax

```
runID = Simulink.sdi.createRunOrAddToStreamedRun mdl, runName,  
varNames, varValues)
```

Description

`runID = Simulink.sdi.createRunOrAddToStreamedRun(mdl, runName, varNames, varValues)` creates a new run with the data identified by `varNames` and `varValues` if no run exists for the model identified by `mdl`. The function adds the data identified by `varNames` and `varValues` to an existing run if a run associated with `mdl` already exists.

Examples

Generate and Add Data to Run

```
%load system  
load_system('slexAircraftExample')  
  
%mark signal for logging  
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', ...  
    4, 'on')  
  
%simulate model  
sim('slexAircraftExample')  
  
%create cells for names and data  
names = cell({'tout', 'xout', 'yout'});  
data = cell({tout, xout, yout});
```

```
%add data to run
Simulink.sdi.createRunOrAddToStreamedRun('slexAircraftExample', ...
    'Run 1', names, data);
```

Input Arguments

mdl — Name of the model the simulation data is from

character vector

Name of the model the simulation data is from, specified as a character vector.

Example: 'sldemo_absbrake'

runName — Run name

character vector

Name for the new or augmented run.

varNames — Names for data

cell array

Cell array containing the names of all signals and variables.

varValues — Run data

cell array

Cell array containing all of the data for signals and variables.

Output Arguments

runID — Unique numerical identifier for the created or augmented run

integer

Numerical identifier for created or augmented run.

See Also

[Simulink.sdi.addToRun](#) | [Simulink.sdi.createRun](#)

Introduced in R2017a

Simulink.sdi.deleteRun

Delete a run from the Simulation Data Inspector

Syntax

```
Simulink.sdi.deleteRun(runID)
```

Description

`Simulink.sdi.deleteRun(runID)` deletes a run associated with the run ID in the Simulation Data Inspector. Deleting the run removes all data included in the run. After deleting a run, the subsequent runs move up the list, and the run index for each run changes. However, the unique run IDs remain the same.

Examples

Remove Run from the Simulation Data Inspector

```
% Configure model 'slexAircraftExample' for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
            'SaveFormat','StructureWithTime');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('First Run','base',{'simOut'});

Simulink.sdi.deleteRun(runID);
```

- “Inspect and Compare Data Programmatically”

Input Arguments

runID — Unique run identifier
integer

A unique simulation run identifier, specified as an integer. The `Simulink.sdi.createRun` function creates a unique run ID. The run ID can also be accessed for a particular run using the `Simulink.sdi.getRunIDByIndex` function.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.clear` | `Simulink.sdi.copyRun` |
`Simulink.sdi.createRun`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.deleteSignal

Delete a signal from the Simulation Data Inspector

Syntax

```
Simulink.sdi.deleteSignal(signalID)
```

Description

`Simulink.sdi.deleteSignal(signalID)` deletes a signal from the Simulation Data Inspector **Inspect** pane.

Examples

Delete Signal From Simulation Data Inspector

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on','SaveFormat',...
             'StructureWithTime');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('First Run','base',{'simOut'});

% Delete the first signal in this run
run = Simulink.sdi.getRun(runID);
signalID = run.getSignalIDByIndex(1);
Simulink.sdi.deleteSignal(signalID);
```

- “Inspect and Compare Data Programmatically”

Input Arguments

signalID — Unique signal identifier

integer

Unique number identifying a signal in a run, specified as an integer. To find the signal ID, you can use the method `Simulink.sdi.Run.getSignalIDByIndex`.

See Also

`Simulink.sdi.clear` | `Simulink.sdi.deleteRun`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2016a

Simulink.sdi.enablePCTSupport

Control how the Simulation Data Inspector works with the Parallel Computing Toolbox

Syntax

```
Simulink.sdi.enablePCTSupport(enable)
```

Description

`Simulink.sdi.enablePCTSupport(enable)` enables or disables support for automatically importing data from Parallel Computing Toolbox workers into the Simulation Data Inspector, depending on the value of the logical input `enable`.

Examples

Enable Support for Parallel Computing Toolbox

Enable support for Parallel Computing Toolbox to automatically import the output from Parallel Computing Toolbox workers to the Simulation Data Inspector.

```
Simulink.sdi.enablePCTSupport(1)
```

Disable Support for Parallel Computing Toolbox

Disable support for Parallel Computing Toolbox to prevent the output from Parallel Computing Toolbox workers from automatically importing to the Simulation Data Inspector.

```
Simulink.sdi.enablePCTSupport(0)
```

Input Arguments

enable — Logical enable

logical

Logical enable for Parallel Computing Toolbox.

- `true` or `1` enables support for Parallel Computing Toolbox.
- `false` or `0` disables support for Parallel Computing Toolbox.

Data Types: `logical`

See Also

`Simulink.sdi.WorkerRun` | `Simulink.sdi.isPCTSupportEnabled`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

Simulink.sdi.exportRun

Create a Simulink.SimulationData.Dataset object of the data in the specified run

Syntax

```
ds = Simulink.sdi.exportRun(runID)
```

Description

`ds = Simulink.sdi.exportRun(runID)` creates a Simulink.SimulationData.Dataset object of the run in the Simulation Data Inspector identified by `runID`.

Examples

Generate and Export a Run in the Simulation Data Inspector

```
%load example system
load_system('sldemo_fuelsys')

%simulate system and create a run in the Simulation Data Inspector
simOut = sim('sldemo_fuelsys', 'SaveFormat', 'StructureWithTime');

runID = Simulink.sdi.createRun(simOut);

%export run to Simulink.SimulationData.Dataset object
ds = Simulink.sdi.exportRun(runID);
```

Input Arguments

runID — Numerical run identifier
integer

Number identifying a run in the Simulation Data Inspector.

Output Arguments

ds — **Simulink.SimulationData.Dataset** object

`Simulink.SimulationData.Dataset`

`Simulink.SimulationData.Dataset` object containing the data of the run corresponding to the `runID`.

See Also

`Simulink.sdi.createRun`

Introduced in R2017a

Simulink.sdi.getAllRunIDs

Get all run identifiers within the Simulation Data Inspector

Syntax

```
runIDs = Simulink.sdi.getAllRunIDs
```

Description

`runIDs = Simulink.sdi.getAllRunIDs` returns an n-by-1 matrix of all numerical run identifiers for runs in the **Inspect** pane of the Simulation Data Inspector.

Examples

Retrieve All Run IDs After Several Simulations

```
%load example system
load_system('slexAircraftExample')

%simulate system and save output
simOut = sim('slexAircraftExample', 'SaveFormat', 'StructureWithTime');
simOut1 = sim('slexAircraftExample', 'SaveFormat', 'StructureWithTime');

%create run from simulation output
Simulink.sdi.createRun(simOut);
Simulink.sdi.createRun(simOut1);

%get run IDs for runs in the Simulation Data Inspector
runIDs = Simulink.sdi.getAllRunIDs;
```

- “Inspect and Compare Data Programmatically”
- “Organize Your Simulation Data Inspector Workspace”

Output Arguments

runIDs — Matrix of all run IDs in the Simulation Data Inspector

n-by-1 matrix

An n-by-1 matrix containing all of the run IDs for runs of the specified group in the Simulation Data Inspector.

See Also

`Simulink.sdi.addToRun` | `Simulink.sdi.compareRuns` | `Simulink.sdi.copyRun` | `Simulink.sdi.createRun` | `Simulink.sdi.createRunOrAddToStreamedRun` | `Simulink.sdi.deleteRun` | `Simulink.sdi.exportRun` | `Simulink.sdi.getRun` | `Simulink.sdi.getRunCount` | `Simulink.sdi.getRunIDByIndex` | `Simulink.sdi.getRunNamingRule` | `Simulink.sdi.setRunNamingRule` | `Simulink.sdi.setRunOverwrite`

Topics

“Inspect and Compare Data Programmatically”

“Organize Your Simulation Data Inspector Workspace”

Introduced in R2017a

Simulink.sdi.getMarkersOn

Package: Simulink.sdi

Return logical indication of marker property

Syntax

```
res = Simulink.sdi.getMarkersOn
```

Description

`res = Simulink.sdi.getMarkersOn` returns the logical value `res` indicating whether data markers are displayed on plots in the Simulation Data Inspector.

Examples

Save Marker State

You can use the `Simulink.sdi.getMarkersOn` property to save or query part of a Simulation Data Inspector configuration.

```
markerState = Simulink.sdi.getMarkersOn
```

Output Arguments

res — Logical indication of marker state

true | false

Logical indication of whether markers are displayed on plots in the Simulation Data Inspector.

- `true` indicates that markers are displayed.

- `false` indicates that markers are not displayed.

See Also

`Simulink.sdi.clearPreferences` | `Simulink.sdi.copyRunViewSettings` |
`Simulink.sdi.getRunNamingRule` | `Simulink.sdi.resetRunNamingRule` |
`Simulink.sdi.setMarkersOn` | `Simulink.sdi.setRunNamingRule` |
`Simulink.sdi.setSubPlotLayout` | `Simulink.sdi.setTableGrouping` |
`Simulink.sdi.view`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

Simulink.sdi.getRun

Get a `Simulink.sdi.Run` object containing simulation output data

Syntax

```
runObj = Simulink.sdi.getRun(runID)
```

Description

`runObj = Simulink.sdi.getRun(runID)` returns a handle to the `Simulink.sdi.Run` object for the run corresponding to `runID` in the Simulation Data Inspector.

Examples

Get Simulink.sdi.Run Object For Run in the Simulation Data Inspector

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput', 'on', ...
            'SaveFormat', 'StructureWithTime', ...
            'ReturnWorkspaceOutputs', 'on');

% Create a Simulation Data Inspector run
[runID,runIndex,signalIDs] = Simulink.sdi.createRun('My Run', 'base', {'simOut'});

runObj = Simulink.sdi.getRun(runID);
```

- “Inspect and Compare Data Programmatically”

Input Arguments

runID — Unique run identifier
integer

Run ID, a unique number identifying a run in the Simulation Data Inspector, specified as an integer.

Output Arguments

runObj — Run object handle

object

Object containing the signal data and metadata, returned as a `Simulink.sdi.Run` object handle.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.createRun`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.getRunCount

Get the number of runs in the Simulation Data Inspector

Syntax

```
runCount = Simulink.sdi.getRunCount
```

Description

`runCount = Simulink.sdi.getRunCount` returns the number of runs that are in the Simulation Data Inspector.

Examples

Check Run Count After Simulation

Call `Simulink.sdi.getRunCount` to get the number of runs currently in the Simulation Data Inspector after a simulation.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
            'SaveFormat','StructureWithTime');

% Create a Simulation Data Inspector run
runID = Simulink.sdi.createRun('First Run','base',{'simOut'});

% Get run count in the Simulation Data Inspector
runCount = Simulink.sdi.getRunCount()

runCount =

     1
```

- “Inspect and Compare Data Programmatically”

Output Arguments

runCount — Number of runs

integer

Number of runs that exist in the Simulation Data Inspector, returned as an integer.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.createRun`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.getRunIDByIndex

Get the run ID corresponding to the specified run index

Syntax

```
runID = Simulink.sdi.getRunIDByIndex(runIndex)
```

Description

`runID = Simulink.sdi.getRunIDByIndex(runIndex)` returns the run ID for the run corresponding to the run index.

Examples

Get Run IDs For All Runs in the Simulation Data Inspector

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
            'SaveFormat','StructureWithTime', ...
            'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run
[runID, runIndex, signalIDs] = Simulink.sdi.createRun('Run1','base',{'simOut'});

% Get the ID of the previously created run by index
runID2 = Simulink.sdi.getRunIDByIndex(runIndex);

% Both runID and runID2 reference the same run and should be equal
isequal(runID, runID2)
```

```
ans =
```

```
1
```

- “Inspect and Compare Data Programmatically”

Input Arguments

runIndex — Simulation run index

integer

Number representing an index to the list of runs currently in the Simulation Data Inspector, specified as an integer. The run index is an output of the `Simulink.sdi.createRun` function.

Output Arguments

runID — Unique run identifier

integer

Run ID, a unique number identifying a run in the Simulation Data Inspector, returned as an integer.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.getRunCount` |
`Simulink.sdi.isValidRunID`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.getRunNamingRule

Get the run naming rule for Simulation Data Inspector runs

Syntax

```
runNamingRule = Simulink.sdi.getRunNamingRule()
```

Description

`runNamingRule = Simulink.sdi.getRunNamingRule()` returns the run naming rule for Simulation Data Inspector runs.

Examples

Change and Restore the Simulation Run Naming Rule

```
% Load the model
sldemo_absbrake;

% Save the prior run naming rule and set a new rule
runNamingRule = Simulink.sdi.getRunNamingRule();
Simulink.sdi.setRunNamingRule('My special simulation');

% Record logged signals and send them to the Simulation Data Inspector
set_param('sldemo_absbrake', 'InspectSignalLogs', 1);

% Run the simulation
set_param('sldemo_absbrake', 'SimulationCommand', 'Start');

% Restore the previous run naming rule
Simulink.sdi.setRunNamingRule(runNamingRule);
```

```
% Open the Simulation Data Inspector to see the new run
Simulink.sdi.view;
```

Output Arguments

runNamingRule — Current Simulation Data Inspector run naming rule
character vector

Current Simulation Data Inspector run naming rule, returned as a character vector containing any of the run naming rule tokens <run_index>, <model_name>, <time_stamp>, <sim_mode>, or a custom run naming rule.

See Also

`Simulink.sdi.resetRunNamingRule` | `Simulink.sdi.setRunNamingRule`

Introduced in R2015a

Simulink.sdi.getSignal

Get a `Simulink.sdi.Signal` object for the specified signal in the Simulation Data Inspector

Syntax

```
signal = Simulink.sdi.getSignal(signalID)
```

Description

`signal = Simulink.sdi.getSignal(signalID)` returns the `Simulink.sdi.Signal` object for the signal corresponding to the signal ID. The `Simulink.sdi.Signal` object manages the signal's time series data and metadata.

Examples

Modify Signal Properties in Simulation Data Inspector

Get the `Simulink.sdi.Signal` object for a signal in the Simulation Data Inspector. With the signal object you can modify its comparison and visualization properties.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput', 'on', ...
            'SaveFormat', 'StructureWithTime');

% Create a Simulation Data Inspector run
[runID,runIndex,signalIDs] = Simulink.sdi.createRun('My Run', 'base', {'simOut'});

signalObj = Simulink.sdi.getSignal(signalIDs(1));

% Specify the comparison and visualization signal properties
signalObj.absTol = .5;
signalObj.syncMethod = 'intersection';
signalObj.interpMethod = 'linear';
```

```
signalObj.lineColor = [1,0.4,0.6];  
signalObj.lineDashed = '-';  
signalObj.checked = true;  
  
% View the signals in Simulation Data Inspector GUI  
Simulink.sdi.view;
```

- “Inspect and Compare Data Programmatically”

Input Arguments

signalID — Unique signal identifier

integer

Signal ID, a unique number identifying a signal in the Simulation Data Inspector, specified as an integer.

Output Arguments

signal — Signal time series object

object

A signal properties object, returned as a `Simulink.sdi.Signal` handle object.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.Signal` | `Simulink.sdi.createRun` | `Simulink.sdi.getRun`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.isPCTSupportEnabled

Determine status for Parallel Computing Toolbox support

Syntax

```
res = Simulink.sdi.isPCTSupportEnabled
```

Description

`res = Simulink.sdi.isPCTSupportEnabled` returns `res`, a logical indication of whether support for the Parallel Computing Toolbox is enabled.

Examples

Check Status of Parallel Computing Toolbox Support

Before running code that depends on whether the Parallel Computing Toolbox is supported, you can use `Simulink.sdi.isPCTSupportEnabled` to check the support status. The default behavior for the Simulation Data Inspector enables Parallel Computing Toolbox support.

```
Simulink.sdi.isPCTSupportEnabled
```

```
ans =
```

```
    logical
```

```
     1
```

Output Arguments

res — Logical indicator of Parallel Computing Toolbox support
logical

Logical indication of Parallel Computing Toolbox support.

- 1 indicates that support for the Parallel Computing Toolbox is enabled.
- 0 indicates that support for the Parallel Computing Toolbox is not enabled.

See Also

`Simulink.sdi.WorkerRun` | `Simulink.sdi.enablePCTSupport`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

Simulink.sdi.isValidRunID

Determine if run ID is valid

Syntax

```
isValid = Simulink.sdi.isValidRunID(runID)
```

Description

`isValid = Simulink.sdi.isValidRunID(runID)` returns true if the `runID` corresponds to a run currently in the Simulation Data Inspector. Otherwise, it returns false.

Examples

Verify Run IDs

Before comparing the simulation data of two runs, you can verify that the run IDs are valid.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on', ...
            'SaveFormat','StructureWithTime', ...
            'ReturnWorkspaceOutputs','on');

% Create a Data Inspector run
run1ID = Simulink.sdi.createRun('First Run','base',{'simOut'});
run2ID = Simulink.sdi.createRun('Second Run','base',{'simOut'});

% Check if run IDs are valid in Simulation Data Inspector
run1ID_valid = Simulink.sdi.isValidRunID(run1ID);
run2ID_valid = Simulink.sdi.isValidRunID(run2ID);

if run1ID_valid & run2ID_valid
```

```
% Compare two runs
difference = Simulink.sdi.compareRuns(run1ID, run2ID);
end
```

- “Inspect and Compare Data Programmatically”

Input Arguments

runID — Unique run identifier

integer

Run ID, a unique number identifying a run in the Simulation Data Inspector, specified as an integer.

Output Arguments

isValid — Valid run indicator

logical

Indication of whether the run is a valid Simulation Data Inspector run, returned as a Boolean value: 1, if the run exists; 0, otherwise.

See Also

[Simulink.sdi.Run](#) | [Simulink.sdi.compareRuns](#) | [Simulink.sdi.createRun](#)

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.load

Load a saved Simulation Data Inspector session or view

Syntax

```
isValidSDIMatFile = Simulink.sdi.load(fileName)
```

Description

`isValidSDIMatFile = Simulink.sdi.load(fileName)` loads the Simulation Data Inspector session or view from the MLDATX-file or MAT-file `fileName`.

Examples

Load Previous Simulation Data Inspector Session from a MAT-File

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample', 'SaveOutput','on', ...
            'SaveFormat', 'StructureWithTime', ...
            'ReturnWorkspaceOutputs', 'on');

% Create a run in the Simulation Data Inspector
runID = Simulink.sdi.createRun('My Run','base',{ 'simOut'});

% Save the current Simulation Data Inspector session
Simulink.sdi.save('my_runs.mat');

% Clear all data from the Simulation Data Inspector
Simulink.sdi.clear;

% Import saved MAT-file into the Simulation Data Inspector
Simulink.sdi.load('my_runs.mat');
```

```
% See the results in Simulation Data Inspector  
Simulink.sdi.view;
```

- “Inspect and Compare Data Programmatically”

Input Arguments

fileName — Session or view file name

character vector

File name, specified as a character vector, of the full or partial path of a Simulation Data Inspector session or view MLDATX-file.

Output Arguments

isValidSDIMatFile — Valid file indicator

logical

Indication of whether the MLDATX-file is a valid Simulation Data Inspector session or view file, returned as a Boolean value. Returns 1 if the file is valid and 0 if the file is not valid.

Alternatives

In the Simulation Data Inspector, click **Open** on the **Visualize** tab. For more information, see “Save and Share Simulation Data Inspector Data and Views”.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.save` | `Simulink.sdi.view`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.markSignalForStreaming

Turn streaming on or off for a signal

Syntax

```
Simulink.sdi.markSignalForStreaming(block,port_index,state)
Simulink.sdi.markSignalForStreaming(port_handle,state)
Simulink.sdi.markSignalForStreaming(line_handle,state)
```

Description

`Simulink.sdi.markSignalForStreaming(block,port_index,state)` turns streaming on or off for a signal by specifying the block, `block`, and port index, `port_index`.

`Simulink.sdi.markSignalForStreaming(port_handle,state)` turns streaming on or off for a signal by specifying the block output port, `port_handle`.

`Simulink.sdi.markSignalForStreaming(line_handle,state)` turns streaming on or off for a signal by specifying the signal line handle, `line_handle`.

Examples

Use Block Path to Mark Signal for Streaming

```
% Open the model
sldemo_absbrake;

% Turn on streaming for the Weight block signal output
% in sldemo_absbrake model
Simulink.sdi.markSignalForStreaming('sldemo_absbrake/Weight',1,'on');
```

Use Port Handle to Mark Signal for Streaming

Open a model, and select a block with a signal output to stream.

```
% Get the selected block port handles
phs = get_param(gcf, 'PortHandles')

% Turn streaming on for the block output
Simulink.sdi.markSignalForStreaming(phs.Outport(1), 'on');
```

Use Line Handle to Turn Signal Streaming Off

Open a model, and select a signal marked for streaming.

```
% Get the signal line handles for the model
slhs = get_param(gcf, 'Lines');

% Turn streaming off for the first signal in the structure
Simulink.sdi.markSignalForStreaming(slhs(1).Handle, 'off');
```

Save, Remove, and Restore Streamed Signals

You can save the set of streamed signals to a file and then restore them to your model at a later time using the `Simulink.HMI.InstrumentedSignals` class.

```
% Save signals that are currently streamed in a model
sigs = get_param(bdroot, 'InstrumentedSignals');
save my_instrumented_signals.mat sigs

% Remove all streamed signals from the model
set_param(bdroot, 'InstrumentedSignals', []);

% Restore streamed signals to the model
load my_instrumented_signals.mat
set_param(bdroot, 'InstrumentedSignals', sigs);
```

- “Iterate Model Design with the Simulation Data Inspector”
- “Inspect and Compare Data Programmatically”

Input Arguments

block — Source block path or handle

character vector | handle

Block path or handle that contains the signal source, specified as a character vector.

Example: 'sf_car/shift_logic'

port_index — Source block output port index

integer

Source block output index of the associated signal, specified as an integer.

Example: 1

state — Streaming state toggle

'on' | 'off'

Turn streaming for a signal on or off, specified as a logical value.

Example: 'on'

port_handle — Output port handle

handle

Source block output port of the associated signal, specified as a handle.

Example: pbs.Outport(1)

line_handle — Signal line handle

handle

Source block signal line, specified as a handle.

Example: sls(1).Handle

See Also

Simulink.sdi.view

Topics

“Iterate Model Design with the Simulation Data Inspector”

“Inspect and Compare Data Programmatically”

Introduced in R2015b

Simulink.sdi.report

Generate a Simulation Data Inspector report

Syntax

```
Simulink.sdi.report  
Simulink.sdi.report(Name,Value)
```

Description

`Simulink.sdi.report` creates a report of the default column set and plotted data in the **Inspect** pane in the Simulation Data Inspector.

`Simulink.sdi.report(Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments to generate a report of the specified view and plotted data in the Simulation Data Inspector.

Examples

Create Report from Inspect Pane Plots

```
% Configure model "slexAircraftExample" for logging and simulate  
simOut = sim('slexAircraftExample','SaveOutput','on', ...  
            'SaveFormat','StructureWithTime', ...  
            'ReturnWorkspaceOutputs','on');  
  
% Create a Data Inspector run  
[~,~,signalIDs] = Simulink.sdi.createRun('My Run','base',{'simOut'});  
  
% Select signals for plotting  
for i = 1:length(signalIDs)  
    signal = Simulink.sdi.getSignal(signalIDs(i));  
    signal.checked = true;  
end
```

```
% Create default report, which is the Runs pane view
Simulink.sdi.report;
```

Create Report from Compare Pane Plots

```
% Configure model "slexAircraftExample" for logging and simulate
set_param('slexAircraftExample/Pilot','WaveForm','square');
simOut = sim('slexAircraftExample','SaveOutput','on', ...
            'SaveFormat','StructureWithTime', ...
            'ReturnWorkspaceOutputs','on');

% Create a Simulation Data Inspector run, Simulink.sdi.Run, from
% simOut in the base workspace
runID1 = Simulink.sdi.createRun('First Run','namevalue',{'simOut'},{simOut});

% Simulate again
set_param('slexAircraftExample/Pilot','WaveForm','sawtooth');
simOut = sim('slexAircraftExample','SaveOutput','on', ...
            'SaveFormat','StructureWithTime', ...
            'ReturnWorkspaceOutputs','on');

% Create another Simulation Data Inspector run
runID2 = Simulink.sdi.createRun('Second Run','namevalue',{'simOut'},{simOut});

% Compare two runs
difference = Simulink.sdi.compareRuns(runID1,runID2);

% Specify columns to include in the report
metaDataOfInterest = [Simulink.sdi.SignalMetaData.Result, ...
                      Simulink.sdi.SignalMetaData.BlockPath1, ...
                      Simulink.sdi.SignalMetaData.RelTol];

% Report on the run comparison
Simulink.sdi.report('ReportToCreate','Compare Runs', ...
                  'ColumnsToReport',metaDataOfInterest, ...
                  'SignalsToReport','ReportAllSignals');
```

- “Inspect and Compare Data Programmatically”

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'ColumnsToReport', 'SignalName'`

ReportToCreate — Simulation Data Inspector pane and plots set

`'Inspect Signals'` (default) | `'Compare Runs'`

Simulation Data Inspector pane and plots to include in the report, specified as the comma-separated pair consisting of `'ReportToCreate'` and one of these values:

- `'Inspect Signals'` — Information in the **Inspect** pane.
- `'Compare Runs'` — Information in the **Compare** pane.

Example: `'ReportToCreate', 'Inspect Signals'`

ReportOutputFolder — File output location

`<current working folder>/slprj/sdi` (default) | character vector

Location of the generated report, specified as the comma-separated pair consisting of `'ReportOutputFolder'` and a character vector.

ReportOutputFile — Report file name

`'SDI_report.html'` (default) | character vector

Report file name, specified as the comma-separated pair consisting of `'ReportOutputFile'` and a character vector.

PreventOverwritingFile — Report overwrite prevention

`true` (default) | `false`

Report overwrite prevention, specified as the comma-separated pair consisting of `'PreventOverwritingFile'` and one of the Boolean `true` or `false` values. If the report file exists and the value is `true`, then the report generator increments the file name. If `false`, the report generator overwrites the report file, if it exists.

ColumnsToReport — Metadata column specifier

array

Signal and run metadata column specifier, specified as the comma-separated pair consisting of 'ColumnsToReport' and an array of column enumerations.

Array specifying values from the enumeration class, `Simulink.sdi.SignalMetaData`, which lists all signal metadata available in the Simulation Data Inspector. For example, to include the **Run** and **Synchronization Method** pane columns in a report, create an array `signal_metadata`.

```
signal_metadata = [Simulink.sdi.SignalMetaData.Run, Simulink.sdi.SignalMetaData.SyncMethod];
```

Specify the array as the value in the comma-separated pair.

```
Simulink.sdi.report('ColumnsToReport', signal_metadata);
```

Table columns available for the `Inspect Signals` report:

Column Value	Description
SignalName (default)	Signal name
Line (default)	Signal line style and color
SID	Signal ID
Units	Signal measurement units
SigDataType	Signal data type
SigSampleTime	Method used to sample the signal
Model	Name of the model that generated the signal
BlockName	Name of the source block for the signal
BlockPath	Path to the source block for the signal
Port	Index of the signal on the output port of its block
Dimensions	Dimensions of the matrix containing the signal
Channel	Index of signal within matrix
Run	Name of the simulation run containing the signal

Column Value	Description
AbsTol	User-specified, positive-valued absolute tolerance for the signal
RelTol	User-specified, positive-valued relative tolerance for the signal
OverrideGlobalTol	User-specified property that determines whether signal tolerances take priority over global tolerances
TimeTol	User-specified, positive-valued time tolerance for the signal
InterpMethod	User-specified interpolation method
SyncMethod	User-specified synchronization method used to coordinate signals for comparison
TimeSeriesRoot	Name of the variable associated with the signal for signals imported from the MATLAB workspace
TimeSource	Name of the array containing the time data for signals imported from the MATLAB workspace
DataSource	Name of the array containing the signal data for signals imported from the MATLAB workspace

Table columns available for the Compare Runs report:

Column Value	Description
Result (default)	Pass/fail result of the signal comparison between the Baseline and Compare To runs
Line1	Line style and color for the Baseline signal
Line2	Line style and color for the Compare To signal
AbsTol (default)	User-specified, positive-valued absolute tolerance for the Baseline signal
RelTol (default)	User-specified, positive-valued relative tolerance for the Baseline signal

Column Value	Description
MaxDifference	The maximum difference between the Baseline and Compare To signals
OverrideGlobalTol	User-specified property that specifies whether the Baseline signal tolerances take priority over global tolerances
TimeTol	User-specified, positive-valued time tolerance for the Baseline signal
SignalName1	Signal name from the Baseline run
SignalName2	Signal name from the Compare To run
Units1	Measurement units for the signal in the Baseline run
Units2	Measurement units for the signal in the Compare To run
SigDataType1	The data type for the signal in the Baseline run
SigDataType2	The data type for the signal in the Compare To run
SigSampleTime1	Method used to sample the signal in the Baseline run
SigSampleTime2	Method used to sample the signal in the Compare To run
Run1	Name of the Baseline run
Run2	Name of the Compare To run
AlignedBy (default)	Signal alignment method used to correlate Baseline and Compare To signals
Model1	Name of the model that generated the Baseline signals
Model2	Name of the model that generated the Compare To signals
BlockName1	Name of the source block for the Baseline signal

Column Value	Description
BlockName2	Name of the source block for the Compare To signal
BlockPath1	Path to the source block for the Baseline signal
BlockPath2	Path to the source block for the Compare To signal
Port1	Index of the Baseline signal on the output port of its block
Port2	Index of the Compare To signal on the output port of its block
Dimensions1	Dimensions of the matrix containing the Baseline signal
Dimensions2	Dimensions of the matrix containing the Compare To signal
Channel1	Index of the Baseline within its matrix
Channel2	Index of the Compare To within its matrix
InterpMethod	User-specified interpolation method for the Baseline signal
SyncMethod	User-specified synchronization method for the Baseline signal
TimeSeriesRoot1	Name of the variable associated with the Baseline signal for signals imported from the MATLAB workspace
TimeSeriesRoot2	Name of the variable associated with the Compare To signal for signals imported from the MATLAB workspace
TimeSource1	Name of the array containing the Baseline time data for signals imported from the MATLAB workspace
TimeSource2	Name of the array containing the Compare To time data for signals imported from the MATLAB workspace

Column Value	Description
DataSource1	Name of the array containing the Baseline signal data for signals imported from the MATLAB workspace
DataSource2	Name of the array containing the Compare To signal data for signals imported from the MATLAB workspace
LinkToPlot (default)	Link to a plot of each comparison result

ShortenBlockPath — Block path name length

true (default) | false

Block path name length, specified as the comma-separated pair consisting of 'ShortenBlockPath' and one of the Boolean true or false values. If the value is true and the block path name is too long, the Simulation Data Inspector shortens the name in the report. If the value is false, then the report displays the entire block path name.

LaunchReport — Launch report after creation

true (default) | false

Launch report after creation, specified as the comma-separated pair consisting of 'LaunchReport' and one of the Boolean true or false values. If the value is true after creation, the generated report opens.

SignalsToReport — Comparison signals set

'ReportOnlyMismatchedSignals' (default) | 'ReportAllSignals'

For the **Compare** pane report only. Comparison signals set, specified as the comma-separated pair consisting of 'SignalsToReport' and one of these values:

- 'ReportOnlyMismatchedSignals' — Include only the mismatched signals from the comparison between the **Baseline** and **Compare To** runs.
- 'ReportAllSignals' — Include all signals from the comparison between the **Baseline** and **Compare To** runs.

Alternatives

In the Simulation Data Inspector, click **Create Report**. For more information, see “Create a Simulation Data Inspector Report”.

See Also

`Simulink.sdi.compareRuns` | `Simulink.sdi.createRun`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.resetRunNamingRule

Reset the Simulation Data Inspector run naming rule to the default

Syntax

```
Simulink.sdi.resetRunNamingRule
```

Description

`Simulink.sdi.resetRunNamingRule` resets the run naming rule to the default rule tokens for Simulation Data Inspector runs.

Examples

Reset the Run Naming Rule to Default

```
% Load the model
sldemo_absbrake;

% Change the run naming rule
Simulink.sdi.setRunNamingRule('My special simulation');

% Record logged signals and send them to the Simulation Data Inspector
set_param('sldemo_absbrake', 'InspectSignalLogs', 1);

% Run the simulation
set_param('sldemo_absbrake', 'SimulationCommand', 'Start');

% Reset the naming rule
Simulink.sdi.resetRunNamingRule;

% Run the simulation again
set_param('sldemo_absbrake', 'SimulationCommand', 'Start');
```

```
% Open the Simulation Data Inspector  
Simulink.sdi.view;
```

The first run in the **Inspect** pane of the Simulation Data Inspector uses the run naming rule `My special simulation`, and the second run uses the default run naming rule `Run <run_index>: <model_name>`, which appears as `Run 2: sldemo_absbrake`.

See Also

`Simulink.sdi.getRunNamingRule` | `Simulink.sdi.setRunNamingRule`

Introduced in R2015a

Simulink.sdi.save

Save current Simulation Data Inspector session

Syntax

```
Simulink.sdi.save(fileName)
```

Description

`Simulink.sdi.save(fileName)` saves all runs, signals, tolerances, and signal selections to a MAT-file `fileName`.

Examples

Save Simulation Data Inspector Runs

Save the Simulation Data Inspector simulation runs and specified tolerances to a MAT-file.

```
% Configure model "sldemo_fuelsys" for logging and simulate
simOut = sim('sldemo_fuelsys', 'SaveOutput','on', ...
            'SaveFormat', 'StructureWithTime', ...
            'ReturnWorkspaceOutputs', 'on');
% Create a run in the Simulation Data Inspector
runID = Simulink.sdi.createRun('My Run','base',{ 'simOut' });

% Save the current Simulation Data Inspector session
Simulink.sdi.save('my_runs.mat');
```

You can also load the information back in to the Simulation Data Inspector using the `Simulink.sdi.load` function.

- “Inspect and Compare Data Programmatically”

Input Arguments

fileName — Session file name

character vector

A character vector specifying the target file to save.

Alternatives

In the Simulation Data Inspector, click **Save** on the **Visualize** tab. For more information, see “Save and Share Simulation Data Inspector Data and Views”.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.load`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b

Simulink.sdi.setMarkersOn

Package: Simulink.sdi

Control whether markers are shown

Syntax

```
Simulink.sdi.setMarkersOn(value)
```

Description

`Simulink.sdi.setMarkersOn(value)` sets the `show markers` parameter in the Simulation Data Inspector according to `value`.

Examples

Display Markers in the Simulation Data Inspector

```
Simulink.sdi.setMarkersOn(true);
```

Input Arguments

value — Logical input

'true' | 'false'

Logical indication of whether markers are shown on plots in the Simulation Data Inspector.

- `True` shows markers on plots in the Simulation Data Inspector.
- `False` does not show markers on plots in the Simulation Data Inspector.

Data Types: `logical`

See Also

`Simulink.sdi.clearPreferences` | `Simulink.sdi.getMarkersOn` |
`Simulink.sdi.getRunNamingRule` | `Simulink.sdi.setRunNamingRule` |
`Simulink.sdi.setSubPlotLayout` | `Simulink.sdi.setTableGrouping` |
`Simulink.sdi.view`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

Simulink.sdi.setRunNamingRule

Specify a run naming rule

Syntax

```
Simulink.sdi.setRunNamingRule(rule)
```

Description

`Simulink.sdi.setRunNamingRule(rule)` specifies a template for naming a run in the Simulation Data Inspector.

Examples

Specify Run Naming Rule

```
Simulink.sdi.setRunNamingRule(...  
    'Run <run_index> : <model_name> : <sim_mode>');
```

After recording a run, in the Signal Browser table of the Simulation Data Inspector, the run appears with a name similar to the following:

```
Run 1 : slexAircraftExample : normal
```

- “Inspect and Compare Data Programmatically”
- “Organize Your Simulation Data Inspector Workspace”

Input Arguments

rule — Run naming rule

```
<run_index> | <model_name> | <time_stamp> | <sim_mode>
```

A character vector using predefined tokens and regular characters to create a template for run names. The available tokens are

- `<run_index>` — Sequential number of each run
- `<model_name>` — Name of model
- `<time_stamp>` — Run creation time
- `<sim_mode>` — Simulation mode for recorded run

The tokens are contained within a character vector. For example:

```
Simulink.sdi.setRunNamingRule('Run <run_index> : <model_name>');
```

Alternatives

You can modify run naming rule using the Simulation Data Inspector UI. The run properties are located on the **New Run** tab of the Simulation Data Inspector preferences menu. For more information, see “Organize Your Simulation Data Inspector Workspace”.

See Also

`Simulink.sdi.clearPreferences` | `Simulink.sdi.getRunNamingRule` | `Simulink.sdi.setRunNamingRule`

Topics

“Inspect and Compare Data Programmatically”

“Organize Your Simulation Data Inspector Workspace”

Introduced in R2011b

Simulink.sdi.setRunOverwrite

Enable overwrite run mode for a Simulation Data Inspector run

Syntax

```
Simulink.sdi.setRunOverwrite(runID, overwrite)
```

Description

`Simulink.sdi.setRunOverwrite(runID, overwrite)` enables overwrite run mode for the run identified by `runID` in the Simulation Data Inspector.

Examples

Mark Run for Overwriting

This example shows how to enable overwrite run mode for a run in the Simulation Data Inspector.

Open the `sldemo_fuelsys` model.

On the Simulink Editor toolbar, click the **Simulation Data Inspector** button arrow and select **Send Logged Workspace Data to Data Inspector**.

Simulate the model.

Open the Simulation Data Inspector.

At the MATLAB Command Window, create a `runID` variable that uses the value of the **Run ID** for a run. You can find the **Run ID** for a run using the function `Simulink.sdi.getRunIDByIndex`.

```
runID = 1;
```

Set the overwrite condition to `true`.

```
Simulink.sdi.setRunOverwrite(runID,true);
```

In the Simulation Data Inspector, you can see the run is now marked to overwrite during the next simulation.

Simulate `sldemo_fuelsys`.

In the Simulation Data Inspector, the new simulation data replaces the previous run.

- “Inspect and Compare Data Programmatically”
- “Iterate Model Design with the Simulation Data Inspector”
- “Organize Your Simulation Data Inspector Workspace”

Input Arguments

runID — Unique run identifier

integer

Run ID, a unique number identifying a run in the Simulation Data Inspector, specified as an integer.

overwrite — Overwrite run setting

`true` | `false`

Overwrite run setting, specified as `true` or `false`. When set to `true`, the subsequent simulations overwrite the run identified by the `runID`.

Example: `Simulink.sdi.setRunOverwrite(1, true);`

Alternatives

You can set and disable run overwrite mode from the right-click context menu for the run in the Simulation Data Inspector UI. For more information see “Iterate Model Design with the Simulation Data Inspector”.

See Also

`Simulink.sdi.clearPreferences` | `Simulink.sdi.getRunNamingRule` |
`Simulink.sdi.resetRunNamingRule` | `Simulink.sdi.setRunNamingRule`

Topics

“Inspect and Compare Data Programmatically”

“Iterate Model Design with the Simulation Data Inspector”

“Organize Your Simulation Data Inspector Workspace”

Introduced in R2011b

Simulink.sdi.setSubPlotLayout

Package: Simulink.sdi

Set subplot layout in the Simulation Data Inspector

Syntax

```
Simulink.sdi.setSubPlotLayout(r,c)
```

Description

`Simulink.sdi.setSubPlotLayout(r,c)` sets the grid layout of plots in the Simulation Data Inspector using the specified number of rows, `r`, and columns, `c`.

Examples

Change Subplot Layout

```
% Change subplot layout to 4 rows and 2 columns  
Simulink.sdi.setSubPlotLayout(4,2);
```

Create a Run and View the Data

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.

Create Data for the Run

This example creates `timeseries` objects for a sine and a cosine. To visualize your data, the Simulation Data Inspector requires at least a time vector that corresponds with your data.

```
% Generate timeseries data
time = linspace(0, 20, 100);

sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals, time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals, time);
cos_ts.Name = 'Cosine, T=8';
```

Create a Simulation Data Inspector Run and Add Your Data

To give the Simulation Data Inspector access to your data, use the `create` method and create a run. This example modifies some of the run's properties to help identify the data. You can easily view run and signal properties with the Simulation Data Inspector.

```
% Create a run
run = Simulink.sdi.Run.create;
run.Name = 'Sinusoids';
run.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
run.add('vars', sine_ts, cos_ts);
```

Plot Your Data Using the `Simulink.sdi.Signal` Object

The `getSignalByIndex` method returns a `Simulink.sdi.Signal` object that can be used to plot the signal in the Simulation Data Inspector. You can also programmatically control aspects of the plot's appearance, such as the color and style of the line representing the signal. This example customizes the subplot layout and signal characteristics.

```
% Get signal, modify its properties, and change Checked property to true
sine_sig = run.getSignalByIndex(1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';
sine_sig.Checked = true;

% Add another subplot for the cosine signal
Simulink.sdi.setSubPlotLayout(2, 1);

% Plot the cosine signal and customize its appearance
cos_sig = run.getSignalByIndex(2);
```

```
cos_sig.LineColor = [0 1 0];  
cos_sig.plotOnSubPlot(2, 1, true);  
  
% View the signal in the Simulation Data Inspector  
Simulink.sdi.view
```

Close the Simulation Data Inspector and Save Your Data

```
Simulink.sdi.close('sinusoids.mat')
```

- “Inspect and Compare Data Programmatically”
- “Create Plots Using the Simulation Data Inspector”

Input Arguments

r — Number of rows

integer

Number of rows in the subplot grid layout, specified as a whole number between 1 and 8, inclusive.

Example: 2

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

c — Number of columns

integer

Number of columns in the subplot grid layout, specified as a whole number between 1 and 8, inclusive.

Example: 2

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

See Also

Simulink.sdi.Signal | Simulink.sdi.Signal.plotOnSubPlot |
Simulink.sdi.clearPreferences | Simulink.sdi.setMarkersOn |
Simulink.sdi.view

Topics

“Inspect and Compare Data Programmatically”

“Create Plots Using the Simulation Data Inspector”

Introduced in R2016a

Simulink.sdi.setTableGrouping

Package: Simulink.sdi

Change signal grouping hierarchy in Inspect pane

Syntax

```
Simulink.sdi.setTableGrouping
Simulink.sdi.setTableGrouping('group1')
Simulink.sdi.setTableGrouping('group1', 'group2')
Simulink.sdi.setTableGrouping('group1', 'group2', 'group3')
```

Description

`Simulink.sdi.setTableGrouping` groups signals in the **Inspect** pane of the Simulation Data Inspector as a flat list.

`Simulink.sdi.setTableGrouping('group1')` groups signals in the **Inspect** pane by the parameter specified by `group1`.

`Simulink.sdi.setTableGrouping('group1', 'group2')` groups signals in the **Inspect** pane first by the parameter specified by `group1` and then by the parameter specified by `group2`.

`Simulink.sdi.setTableGrouping('group1', 'group2', 'group3')` groups signals in the **Inspect** pane first by the parameter specified by `group1`, then by the parameter specified by `group2`, and finally by the parameter specified by `group3`. If you have a Simscape license, you have three options for table grouping.

Examples

Group Data by Model and Then Data Hierarchy

You can group your data in the navigation pane to visualize subsystem and data hierarchy relationships among signals clearly.

```
Simulink.sdi.setTableGrouping('Subsystems','DataHierarchy');
```

- “Inspect and Compare Data Programmatically”
- “Organize Your Simulation Data Inspector Workspace”

Input Arguments

'group1' — Signal grouping parameter

character vector

Parameter used to group signals in the **Inspect** pane of the Simulation Data Inspector. You can specify two grouping parameters, or three if you have a Simscape license.

- `DataHierarchy` groups signals according to any data hierarchy in the model, for example grouping signals in a bus together.
- `SubSystems` groups signals according to the model's subsystem hierarchy.
- `PhysmodHierarchy` groups signals according to the Simscape block structure. This parameter is only available with a Simscape license.

Example: 'SubSystems'

Example: 'DataHierarchy'

Data Types: char

See Also

`Simulink.sdi.clearPreferences` | `Simulink.sdi.view`

Topics

“Inspect and Compare Data Programmatically”

“Organize Your Simulation Data Inspector Workspace”

Introduced in R2016a

Simulink.sdi.view

Package: Simulink.sdi

Open the Simulation Data Inspector

Syntax

```
Simulink.sdi.view
```

Description

`Simulink.sdi.view` opens the Simulation Data Inspector. You can write a script to plot your data and customize the Simulation Data Inspector properties and then use `Simulink.sdi.view` to see the results.

Examples

Open the Simulation Data Inspector from the Command Line

You can open the Simulation Data Inspector from the MATLAB command line to visualize, inspect, and analyze your data.

```
Simulink.sdi.view
```

Create a Run and View the Data

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.

Create Data for the Run

This example creates `timeseries` objects for a sine and a cosine. To visualize your data, the Simulation Data Inspector requires at least a time vector that corresponds with your data.

```
% Generate timeseries data
time = linspace(0, 20, 100);

sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals, time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals, time);
cos_ts.Name = 'Cosine, T=8';
```

Create a Simulation Data Inspector Run and Add Your Data

To give the Simulation Data Inspector access to your data, use the `create` method and create a run. This example modifies some of the run's properties to help identify the data. You can easily view run and signal properties with the Simulation Data Inspector.

```
% Create a run
run = Simulink.sdi.Run.create;
run.Name = 'Sinusoids';
run.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
run.add('vars', sine_ts, cos_ts);
```

Plot Your Data Using the `Simulink.sdi.Signal` Object

The `getSignalByIndex` method returns a `Simulink.sdi.Signal` object that can be used to plot the signal in the Simulation Data Inspector. You can also programmatically control aspects of the plot's appearance, such as the color and style of the line representing the signal. This example customizes the subplot layout and signal characteristics.

```
% Get signal, modify its properties, and change Checked property to true
sine_sig = run.getSignalByIndex(1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';
sine_sig.Checked = true;
```

```
% Add another subplot for the cosine signal
Simulink.sdi.setSubPlotLayout(2, 1);

% Plot the cosine signal and customize its appearance
cos_sig = run.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.plotOnSubPlot(2, 1, true);

% View the signal in the Simulation Data Inspector
Simulink.sdi.view
```

Close the Simulation Data Inspector and Save Your Data

```
Simulink.sdi.close('sinusoids.mat')
```

- “Inspect and Compare Data Programmatically”
- “Populate the Simulation Data Inspector with Your Data”

Alternatives

You can open the Simulation Data Inspector from the Simulink Editor toolbar with the

Simulation Data Inspector button .

See Also

`Simulink.sdi.clear` | `Simulink.sdi.clearPreferences` | `Simulink.sdi.close`
| `Simulink.sdi.setSubPlotLayout`

Topics

- “Inspect and Compare Data Programmatically”
- “Populate the Simulation Data Inspector with Your Data”

Introduced in R2011b

Simulink.SimulationData.createStructOfTimeseries

Create a structure with MATLAB timeseries object leaf nodes

Syntax

```
struct_of_ts = Simulink.SimulationData.createStructOfTimeseries (
tsArrayObject)
```

```
struct_of_ts = Simulink.SimulationData.createStructOfTimeseries (
busObj, structOfTimeseries)
```

```
struct_of_ts = Simulink.SimulationData.createStructOfTimeseries (
busObj, cellofTimeseries)
```

```
struct_of_ts = Simulink.SimulationData.createStructOfTimeseries (
busObj, cellofTimeseries, dims)
```

Description

`struct_of_ts = Simulink.SimulationData.createStructOfTimeseries (tsArrayObject)` creates a structure of MATLAB timeseries objects from a Simulink.TsArray object. Use this syntax for signal logging data for a model simulated in a release earlier than R2016a that used ModelDataLogs signal logging format.

`struct_of_ts = Simulink.SimulationData.createStructOfTimeseries (busObj, structOfTimeseries)` creates a structure that matches the attributes of the bus object busObj and sets the values of structure leaf nodes using a structure of MATLAB timeseries objects structOfTimeseries. Use this syntax when using a partial structure as the basis for creating a full structure to load into a model.

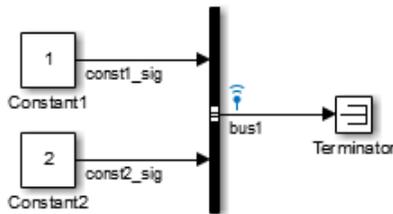
`struct_of_ts = Simulink.SimulationData.createStructOfTimeseries (busObj, cellofTimeseries)` creates a structure that matches the attributes of the bus object busObj and sets the values of structure leaf nodes using a cell array of MATLAB timeseries objects cellofTimeseries.

`struct_of_ts = Simulink.SimulationData.createStructOfTimeseries(busObj, cellOfTimeseries, dims)` creates a structure with the dimensions `dims`. Use this syntax to create a structure to load into an array of buses.

Examples

Structure Based on `Simulink.TsArray`

Suppose you had signal logging data from simulating a model in a release earlier than R2016a, using the `ModelDataLogs` format. The logged output is `logstdout`.



View the logged data.

```
logstdout
```

```
logstdout =
```

```
Simulink.ModelDataLogs (log_modeldatalogs):
  Name           Elements  Simulink Class
  bus1           2        TsArray
```

Convert the logged data to a structure of MATLAB timeseries objects.

```
struct_of_ts = ...
Simulink.SimulationData.createStructOfTimeseries(logstdout.bus1)

struct_of_ts =
```

```

const1_sig: [1x1 timeseries]
const2_sig: [1x1 timeseries]

```

Structure Based on Bus Object and a Partial Structure of Timeseries Data

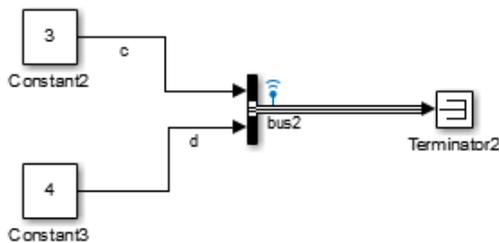
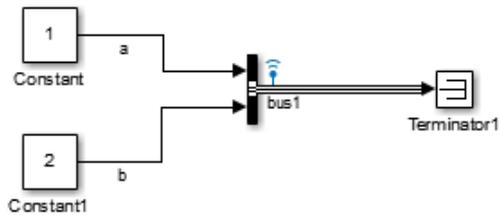
Create a structure of MATLAB `timeseries` objects based on a `Simulink.Bus` object and a partial structure of MATLAB `timeseries` objects. Use this structure to load into another model. Open a model and simulate it, producing signal logging data.

Open a model and simulate it, producing signal logging data.

```

open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_log_structTimeSeries')))
sim('ex_log_structTimeSeries')

```



View the logged signal data.

```
ex_log_structTimeSeries_logargout
```

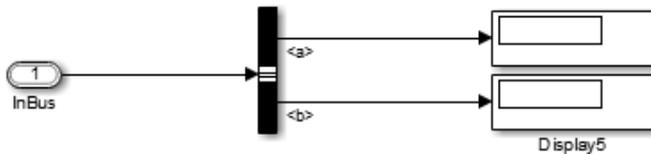
```
Simulink.SimulationData.Dataset 'ex_log_structTimeSeries_logargout' with 2 elements
```

	Name	BlockPath
1	[1x1 Signal]	bus1 ex_log_structTimeSeries/Bus Creator
2	[1x1 Signal]	bus2 ex_log_structTimeSeries/Bus Creator1

- Use braces { } to access, modify, or add elements using index.

Open the model to load the logged signal data into.

```
open_system(docpath(fullfile(docroot,'toolbox','simulink',...
'examples','ex_load_structTimeSeries_Bus')))
```



The `ex_load_structTimeSeries_Bus` model **Configuration Parameters > Data Import/Export > Input** parameter lists the `ex_load_structTimeSeries_inputBus` variable. However, you have not yet defined that variable in the MATLAB workspace. Use `Simulink.SimulationData.createStructOfTimeseries` to define that variable.

```
ex_load_structTimeSeries_inputBus = ...
Simulink.SimulationData.createStructOfTimeseries...
('bus', ex_log_structTimeSeries_logout.get(2).Values)

ex_load_structTimeSeries_inputBus =

    a: [1x1 timeseries]
    b: [1x1 timeseries]
```

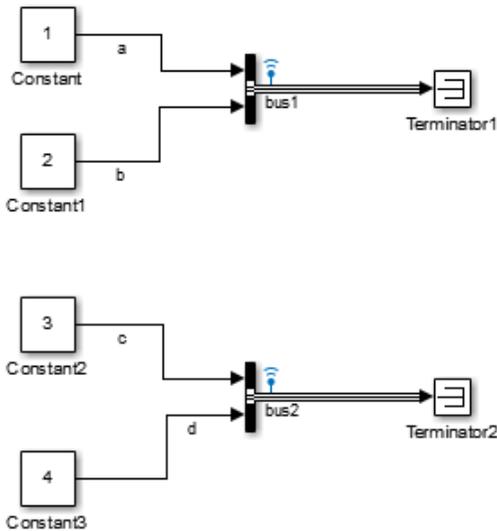
Structure to Use with an Array of Buses

Create a structure of MATLAB `timeseries` objects to load into an array of buses. Specify the dimensions of the created structure and a cell array of MATLAB `timeseries` objects.

Open a model and simulate it, producing signal logging data.

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
    'examples', 'ex_log_structTimeSeries')))
sim('ex_log_structTimeSeries')
```

The simulated `ex_log_structTimeSeries` model looks like this:



View the logged signal data.

```
ex_log_structTimeSeries_logargout
```

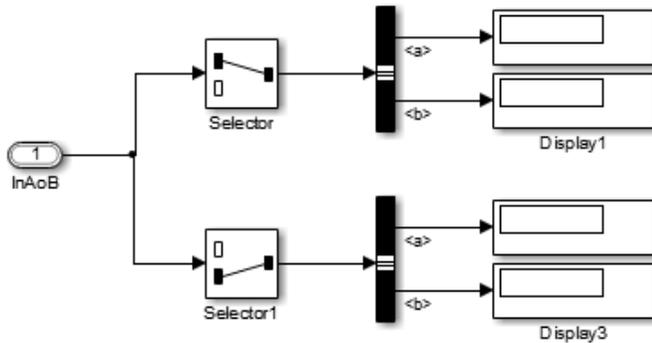
```
Simulink.SimulationData.Dataset 'ex_log_structTimeSeries_logargout' with 2 elements
```

	Name	BlockPath
1	[1x1 Signal]	bus1
2	[1x1 Signal]	bus2

- Use braces { } to access, modify, or add elements using index.

Open the model to load the logged signal data into.

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
    'examples', 'ex_load_structTimeSeries_AoB')))
```



The `ex_load_structTimeSeries_AoB` model's **Configuration Parameters > Data Import/Export > Input** parameter lists the `ex_load_structTimeSeries_inputAoB` variable. However, you have not yet defined that variable in the MATLAB workspace. Use `Simulink.SimulationData.createStructOfTimeseries` to define that variable.

```
ex_load_structTimeSeries_inputAoB = ...
Simulink.SimulationData.createStructOfTimeseries...
('bus',{ex_log_structTimeSeries_logargout.get(1).Values.a,...
ex_log_structTimeSeries_logargout.get(1).Values.b,...
ex_log_structTimeSeries_logargout.get(2).Values.c,...
ex_log_structTimeSeries_logargout.get(2).Values.d},[2, 1])

ex_load_structTimeSeries_inputAoB =

2x1 struct array with fields:
    a
    b
```

Input Arguments

tsArrayObject — `Simulink.TsArray` object to convert

`Simulink.TsArray` object

`Simulink.TsArray` object to convert to a structure of MATLAB timeseries objects

In releases earlier than R2016a, when you log signals using the `ModelDataLogs` format, the logged data is a collection of `Simulink.TsArray` objects.

busObj — Bus object for creating a structure of MATLAB timeseries objects`Simulink.Bus` object

Bus object for creating a structure of MATLAB timeseries objects, specified as the name of a `Simulink.Bus` object.

Data Types: `char`

structOfTimeseries — Structure object for values to override ground values, specified as a structure of MATLAB timeseries objects.`structure of MATLAB timeseries objects`

Structure object for values to override ground values, specified as a structure of MATLAB timeseries objects. The structure must have the same hierarchy as the bus object. However, the names of the fields in the structure do not have to match the names of the corresponding bus object nodes.

Data Types: `struct`

cellOfTimeseries — Cell array objects for values to override ground values, specified as a cell array of MATLAB timeseries objects.`cell array of MATLAB timeseries objects`

Cell array object for values to override ground values, specified as a cell array of MATLAB timeseries objects. If you specify a cell array of MATLAB timeseries objects and you specify a `dims` argument, then the length of the cell array must be equal to the result of `Simulink.BusObject.getNumLeafBusElements` times the product of the specified dimensions.

Data Types: `cell`

dims — Dimensions of the structure that this function creates.`vector`

Dimensions of the structure that this function creates, specified as a vector. The length of the cell array is equal to the result of `Simulink.BusObject.getNumLeafBusElements` times the product of the specified dimensions.

If you specify a dimension in the form `[n]`, then Simulink interprets the dimension to be `1xn`.

Data Types: `double`

Output Arguments

struct_of_ts — Structure of MATLAB timeseries objects.

MATLAB structure

MATLAB timeseries objects, returned as a structure. The structure has the same hierarchy and attributes as the `Simulink.TsArray` object or `Simulink.Bus` object that you specify.

The dimensions of `structOfTimeseries` depend on the input arguments:

- If you specify `tsArrayObject`, then the dimension is 1.
- If you specify the `busObj` and a structure of MATLAB timeseries, then the dimension matches the dimensions of the specified structure.
- If you specify only the `busObj` and a cell array of MATLAB timeseries, then the dimension is 1.
- If you specify the `busObj` argument, a cell array of MATLAB timeseries, and the `dims` argument, then the dimensions match the dimensions of `dims`.

Related Links

[Simulink.Bus](#) | [Simulink.TsArray](#) | [Simulink.ModelDataLogs](#) | [Simulink.ModelDataLogs.convertToDataset](#)

Introduced in R2013a

getAsDatastore

Class: Simulink.SimulationData.DatasetRef

Package: Simulink.SimulationData

Get matlab.io.datastore.SimulationDatastore representation of element from referenced Dataset object

Syntax

```
element =  
Simulink.SimulationData.DatasetRef.getAsDatastore(datasetref_elements)
```

Description

```
element =  
Simulink.SimulationData.DatasetRef.getAsDatastore(datasetref_elements)
```

returns a matlab.io.datastore.SimulationDatastore representation of an element or collection of elements from the referenced dataset, based on index, name, or block path of the element.

You can represent a Dataset element as a matlab.io.datastore.SimulationDatastore object if the element was placed into the MAT-file using either of these approaches:

- Log Dataset format data to persistent storage (MAT-file).
- Place the element into a Simulink.SimulationData.Dataset object and saved the Dataset object to a v7.3 MAT-file.

The SimulationDatastore representation for a Dataset element creates a SimulationDatastore object for the Values field of that element. The SimulationDatastore representation supports streaming of the data for the Values property of the element into other simulations or into MATLAB.

Note You cannot use create a `SimulationDatastore` for Dataset elements that contain these types of data:

- Array
-

You can use `SimulationDatastore` objects to:

- Refer to logged simulation data that is stored on disk in a MAT-file.
- Specify signals to stream incrementally from disk to a simulation.
- Provide a basis for big data analysis using MATLAB functions.

Input Arguments

datasetref_element — Element of referenced dataset in MAT-file
index of the element

Element of a referenced dataset in a MAT-file, specified as an index, name (as a character vector), or block path (as a character vector).

Output Arguments

element — Element accessed using `SimulationDatastore` object

`matlab.io.datastore.SimulationDatastore` object |
`Simulink.SimulationData.Signal`, `Simulink.SimulationData.State` or similar object, whose `Values` data uses a `matlab.io.datastore.SimulationDatastore` object

Element accessed using `SimulationDatastore` object, returned as either a `matlab.io.datastore.SimulationDatastore` object or a `Simulink.Signal`, `Simulink.State`, or similar object, whose `Values` data uses a `matlab.io.datastore.SimulationDatastore` object.

Examples

Use a SimulationDatastore to Reference a Signal's Data in a DatasetRef

Log signal data to persistent storage (select the **Log Dataset data to file** configuration parameter) and simulate a model.

Create a DatasetRef for the signal logging Dataset data (logout) in the out.mat MAT-file.

```
sigLogRef = Simulink.SimulationData.DatasetRef('out.mat', 'logout');
firstSig = sigLogRef.getAsDatastore(1)
```

```
firstSig =
```

```
Simulink.SimulationData.Signal
Package: Simulink.SimulationData
```

```
Properties:
```

```
    Name: 'x1'
  PropagatedName: ''
    BlockPath: [1x1 Simulink.SimulationData.BlockPath]
    PortType: 'outport'
    PortIndex: 1
    Values: [1x1 matlab.io.datastore.SimulationDatastore]
```

Load the data into another model. This approach streams the values of the signal x1 for another simulation.

```
ds = Simulink.SimulationData.Dataset;
ds{1} = sigLogRef{1};
sim('other_model', 'ExternalInput', 'ds');
```

- “Load Big Data for Simulations”

Alternative

To streamline the use of indexing, you can use curly braces ({}) syntax to obtain a SimulationDatastore object for DatasetRef object signal values. The requirements and results are the same as using getAsDatastore. For example, if you log signal data to persistent storage (select the **Log Dataset data to file** configuration parameter) and simulate a model.

```
sigLogRef = Simulink.SimulationData.DatasetRef('out.mat', 'logout');
firstSig = sigLogRef{1}
```

```
ans =  
  
Simulink.SimulationData.Signal  
Package: Simulink.SimulationData  
  
Properties:  
    Name: 'x1'  
    PropagatedName: ''  
    BlockPath: [1x1 Simulink.SimulationData.BlockPath]  
    PortType: 'outport'  
    PortIndex: 1  
    Values: [1x1 matlab.io.datastore.SimulationDatastore]
```

See Also

[Simulink.SimulationData.Dataset](#) | [Simulink.SimulationData.DatasetRef](#) | [matlab.io.datastore.SimulationDatastore](#)

Topics

“Load Big Data for Simulations”

Introduced in R2017a

Simulink.SimulationData.DatasetRef.getDatasetVariableNames

List names of Dataset variables in MAT-file

Syntax

```
varNames =  
Simulink.SimulationData.DatasetRef.getDatasetVariableNames(matFile)
```

Description

```
varNames =  
Simulink.SimulationData.DatasetRef.getDatasetVariableNames(matFile)
```

lists the names of variables for Dataset data in a MAT-file.

Examples

List Variable Names in MAT-File

Suppose that you simulate a model using the default variable names for signal logging data and states data. You enable the **Configuration Parameters > Data Import/Export > Log Dataset data to file** and use the default MAT-file name of `out.mat`.

List the variable names in the MAT-file.

```
varNames = Simulink.SimulationData.DatasetRef.getDatasetVariableNames('out.mat')
```

```
varNames =  
  
    'xout'    'logout'
```

- “Log Data to Persistent Storage”

- “Load Big Data for Simulations”

Tips

To get the names of `Dataset` variables in the MAT-file, using the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function processes faster than using the `who`, or `whos` functions.

Input Arguments

matFile — MAT-file that contains `Dataset` variables
character vector

MAT-file that contains `Dataset` variables, specified as a character vector. The character vector specifies the path to the MAT-file.

Output Arguments

varNames — Names of `Dataset` variables in MAT-file
cell array

Names of `Dataset` variables in MAT-file, returned as a cell array.

See Also

`Simulink.SimulationData.Dataset` | `Simulink.SimulationData.DatasetRef`

Topics

“Log Data to Persistent Storage”
“Load Big Data for Simulations”

Introduced in R2016a

Simulink.SimulationData.forEachTimeseries

Call function on each `timeseries` object

Syntax

```
dataResults = Simulink.SimulationData.forEachTimeseries(  
functionHandle,inputData)
```

Description

`dataResults = Simulink.SimulationData.forEachTimeseries(functionHandle,inputData)` runs the specified function handle on all MATLAB `timeseries` objects contained in `inputData`.

Examples

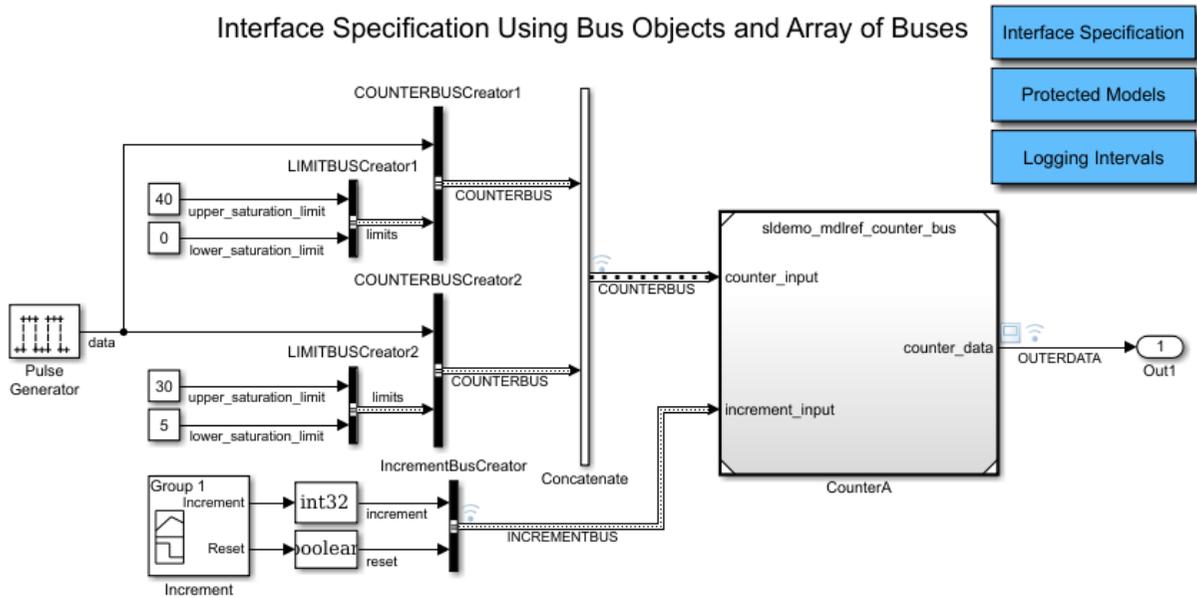
Find Minimum for Logged Bus Signal Data

This example shows how to use the `forEachTimeseries` function to run the `min` function on each `timeseries` object in the logged data for the `COUNTERBUS` signal.

Open the model and simulate it.

```
open_system('sldemo_mdref_bus')  
sim('sldemo_mdref_bus');
```

Interface Specification Using Bus Objects and Array of Buses



Access the signal logging data. For this model, that data is stored in the topOut variable.

topOut

Simulink.SimulationData.Dataset 'topOut' with 4 elements

	Name	BlockPath
1 [1x1 Signal]	COUNTERBUS	sldemo_mdref_bus/Concatenate
2 [1x1 Signal]	OUTERDATA	sldemo_mdref_bus/CounterA
3 [1x1 Signal]	INCREMENTBUS	sldemo_mdref_bus/IncrementBusCreator
4 [1x1 Signal]	INNERDATA	...erA\sldemo_mdref_counter_bus/COUNTER

- Use braces { } to access, modify, or add elements using index.

Find the values for the COUNTERBUS element.

```
counterbusData = topOut{1}.Values
```

```
counterbusData =
```

```
2x1 struct array with fields:
```

```
    data
    limits
```

Run the `min` function on the counterbus data.

```
ret = Simulink.SimulationData.forEachTimeseries(@min,counterbusData)
```

```
ret =
```

```
2x1 struct array with fields:
```

```
    data
    limits
```

Explore the returned data.

```
ret(1)
```

```
ans =
```

```
    data: 0
    limits: [1x1 struct]
```

```
ret(2).limits
```

```
ans =
```

```
    upper_saturation_limit: 40
    lower_saturation_limit: 0
```

Input Arguments

functionHandle — Function to run on MATLAB timeseries objects

function handle

Function to run on `timeseries` objects, specified as a function handle. For information about specifying function handles, see “Pass Function to Another Function” (MATLAB).

The function that you use with `forEachTimeseries`:

- Can be either a built-in function or a user-specified function

- Must return a scalar

If the function that you use with `forEachTimeseries` takes:

- One argument, specify the function handle and the input data. For example:

```
ret = Simulink.SimulationData.forEachTimeseries(@min,data);
```

- More than one argument, specify the function handle as `@(x)` and then specify the function, using `x` as the first argument. For remaining arguments, specify values. For example, this command runs the `resample` function on MATLAB timeseries objects in `data`, for the time vector `[2.5 3]`.

```
ret = Simulink.SimulationData.forEachTimeseries(@(x) ...  
    (resample(x,[2.5 3]),data);
```

inputData — Data to run specified function on

MATLAB `timeseries` object | array of `timeseries` | structure with `timeseries` at leaf nodes | array of structures with `timeseries` at leaf nodes

Data to run specified function on, specified as `timeseries` data.

Output Arguments

dataResults — Data resulting from running specified function

MATLAB `timeseries` object | array of `timeseries` | structure with `timeseries` at leaf nodes | array of structures with `timeseries` at leaf nodes

Data resulting from running specified function, returned using the format and hierarchy of the input data.

Related Links

MATLAB `timeseries` “Function Handles”

Introduced in R2016b

Simulink.SimulationData.signalLoggingSelector

Open Signal Logging Selector

Syntax

```
Simulink.SimulationData.signalLoggingSelector(modelName)
```

Description

`Simulink.SimulationData.signalLoggingSelector(modelName)` opens the Signal Logging Selector dialog box for the model that you specify with `modelName`.

Input Arguments

modelName

Character vector that specifies the name of the model for which you want to open the Signal Logging Selector dialog box.

Example

Open the Signal Logging Selector dialog box for the `sldemo_mdrefref_bus.mdl`.

```
Simulink.SimulationData.signalLoggingSelector('sldemo_mdrefref_bus')
```

See Also

`Simulink.ModelDataLogs` | `Simulink.SimulationData.Dataset`

Topics

“Override Signal Logging Settings”

Introduced in R2011a

setName

Class: Simulink.SimulationData.Unit

Package: Simulink.SimulationData

Specify name of logging data units

Syntax

```
unitObject = setName(unitObj,unitName)
```

Description

`unitObject = setName(unitObj,unitName)` sets the name for the `Simulink.SimulationData.Unit` object to the name specified in `unitName`.

Input Arguments

unitObj — Logging data unit object to name

`Simulink.SimulationData.Unit` object

Logging data unit object to name, specified as a `Simulink.SimulationData.Unit` object.

unitName — Name of logging data unit

character vector

Name of logging data unit, specified as a character vector.

Output Arguments

name — Name of logging data units

character vector

Name of logging data units, returned as a character vector.

Examples

Name a Logging Data Unit Object

```
inchesUnit = Simulink.SimulationData.Unit('in');  
inchesUnit = setName(inchesUnit, 'inches')
```

```
inchesUnit =
```

```
Units with properties:
```

```
Name: 'inches'
```

- “Log Signal Data That Uses Units”
- “Convert Logged Data to Dataset Format”
- “Prepare Model Inputs and Outputs”

See Also

`Simulink.SimulationData.Unit`

Topics

“Log Signal Data That Uses Units”
“Convert Logged Data to Dataset Format”
“Prepare Model Inputs and Outputs”

Introduced in R2011a

Simulink.SimulationData.updateDatasetFormatLogging

Convert model and its referenced models to use `Dataset` format for signal logging

Syntax

```
Simulink.SimulationData.updateDatasetFormatLogging(top_model)
Simulink.SimulationData.updateDatasetFormatLogging(top_model,
variants)
```

Description

Note The `ModelDataLogs` class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format. You do not need to use this command to update the signal logging format for a model that uses model referencing. Opening the model in R2016a or later uses `Dataset` format for all signal logging.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy `ModelDataLogs` API”.

`Simulink.SimulationData.updateDatasetFormatLogging(top_model)` converts the top-level model and all of its referenced models to use the `Dataset` format for signal logging instead of the `ModelDataLogs` format. You can convert signal logging data from

ModelDataLogs to Dataset format. Converting to Dataset format makes it easier to post-process with other logged data (for example, logged states), which can also use Dataset format. For more information, see “Convert Logged Data to Dataset Format”.

If a Model block has the **Generate preprocessor conditionals** option selected, the function converts all the variants; otherwise, the function converts only the active variant.

`Simulink.SimulationData.updateDatasetFormatLogging(top_model, variants)` specifies which variant models to convert to use the Dataset signal logging format. For details about the `variants` argument, see “Input Arguments” on page 2-814

Input Arguments

`top_model`

Character vector that specifies the name of the top-level model.

`variants`

Character vector that specifies which variant models to update:

- 'ActivePlusCodeVariants' — (Default) Search all variants if any generate preprocessor conditionals. Otherwise, search only the active variant.
- 'ActiveVariants' — Convert only the active variant.
- 'AllVariants' — Convert all variants.

Definitions

Dataset

The *Dataset* format causes Simulink to use a `Simulink.SimulationData.Dataset` object to store the logged signal data. The Dataset format use MATLAB timeseries objects to formatting the data.

ModelDataLogs

The *ModelDataLogs* format causes Simulink to use a `Simulink.ModelDataLogs` object to store the logged signal data. `Simulink.Timeseries` and `Simulink.TsArray` objects provide the format for the data.

Tips

- The conversion function sets the `SignalLoggingSaveFormat` parameter value to `Dataset` for all the updated models.
- If you want to save the format updates that the conversion function makes, then ensure that the top-level model, referenced models, and variant models are accessible and writable.
- If a model has no other unsaved changes, the conversion function saves the format updates to the model. If the model has unsaved changes, the function updates the format, but does not save those changes.
- If you use this function for a model that does not include any referenced models, the function converts the top-level model use the `Dataset` format.

See Also

`Simulink.ModelDataLogs` | `Simulink.ModelDataLogs.convertToDataset` | `Simulink.SimulationData.Dataset`

Topics

“Migrate Scripts That Use Legacy ModelDataLogs API”

Introduced in R2011a

find

Class: Simulink.SimulationOutput

Package: Simulink

Access and display values of simulation results

Syntax

```
output = simOut.find('VarName')
```

Description

`output = simOut.find('VarName')` accepts one variable name. Specify *VarName* inside single quotes.

Input Arguments

VarName

Name of logged variable for which you seek values.

Default:

Output Arguments

Value

Value of the logged variable name specified in input.

Examples

Simulate `vdp` and store the values of the variable `youtNew` in `yout`.

```
simOut = sim('vdp','SimulationMode','rapid','AbsTol','1e-5',...
            'SaveState','on','StateSaveName','xoutNew',...
            'SaveOutput','on','OutputSaveName','youtNew');
yout = simOut.find('youtNew')
```

Alternatives

A simpler alternative to use dot notation. For example, to access data for the `xoutNew` output variable, you can use this command:

```
simOut.xoutNew
```

Another alternative is to use `Simulink.SimulationOutput.who` and then `Simulink.SimulationOutput.get`.

See Also

`Simulink.SimulationOutput.get` | `Simulink.SimulationOutput.who`

get

Class: Simulink.SimulationOutput

Package: Simulink

Access and display values of simulation results

Syntax

```
output = simOut.get('VarName')
```

Description

`output = simOut.get('VarName')` accepts one variable name. Specify *VarName* inside single quotes.

Tip A simpler alternative to using the `get` function is to use dot notation. For example, to access data for the `xout` output variable, you can use this command:

```
simOut.xout
```

Input Arguments

VarName

Name of logged variable for which you seek values.

Default:

Output Arguments

Value

Value of the logged variable name specified in input.

Examples

Simulate `vdp` and store the values of the variable `youtNew` in `yout`.

```
simOut = sim('vdp','SimulationMode','rapid','AbsTol','1e-5',...
            'SaveState','on','StateSaveName','xoutNew',...
            'SaveOutput','on','OutputSaveName','youtNew');
yout = simOut.get('youtNew')
```

Alternatives

A simpler alternative to use dot notation. For example, to access data for the `xout` output variable, you can use this command:

```
simOut.xout
```

Another alternative is to use `Simulink.SimulationOutput.who` and then `Simulink.SimulationOutput.find`.

See Also

`Simulink.SimulationOutput.find` | `Simulink.SimulationOutput.who`

getSimulationMetadata

Class: Simulink.SimulationOutput

Package: Simulink

Return `SimulationMetadata` object for simulation

Syntax

```
mData = simout.getSimulationMetadata()
```

Description

`mData = simout.getSimulationMetadata()` retrieves metadata information in a `SimulationMetadata` object from the `simout` `SimulationOutput` object.

Input Arguments

simout — Simulation object to get metadata from
object

Simulation object to get metadata from, specified as a `SimulationOutput` object.

Output Arguments

mData — `SimulationMetadata` object stored in the `simout` `SimulationOutput` object
object

`SimulationMetadata` object stored in the `simout` `SimulationOutput` object, returned as an object.

Examples

Retrieve Metadata From vdp Simulation

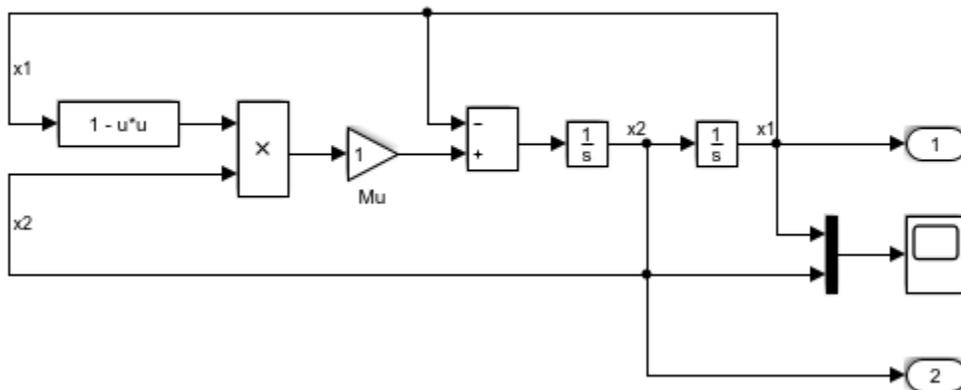
Simulate the vdp model and retrieve metadata information from the simulation.

Simulate the vdp model. Save the results of the Simulink.SimulationOutput object in simout

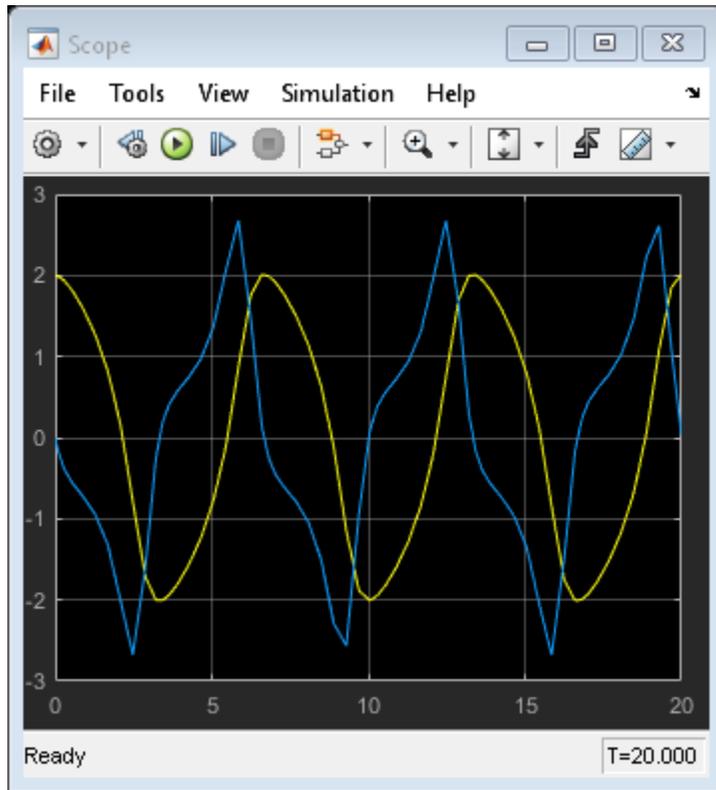
```
open_system('vdp')  
simout = sim(bdroot, 'ReturnWorkspaceOutputs', 'on');
```



van der Pol Equation



Copyright 2004-2013 The MathWorks, Inc.



Retrieve metadata information about this simulation using `mData`.

```
mData=simout.getSimulationMetadadata()
```

```
mData =
```

```
SimulationMetadadata with properties:
```

```
    ModelInfo: [1x1 struct]  
    TimingInfo: [1x1 struct]  
    ExecutionInfo: [1x1 struct]  
    UserString: ''
```

```
UserData: []
```

Alternatives

A simpler alternative to use dot notation with the `SimulationMetadata` property. For example:

```
simOut.SimulationMetadata.ModelInfo
```

Another alternative is to display simulation metadata in the Variable Editor using one of these approaches:

- Select the **Show Simulation Metadata** check box (which displays the data in a tree structure).
- Double-click the **SimulationMetadata** row.
- View the `SimulationMetadata` object.

See Also

`Simulink.SimulationMetadata` | `Simulink.SimulationOutput.setUserData` | `Simulink.SimulationOutput.setUserString`

setUserData

Class: Simulink.SimulationOutput

Package: Simulink

Store custom data in `SimulationMetadata` object that `SimulationOutput` object contains

Syntax

```
simoutNew = simout.setUserData (CustomData)
```

Description

`simoutNew = simout.setUserData (CustomData)` assigns a copy of the `simout` `SimulationOutput` object to `simoutNew`. The copy contains `CustomData` in its `SimulationMetadata` object.

Input Arguments

simout — Simulation object to get metadata from
object

Simulation object to get metadata from, specified as a `SimulationOutput` object.

CustomData — Data to store in a metadata object
data

Any custom data you want to store in the metadata object.

Output Arguments

simoutNew — Simulation object that stores metadata object with custom data
object

A copy of the `simout` `SimulationOutput` object that contains `CustomData` in its `SimulationMetadata` object, returned as an object.

Examples

Store Data in SimulationMetadata Object of vdp Simulation

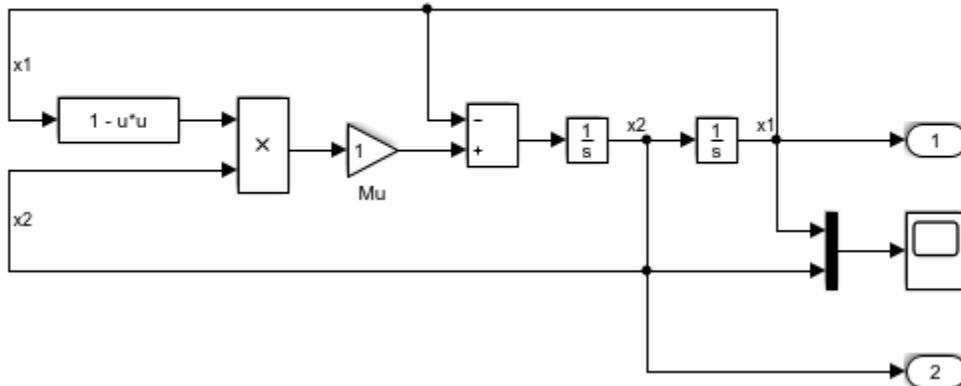
Simulate the `vdp` model. Store custom data in the `SimulationMetadata` object that the `SimulationOutput` object contains.

Simulate the `vdp` model. Save the results of the `Simulink.SimulationOutput` object in `simout`.

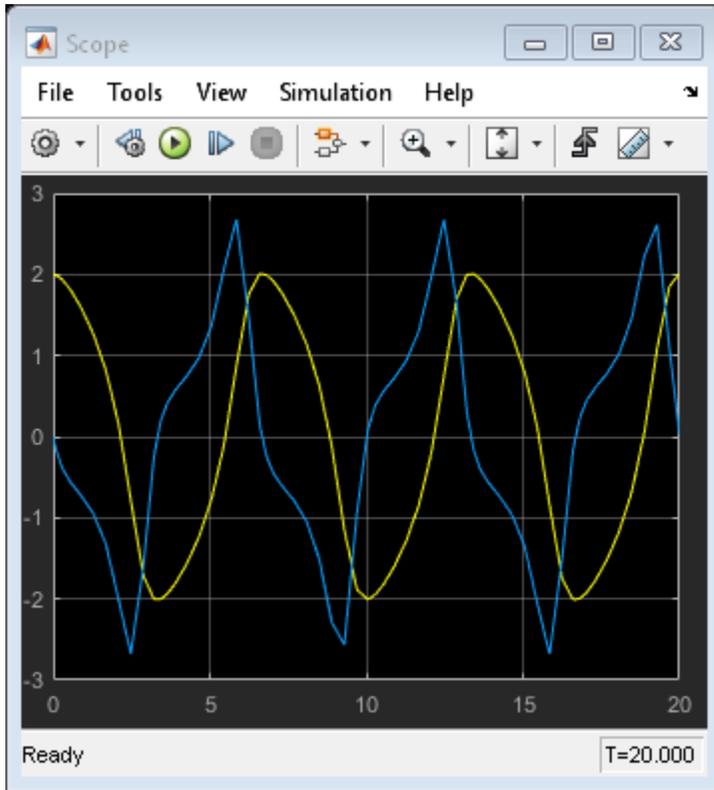
```
open_system('vdp')
simout=sim(bdroot,'ReturnWorkspaceOutputs','on');
```



van der Pol Equation



Copyright 2004-2013 The MathWorks, Inc.



Store custom data about the simulation in the `SimulationMetadata` object that `simout` contains.

```
simout=simout.setUserData(struct('param1','value1','param2','value2','param3','value3'))
```

Use `SimulationOutput.getSimulationMetadata` to retrieve the information you stored.

```
mData=simout.getSimulationMetadata();
disp(mData.UserData)

    param1: 'value1'
    param2: 'value2'
```

```
param3: 'value3'
```

See Also

[Simulink.SimulationMetadata](#) |

[Simulink.SimulationOutput.getSimulationMetadata](#) |

[Simulink.SimulationOutput.setUserString](#)

setUserString

Class: Simulink.SimulationOutput

Package: Simulink

Store custom character vector in `SimulationMetadata` object that `SimulationOutput` object contains

Syntax

```
simoutNew = simout.setUserString(CustomString)
```

Description

`simoutNew = simout.setUserString(CustomString)` assigns a copy of the `simout` `SimulationOutput` object to `simoutNew`. The copy contains `CustomString` in its `SimulationMetadata` object.

Input Arguments

simout — Simulation object to get metadata from
object

Simulation object to get metadata from, specified as a `SimulationOutput` object.

CustomString — Character vector to store in a metadata object
character vector

Any custom character vector you want to store in the metadata object.

Output Arguments

simoutNew — Simulation object that stores metadata object with custom character vector
object

A copy of the `simout` `SimulationOutput` object that contains `CustomString` in its `SimulationMetadata` object, returned as an object.

Examples

Store a Character Vector in SimulationMetadata Object of vdp Simulation

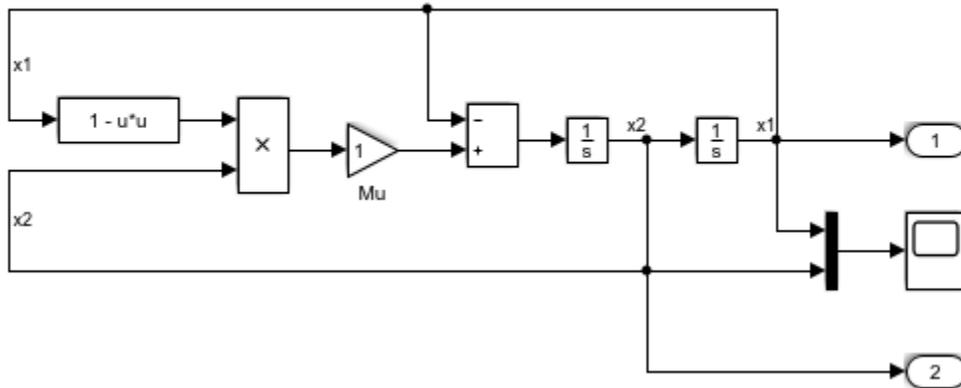
Simulate the `vdp` model. Store a custom character vector in the `SimulationMetadata` object that the `SimulationOutput` object contains.

Simulate the `vdp` model. Save the results of the `Simulink.SimulationOutput` object in `simout`.

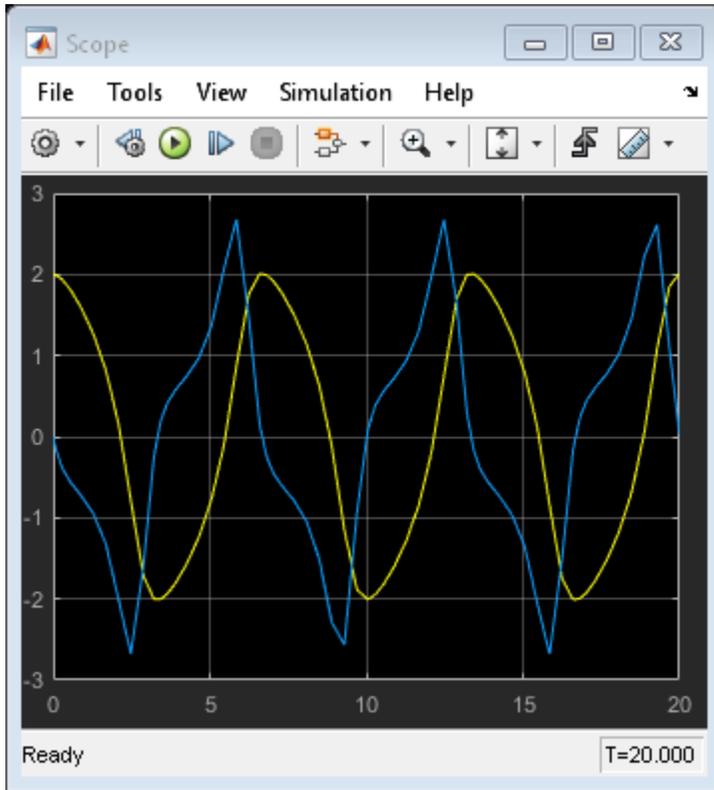
```
open_system('vdp')
simout=sim(bdroot,'ReturnWorkspaceOutputs','on');
```



van der Pol Equation



Copyright 2004-2013 The MathWorks, Inc.



Store a character vector to describe the simulation.

```
simout=simout.setUserString('First Simulation');
```

Use `SimulationOutput.getSimulationMetadata` to retrieve the information you stored.

```
mData=simout.getSimulationMetadata();  
disp(mData.UserString)
```

First Simulation

See Also

[Simulink.SimulationMetadata](#) |
[Simulink.SimulationOutput.getSimulationMetadata](#) |
[Simulink.SimulationOutput.setUserData](#)

who

Class: Simulink.SimulationOutput

Package: Simulink

Access and display output variable names of simulation

Syntax

```
simOutVar = simOut.who
```

Description

simOutVar = simOut.who returns the names of all simulation output variables, including workspace variables.

Output Arguments

simOutVar

Character vector array of output variable names of simulation.

Examples

Simulate vdp and store the character vector values of the output variable names.

```
simOut = sim('vdp','SimulationMode','rapid','AbsTol','1e-5',...  
            'SaveState','on','StateSaveName','xoutNew',...  
            'SaveOutput','on','OutputSaveName','youtNew');  
simOutVar = simOut.who
```

Alternatives

A simpler alternative to use dot notation. For example, to access data for the `xoutNew` output variable, you can use this command:

```
simOut.xoutNew
```

See Also

```
Simulink.SimulationOutput.find | Simulink.SimulationOutput.get
```

Simulink.SubSystem.convertToModelReference

Convert subsystem to model reference

Syntax

```
Simulink.SubSystem.convertToModelReference(gcb, 'UseConversionAdvisor', true)
```

```
[success, mdlRefBlkHs] = Simulink.SubSystem.convertToModelReference(subsys, mdlRefs)
```

```
[success, mdlRefBlkHs] = Simulink.SubSystem.convertToModelReference(subsys, mdlRefs, Name, Value)
```

Description

`Simulink.SubSystem.convertToModelReference(gcb, 'UseConversionAdvisor', true)` opens the Model Reference Conversion Advisor for the currently selected subsystem block.

`[success, mdlRefBlkHs] = Simulink.SubSystem.convertToModelReference(subsys, mdlRefs)` converts the specified subsystems to referenced models using the `mdlRefs` value.

For each subsystem that the function converts, it:

- Creates a model
- Copies the contents of the subsystem into the new model
- Updates any root-level Inport, Outport, Trigger, and Enable blocks and the configuration parameters of the model to match the compiled attributes of the original subsystem
- Copies the contents of the model workspace of the original model to the new model

Before you use the function, load the model containing the subsystem.

[success,mdlRefBlkHs] = Simulink.SubSystem.convertToModelReference (subsys,mdlRefs,Name,Value) uses additional options specified by one or more Name,Value pair arguments.

Examples

Open the Model Reference Conversion Advisor

Open the f14 model.

```
open_system('f14');
```

In the f14 model, select the Controller subsystem output signal, click the **Simulation**



Data Inspector button arrow, and select **Log Selected Signals**.

In the Simulink Editor, select the Controller subsystem. Then open the Model Reference Conversion Advisor from the command line.

```
Simulink.SubSystem.convertToModelReference(gcb,'UseConversionAdvisor',true);
```

Perform the conversion using the advisor.

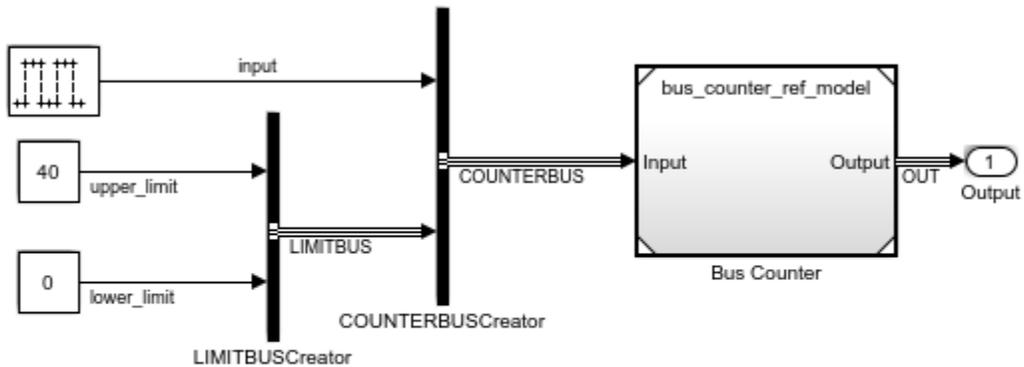
Convert Subsystem to Referenced Model

Convert the Bus Counter subsystem to a referenced model named bus_counter_ref_model.

```
open_system('sldemo_mdhref_conversion');
Simulink.SubSystem.convertToModelReference(...
    'sldemo_mdhref_conversion/Bus Counter', ...
    'bus_counter_ref_model', ...
    'AutoFix',true,...
    'ReplaceSubsystem',true,...
    'CheckSimulationResults',true);

### Successfully converted Subsystem to Model reference block
```

Conversion to Model Reference Example



Double click here to convert
'Bus Counter' Subsystem to
Referenced Model

Copyright 1990-2014 The MathWorks, Inc.

Convert Multiple Subsystems to Referenced Models

Convert the two subsystems with one command.

```
open_system('f14');
set_param(gcs, 'SaveOutput', 'on', 'SaveFormat', 'Dataset');
set_param(gcs, 'SignalResolutionControl', 'UseLocalSettings');
Simulink.SubSystem.convertToModelReference(...
{'f14/Controller', 'f14/Aircraft Dynamics Model'},...
{'controller_ref_model', 'aircraft_dynamics_ref_model'},...
'ReplaceSubsystem', true, ...
```

```
'AutoFix', true, ...
'CheckSimulationResults', true)
```

- sldemo_mdhref_conversion
- “Convert a Subsystem to a Referenced Model”
- “Model Referencing”

Input Arguments

subsys — Subsystems to convert

character vector | subsystem handle | cell array of character vectors | array of subsystem handles

Subsystems to convert, specified as a character vector, subsystem handle, or cell array of strings or array of subsystem handles.

For information about which subsystems you can convert, see “Limitations on Subsystems That You Can Convert”.

Note You cannot convert a parent subsystem a child of that subsystem at the same time.

Data Types: `double`

mdlRefs — Referenced model names

character vector | cell array of character vectors

Referenced model names, specified as a character vector or cell array of character vectors. Each model name must be 59 characters or less.

If you specify a cell array of subsystems to convert, specify a cell array of referenced model names. Each model name corresponds to the specified subsystem, in the same order.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `Simulink.SubSystem.convertToModelReference...
(engineSubsys, engineModelRef, 'ReplaceSubsystem', true)`

AutoFix — Fix all conversion issues that can be fixed automatically

`false` (default) | `true`

If you set `AutoFix` to `true`, the function fixes all conversion issues that it can fix.

For issues that the function cannot fix, the conversion process generates error messages that you address by modifying the model.

Note If you set 'Force' to `true`, then the function does not automatically fix conversion issues.

Data Types: `logical`

Force — Complete conversion even with errors

`false` (default) | `true`

If you set 'Force' to `true`, the function returns conversion errors as warnings and continues with the conversion without fixing the errors. This option allows you to use the function to do the initial steps of conversion and then complete the conversion process yourself.

If you set `Force` to `true`, then the function does not fix conversion issues, even if you set 'AutoFix' to `true`. However, the success output argument is `true`, regardless of whether any conversion errors occurred.

CheckSimulationResults — Compare simulation results before and after conversion

`false` (default) | `true`

Compare simulation results before and after conversion, specified as `true` or `false`.

Before performing the conversion, enable signal logging for the subsystem output signals of interest in the model.

For the `Simulink.SubSystem.convertToModelReference` command, set:

- 'CheckSimulationResults' to true
- 'AbsoluteTolerance'
- 'RelativeTolerance'
- 'SimulationModes' to the same as the simulation mode as in the original model

If the difference between simulation results exceeds the tolerance level, the function displays a message.

AbsoluteTolerance — Absolute signal tolerance for comparison

'1e-06' (default) | double

Absolute signal tolerance for comparison, specified as a double. Use the option only if you set `CheckSimulationResults` to true.

Data Types: double

RelativeTolerance — Relative signal tolerance for comparison

'1e-06' (default) | double

Relative signal tolerance for comparison, specified as a double. Use the option only if you set `CheckSimulationResults` to true.

Data Types: double

DataFileName — Name of file for storing conversion data

character vector

Name of file for storing conversion data, specified as a character vector. You can specify an absolute or relative path.

You can save the conversion data in a MAT-file (default) or a MATLAB file. If you use a `.m` file extension, the function serializes all variables to a MATLAB file.

By default, the function uses a file name consisting of the model name plus `_conversion_data.mat`.

ReplaceSubsystem — Replace content of each subsystem with Model blocks

false (default) | true

Replace subsystem blocks with Model blocks, specified as true or false. The Model block references the referenced model.

By default, the function displays the referenced models in separate Simulink Editor windows.

If you set the value to `true`, consider making a backup of the original model before you convert the subsystems. If you want to undo the conversion, having a backup makes it easier to restore the model.

If you set `ReplaceSubsystem` to `true`, the conversion action depends on whether you use the automatic fix options.

- If you use the automatic fixes, then the conversion replaces the Subsystem block with a Model block unless the automatic fixes change the input or output ports. If the ports change, then the conversion includes the contents of the subsystem in a Model block that is inserted in the Subsystem block.
- If you do not use the automatic fixes, then the conversion replaces the Subsystem block with a Model block.

Data Types: `logical`

CreateWrapperSubsystem — Insert wrapper subsystem to preserve model layout

`false` (default) | `true`

Insert wrapper subsystem to preserve model layout, specified as `true` or `false`. When you convert a subsystem to a referenced model, you can have the conversion process insert a wrapper subsystem to preserve the layout of a model. The subsystem wrapper contains the Model block from the conversion.

The conversion creates a wrapper subsystem automatically if the conversion modifies the Model block interface by adding ports.

Data Types: `logical`

SimulationModes — Simulation mode for Model blocks

`'Normal'` (default) | `'Accelerator'`

Simulation mode for Model blocks, specified as `'Normal'` or `'Accelerator'`. The simulation mode setting applies to the Model blocks that reference the models that the conversion creates.

BuildTarget — Model reference targets to generate

`'Sim'` | `'RTW'`

Model reference targets to generate.

- 'Sim' — Model reference simulation target
- 'RTW' — Code generation target

Output Arguments

success — Conversion status

1 | 0

Conversion status. A value of 1 indicates a successful conversion.

If you set 'Force' to `true`, the function returns a value of 1 if the conversion completes. However, the simulation results can differ from the simulation results for the model before conversion.

mdlRefBlkHs — Handles of created Model blocks

handle of Model block | array of handles of Model blocks

Handles of created Model blocks, returned as a double or cell array.

Data Types: `double`

Tips

- You cannot convert a parent subsystem a child of that subsystem at the same time.
- Specifying multiple subsystems to convert with one command can save time, compared to converting each subsystem separately. The multiple-subsystem conversion process compiles the model one time.
- If you specify multiple subsystems to convert, the conversion process attempts to convert each subsystem. Successfully converted subsystems produce referenced models, even if the conversions of other subsystems fail.
- If you specify multiple subsystems, consider:
 - Specifying these 'Autofix', 'ReplaceSubsystem', 'CheckSimulationResults' name and value pairs, set to `true`.
 - In the model, setting a short simulation time.

- Simulink uses the data dictionary to save the bus objects that it creates as part of the conversion processing when both these conditions exist:
 - The top model uses a data dictionary.
 - All changes to the top model are saved.
- After you complete the conversion, update the model as necessary to meet your modeling requirements. For details, see “Integrate the Referenced Model into the Parent Model”.
- Converting a masked subsystem can require you to perform additional tasks to maintain the same general behavior that the masked subsystem provided.

If the subsystem that you convert contains a masked block, consider masking the Model block in your new referenced model (see “Block Masks”). Configure the referenced model to support the functionality of the masked subsystem.

Note A referenced model does not support the functionality that you can achieve with mask initialization code to create masked parameters.

For mask parameters, replace the mask parameters with model arguments (see “Parameterize Instances of a Reusable Referenced Model”):

- 1** In the model workspace of the referenced model, create a variable for each mask parameter.
- 2** In the Model Explorer, select the `Model Workspace` node. In the **Contents** pane, select the **Argument** check box to identify the variables as model arguments.
- 3** In the new Model block, on the **Arguments** tab, in the **Model arguments** table, specify the values for the model arguments.

For masked callbacks, icons, ports, and documentation:

- 1** In the backup copy, open the Mask Editor on the masked subsystem and copy the content you want into the masked Model block.
- 2** In the Mask Editor for the new Model block, paste the masked subsystem content.

See Also

`Simulink.BlockDiagram.copyContentsToSubsystem` | `Simulink.Bus.save` |
`Simulink.SubSystem.copyContentsToBlockDiagram`

Topics

`sldemo_mdhref_conversion`

“Convert a Subsystem to a Referenced Model”

“Model Referencing”

Introduced in R2006a

Simulink.SubSystem.copyContentsToBlockDiagram

Copy contents of subsystem to empty block diagram

Syntax

```
Simulink.SubSystem.copyContentsToBlockDiagram(subsys, bdiag)
```

Description

`Simulink.SubSystem.copyContentsToBlockDiagram(subsys, bdiag)` copies the contents of the subsystem *subsys* to the block diagram *bdiag*. The subsystem and block diagram must have already been loaded. The subsystem cannot be part of the block diagram. The function affects only blocks, lines, and annotations; it does not affect nongraphical information such as configuration sets.

This function cannot be used if the destination block diagram contains any blocks or signals. Other types of information can exist in the destination block diagram and are unaffected by the function. Use `Simulink.BlockDiagram.deleteContents` if necessary to empty the block diagram before using `Simulink.SubSystem.copyContentsToBlockDiagram`.

Tip To flatten a model hierarchy by expanding the contents of a subsystem to the system that contains that subsystem, do not use the `Simulink.SubSystem.copyContentsToBlockDiagram` function. Instead, expand the subsystem, as described in “Expand Subsystem Contents”.

Input Arguments

subsys

Subsystem name or handle

bdiag

Block diagram name or handle

Examples

Copy the graphical contents of `f14/Controller`, including all nested subsystems, to a new block diagram:

```
% open f14
open_system('f14');

% create a new model
newbd = new_system;
open_system(newbd);

% copy the subsystem
Simulink.SubSystem.copyContentsToBlockDiagram('f14/Controller', newbd);

% close f14 and the new model
close_system('f14', 0);
close_system(newbd, 0);
```

See Also

[Simulink.BlockDiagram.copyContentsToSubsystem](#) |
[Simulink.BlockDiagram.deleteContents](#) |
[Simulink.SubSystem.convertToModelReference](#) |
[Simulink.SubSystem.deleteContents](#)

Topics

“Model Editing Fundamentals”
“Create a Subsystem”
“Expand Subsystem Contents”

Introduced in R2007a

Simulink.SubSystem.deleteContents

Delete contents of subsystem

Syntax

```
Simulink.SubSystem.deleteContents(subsys)
```

Description

`Simulink.SubSystem.deleteContents(subsys)` deletes the contents of the subsystem *subsys*. The function affects only blocks, lines, and annotations. The subsystem must have already been loaded.

Note This function does not delete library blocks in a subsystem.

Input Arguments

subsys

Subsystem name or handle

Examples

Delete the graphical contents of `Controller`, including all nested subsystems:

```
Simulink.SubSystem.deleteContents('f14/Controller');
```

See Also

`Simulink.BlockDiagram.copyContentsToSubsystem` |
`Simulink.BlockDiagram.deleteContents` |

`Simulink.SubSystem.convertToModelReference |`
`Simulink.SubSystem.copyContentsToBlockDiagram`

Topics

“Model Hierarchy”

“Create a Subsystem”

Introduced in R2007a

Simulink.SubSystem.getChecksum

Return checksum of nonvirtual subsystem

Syntax

```
[checksum,details] = Simulink.SubSystem.getChecksum(subsys)
```

Description

`[checksum,details] = Simulink.SubSystem.getChecksum(subsys)` returns the checksum of the specified nonvirtual subsystem. Simulink computes the checksum based on the subsystem parameter settings and the blocks the subsystem contains. Virtual subsystems do not have checksums.

One use of this command is to determine why code generated for a subsystem is not being reused.

Note `Simulink.SubSystem.getChecksum` compiles the model that contains the specified subsystem, if the model is not already in a compiled state. If you need to get the checksum for multiple subsystems and want to avoid multiple compiles, use the command `model([], [], [], 'compile')` to place the model in a compiled state before using `Simulink.SubSystem.getChecksum`.

This command accepts the argument `subsys`, which is the full name or handle of the nonvirtual subsystem block for which you are returning checksum data.

Examples

Run `getChecksum` on Model

Run the function `Simulink.SubSystem.getChecksum` on the model `rtwdemo_ssreuse`. In the MATLAB editor window, both output structures are

displayed. In the workspace pane, double-click on either of the structures to view its contents.

Load the model `rtwdemo_ssreuse`.

```
rtwdemo_ssreuse
```

Select subsystem `SS1` and execute the follow line of code in the MATLAB editor to get the full name and path to the subsystem `SS1`:

```
path_ss1 = gcb
```

Run the function `getChecksum` on the subsystem with the following command:

```
[chksum1, chksum1_details] = Simulink.SubSystem.getChecksum(path_ss1)
```

The output structures `chksum1` and `chksum1_details` will store the output of the `getChecksum` function call.

```
chksum1 =
```

```
struct with fields:
    Value: [4x1 uint32]
    MarkedUnique: 0
```

```
chksum1_details =
```

```
struct with fields:
    ContentsChecksum: [1x1 struct]
    InterfaceChecksum: [1x1 struct]
    ContentsChecksumItems: [359x1 struct]
    InterfaceChecksumItems: [60x1 struct]
```

Input Arguments

subsys — Name or handle of nonvirtual subsystem

character vector

Input the full name of the nonvirtual subsystem for which you want to calculate the checksum.

Data Types: `char`

Output Arguments

checksum — A structure that stores the value of the checksum and indicates whether `subsys` contains unique block or subsystem properties which prevent generated code reuse
structure

Checksum information, returned as a structure with the fields:

value — Array of four 32-bit integers that represents the subsystem's 128-bit checksum
`4x1 uint32`

MarkedUnique — True if the subsystem or the blocks it contains have properties that would prevent the code generated for the subsystem from being reused; otherwise, false
`bool`

details — A structure that stores checksum data on model contents and the interface
structure

Checksum information, returned as a structure with the fields:

ContentsChecksum — A structure of the same form as `checksum`, representing a checksum that provides information about all blocks in the system
structure

InterfaceChecksum — A structure of the same form as `checksum`, representing a checksum that provides information about the subsystem's block parameters and connections
structure

ContentsChecksumItems — Structure array that Simulink uses to compute the checksum for `ContentsChecksum`
structure

Structure array returned with the following fields:

handle — Object for which Simulink added an item to the checksum. For a block, the handle is a full block path. For a block port, the handle is the full block path and a character vector that identifies the port
`char array`

Identifier — Descriptor of the item Simulink added to the checksum. If the item is a documented parameter, the identifier is the parameter name

char array

value — Value of the item Simulink added to the checksum. If the item is a parameter, value is the value returned by `get_param(handle, identifier)`

type

InterfaceChecksumItems — Structure array that Simulink uses to compute the checksum for `InterfaceChecksum`

structure

Structure array returned with the following fields:

Handle — Object for which Simulink added an item to the checksum. For a block, the handle is a full block path. For a block port, the handle is the full block path and a character vector that identifies the port

char array

Identifier — Descriptor of the item Simulink added to the checksum. If the item is a documented parameter, the identifier is the parameter name

char array

value — Value of the item Simulink added to the checksum. If the item is a parameter, value is the value returned by `get_param(handle, identifier)`

type

See Also

`Simulink.BlockDiagram.getChecksum`

Introduced in R2006b

Simulink.suppressDiagnostic

Suppress a diagnostic from a specific block

Syntax

```
Simulink.suppressDiagnostic(source, message_id)
Simulink.suppressDiagnostic(diagnostic)
```

Description

`Simulink.suppressDiagnostic(source, message_id)` suppresses all instances of diagnostics represented by `message_id` thrown by the blocks specified by `source`.

`Simulink.suppressDiagnostic(diagnostic)` suppresses the diagnostics associated with `MSLDiagnostic` object `diagnostic`.

Examples

Suppress a Warning Thrown By a Block

Using the model from “Suppress Diagnostic Messages Programmatically”, use the `Simulink.suppressDiagnostic` function to suppress the parameter precision loss warning thrown by the Constant block, `one`.

```
Simulink.suppressDiagnostic('Suppressor_CLI_Demo/one', ...
    'SimulinkFixedPoint:util:fxpParameterPrecisionLoss');
```

- “Suppress Diagnostic Messages Programmatically”

Input Arguments

source — Block or model object throwing the diagnostic
block path | block handle

The source of the diagnostic, specified as a block path, block handle, cell array of block paths, or cell array of block handles.

To get the block path, use the `gcb` function.

To get the block handle, use the `getSimulinkBlockHandle` function.

Data Types: `char` | `cell`

message_id — message identifier of diagnostic

message identifier | cell array of message identifiers

Message identifier of the diagnostic, specified as a character vector or a cell array of character vectors. You can find the message identifier of diagnostics thrown during simulation by accessing the `ExecutionInfo` property of the `Simulink.SimulationMetadata` object associated with a simulation. You can also use the `lastwarn` function.

Data Types: `char` | `cell`

diagnostic — Diagnostic object

`MSLDiagnostic` object

Diagnostic specified as an `MSLDiagnostic` object. Access the `MSLDiagnostic` object through the `ExecutionInfo` property of the `Simulink.SimulationMetadata` object.

Data Types: `struct`

See Also

`Simulink.SuppressedDiagnostic` | `Simulink.SuppressedDiagnostic.restore`
| `Simulink.getSuppressedDiagnostics` | `Simulink.restoreDiagnostic`

Topics

“Suppress Diagnostic Messages Programmatically”

sint

Create `Simulink.NumericType` object describing signed integer data type

Syntax

```
a = sint(WordLength)
```

Description

`sint(WordLength)` returns a `Simulink.NumericType` object that describes the data type of a signed integer with a word size given by `WordLength`.

Note `sint` is a legacy function. In new code, use `fixdt` instead. In existing code, replace `sint(WordLength)` with `fixdt(1,WordLength,0)`.

Examples

Define a 16-bit signed integer data type.

```
a = sint(16)
```

```
a =
```

```
    NumericType with properties:
```

```
        DataTypeMode: 'Fixed-point: binary point scaling'  
        Signedness: 'Signed'  
        WordLength: 16  
        FractionLength: 0  
        IsAlias: 0  
        DataScope: 'Auto'  
        HeaderFile: ''  
        Description: ''
```

See Also

`Simulink.NumericType` | `fixdt` | `float` | `sfix` | `sfrac` | `ufix` | `ufrac` | `uint`

Introduced before R2006a

slbuild

Build standalone executable or model reference target for model; except where noted, this function requires a Simulink Coder license

Syntax

```
slbuild(model,buildSpec,varArgIn)
```

Description

`slbuild(model,buildSpec,varArgIn)` builds a standalone Simulink Coder binary executable file from the *model*, using the current model configuration settings. If the *model* has not been loaded, `slbuild` loads it before initiating the build process. The *buildSpec* and *varArgIn* arguments are optional.

Do not use `rtwbuild`, `rtwrebuild`, or `slbuild` commands with parallel language features (Parallel Computing Toolbox) (for example, within a `parfor` or `spm` loop). For information about parallel builds of referenced models, see “Reduce Build Time for Referenced Models” (Simulink Coder).

You cannot use `slbuild` to build subsystems.

Examples

Generate Code and Build Executable Image for Model

Generate C code for model `rtwdemo_rtwinintro`.

```
slbuild('rtwdemo_rtwinintro')  
% same operation as ...  
% slbuild('rtwdemo_rtwinintro','StandaloneRTWTarget')
```

For the GRT target, the coder generates the following code files and places them in folders `rtwdemo_rtwinintro_grt_rtw` and `slprj/grt/_sharedutils`.

Model Files	Shared Files	Interface Files	Other Files
rtwdemo_rtwinintro.c	rtwtypes.h	rtmodel.h	none
rtwdemo_rtwinintro.h	multiword_types.h		
rtwdemo_rtwinintro_private.h	builtin_typeid_types.h		
rtwdemo_rtwinintrotypes.h			

If the following model configuration parameters settings apply, the coder generates additional results.

Parameter Setting	Results
Code Generation > Generate code only pane is cleared	Executable image rtwdemo_rtwinintro.exe
Code Generation > Report > Create code generation report is selected	Report appears, providing information and links to generated code files, subsystem and code interface reports, entry-point functions, inports, outports, interface parameters, and data stores

Force Top Model Build

Generate code and build an executable image for `rtwdemo_mdleftop`, which refers to model `rtwdemo_mdleftop`, regardless of model checksums and parameter settings.

```
sbuild('rtwdemo_mdleftop', 'StandaloneRTWTarget', ...
      'ForceTopModelBuild', true)
```

Clean Top Model Build

Clean the model build area enough to trigger regeneration of the top model code at the next build.

```
slbuild('rtwdemo_rtwinintro','CleanTopModel')
```

Input Arguments

model — Specifies model for the build process

handle | character vector

Model for which to build a standalone executable or model reference target, specified as a handle or a character vector representing the model name.

Example: `gcs`

buildSpec — Specifies the code generation action for the build process

'StandaloneRTWTarget' (default) | 'ModelReferenceSimTarget' |
'ModelReferenceRTWTarget' | 'ModelReferenceRTWTargetOnly' |
'CleanTopModel'

The *buildSpec* directs the code generator to perform the selected build action for the *model* and the build process:

- Honors the setting of the **Rebuild** parameter on the **Model Referencing** pane of the Configuration Parameters dialog box.
- Requires a Simulink Coder license only if you build a model reference Simulink Coder target, not if you build only a model reference simulation target.

The *buildSpec* argument must be one of the following:

buildSpec	Build Action
'StandaloneRTWTarget'	Builds a standalone Simulink Coder binary executable file from the <i>model</i> , using the current model configuration settings. If the <i>model</i> has not been loaded, <code>slbuild</code> loads it before initiating the build process.
'ModelReferenceSimTarget'	Builds a model reference simulation target (does not require a Simulink Coder license)
'ModelReferenceRTWTarget'	Builds a model reference Simulink Coder target and the corresponding model reference simulation target
'ModelReferenceRTWTargetOnly'	Builds only a model reference Simulink Coder target

buildSpec	Build Action
'CleanTopModel'	Cleans the model build area enough to trigger regeneration of the top model code at the next build

Example: 'ModelReferenceSimTarget'

varArgIn — Name-value pair parameters that provide added arguments for the build process

name-value pairs

```
sbuild(myModel, 'StandaloneRTWTarget', 'ForceTopModelBuild', true)
```

Name-Value Pair Arguments

Specify optional comma-separated pairs of *Name*, *Value* arguments. *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as *Name1*, *Value1*, . . . , *NameN*, *ValueN*.

Example: 'UpdateThisModelReferenceTarget', 'Force'

UpdateThisModelReferenceTarget — Specifies a conditional rebuild option for the model reference build

'Force' | 'IfOutOfDateOrStructuralChange' | 'IfOutOfDate'

The 'UpdateThisModelReferenceTarget' option only applies when the *buildSpec* selection is 'ModelReferenceSimTarget', 'ModelReferenceRTWTarget', or 'ModelReferenceRTWTargetOnly'.

The 'UpdateThisModelReferenceTarget' value specifies a conditional rebuild option for the model reference target build when the **Rebuild** parameter on the **Model Referencing** pane of the Configuration Parameters dialog box is set to *Never*.

The 'UpdateThisModelReferenceTarget' value applies only to *model*, not to any models referenced by *model*.

The 'UpdateThisModelReferenceTarget' value must be one of the following:

UpdateThisModelReferenceTarget	Conditional Rebuild Action
'Force'	Unconditionally rebuilds the model. This option is equivalent to the Always rebuild option on the Model Referencing pane of the Configuration Parameters dialog box.
'IfOutOfDateOrStructuralChange'	Rebuilds the model if the build process detects any changes. This option is equivalent to the If any changes detected rebuild option on the Model Referencing pane of the Configuration Parameters dialog box.
'IfOutOfDate'	Rebuilds the model if the build process detects any changes in known dependencies of this model. This option is equivalent to the If any changes in known dependencies detected rebuild option on the Model Referencing pane of the Configuration Parameters dialog box.

Example: 'UpdateThisModelReferenceTarget', 'Force'

ForceTopModelBuild — Directs the code generator to generate code and build an executable image for the top model of the referenced model hierarchy, regardless of model checksums and parameter settings

false (default) | true

Use 'ForceTopModelBuild' value true to force the top model build.

Example: 'ForceTopModelBuild', true

See Also

rtwbuild | rtwrebuild

Topics

“Model Reference Simulation Targets”

“What Is Acceleration?”

“Perform Acceleration”

“Reuse Simulation Builds for Faster Simulations”

Introduced before R2006a

slCharacterEncoding

Change MATLAB character set encoding

Syntax

```
currentCharacterEncoding = slCharacterEncoding()  
slCharacterEncoding(encoding)
```

Description

This command allows you to change the current MATLAB character set encoding to be compatible with the character encoding of a model that you want to open.

`currentCharacterEncoding = slCharacterEncoding()` returns the current MATLAB character set encoding.

`slCharacterEncoding(encoding)` changes the MATLAB character set encoding to the specified encoding. You should only specify these values:

- 'US-ASCII'
- 'Windows-1252'
- 'ISO-8859-1'
- 'Shift_JIS'
- 'UTF-8'

If you want to use a different character encoding, you need to start MATLAB with the appropriate locale settings for your operating system. Consult your operating system manual to change the locale setting. Simulink can support any character encoding that uses single-byte or double-byte characters.

If you open a model that uses a particular character set encoding in a MATLAB session that uses a different encoding, a warning appears. For example, suppose that you create a model in a MATLAB session configured for `Shift_JIS` and open it in a session configured for `US_ASCII`. The warning message shows the encoding of the current

session and the encoding used to create the model. If you encounter any problems with corrupted characters, for example when using MATLAB files associated with the model, then try using the `sICharacterEncoding` function to change the character encoding

- 1 Close all open models.
- 2 Use `sICharacterEncoding` to change the character encoding of the current MATLAB session to match the model character encoding.
- 3 Reopen the model.

Note You must close all open models or libraries before changing the MATLAB character set encoding except when changing from 'US-ASCII' to another encoding.

See Also

Topics

“Open a Model with Different Character Encoding”

“Save Models with Different Character Encodings”

Introduced before R2006a

sldebug

Start simulation in debug mode

Syntax

```
sldebug('sys')
```

Description

`sldebug('sys')` starts a simulation in debug mode. See “Debugger Command-Line Interface” for information about using the debugger.

Examples

The following command:

```
sldebug('vdp')
```

loads the Simulink example model `vdp` into memory and starts the simulation in debug mode. Alternatively, you can achieve the same result by using the `sim` command:

```
sim('vdp', 'debug', 'on')
```

See Also

`sim`

Introduced in R2006a

sldiagnostics

Display diagnostic information about Simulink system

Syntax

```
sldiagnostics('sys')  
[txtRpt, sRpt] = sldiagnostics('sys')  
[txtRpt, sRpt] = sldiagnostics('sys', options)  
[txtRpt, sRpt] = sldiagnostics('sys', 'CompileStats')  
[txtRpt, sRpt] = sldiagnostics('sys', 'RTWBuildStats')
```

Description

`sldiagnostics('sys')` displays the following diagnostic information associated with the model or subsystem specified by `sys`:

- Number of each type of block
- Number of each type of Stateflow object
- Number of states, outputs, inputs, and sample times of the root model.
- Names of libraries referenced and instances of the referenced blocks
- Time and additional memory used for each compilation phase of the root model

If the model specified by `sys` is not loaded, then `sldiagnostics` loads the model before performing the analysis.

The command `sldiagnostics('sys', options)` displays only the diagnostic information associated with the specific operations listed as `options` character vectors. The table below summarizes the options available and their corresponding valid input and output.

With `sldiagnostics`, you can input the name of a model or the path to a subsystem. For some analysis options, `sldiagnostics` can analyze only a root model. If you provide an incompatible input for one of these analyses, then `sldiagnostics` issues a warning. Finally, if you input a Simulink Library, then `sldiagnostics` cannot perform options

that require a model compilation (**Update Diagram**). Instead, `sldiagnostics` issues a warning.

During the analysis, `sldiagnostics` will follow library links but will not follow or analyze Model References. See `find_mdhrefs` for more information on finding all Model blocks and referenced models in a specified model.

Option	Valid Inputs	Output
<code>CountBlocks</code>	root model, library, or subsystem	Lists all unique blocks in the system and the number of occurrences of each. This includes blocks that are nested in masked subsystems or hidden blocks.
<code>CountSF</code>	root model, library, or subsystem	Lists all unique Stateflow objects in the system and the number of occurrences of each.
<code>Sizes</code>	root model	Lists the number of states, outputs, inputs, and sample times, as well as a flag indicating direct feedthrough, used in the root model.
<code>Libs</code>	root model, library, or subsystem	Lists all unique libraries referenced in the root model, as well as the names and numbers of the library blocks.
<code>CompileStats</code>	root model	Lists the time and additional memory used for each compilation phase of the root model. This information helps users troubleshoot model compilation speed and memory issues.
<code>RTWBuildStats</code>	root model	Lists the same information as the <code>CompileStats</code> diagnostic. When issued with the second output argument <code>sRpt</code> , it captures the same statistics included in <code>CompileStats</code> and also the Simulink Coder build statistics. You must explicitly specify this option, because it is not part of the default analysis.

Option	Valid Inputs	Output
All	not applicable	Performs all diagnostics.

Note Running the `CompileStats` diagnostic before simulating a model for the first time will show greater memory usage. However, subsequent runs of the `CompileStats` diagnostic on the model will require less memory usage.

`[txtRpt, sRpt] = sldiagnostics('sys')` returns the diagnostic information as a textual report `txtRpt` and a structure array `sRpt`, which contains the following fields that correspond to the diagnostic options:

- `blocks`
- `stateflow`
- `sizes`
- `links`
- `compilestats`

`[txtRpt, sRpt] = sldiagnostics('sys', options)` returns only the specified options. If your chosen options specify just one type of analysis, then `sRpt` contains the results of only that analysis.

`[txtRpt, sRpt] = sldiagnostics('sys', 'CompileStats')` returns information on time and memory usage in `txtRpt` and `sRpt`.

`[txtRpt, sRpt] = sldiagnostics('sys', 'RTWBuildStats')` includes Simulink Coder build statistics in addition to the information reported for `CompileStats` in the `sRpt` output.

- `txtRpt` contains the formatted textual output of time spent in each of the phases in Simulink and Simulink Coder (if you specified `RTWBuildStats`), for example:

```
Compile Statistics For: rtwdemo_counter
  Cstat1: 0.00 seconds Model compilation pre-start
  Cstat2: 0.00 seconds Stateflow compile pre-start notification
  Cstat3: 0.10 seconds Post pre-comp-start engine event
  Cstat4: 10.00 seconds Stateflow compile start notification
  Cstat5: 0.00 seconds Model compilation startup completed
```

- `sRpt` is a MATLAB structure containing time and memory usage for each of the phases, for example:

```
sRpt =  
Model:      'myModel1'  
Statistics:  [1x134 struct]
```

The size of the `sRpt.Statistics` array indicates the number of compile and build phases executed during the operation. Examine the `Statistics` fields:

```
sRpt.Statistics(1) =  
Description:      'Phase1'  
CPUTime:         7.2490  
WallClockTime    4.0092  
ProcessMemUsage: 26.2148  
ProcessMemUsagePeak: 28.6680  
ProcessVMSize:   15.9531
```

`CPUTime` and `WallClockTime` show the elapsed time for the phase in seconds.

`ProcessMemUsage`, `ProcessMemUsagePeak` and `ProcessVMSize` show the memory consumption during execution of the phase in MB.

Examine these key metrics to understand the performance:

- `WallClockTime`—The real-time elapsed in each phase in seconds. Sum the `WallClockTime` in each phase to get the total time taken to perform the operation:

```
ElapsedTime = sum([statRpt.Statistics(:).WallClockTime]);
```

- `ProcessMemUsage`—The amount of memory consumed in each phase. Sum the `ProcessMemUsage` across all the phases to get the memory consumption during the entire operation:

```
TotalMemory = sum([statRpt.Statistics(:).ProcessMemUsage]);
```

- `ProcessMemUsagePeak`—The maximum amount of allocated memory in each phase. Get the maximum of this metric across all the phases to find the peak memory allocation during the operation:

```
PeakMemory = max([statRpt.Statistics(:).ProcessMemUsagePeak]);
```

Note Memory statistics are available only on the Microsoft Windows platform.

Examples

The following command counts and lists each type of block used in the `sldemo_bounce` model that comes with Simulink software.

```
sldiagnostics('sldemo_bounce', 'CountBlocks')
```

The following command counts and lists both the unique blocks and Stateflow objects used in the `sf_boiler` model that comes with Stateflow software; the textual report returned is captured as `myReport`.

```
myReport = sldiagnostics('sf_boiler', 'CountBlocks', 'CountSF')
```

The following commands open the `f14` model that comes with Simulink software, and counts the number of blocks used in the `Controller` subsystem.

```
sldiagnostics('f14/Controller', 'CountBlocks')
```

The following command runs the `Sizes` and `CompileStats` diagnostics on the `f14` model, capturing the results as both a textual report and structure array.

```
[txtRpt, sRpt] = sldiagnostics('f14', 'Sizes', 'CompileStats')
```

See Also

`find_system` | `get_param`

Introduced in R2006a

sldiagviewer.diary

Log simulation warnings and errors and build information to file

Syntax

```
sldiagviewer.diary  
sldiagviewer.diary(filename)  
sldiagviewer.diary(toggle)  
sldiagviewer.diary(filename, 'UTF-8')
```

Description

`sldiagviewer.diary` intercepts build information, warnings, and errors transmitted to the Command Window or the Diagnostic Viewer and logs them to a text file `diary.txt` in the current folder.

`sldiagviewer.diary(filename)` toggles the logging state of the text file specified by `filename`.

`sldiagviewer.diary(toggle)` turns logging to the log file on or off. The setting applies to the last file name you specified for logging or to `diary.txt` if you did not specify a file name.

`sldiagviewer.diary(filename, 'UTF-8')` specifies the character encoding for the log file `filename`.

Examples

Log Build Information and Simulation Warnings and Errors

Start logging build information and simulation warnings and errors to `diary.txt`.

```
sldiagviewer.diary
open_system('vdp')
rtwbuild('vdp')
```

Open diary.txt to view logs.

```
### Starting build procedure for model: vdp
### Build procedure for model: 'vdp' aborted due to an error.
...
```

Log to Specific File

Set up logging to a file.

```
sldiagviewer.diary('C:\MyLogs\log1.txt')
```

Toggle File Logging State

Switch the logging state of a file.

```
sldiagviewer.diary('C:\MyLogs\log1.txt') % Start logging
open_system('vdp')
rtwbuild('vdp')

sldiagviewer.diary('off') % Switch off logging
open_system('sldemo_fuelsys')
rtwbuild('sldemo_fuelsys')

sldiagviewer.diary('on') % Resume logging
```

Specify Log File Name and Character Encoding

Set the file name to log to and the character encoding to use.

```
sldiagviewer.diary('C:\MyLogs\log1.txt', 'UTF-8')
```

- “View Diagnostics”
- “Customize Diagnostic Messages”

Input Arguments

toggle — Logging state

'off' | 'on'

Logging state, specified as 'on' or 'off'.

Example: `sldiagviewer.diary('on')`

filename — Name of file to log data to

character vector

Name of file to log data to, specified as a character vector.

Example: `sldiagviewer.diary('C:\Simulations\mySimulationDiary.txt')`

See Also

Topics

“View Diagnostics”

“Customize Diagnostic Messages”

Introduced in R2014a

sldiscmdl

Discretize model that contains continuous blocks

Syntax

```
sldiscmdl('model_name', sample_time)
sldiscmdl('model_name', sample_time, method)
sldiscmdl('model_name', sample_time, options)
sldiscmdl('model_name', sample_time, method, freq)
sldiscmdl('model_name', sample_time, method, options)
sldiscmdl('model_name', sample_time, method, freq, options)
[old_blks, new_blks] =
sldiscmdl('model_name', sample_time, method, freq, options)
```

Description

`sldiscmdl('model_name', sample_time)` discretizes the model named 'model_name' using the specified `sample_time`. The model does not need to be open, and the units for `sample_time` are simulation seconds.

`sldiscmdl('model_name', sample_time, method)` discretizes the model using `sample_time` and the transform method specified by `method`.

`sldiscmdl('model_name', sample_time, options)` discretizes the model using `sample_time` and criteria specified by the `options` cell array. This array consists of four elements: `{target, replace_with, put_into, prompt}`.

`sldiscmdl('model_name', sample_time, method, freq)` discretizes the model using `sample_time`, `method`, and the critical frequency specified by `freq`. The units for `freq` are Hz. When you specify `freq`, `method` must be 'prewarp'.

`sldiscmdl('model_name', sample_time, method, options)` discretizes the model using `sample_time`, `method`, and `options`.

`sldiscmdl('model_name', sample_time, method, freq, options)` discretizes the model using *sample_time*, *method*, *freq*, and *options*. When you specify *freq*, *method* must be 'prewarp'.

`[old_blks, new_blks] = sldiscmdl('model_name', sample_time, method, freq, options)` discretizes the model using *sample_time*, *method*, *freq*, and *options*. When you specify *freq*, *method* must be 'prewarp'. The function also returns two cell arrays that contain full path names of the original, continuous blocks and the new, discretized blocks.

Input Arguments

model_name

Name of the model to discretize.

sample_time

Sample-time specification for the model:

Scalar value

Sample time with zero offset, such as 1

Two-element vector

Sample time with nonzero offset, such as
[1 0.1]

method

Method of converting blocks from continuous to discrete mode:

'zoh' (default)

Zero-order hold on the inputs

'foh'

First-order hold on the inputs

'tustin'

Bilinear (Tustin) approximation

'prewarp'

Tustin approximation with frequency prewarping

'matched'

Matched pole-zero method

For single-input, single-output (SISO) systems only

freq

Critical frequency in Hz. This input applies only when the *method* input is 'prewarp'.

options

Cell array {*target*, *replace_with*, *put_into*, *prompt*}, where each element can take the following values:

<i>target</i>	'all' (default)	Discretize all continuous blocks
	'selected'	Discretize only selected blocks in the model
	'full_blk_path'	Discretize specified block
<i>replace_with</i>	'parammask' (default)	Create discrete blocks whose parameters derive from the corresponding continuous blocks
	'hardcoded'	Create discrete blocks with hard-coded parameters placed directly into each block dialog box
<i>put_into</i>	'copy' (default)	Create discretization in a copy of the original model
	'configurable'	Create discretization candidate in a configurable subsystem
	'current'	Apply discretization to the current model
	'untitled'	Create discretization in a new untitled window
<i>prompt</i>	'on' (default)	Show discretization information at the command prompt
	'off'	Do not show discretization information at the command prompt

Examples

Discretize all continuous blocks in the `slexAircraftExample` model using a 1-second sample time:

```
sldiscmdl('slexAircraftExample',1);
```

Discretize the Aircraft Dynamics Model subsystem in the `slexAircraftExample` model using a 1-second sample time, a 0.1-second offset, and a first-order hold transform method:

```
sldiscmdl('slexAircraftExample',[1 0.1],'foh',...  
{'slexAircraftExample/Aircraft Dynamics Model',...  
'parammask','copy','on'});
```

Discretize the Aircraft Dynamics Model subsystem in the `slexAircraftExample` model and retrieve the full path name of the second discretized block:

```
[old_blks,new_blks] = sldiscmdl('slexAircraftExample',[1 0.1],...  
'foh',{'slexAircraftExample/Aircraft Dynamics Model','parammask',...  
'copy','on'});  
% Get full path name of the second discretized block  
new_blks{2}
```

See Also

`slmdliscui`

Topics

“Discretize a Model with the `sldiscmdl` Function”

Introduced before R2006a

slIsFileChangedOnDisk

Determine whether model has changed since it was loaded

Syntax

```
Changed = slIsFileChangedOnDisk('sys')
```

Description

`Changed = slIsFileChangedOnDisk('sys')` Returns true if the file which contains block diagram `sys` was changed on disk since the block diagram was loaded.

Examples

To ensure that code is not generated for a model whose file has changed on disk since it was loaded, include the following in the 'entry' section of the `STF_make_rtw_hook.m` file:

```
if (slIsFileChangedOnDisk(sys))  
    error('File has changed on disk since it was loaded. Aborting code generation.');
```

```
end
```

See Also

Topics

“Customize Build Process with STF_make_rtw_hook File” (Simulink Coder)

“Model File Change Notification”

Introduced in R2007b

slLibraryBrowser

Open Simulink Library Browser

Syntax

```
slLibraryBrowser
slLibraryBrowser('open')
slLibraryBrowser('noshow')
libraryhandle = slLibraryBrowser
slLibraryBrowser('close')
```

Description

`slLibraryBrowser` opens the Simulink Library Browser.

If you want to load the Simulink block library, use `load_system simulink` instead.

If you want to start Simulink without opening any windows, use the faster `start_simulink` instead.

`slLibraryBrowser('open')` opens the Library Browser.

`slLibraryBrowser('noshow')` loads the Library Browser in memory without making it visible. Use this to make future calls to `slLibraryBrowser('open')` faster.

`libraryhandle = slLibraryBrowser` returns the handle of the Library Browser object.

`slLibraryBrowser('close')` closes the Library Browser.

Examples

Open and Close the Library Browser

```
sLibraryBrowser  
sLibraryBrowser('close')
```

Load the Library Browser and Get a Handle

```
libraryhandle = sLibraryBrowser('noshow')
```

- “Build and Edit a Model in the Simulink Editor”

See Also

[simulink](#) | [start_simulink](#)

Topics

“Build and Edit a Model in the Simulink Editor”
“Model Editing Environment”

Introduced in R2016a

slmdliscui

Open Model Discretizer GUI

Syntax

```
slmdliscui  
slmdliscui('model')
```

Description

`slmdliscui` opens the Model Discretizer. A model does not need to be open.

`slmdliscui('model')` opens the Model Discretizer for the model or library called *'name'*.

To use the Model Discretizer, you must have a Control System Toolbox license, version 5.2 or later.

Examples

Open the Model Discretizer for the `slexAircraftExample` model:

```
slmdliscui('slexAircraftExample')
```

Open the Model Discretizer for the discretizing library:

```
slmdliscui('discretizing')
```

See Also

`sldiscmdl`

Topics

“Discretize a Model with the Model Discretizer”

Introduced before R2006a

slprofreport

Regenerate profiler report from data, `ProfileData`, saved from previous run

Syntax

```
slprofreport(model_nameProfileData)
```

Description

When you run a model with the profiler enabled, the simulation generates the data and saves it in the variable, `model_nameProfileData`. `slprofreport(model_nameProfileData)` generates a profiler report based on the data in `model_nameProfileData`, saved from the model run.

Input Arguments

ProfileData

Variable that contains profiler data from a model run. The variable name consists of the model name and `ProfileData`, for example, `vdpProfileData`.

Default: None

Examples

Regenerate Simulink Profiler Results

Regenerate the Profiler report for model `vdp`

In the MATLAB Command Window, start the `vdp` model.

In the Simulink editor window, run `vdp` model with Simulink Profiler enabled.

Simulink stores the data to the variable `vdpProfileData`.

To review the report, in the MATLAB Command Window

```
slprofreport(vdpProfileData)
```

The Simulink Profiler Report window is displayed.

- “Save Profiler Results”

See Also

Topics

“Save Profiler Results”

“How Profiler Captures Performance Data”

Introduced in R2012a

slproject.create

Create blank Simulink project

Syntax

```
proj = slproject.create  
proj = slproject.create(path)  
proj = slproject.create(name)
```

Description

`proj = slproject.create` creates and opens a Simulink project using the blank project template from the start page, and returns a project object. Use the project object to manipulate the currently open Simulink project at the command line. The new project is created in the default folder. To change the default folder for new Simulink projects, on the Simulink Project tab, click **Preferences**, and then set the **Default folder**.

`proj = slproject.create(path)` creates the project at the location specified by `path`.

`proj = slproject.create(name)` creates the project in the default folder, with the name specified by `name`.

Examples

Create a Blank Project in the Default Folder

```
slproject.create
```

You can control the default folder for new projects using the project preferences.

Create a Blank Project in a Specified Folder

```
proj = slproject.create('C:\work\myprojectname');
```

Create a Named Blank Project in the Default Folder

```
proj = slproject.create('myprojectname');
```

- “Creating Simulink Projects Programmatically”
- “Automate Simulink Project Tasks Using Scripts”

Input Arguments

path — Path for the new project location

character vector

Path for the new project location, specified as a character vector. If you do not specify the path, `slproject.create` creates the project in the default location. You can change the default location in the project preferences.

Example: `C:\work\projectname`

Data Types: `char`

name — Name for the new project

character vector

Name for the new project, specified as a character vector.

Example: `myproject`

Data Types: `char`

Output Arguments

proj — Project

project object

Project, returned as a project object. Use the project object to manipulate the currently open Simulink project at the command line.

Properties of `proj` output argument.

Project Property	Description
Name	Project name
Information	Information about the project such as the description, source control integration, repository location, and whether it is a top-level project.
Dependencies	Dependencies between project files in a MATLAB digraph object.
Shortcuts	Shortcut files in the project.
ProjectPath	Folders that the project puts on the MATLAB path.
ProjectReferences	Folders that contain referenced projects. Contains read-only project objects for referenced projects.
Categories	Categories of project labels.
Files	Paths and names of project files.
RootFolder	Full path to project root folder.

See Also

`Simulink.createFromTemplate` | `addFile` | `addFolderIncludingChildFiles` | `addPath` | `addReference` | `addShortcut` | `simulinkproject`

Topics

“Creating Simulink Projects Programmatically”
“Automate Simulink Project Tasks Using Scripts”

Introduced in R2017a

addPath

Add folder to path of Simulink project

Syntax

```
folderpath = addPath(project, folder)
```

Description

`folderpath = addPath(project, folder)` adds a folder in a Simulink project to the current project path. The folder must be in the project. The project puts the folders on the MATLAB search path when it loads and removes them from the path when it closes. To learn more, see “Specify Project Path”.

Examples

Add a Folder to the Project Path

```
sldemo_slproject_airframe;  
project = simulinkproject;
```

Create a new folder.

```
folderpath = fullfile(project.RootFolder, 'folder');  
mkdir(folderpath);
```

Add this new folder to the project.

```
projectFile = addFile(project, folderpath);
```

Add this new folder to the project path.

```
folderpath = addPath(project, folderpath);
```

- “Specify Project Path”

Input Arguments

project — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink project at the command line.

folder — Path of folder

character vector

Path of the folder to add relative to the project root folder, specified as a character vector. The folder must be within the root folder.

Example: `models/myfolder`

Output Arguments

folderpath — Path folder

path folder object

Path folder object containing the specified folder path. The project puts the folders on the MATLAB search path when it loads and removes them from the path when it closes.

See Also

`addFile` | `addFolderIncludingChildFiles` | `removePath` | `simulinkproject`

Topics

“Specify Project Path”

Introduced in R2017a

removePath

Remove folder from Simulink project path

Syntax

```
removePath(project, folder)
```

Description

`removePath(project, folder)` removes a folder in a Simulink project from the current project path. The folder must be in the project.

Examples

Remove a Folder from the Project Path

```
sldemo_slproject_airframe;  
project = simulinkproject;
```

Create a new folder.

```
folderpath = fullfile(project.RootFolder, 'folder');  
mkdir(folderpath);
```

Add this new folder to the project.

```
projectFile = addFile(project, folderpath);
```

Add the new folder to the project path.

```
folderpath = addPath(project, folderpath);
```

Remove the new folder from the project path.

```
removePath(project, folderpath)
```

Input Arguments

project — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink project at the command line.

folder — Path of folder

character vector

Path of the folder to remove relative to the project root folder, specified as a character vector. The folder must be within the root folder.

Example: `models/myfolder`

See Also

`addPath` | `simulinkproject`

Introduced in R2017a

addReference

Add referenced project to Simulink project

Syntax

```
projreference = addReference(project, folder)
projreference = addReference(project, folder, type)
```

Description

`projreference = addReference(project, folder)` adds a reference to the Simulink project specified by `folder`. The reference is added to the current project, `project`.

`projreference = addReference(project, folder, type)` specifies the type of reference to create. Specify relative or absolute reference.

Examples

Add a Referenced Project

Create a project and get a project object.

```
sldemo_slproject_airframe;
project = simulinkproject;
```

Create a new blank project.

```
projectToReference = slproject.create();
```

Reload the first project and add a reference to the new blank project.

```
reload(project);
addReference(project, projectToReference, 'absolute');
```

Find out if a project is a top-level project. 1 indicates a top-level project.

```
project.Information.TopLevel  
  
ans =  
  
    logical  
  
    1
```

Input Arguments

project — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink project at the command line.

folder — Path of folder

character vector

Path of the folder to add the reference, relative to the project root folder, specified as a character vector. The folder must be within the root folder.

Example: `models/myfolder`

type — Type of reference

relative | absolute

Type of reference, relative to the project root folder, specified as a character vector.

Output Arguments

projreference — Project reference

project reference object

Project reference object containing information about the referenced project.

See Also

`removeReference` | `simulinkproject`

Introduced in R2017a

removeReference

Add folder to Simulink project path

Syntax

```
removeReference(project, folder)
```

Description

`removeReference(project, folder)` removes the reference to the Simulink project `project` from the current project.

Examples

Remove a Referenced Project

Create a project and get a project object.

```
sldemo_slproject_airframe;  
project = simulinkproject;
```

Create a new blank project.

```
projectToReference = slproject.create();
```

Reload the first project and add a reference to the new blank project.

```
project.reload();  
addReference(project, projectToReference, 'absolute');
```

Remove the reference to the blank project.

```
removeReference(project, projectToReference);
```

Input Arguments

project — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink project at the command line.

folder — Path of folder

character vector

Path of the folder to the reference to be removed, relative to the project root folder, specified as a character vector. The folder must be within the root folder.

Example: `models/myfolder`

See Also

`addReference` | `simulinkproject`

Introduced in R2017a

addShortcut

Add shortcut to Simulink project

Syntax

```
shortcut = addShortcut(project, file)
```

Description

`shortcut = addShortcut(project, file)` adds a shortcut to the specified file in the Simulink project.

To set the shortcut to run at startup or shutdown, use Simulink Project. See “Automate Startup Tasks”.

Examples

Add a Shortcut

Create a project and get a project object.

```
sldemo_slproject_airframe;  
project = simulinkproject;
```

Create a new file.

```
filepath = fullfile(project.RootFolder, 'new_model.slx')  
    new_system('new_model');  
    save_system('new_model', filepath)
```

Add this new model to the project.

```
projectFile = addFile(project, filepath)
```

Add a new shortcut to the new model.

```
shortcut = addShortcut(project,filepath);
```

- “Automate Startup Tasks”

Input Arguments

project — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink project at the command line.

file — Path of file

character vector

Path of the file to add a shortcut to, relative to the project root folder, including the file extension, specified as a character vector. The file must be within the root folder.

Example: ‘models/myModelName.slx’

Output Arguments

shortcut — Shortcut

shortcut object

Shortcut object containing information about the shortcut.

See Also

`removeShortcut` | `simulinkproject`

Topics

“Automate Startup Tasks”

Introduced in R2017a

removeShortcut

Remove shortcut from Simulink project

Syntax

```
removeShortcut(project, file)
```

Description

`removeShortcut(project, file)` removes the shortcut to the specified file in the Simulink project.

Examples

Remove a Shortcut

Create a project and get a project object.

```
sldemo_slproject_airframe;  
project = simulinkproject;
```

Create a new file.

```
filepath = fullfile(project.RootFolder, 'new_model.slx')  
    new_system('new_model');  
    save_system('new_model', filepath)
```

Add this new model to the project.

```
projectFile = addFile(project, filepath)
```

Add a new shortcut to the new model.

```
shortcut = addShortcut(project, filepath);
```

Remove the shortcut.

```
removeShortcut(project, shortcut);
```

Input Arguments

project — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink project at the command line.

file — Path of file

character vector

Path of the shortcut file, relative to the project root folder, including the file extension, specified as a character vector. The file must be within the root folder.

Example: 'models/myModelName.slx'

See Also

`addShortcut` | `simulinkproject`

Introduced in R2017a

slproject.getCurrentProject

Manipulate current Simulink Project at command line

Syntax

```
proj = slproject.getCurrentProject
```

Description

`proj = slproject.getCurrentProject` gets the current project open in the Simulink Project Tool and returns a project object `proj` that you can use to manipulate the project programmatically. If no project is open, then you see an error.

Note `slproject.getCurrentProject` will be removed in a future release. Use `slproject.getCurrentProjects` instead.

Examples

Get Airframe Example Project

Open the Airframe project and use `slproject.getCurrentProject` to get a project object to manipulate the project at the command line.

```
sldemo_slproject_airframe
proj = slproject.getCurrentProject
```

```
proj =
```

```
    ProjectManager with properties:
```

```
    Name: 'Simulink Project Airframe Example'
    Categories: [1x1 slproject.Category]
    Shortcuts: [1x8 slproject.Shortcut]
```

```
ProjectPath: [1x7 slproject.PathFolder]
ProjectReferences: [1x0 slproject.ProjectReference]
Files: [1x30 slproject.ProjectFile]
RootFolder: 'C:\Work\Simulink\Projects\slexamples\airframe'
```

Output Arguments

proj — Project

project object

Project, returned as a project object. Use the project object to manipulate the currently open Simulink Project at the command line.

See Also

Functions

`simulinkproject` | `slproject.getCurrentProjects` | `slproject.loadProject`

Introduced in R2013a

slproject.getCurrentProjects

List all top-level Simulink projects

Syntax

```
projects = slproject.getCurrentProjects
```

Description

`projects = slproject.getCurrentProjects` returns a list of all top-level projects open in Simulink Project. Currently only one or zero top-level projects can be loaded. Returns an object array of 1 or 0 `ProjectManager` objects `projects` that you can use to manipulate the project programmatically. Use `slproject.getCurrentProjects` for project automation scripts.

If you execute `slproject.getCurrentProjects` inside a project shortcut, it returns only the project that the shortcut belongs to. If the shortcut belongs to a referenced project, it returns the referenced project.

Examples

Get Airframe Example Project

Open the Airframe project and use `slproject.getCurrentProjects` to get a project object to manipulate the project at the command line.

```
sldemo_slproject_airframe  
proj = slproject.getCurrentProjects  
  
proj =
```

```
    ProjectManager with properties:
```

```
        Name: 'Simulink Project Airframe Example'
```

```

Categories: [1x1 slproject.Category]
Shortcuts: [1x8 slproject.Shortcut]
ProjectPath: [1x7 slproject.PathFolder]
ProjectReferences: [1x0 slproject.ProjectReference]
Files: [1x30 slproject.ProjectFile]
RootFolder: 'C:\Work\Simulink\Projects\airframe'

```

Find Project Commands

Open the airframe project and create a project object.

```

sldemo_slproject_airframe
proj = slproject.getCurrentProject

```

```

proj =

```

```

ProjectManager with properties:

```

```

Name: 'Simulink Project Airframe Example'
Categories: [1x1 slproject.Category]
Shortcuts: [1x8 slproject.Shortcut]
ProjectPath: [1x7 slproject.PathFolder]
ProjectReferences: [1x0 slproject.ProjectReference]
Files: [1x30 slproject.ProjectFile]
RootFolder: 'C:\Work\Simulink\Projects\airframe'

```

Find out what you can do with your project.

```

methods(proj)

```

```

Methods for class slproject.ProjectManager:

```

```

addFile                findCategory
addFolderIncludingChildFiles findFile
close                  isLoading
createCategory         listModifiedFiles
export                 refreshSourceControl

```

```

reload
removeCategory
removeFile

```

Examine Project Properties

After you get a project object, you can examine project properties.

Open the airframe project and create a project object.

```
sldemo_slproject_airframe
proj = slproject.getCurrentProjects;
```

Examine the project files.

```
files = proj.Files
files =

    1x30 ProjectFile array with properties:

        Path
        Labels
        Revision
        SourceControlStatus
```

Examine the labels of the eighth file.

```
proj.Files(8).Labels
ans =

    Label with properties:

    File: 'C:\Work\airframe\data\system_model.sldd'
        Data: []
        DataType: 'none'
        Name: 'Design'
        CategoryName: 'Classification'
```

Get a particular file.

```
myfile = findFile(proj, 'models/AnalogControl.mdl')
myfile =

    ProjectFile with properties:

    Path: 'C:\Temp\airframe\models\AnalogControl.mdl'
    Labels: [1x1 slproject.Label]
```

```
Revision: '2'
SourceControlStatus: Unmodified
```

Find out what you can do with the file.

```
methods(myfile)
```

```
Methods for class slproject.ProjectFile:
```

```
addLabel
removeLabel
findLabel
```

Output Arguments

projects — Projects

object array of 1 or 0 `ProjectManager` objects

Projects, returned as an object array of 1 or 0 `ProjectManager` objects. Use the project object to manipulate the currently open Simulink Project at the command line.

Properties of `ProjectManager` objects in output argument.

Project Property	Description
Name	Project name
Categories	Categories of project labels
Shortcuts	Shortcut files in project
ProjectPath	Folders that the project puts on the MATLAB path
ProjectReferences	Folders that contain referenced projects
Files	Paths and names of project files
RootFolder	Full path to project root folder

Tips

Alternatively, you can use `simulinkproject` to get a project object, but `simulinkproject` also opens and gives focus to the Simulink Project Tool. Use

`simulinkproject` to open projects and explore projects interactively. Use `slproject.getCurrentProjects` for project automation scripts.

See Also

Functions

`simulinkproject` | `slproject.getCurrentProject` | `slproject.loadProject`

Introduced in R2016a

slproject.loadProject

Load Simulink project

Syntax

```
slproject.loadProject(projectPath);  
proj = slproject.loadProject(projectPath)
```

Description

`slproject.loadProject(projectPath)` loads the project specified by the `.prj` file or folder `projectPath` in the Simulink Project Tool, and closes any currently open project.

`proj = slproject.loadProject(projectPath)` loads the project and returns a project object `proj` for manipulating the project. Use `slproject.loadProject` for project automation scripts.

Examples

Load Project

Load a project from a folder called `'C:/projects/project1/'`. Replace this path with the location of your project.

```
proj = slproject.loadProject('C:/projects/project1/')
```

Get Airframe Example Project

Open the Airframe project and use `slproject.getCurrentProjects` to get a project object to manipulate the project at the command line.

```
sldemo_slproject_airframe
proj = slproject.getCurrentProjects

proj =

    ProjectManager with properties:

        Name: 'Simulink Project Airframe Example'
        Categories: [1x1 slproject.Category]
        Shortcuts: [1x8 slproject.Shortcut]
        ProjectPath: [1x7 slproject.PathFolder]
        ProjectReferences: [1x0 slproject.ProjectReference]
        Files: [1x30 slproject.ProjectFile]
        RootFolder: 'C:\Work\Simulink\Projects\airframe'
```

Find Project Commands

Get the Airframe project.

```
sldemo_slproject_airframe
proj = slproject.getCurrentProjects;
```

Find project commands.

```
methods(proj)
```

```
Methods for class slproject.ProjectManager:
```

```
addFile                findCategory
addFolderIncludingChildFiles  findFile
close                  isLoaded
createCategory         listModifiedFiles
export                 refreshSourceControl
```

```
reload
removeCategory
removeFile
```

Examine Project Properties

After you get a project object, you can examine project properties.

Get the airframe project.

```
sldemo_slproject_airframe
proj = slproject.getCurrentProjects;
```

Examine the project files.

```
files = proj.Files
files =
    1x30 ProjectFile array with properties:
        Path
        Labels
        Revision
        SourceControlStatus
```

Examine the labels of the 13th file.

```
proj.Files(13).Labels
ans =
    Label with properties:
        File: 'C:\Temp\airframe\models\AnalogControl.mdl'
           Data: []
        DataType: 'none'
           Name: 'Design'
        CategoryName: 'Classification'
```

Get a particular file by name.

```
myfile = findFile(proj, 'models/AnalogControl.mdl')
myfile =
    ProjectFile with properties:
        Path: 'C:\Temp\airframe\models\AnalogControl.mdl'
        Labels: [1x1 slproject.Label]
        Revision: '2'
        SourceControlStatus: Unmodified
```

Find out what you can do with the file.

```
methods(myfile)

Methods for class slproject.ProjectFile:

addLabel
removeLabel
findLabel
```

Input Arguments

projectPath — Full path to project file or folder
character vector

Full path to project .prj file or the path to the project root folder, specified as a character vector.

Example: 'C:/projects/project1/myProject.prj'

Example: 'C:/projects/project1/'

Output Arguments

proj — Project
project object

Project, returned as a project object. Use the project object to manipulate and explore the Simulink Project at the command line.

Properties of `proj` output argument.

Project Property	Description
Name	Project name
Categories	Categories of project labels
Shortcuts	Shortcut files in project
ProjectPath	Folders that the project puts on the MATLAB path
ProjectReferences	Folders that contain referenced projects
Files	Paths and names of project files

Project Property	Description
RootFolder	Full path to project root folder

See Also

Functions

`simulinkproject` | `slproject.getCurrentProjects`

Topics

“What Are Simulink Projects?”

Introduced in R2013a

listModifiedFiles

List modified files in Simulink project

Syntax

```
modifiedfiles = listModifiedFiles(proj)
```

Description

`modifiedfiles = listModifiedFiles(proj)` returns the list of modified project files in the project object `proj`. `listModifiedFiles` refreshes the source control statuses in the project and then returns an array of the project files which are listed in the Modified Files view of the Simulink Project.

Examples

Get a List of Modified Files in the Project

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

Modify a project model file by adding an arbitrary block.

```
open_system('AnalogControl')  
add_block('built-in/SubSystem', 'AnalogControl/test')  
save_system('AnalogControl')
```

Get all the modified files in the project.

```
modifiedfiles = listModifiedFiles(proj)  
  
modifiedfiles =
```

```
1x2 ProjectFile array with properties:
```

```
Path
Labels
Revision
SourceControlStatus
```

Observe two modified files. Compare with the Modified Files view in Simulink Project, where you can see a modified model file, and the corresponding `.SimulinkProject` definition file.

Get the second modified file.

```
modifiedfiles(2)
```

```
ans =
```

```
ProjectFile with properties:
```

```
Path: 'C:\Work\temp\slexamples\airframe2\models\AnalogControl.mdl'
      Labels: [1x1 slproject.Label]
      Revision: '2'
      SourceControlStatus: Modified
```

Observe the file `SourceControlStatus` property is `Modified`. Similarly, `listModifiedFiles` returns any files that are added, conflicted, deleted, etc., that show up in the Modified Files view in Simulink Project.

Get all the project files with a particular source control status. For example, get the files that are `Unmodified`.

```
proj.Files(ismember([proj.Files.SourceControlStatus], matlab.sourcecontrol.Status.Unmod
```

```
ans =
```

```
1x29 ProjectFile array with properties:
```

```
Path
Labels
```

Revision
SourceControlStatus

Input Arguments

proj — Project

project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

Output Arguments

modifiedfiles — Modified files

file object | array

Modified files, returned as an array of file objects.

See Also

Functions

`refreshSourceControl` | `simulinkproject`

Introduced in R2016a

listRequiredFiles

Get project file dependencies

Syntax

```
files = listRequiredFiles(proj, file)
```

Description

`files = listRequiredFiles(proj, file)` returns the files that the specified file requires to run.

Examples

Get Required Files

Open the airframe project, create a project object and get a file.

```
sldemo_slproject_airframe;  
proj = simulinkproject;  
file = 'models/slproject_f14.slx'
```

Get the files required by the specified file.

```
files = listRequiredFiles(project, file);
```

- “Perform Impact Analysis”

Input Arguments

proj — Project
project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

file — Path of file

character vector | project file object

Path of the file to add relative to the project root folder, including the file extension, specified as a character vector, an absolute file path or an instance of a project file object. The file must be within the root folder.

Example: `'models/myModelName.slx'`

Output Arguments

file — Project file

cell array of character vectors

Required files, returned as a cell array of character vectors.

See Also

`simulinkproject`

Topics

“Perform Impact Analysis”

Introduced in R2017a

refreshSourceControl

Update source control status of Simulink project files

Syntax

```
refreshSourceControl(proj)
```

Description

`refreshSourceControl(proj)` updates the source control status for all files in the Simulink project `proj`. Use this to get the latest source control information before querying the `SourceControlStatus` property on individual files.

If you use `listModifiedFiles` to find all modified files in the project, you do not need to call `refreshSourceControl` first.

Examples

Refresh Source Control Information on Files in the Project

Open the airframe project and create a project object.

```
sldemo_slproject_airframe;  
proj = simulinkproject;
```

Refresh source control status before querying individual files.

```
refreshSourceControl(proj)
```

Input Arguments

proj — Project
project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a Simulink Project at the command line.

See Also

Functions

`listModifiedFiles` | `simulinkproject`

Introduced in R2016a

solverprofiler.profileModel

Examine model for performance analysis

Syntax

```
res = solverprofiler.profileModel(model)
res = solverprofiler.profileModel(model, Name, Value)
```

Description

`res = solverprofiler.profileModel(model)` runs the Solver Profiler on the specified model and stores the results in `res`

`res = solverprofiler.profileModel(model, Name, Value)` specifies the Solver Profiler parameters using one or more `Name, Value` pair arguments.

Examples

Examine a Model with Default Settings

Examine the model `f14` using the default commandline settings.

```
model = 'f14';
res = solverprofiler.profileModel(model);
```

You can see a summary of the results by calling `res.summary`.

```
res.summary

struct with fields:

    solver: 'ode45'
    tStart: 0
    tStop: 60
```

```
absTol: 1.0000e-06
relTol: 1.0000e-04
hMax: 0.1000
hAverage: 0.0444
steps: 1352
profileTime: 0.9974
zcNumber: 0
resetNumber: 600
jacobianNumber: 0
exceptionNumber: 195
```

Open the results in the Solver Profiler dialog to visualize them. This step is equivalent to enabling OpenSP when calling the function.

```
solverprofiler.exploreResult(res)
```

Configure Solver Profiler and Examine a Model

Examine the model `ssc_actuator_custom_pneumatic` with a fully specified configuration.

```
model = 'ssc_actuator_custom_pneumatic';
res = solverprofiler.profileModel(model, ...
    'SaveStates' , 'on', ...
    'SaveSimscapeStates' , 'On', ...
    'SaveJacobian' , 'On', ...
    'StartTime' , 5, ...
    'StopTime' , 50, ...
    'BufferSize' , 10000, ...
    'TimeOut' , 5, ...
    'OpenSP' , 'On', ...
    'DataFullFile' , fullfile(pwd, 'ssc_profiling_result.mat'));
```

Input Arguments

model — Model to examine

character vector (default)

Name of model to be profiled, specified as a character vector.

Example: `h = solverprofiler.profileModel('vdp')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'StartTime', 0, 'StopTime', 10, 'SaveStates', 'On'`

SaveStates — Save model states to file

off (default) | on

By default, the profiler does not save the states of the model. Enabling this parameter configures the profiler to save the states to a MAT-file.

Example: `'SaveStates', 'On'`

SaveSimscapeStates — Save states to file

off (default) | on

Enable this parameter to save states to a MAT-file.

Example: `'SaveSimscapeStates', 'On'`

SaveJacobian — Save model Jacobian

off (default) | on

Option to log the solver Jacobian matrices to memory. This option is useful for simulations that use implicit solvers. For a comparison of solvers, see “Solver Classification Criteria”.

Example: `'SaveJacobian', 'On'`

StartTime — Profiler start time

model start time (default) | scalar

Time, in seconds, of the simulation that the profiler starts analyzing the model. This is not the same as the start time of the simulation.

Example: `'StartTime', 5`

StopTime — Profiler stop time

model stop time (default) | scalar

Time, in seconds, of the simulation to which the profiler should profile the model. By default, the analysis continues until the end of the simulation. Changing this parameter does not change the stop time of the model which you specify in the Model Configuration Parameters.

A value less than the configured stop time of the model stops the profiling and simulation at `StopTime`.

Example: `'StopTime', 30`

BufferSize — Memory impact of logging

50000 (default) | positive scalar

Maximum number of events that are logged. If the number of logged events reaches this value and memory is available, increase `BufferSize`. If memory is limited, consider lowering the value.

Example: `'BufferSize', 60000`

TimeOut — Maximum time to wait for solver to resume

positive scalar

Time, in seconds, to wait before the profiler stops running. This option is useful in situations where the simulation is unable to proceed. The profiler waits for the specified time and quits if no progress has been made.

Example: `'TimeOut', 10`

OpenSP — Open the Solver Profiler dialog box

off (default) | on

Choice to open the Solver Profiler dialog box after profiling has completed.

Example: `'OpenSP', 'On'`

DataFullFile — Path and name of saved results

character vector of full file path

By default, the profiling results are saved in a MAT-file named `model_@_dd_Month_YYYY_hh_mm_ss.mat` in the current working folder. You can specify a different file name by which to save the results in the current working folder. To save the file in a different location, specify the full path of the file, including the file name.

Example: 'DataFullFile', 'C:\Users\myusername\Documents\profiled
 \vdp_results.mat'

Output Arguments

res — High-level summary of profiling results

structure

Profiling results, returned as a structure with the fields:

file — Full path and name of saved results

character vector

Path and name of the MAT-file where the results of the profiling operation are stored as MAT file. By default, they are stored in the current working folder with a file name having the pattern: model_@_dd_Month_yyyy_hh_mm_ss.mat. To store them in a different location or by a different name, specify DataFullFile when calling solverprofiler.profileModel.

summary — Summary of profiling results

structure

A high-level summary of the results of the profiling operation, returned as a structure. The summary provides an overview of the performance of the simulation and health of the model.

The summary structure contains these fields.

Field	Purpose	Values	Description
solver	Solver used by simulation	any of the solvers supported by Solver Profiler	Solver used by the simulation as configured in the Configuration Parameters for the model. For a list of all the solvers, see “Solver”. The Solver Profiler does not support models without any continuous states.
tStart	Start time of simulation	scalar	Start time, in seconds, for the simulation of the model during the profiling operation.

Field	Purpose	Values	Description
tStop	Stop time of simulation	scalar	Stop time, in seconds, of the simulation during the profiling operation. If StopTime is set to be earlier than the configured Stop Time of the model, the simulation stops at StopTime.
absTol	Absolute tolerance of the solver	positive scalar	Absolute tolerance of the solver as specified in the configuration settings for the model. For more information, see “Absolute tolerance”
relTol	Relative tolerance of the solver	positive scalar	Relative tolerance of the solver as specified in the configuration settings of the model. For more information, see “Relative tolerance”
hMax	Maximum step size	positive scalar	Largest time step that the solver can take. See “Max step size”.
hAverage	Average step size	positive scalar	Average size of the time step taken by the solver.
steps	Total steps taken	positive scalar	Total number of time steps taken by the solver.
profileTime	Time to profile	positive scalar	Time, in seconds, taken by the Solver Profiler to examine the model.
zcNumber	Total number of zero crossings	nonnegative scalar	Number of times zero crossings occur during the simulation of the model. The detection of these zero crossings incurs computational cost and can slow down the simulation. For information on zero-crossing detection, see “Zero-Crossing Detection”.
resetNumber	Number of solver resets	nonnegative scalar	Number of times the solver has to reset its parameters.

Field	Purpose	Values	Description
jacobi anNum ber	Number of Jacobian updates	nonnegative scalar	Number of times the solver Jacobian matrix is updated during a simulation. For more information, see “Explicit Versus Implicit Continuous Solvers”.
except ionNum ber	Number of solver exceptions	nonnegative scalar	Total number of solver exceptions encountered during a simulation. These exceptions are events where the solver is unable to solve the model states to the specified accuracy. As a result, the solver runs adjusted trials which increase computational cost.

Data Types: struct

See Also

“Understand Profiling Results” | “Examine Model Dynamics Using Solver Profiler”

Topics

“About Solvers”

“Choose a Solver”

Introduced in R2017b

start_simulink

Start Simulink without opening any windows

Syntax

```
start_simulink
```

Description

`start_simulink` starts Simulink without opening any models, the Start Page, or the Simulink Library Browser. Use this in startup scripts to start Simulink without any other window taking the focus away from the MATLAB Desktop. For example, use `start_simulink` in the MATLAB `startup.m` file, when starting MATLAB with the `-r` command line option, or in Simulink project startup scripts. Opening a model for the first time in a MATLAB session is much quicker after running `start_simulink`.

If you want to open the Simulink Start Page to create or open models, use the `simulink` function instead.

If you want to open the Library Browser, use `slLibraryBrowser`.

Examples

Start Simulink When Starting MATLAB

Use the `-r` command line option to start Simulink when starting MATLAB, without opening any windows.

On Windows, create a desktop shortcut with the following target:

```
matlabroot\bin\win64\matlab.exe -r start_simulink
```

On Linux® and Mac, enter:

```
matlab -r start_simulink
```

- “Automate Startup Tasks”

See Also

[simulink](#) | [simulinkproject](#) | [slLibraryBrowser](#)

Topics

“Automate Startup Tasks”

Introduced in R2015b

slupdate

Replace blocks from previous releases with latest versions

Note `slupdate` will be removed in a future release. The `slupdate` command can only upgrade some parts of your model. Use the Upgrade Advisor instead. See “Model Upgrades”.

Syntax

```
slupdate('sys')
slupdate('sys', prompt)
AnalysisResult = slupdate('sys', 'OperatingMode', 'Analyze')
```

Description

`slupdate('sys')` replaces blocks in model `sys` from a previous release of Simulink software with the latest versions. The `slupdate` function alone cannot perform all upgrade checks on your model. Use the Upgrade Advisor to access the `slupdate` checks and also advice and fixes for all other upgrade checks. See “Model Upgrades”.

Note Best practice is to first open the model, and press CTRL+D to update the model, before you call `slupdate`.

`slupdate('sys', prompt)` specifies whether to prompt you before replacing a block. If `prompt` equals 1, the command prompts you before replacing the block. The prompt asks whether you want to replace the block. Valid responses are

- `y`

Replace the block (the default).

- `n`

Do not replace the block.

- `a`

Replace this and all subsequent obsolete blocks without further prompting.

If `prompt` equals 0, the command replaces all obsolete blocks without prompting you.

In addition to replacing obsolete blocks, `slupdate`

- Reconnects broken links to masked blocks in libraries provided by MathWorks to ensure that the model reflects changes made to the blocks in this release. This will overwrite any custom changes you made to the masks of these blocks.
- Updates obsolete configuration settings for the model.

`AnalysisResult = slupdate('sys', 'OperatingMode', 'Analyze')` performs only the analysis portion without updating or changing the model. This command analyzes referenced models, linked libraries, and S-functions, and then returns a data structure with the following fields:

- `Message` — character vector containing a message summarizing the results
- `blockList` — cell array listing blocks that need to be updated
- `blockReasons` — cell array listing reasons for updating the corresponding blocks
- `modelList` — cell array listing referenced models and the parent model
- `libraryList` — cell array listing non-MathWorks libraries referenced
- `configSetList` — for internal use
- `sfunList` — cell array listing S-functions referenced
- `sfunOK` — logical array representing S-function status, where `false` indicates that an S-function needs updating and `true` indicates otherwise
- `sfunType` — cell array listing apparent S-function type (e.g., `.mex`)

See Also

`upgradeadvisor`

Topics

“Model Upgrades”

Introduced before R2006a

trim

Find trim point of dynamic system

Syntax

```
[x,u,y,dx] = trim('sys')
[x,u,y,dx] = trim('sys',x0,u0,y0)
[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy)
[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx)
[x,u,y,dx,options] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options)
[x,u,y,dx,options] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options,t)
```

Description

A trim point, also known as an equilibrium point, is a point in the parameter space of a dynamic system at which the system is in a steady state. For example, a trim point of an aircraft is a setting of its controls that causes the aircraft to fly straight and level. Mathematically, a trim point is a point where the system's state derivatives equal zero. `trim` starts from an initial point and searches, using a sequential quadratic programming algorithm, until it finds the nearest trim point. You must supply the initial point implicitly or explicitly. If `trim` cannot find a trim point, it returns the point encountered in its search where the state derivatives are closest to zero in a min-max sense; that is, it returns the point that minimizes the maximum deviation from zero of the derivatives. `trim` can find trim points that meet specific input, output, or state conditions, and it can find points where a system is changing in a specified manner, that is, points where the system's state derivatives equal specific nonzero values.

`[x,u,y,dx] = trim('sys')` finds the equilibrium point of the model `'sys'`, nearest to the system's initial state, `x0`. Specifically, `trim` finds the equilibrium point that minimizes the maximum absolute value of $[x-x_0, u, y]$. If `trim` cannot find an equilibrium point near the system's initial state, it returns the point at which the system is nearest to equilibrium. Specifically, it returns the point that minimizes $\text{abs}(dx)$ where `dx` represents the derivative of the system. You can obtain `x0` using this command.

```
[sizes,x0,xstr] = sys([],[],[],0)
```

`[x,u,y,dx] = trim('sys',x0,u0,y0)` finds the trim point nearest to `x0`, `u0`, `y0`, that is, the point that minimizes the maximum value of

```
abs([x-x0; u-u0; y-y0])
```

`[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy)` finds the trim point closest to `x0`, `u0`, `y0` that satisfies a specified set of state, input, and/or output conditions. The integer vectors `ix`, `iu`, and `iy` select the values in `x0`, `u0`, and `y0` that must be satisfied. If `trim` cannot find an equilibrium point that satisfies the specified set of conditions exactly, it returns the nearest point that satisfies the conditions, namely,

```
abs([x(ix)-x0(ix); u(iu)-u0(iu); y(iy)-y0(iy)])
```

`[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx)` finds specific nonequilibrium points, that is, points at which the system's state derivatives have some specified nonzero value. Here, `dx0` specifies the state derivative values at the search's starting point and `idx` selects the values in `dx0` that the search must satisfy exactly.

`[x,u,y,dx,options] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options)` specifies an array of optimization parameters that `trim` passes to the optimization function that it uses to find trim points. The optimization function, in turn, uses this array to control the optimization process and to return information about the process. `trim` returns the `options` array at the end of the search process. By exposing the underlying optimization process in this way, `trim` allows you to monitor and fine-tune the search for trim points.

The following table describes how each element affects the search for a trim point. Array elements 1, 2, 3, 4, and 10 are particularly useful for finding trim points.

No.	Default	Description
1	0	Specifies display options. 0 specifies no display; 1 specifies tabular output; -1 suppresses warning messages.
2	10^{-4}	Precision the computed trim point must attain to terminate the search.
3	10^{-4}	Precision the trim search goal function must attain to terminate the search.
4	10^{-6}	Precision the state derivatives must attain to terminate the search.

No.	Default	Description
5	N/A	Not used.
6	N/A	Not used.
7	N/A	Used internally.
8	N/A	Returns the value of the trim search goal function (λ in goal attainment).
9	N/A	Not used.
10	N/A	Returns the number of iterations used to find a trim point.
11	N/A	Returns the number of function gradient evaluations.
12	0	Not used.
13	0	Number of equality constraints.
14	100*(Number of variables)	Maximum number of function evaluations to use to find a trim point.
15	N/A	Not used.
16	10^{-8}	Used internally.
17	0.1	Used internally.
18	N/A	Returns the step length.

`[x,u,y,dx,options] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options,t)`
sets the time to t if the system is dependent on time.

Note If you fix any of the state, input or output values, `trim` uses the unspecified free variables to derive the solution that satisfies these constraints.

Examples

Consider a linear state-space system modeled using a State-Space block

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

The A , B , C , and D matrices to enter at the command line or in the block parameters dialog are:

```
A = [-0.09 -0.01; 1 0];  
B = [ 0 -7; 0 -2];  
C = [ 0 2; 1 -5];  
D = [-3 0; 1 0];
```

Example 1

To find an equilibrium point in this model called `sys`, use:

```
[x,u,y,dx,options] = trim('sys')  
x =  
    0  
    0  
u =  
    0  
    0  
y =  
    0  
    0  
dx =  
    0  
    0
```

The number of iterations taken is:

```
options(10)  
ans =  
     7
```

Example 2

To find an equilibrium point near $x = [1;1]$, $u = [1;1]$, enter

```
x0 = [1;1];  
u0 = [1;1];  
[x,u,y,dx,options] = trim('sys', x0, u0);  
x =  
    1.0e-13 *  
   -0.5160  
   -0.5169  
u =  
    0.3333  
    0.0000
```

```

y =
  -1.0000
   0.3333
dx =
  1.0e-12 *
   0.1979
   0.0035

```

The number of iterations taken is

```

options(10)
ans =
    25

```

Example 3

To find an equilibrium point with the outputs fixed to 1, use:

```

y = [1;1];
iy = [1;2];
[x,u,y,dx] = trim('sys', [], [], y, [], [], iy)
x =
    0.0009
   -0.3075
u =
   -0.5383
    0.0004
y =
    1.0000
    1.0000
dx =
    1.0e-15 *
   -0.0170
    0.1483

```

Example 4

To find an equilibrium point with the outputs fixed to 1 and the derivatives set to 0 and 1, use

```

y = [1;1];
iy = [1;2];
dx = [0;1];

```

```
idx = [1;2];
[x,u,y,dx,options] = trim('sys',[],[],y,[],[],iy,dx,idx)
x =
    0.9752
   -0.0827
u =
   -0.3884
   -0.0124
y =
    1.0000
    1.0000
dx =
    0.0000
    1.0000
```

The number of iterations taken is

```
options(10)
ans =
    13
```

Limitations

The trim point found by `trim` starting from any given initial point is only a local value. Other, more suitable trim points may exist. Thus, if you want to find the most suitable trim point for a particular application, it is important to try a number of initial guesses for `x`, `u`, and `y`.

Algorithms

`trim` uses a sequential quadratic programming algorithm to find trim points. See “Sequential Quadratic Programming (SQP)” (Optimization Toolbox) for a description of this algorithm.

Introduced before R2006a

tunablevars2parameterobjects

Create Simulink parameter objects from tunable parameters

Syntax

```
tunablevars2parameterobjects ('modelName')  
tunablevars2parameterobjects ('modelName', class)
```

Description

`tunablevars2parameterobjects ('modelName')` creates `Simulink.Parameter` objects in the base workspace for the variables listed in the specified model's Tunable Parameters dialog, then deletes the source information from the dialog. To preserve the information, save the resulting Simulink parameter objects into a MAT-file.

If a tunable variable is already defined as a numeric variable in the base workspace, the variable will be replaced by a parameter object and the original variable will be copied to the object's Value property.

If a tunable variable is already defined as a Simulink parameter object, the object will not be modified but the information for the variable will still be deleted from the Tunable Parameters dialog.

If a tunable variable is defined as any other class of variable, the variable will not be modified and the information for the variable will not be deleted from the Tunable Parameters dialog.

`tunablevars2parameterobjects ('modelName', class)` creates objects of the specified class rather than `Simulink.Parameter` objects.

Input Arguments

modelName

Model name or handle

class

Parameter class to use for creating objects

Default: `Simulink.Parameter`

See Also

`Simulink.Parameter`

Topics

“Tunable Parameters”

Introduced in R2007b

ufix

Create `Simulink.NumericType` object describing unsigned fixed-point data type

Syntax

```
a = ufix(WordLength)
```

Description

`ufix(WordLength)` returns a `Simulink.NumericType` object that describes an unsigned fixed-point data type with the specified word length and unspecified scaling.

Note `ufix` is a legacy function. In new code, use `fixdt` instead. In existing code, replace `ufix(WordLength)` with `fixdt(0,WordLength)`.

Examples

Define a 16-bit unsigned fixed-point data type.

```
a = ufix(16)
```

```
a =
```

```
NumericType with properties:
```

```
    DataTypeMode: 'Fixed-point: unspecified scaling'  
    Signedness: 'Unsigned'  
    WordLength: 16  
    IsAlias: 0  
    DataScope: 'Auto'  
    HeaderFile: ''  
    Description: ''
```

See Also

`Simulink.NumericType` | `fixdt` | `float` | `sfix` | `sfrac` | `sint` | `ufrac` | `uint`

Introduced before R2006a

ufrac

Create `Simulink.NumericType` object describing unsigned fractional data type

Syntax

```
a = ufrac(WordLength)
a = ufrac(WordLength, GuardBits)
```

Description

`ufrac(WordLength)` returns a `Simulink.NumericType` object that describes the data type of an unsigned fractional data type with a word size given by `WordLength`.

`ufrac(WordLength, GuardBits)` returns a `Simulink.NumericType` object that describes the data type of an unsigned fractional data type. The total word size is given by `WordLength` with `GuardBits` bits located to the left of the binary point.

Note `ufrac` is a legacy function. In new coder, use `fixdt` instead. In existing code, replace `ufrac(WordLength)` with `fixdt(0,WordLength,WordLength)` and `ufrac(WordLength,GuardBits)` with `fixdt(0,WordLength,(WordLength-GuardBits))`.

Examples

Define an 8-bit unsigned fractional data type with 4 guard bits. Note that the range of this data type is from 0 to $(1 - 2^{-8}) \cdot 2^4 = 15.9375$.

```
a = ufrac(8,4)
```

```
a =
```

```
    NumericType with properties:
```

```
    DataTypeMode: 'Fixed-point: binary point scaling'
```

```
Signedness: 'Unsigned'  
WordLength: 8  
FractionLength: 4  
  IsAlias: 0  
  DataScope: 'Auto'  
  HeaderFile: ''  
  Description: ''
```

See Also

`Simulink.NumericType` | `fixdt` | `float` | `sfix` | `sfrac` | `sint` | `ufix` | `uint`

Introduced before R2006a

uint

Create `Simulink.NumericType` object describing unsigned integer data type

Syntax

```
a = uint(WordLength)
```

Description

`uint(WordLength)` returns a `Simulink.NumericType` object that describes the data type of an unsigned integer with a word size given by `WordLength`.

Note `uint` is a legacy function. In new code, use `fixdt` instead. In existing code, replace `uint(WordLength)` with `fixdt(0,WordLength,0)`.

Examples

Define a 16-bit unsigned integer.

```
a = uint(16)
```

```
a =
```

```
NumericType with properties:
```

```
    DataTypeMode: 'Fixed-point: binary point scaling'  
    Signedness: 'Unsigned'  
    WordLength: 16  
    FractionLength: 0  
    IsAlias: 0  
    DataScope: 'Auto'  
    HeaderFile: ''  
    Description: ''
```

See Also

`Simulink.NumericType` | `fixdt` | `float` | `sfix` | `sfrac` | `sint` | `ufix` | `ufrac`

Introduced before R2006a

unpack

Extract signal logging objects from signal logs and write them into MATLAB workspace

Syntax

```
log.unpack  
tsarray.unpack  
log.unpack('systems')  
log.unpack('all')
```

Description

Note The `ModelDataLogs` class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy `ModelDataLogs` API”.

`log.unpack` or `unpack(log)` extracts the top level elements of the `Simulink.ModelDataLogs` or `Simulink.SubsysDataLogs` object named **log** (e.g., `log.out`).

`log.unpack('systems')` or `unpack(log, 'systems')` extracts `Simulink.Timeseries` and `Simulink.TsArray` objects from the `Simulink.ModelDataLogs` or `Simulink.SubsysDataLogs` object named `log`. This

`command` does not extract `Simulink.Timeseries` objects from `Simulink.TsArray` objects nor does it write intermediate `Simulink.ModelDataLogs` or `Simulink.SubsysDataLogs` objects to the MATLAB workspace.

`log.unpack('all')` or `unpack(log, 'all')` extracts all the `Simulink.Timeseries` objects contained by the `Simulink.ModelDataLogs`, `Simulink.TsArray`, or `Simulink.SubsysDataLogs` object named `log`.

`tsarray.unpack` extracts the time-series objects of class `Simulink.Timeseries` from the `Simulink.TsArray` object named `tsarray`.

See Also

`Simulink.ModelDataLogs` | `Simulink.SubsysDataLogs` | `Simulink.Timeseries`
| `Simulink.TsArray` | `who` | `whos`

Topics

“Export Signal Data Using Signal Logging”

Introduced before R2006a

upgradeadvisor

Open Upgrade Advisor

Syntax

```
upgradeadvisor('modelname')  
upgrader = upgradeadvisor('modelname')
```

Description

`upgradeadvisor('modelname')` opens the Upgrade Advisor for the model specified by `modelname`. This command loads the model if necessary, but does not open it in the Simulink Editor. Use the Upgrade Advisor to help you upgrade and improve models with the current release.

`upgrader = upgradeadvisor('modelname')` returns an object that you can use to analyze and upgrade a hierarchy of models programmatically. If you specify an output, then the Upgrade Advisor does not open. You can use the methods `analyze` and `upgrade` with the `upgrader` object output of the `upgradeadvisor` function.

- To programmatically analyze a model for recommended upgrades, create an `upgrader` object and use the method `analyze`.
- To programmatically analyze and upgrade a model, create an `upgrader` object and use the method `upgrade`.
- To configure options before running `analyze` or `upgrade`, see “Examples” on page 2-948.

Tip For an example showing how to programmatically upgrade a whole project, see “Upgrade Simulink Models Using a Simulink Project”.

Input Arguments

modelName

Name or handle to the model, specified as a character vector.

Output Arguments

upgrader

Object for analyzing and upgrading the hierarchy of models programmatically.

Examples

Open Upgrade Advisor on a Model

To open the Upgrade Advisor on the vdp example model:

```
upgradeadvisor('vdp')
```

To open the Upgrade Advisor on the currently selected model:

```
upgradeadvisor(bdroot)
```

Programmatically Analyze and Upgrade a Model

- 1 Get an upgrader object. This example uses a writable copy of the vdp model.

```
load_system('vdp'); save_system('vdp',fullfile(tempdir, 'myvdp'))
upgrader = upgradeadvisor('myvdp')
```

```
upgrader =
```

```
Upgrader with properties:
```

```
ChecksToSkip: {}
SkipLibraries: 0
SkipBlocksets: 1
OneLevelOnly: 0
```

```
ShowReport: 1
RootModel: 'myvdp'
ReportFile: ''
```

- 2 To analyze the model for recommended upgrades, following library links and model references, run:

```
analyze(upgrader);
```

You see a report of issues found.

- 3 To analyze the model and automatically fix all issues (where automated fixes are available), run:

```
upgrade(upgrader);
```

This command follows library links and model references, and saves any fixes to the model files.

You see a report of issues found and actions taken.

- 4 To find the location of the report:

```
reportLocation = upgrader.ReportFile
```

- 5 You can configure options before running analyze or upgrade.

- Specify checks to skip before running analyze or upgrade. Find the ID for a check in the Upgrade Advisor by right-clicking the check and selecting **Send Check ID to Workspace**. Then set the `advisor.ChecksToSkip` property. For example:

```
advisor.ChecksToSkip = {'mathworks.design.CSStoVSSConvert'};
upgrade(upgrader);
```

- Specify running the Upgrade Advisor on only the current model without following library links or model references:

```
upgrader.OneLevelOnly = true; % default false
```

- Specify running the Upgrade Advisor on the current model, following model references but not library links:

```
upgrader.SkipLibraries = true; % default false
```

- Specify running the Upgrade Advisor on the current model, including upgrading files in blocksets or toolboxes:

```
upgrader.SkipBlocksets = false; % default true
```

By default, the Upgrade Advisor does not upgrade files in blocksets or toolboxes. The Upgrade Advisor detects blocksets from the output of `ver` and the existence of a `Contents.m` file.

- To turn off showing the report after analyze or upgrade, set:

```
upgrader.ShowReport = false; % default true
```

Tip For an example showing how to programmatically upgrade a whole project, see “Upgrade Simulink Models Using a Simulink Project”.

Tips

- The Upgrade Advisor can identify cases where you can benefit by changing your model to use new features and settings in Simulink. The Advisor provides advice for transitioning to new technologies, and upgrading a model hierarchy.

The Upgrade Advisor can also identify cases where a model will not work because changes and improvements in Simulink require changes to a model.

The Upgrade Advisor offers options to perform recommended actions automatically or instructions for manual fixes.

Alternatives

You can also open the Upgrade Advisor from the Simulink Editor, by selecting **Analysis > Model Advisor > Upgrade Advisor**.

Alternatively, you can open the Upgrade Advisor from the Model Advisor. In the Model Advisor, under `By Task` checks, expand the folder `Upgrading to the Current Simulink Version` and select the check `Open the Upgrade Advisor`.

See Also

`modeladvisor`

Topics

“Consult the Upgrade Advisor”

“Run Model Checks”

“Upgrade Simulink Models Using a Simulink Project”

Introduced in R2012b

view_mdrefs

Display graph of model reference dependencies

Syntax

```
view_mdrefs('modelName')
```

Description

`view_mdrefs('modelName')` launches the Model Dependency Viewer, which displays a graph of model reference dependencies for the model specified by `modelName`. The nodes in the graph represent Simulink models. The directed lines indicate model dependencies.

The default display omits library blocks. You could see this same display by opening `modelName` and choosing **Analysis > Model Dependencies > Model Dependency Viewer > Models Only** from the model menu. Use **Analysis > Model Dependencies > Model Dependency Viewer** to see other dependency displays.

The Model Dependency Viewer is the same tool, and provides the same options, whether you launch it by typing `view_mdrefs('modelName')` or by using the Simulink GUI. To see an example of using the Model Dependency Viewer, type `sldemo_mdref_depgraph` in the MATLAB Command Window.

See Also

Model | `find_mdrefs`

Topics

`sldemo_mdref_depgraph`

“Model Referencing”

“Model Dependency Viewer”

Introduced before R2006a

who

List names of top-level data logging objects in Simulink `ModelDataLogs` data log

Syntax

```
log.who  
tsarray.who  
log.who('systems')  
log.who('all')
```

Description

Note To list names of top-level data logging objects in Dataset format, use `Simulink.SimulationData.Dataset.find`.

The `ModelDataLogs` class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the Dataset format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use Dataset format.

You can convert signal logging data from `ModelDataLogs` to Dataset format. Converting to Dataset format makes it easier to post-process with other logged data (for example, logged states), which can also use Dataset format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy `ModelDataLogs` API”.

`log.who` or `who(log)` lists the names of the top-level signal logging objects contained by `log`, where `log` is the handle of a `Simulink.ModelDataLogs` object name.

`tsarray.who` or `who(tsarray)` lists the names of `Simulink.TimeSeries` objects contained by the `Simulink.TsArray` object named `tsarray`.

`log.who('systems')` or `who(log, 'systems')` lists the names of all signal logging objects contained by `log` except for `Simulink.Timeseries` objects stored in `Simulink.TsArray` objects contained by `log`.

`log.who('all')` or `who(log, 'all')` lists the names of all the `Simulink.Timeseries` objects contained by the `Simulink.ModelDataLogs`, `Simulink.TsArray`, or `Simulink.SubsysDataLogs` object named `log`.

For information about other uses of `who`, execute `help who` in the MATLAB Command Window.

Tip To get the names of Dataset variables in the MAT-file, using the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function processes faster than using the `who` or `whos` functions.

See Also

`Simulink.ModelDataLogs` | `Simulink.SimulationData.Dataset.find` |
`Simulink.SimulationData.DatasetRef.getDatasetVariableNames` |
`Simulink.SubsysDataLogs` | `Simulink.Timeseries` | `Simulink.TsArray` |
`unpack` | `whos`

Topics

“Load Big Data for Simulations”

Introduced before R2006a

whos

List names and types of top-level data logging objects in Simulink `ModelDataLogs` data log

Syntax

```
log.whos
```

```
tsarray.whos
```

```
log.whos('systems')
```

```
log.whos('all')
```

Description

Note To list names of top-level data logging objects in `Dataset` format, use `Simulink.SimulationData.Dataset.find`.

The `ModelDataLogs` class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy `ModelDataLogs` API”.

`log.whos` or `whos(log)` lists the names and types of the top-level signal logging objects contained by `log`, where `log` is the handle of a `Simulink.ModelDataLogs` object name.

`tsarray.whos` or `whos(tsarray)` lists the names and types of `Simulink.TimeSeries` objects contained by the `Simulink.TsArray` object named `tsarray`.

`log.whos('systems')` or `whos(log, 'systems')` lists the names and types of all signal logging objects contained by `log` except for `Simulink.Timeseries` objects stored in `Simulink.TsArray` objects contained by `log`.

`log.whos('all')` or `whos(log, 'all')` lists the names and types of all the `Simulink.Timeseries` objects contained by the `Simulink.ModelDataLogs`, `Simulink.TsArray` or `Simulink.SubsysDataLogs` object named `log`.

For information about other uses of `whos`, execute `help whos` in the MATLAB Command Window.

Tip To get the names of Dataset variables in the MAT-file, using the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function processes faster than using the `who` or `whos` functions.

See Also

`Simulink.ModelDataLogs` | `Simulink.SimulationData.Dataset.find` |
`Simulink.SimulationData.DatasetRef.getDatasetVariableNames` |
`Simulink.SubsysDataLogs` | `Simulink.Timeseries` | `Simulink.TsArray` |
`unpack` | `who`

Topics

“Load Big Data for Simulations”

Introduced before R2006a

Mask Icon Drawing Commands

color	Change drawing color of subsequent mask icon drawing commands
disp	Display text on masked subsystem icon
dpoly	Display transfer function on masked subsystem icon
droots	Display transfer function on masked subsystem icon
fprintf	Display variable text centered on masked subsystem icon
image	Display RGB image on masked subsystem icon
patch	Draw color patch of specified shape on masked subsystem icon
plot	Draw graph connecting series of points on masked subsystem icon
port_label	Draw port label on masked subsystem icon
text	Display text at specific location on masked subsystem icon

color

Change drawing color of subsequent mask icon drawing commands

Syntax

```
color(colorstr)
```

Here, *colorostr* must be a character vector.

Description

`color(colorstr)` sets the drawing color of all subsequent mask drawing commands to the color specified by the string *colorstr*.

colorstr must be one of the following supported color strings.

```
blue  
green  
red  
cyan  
magenta  
yellow  
black
```

Entering any other string or specifying the color using RGB values results in a warning at the MATLAB command prompt; Simulink ignores the color change. The specified drawing color does not influence the color used by the `patch` or `image` drawing commands.

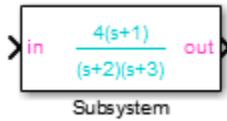
Examples

The following commands

```
color('cyan');  
roots([-1], [-2 -3], 4)
```

```
color('magenta')
port_label('input',1,'in')
port_label('output',1,'out')
```

draw the following mask icon.



See Also

roots | port_label

Introduced in R2006b

disp

Display text on masked subsystem icon

Syntax

```
disp(text)
disp(text, 'texmode', 'on')
```

Description

`disp(text)` displays *text* centered on the block icon. *text* is any MATLAB expression that evaluates to a string.

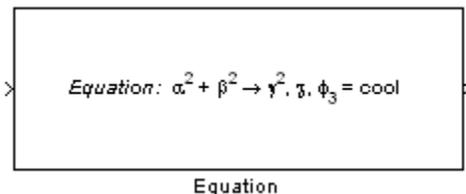
`disp(text, 'texmode', 'on')` allows you to use TeX formatting commands in *text*. The TeX formatting commands in turn allow you to include symbols and Greek letters in icon text. See “Interpreter” (MATLAB) for information on the TeX formatting commands supported by Simulink software.

Examples

The following command

```
disp({'\itEquation:' \alpha^2 + \beta^2 \rightarrow \gamma^2,
\chi, \phi_3 = {\bfcool}'}, 'texmode','on')
```

draws the equation that appears on this masked block icon.



See Also

`fprintf` | `port_label` | `text`

Introduced in R2007a

dpoly

Display transfer function on masked subsystem icon

Syntax

```
dpoly(num, den)
```

```
dpoly(num, den, 'character')
```

Description

`dpoly(num, den)` displays the transfer function whose numerator is *num* and denominator is *den*.

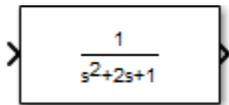
`dpoly(num, den, 'character')` specifies the name of the transfer function independent variable. The default is *s*.

When Simulink draws the block icon, the initialization commands execute and the resulting equation appears on the block icon, as in the following examples:

- To display a continuous transfer function in descending powers of *s*, enter

```
dpoly(num, den)
```

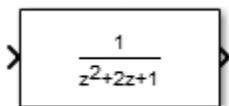
For example, for `num = [0 0 1];` and `den = [1 2 1]` the icon looks like:


$$\frac{1}{s^2+2s+1}$$

- To display a discrete transfer function in descending powers of *z*, enter

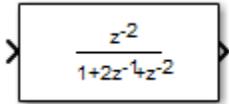
```
dpoly(num, den, 'z')
```

For example, for `num = [0 0 1];` and `den = [1 2 1];` the icon looks like:


$$\frac{1}{z^2+2z+1}$$

- To display a discrete transfer function in ascending powers of $1/z$, enter
`dpoly(num, den, 'z-')`

For example, for *num* and *den* as defined previously, the icon looks like:


$$\frac{z^{-2}}{1+2z^{-1}+z^{-2}}$$

If the parameters are not defined or have no values when you create the icon, Simulink software displays three question marks (? ? ?) in the icon. When you define parameter values in the Mask Settings dialog box, Simulink software evaluates the transfer function and displays the resulting equation in the icon.

See Also

`disp` | `droots` | `port_label` | `text`

droots

Display transfer function on masked subsystem icon

Syntax

```
droots(zero, pole, gain)
droots(zero, pole, gain, 'z')
droots(zero, pole, gain, 'z-')
```

Description

`droots(zero, pole, gain)` displays the transfer function whose zero is *zero*, pole is *pole*, and gain is *gain*.

`droots(zero, pole, gain, 'z')` and `droots(zero, pole, gain, 'z-')` expresses the transfer function in terms of *z* or *1/z*.

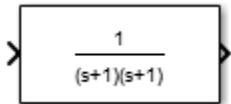
When Simulink draws the block icon, the initialization commands execute and the resulting equation appears on the block icon, as in the following examples:

- To display a zero-pole gain transfer function, enter

```
droots(z, p, k)
```

For example, the preceding command creates this icon for these values:

```
z = []; p = [-1 -1]; k = 1;
```



If the parameters are not defined or have no values when you create the icon, Simulink software displays three question marks (? ? ?) in the icon. When you define parameter values in the Mask Settings dialog box, Simulink software evaluates the transfer function and displays the resulting equation in the icon.

See Also

`disp` | `dpoly` | `port_label` | `text`

Introduced in R2007a

fprintf

Display variable text centered on masked subsystem icon

Syntax

```
fprintf(text)
fprintf(format, var)
```

Description

The `fprintf` command displays formatted text centered on the icon and can display *format* along with the contents of *var*.

Note While this `fprintf` function is identical in name to its corresponding MATLAB function, it provides only the functionality described on this page.

Examples

The command

```
fprintf('Hello');
```

displays the text 'Hello' on the icon.

The command

```
fprintf('Juhi = %d',17);
```

uses the decimal notation format (%d) to display the variable 17.

See Also

`disp` | `port_label` | `text`

Introduced before R2006a

image

Display RGB image on masked subsystem icon

Syntax

```
image(a)  
image(a, position)  
image(a, position, rotation)
```

Description

`image(a)` displays the image *a*, where *a* is an *m*-by-*n*-by-3 array of RGB values. If necessary, use the MATLAB commands `imread` and `ind2rgb` to read and convert bitmap files (such as GIF) to the necessary matrix format.

`image(a, position)` creates the image at the specified position as follows.

Position	Description
<code>[x, y, w, h]</code>	Position (<i>x</i> , <i>y</i>) and size (<i>w</i> , <i>h</i>) of the image where the position is relative to the lower-left corner of the mask. The image scales to fit the specified size.
<code>'center'</code>	Center of the mask
<code>'top-left'</code>	Top left corner of the mask, unscaled
<code>'bottom-left'</code>	Bottom left corner of the mask, unscaled
<code>'top-right'</code>	Top right corner of the mask, unscaled
<code>'bottom-right'</code>	Bottom right corner of the mask, unscaled

`image(a, position, rotation)` allows you to specify whether the image rotates ('on') or remains stationary ('off') as the icon rotates. The default is 'off'.

Note Images in formats `.cur`, `.hdf4`, `.ico`, `.pcx`, `.ras`, `.xwd`, `.svg` (full version) cannot be used as block mask images.

Examples

You can use different commands depending on your requirement to add an image. These commands can be added in the **Icon & Ports** pane of the Mask Editor dialog box.

Syntax	Description
<code>image('icon.jpg')</code>	Reads the icon image from a JPEG file named <code>icon.jpg</code> in the MATLAB path.
<code>[data, map]=image('label.gif')</code> <code>pic=ind2rgb(data,map);</code>	Reads and converts a GIF file, <code>label.gif</code> , to the appropriate matrix format.
<code>image(pic)</code>	Reads the converted label image.

See Also

`patch` | `plot`

Introduced before R2006a

patch

Draw color patch of specified shape on masked subsystem icon

Syntax

```
patch(x, y)  
patch(x, y, [r g b])
```

Description

`patch(x, y)` creates a solid patch having the shape specified by the coordinate vectors `x` and `y`. The patch's color is the current foreground color.

`patch(x, y, [r g b])` creates a solid patch of the color specified by the vector `[r g b]`, where `r` is the red component, `g` the green, and `b` the blue. For example,

```
patch([0 .5 1], [0 1 0], [1 0 0])
```

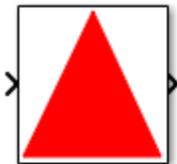
creates a red triangle on the mask's icon.

Examples

The command

```
patch([0 .5 1], [0 1 0], [1 0 0])
```

creates a red triangle on the mask's icon.



Pyramid

See Also

image | plot

Introduced before R2006a

plot

Draw graph connecting series of points on masked subsystem icon

Syntax

```
plot(Y)  
plot(X1, Y1, X2, Y2, ...)
```

Description

`plot(Y)` plots, for a vector Y , each element against its index. If Y is a matrix, it plots each column of the matrix as though it were a vector.

`plot(X1, Y1, X2, Y2, ...)` plots the vectors $Y1$ against $X1$, $Y2$ against $X2$, and so on. Vector pairs must be the same length and the list must consist of an even number of vectors.

Plot commands can include `NaN` and `inf` values. When Simulink software encounters `NaN`s or `infs`, it stops drawing, and then begins redrawing at the next numbers that are not `NaN` or `inf`. The appearance of the plot on the icon depends on the units defined by the **Icon units** option in the Mask Editor.

Simulink software displays three question marks (? ? ?) in the block icon and issues warnings in these situations:

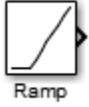
- When you have not defined values for the parameters used in the drawing commands (for example, when you first create the mask, but have not yet entered values in the Mask Settings dialog box)
- When you enter a masked block parameter or drawing command incorrectly

Examples

The command

```
plot([0 1 5], [0 0 4])
```

generates the plot that appears on the icon for the Ramp block, in the Sources library.



See Also

image

Introduced before R2006a

port_label

Draw port label on masked subsystem icon

Syntax

```
port_label('port_type', port_number, 'label')  
port_label('port_type', port_number, 'label', 'texmode', 'on')
```

Description

`port_label('port_type', port_number, 'label')` draws a label on a port. Valid values for `port_type` include the following.

Value	Description
input	Simulink input port
output	Simulink output port
lconn	Physical Modeling connection port on the left side of a masked subsystem
rconn	Physical Modeling connection port on the right side of a masked subsystem
Enable	Label for the trigger port in a masked Triggered or Enabled and Triggered subsystem.
trigger	Label for the trigger port in a masked Triggered or Enabled and Triggered subsystem.
action	Label for the action port in a masked Switch Case Action Subsystem.

The input argument `port_number` is an integer, and `label` is text specifying the port's label.

Note Physical Modeling port labels are assigned based on the nominal port location. If the masked subsystem has been rotated or flipped, for example, a port labeled using 'lconn' as the `port_type` may not appear on the left side of the block.

`port_label('port_type', port_number, 'label', 'texmode', 'on')` lets you use TeX formatting commands in `label`. The TeX formatting commands allow you to include symbols and Greek letters in the port label. See “Interpreter” (MATLAB) for information on the TeX formatting commands that the Simulink software supports.

Examples

The command

```
port_label('input', 1, 'a')
```

defines `a` as the label of input port 1.

The command

```
port_label('Enable','En')
```

defines `En` as the label of Enable port.

The command

```
port_label('trigger','Tr')
```

defines `Tr` as the label of trigger port.

The command

```
port_label('action','Switch():')
```

defines `Switch():` as the label of action port.

The command

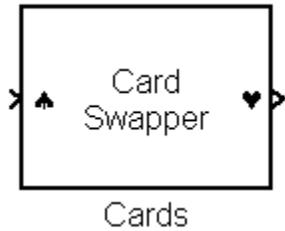
```
port_label('trigger','$\sqrt{m}$','interpreter','tex')
```

defines the label of trigger port with latex interpretation.

The commands

```
disp('Card\nSwapper');  
port_label('input',1,'\spadesuit','texmode','on');  
port_label('output',1,'\heartsuit','texmode','on');
```

draw playing card symbols as the labels of the ports on a masked subsystem.



See Also

`disp` | `fprintf` | `text`

Introduced before R2006a

text

Display text at specific location on masked subsystem icon

Syntax

```
text(x, y, 'text')
text(x, y, 'text', 'horizontalAlignment', 'halign',
     'verticalAlignment', 'valign')
text(x, y, 'text', 'texmode', 'on')
```

Description

The `text` command places a character vector at a location specified by the point (x, y) whose units are defined by the **Icon units** option in the Mask Editor.

`text(x,y, text, 'texmode', 'on')` allows you to use TeX formatting commands in `text`. The TeX formatting commands in turn allow you to include symbols and Greek letters in icon text. See “Interpreter” (MATLAB) for information on the TeX formatting commands supported by Simulink software.

You can optionally specify the horizontal and/or vertical alignment of the text relative to the point (x, y) in the `text` command.

The `text` command offers the following horizontal alignment options.

Option	Aligns
'left'	The left end of the text at the specified point
'right'	The right end of the text at the specified point
'center'	The center of the text at the specified point

The `text` command offers the following vertical alignment options.

Option	Aligns
'base'	The baseline of the text at the specified point
'bottom'	The bottom line of the text at the specified point

Option	Aligns
'middle'	The midline of the text at the specified point
'cap'	The capitals line of the text at the specified point
'top'	The top of the text at the specified point

Note While this `text` function is identical in name to its corresponding MATLAB function, it provides only the functionality described on this page.

Examples

Text Alignment

Center the mask icon text foobar.

```
text(0.5, 0.5, 'foobar', 'horizontalAlignment', 'center')
```

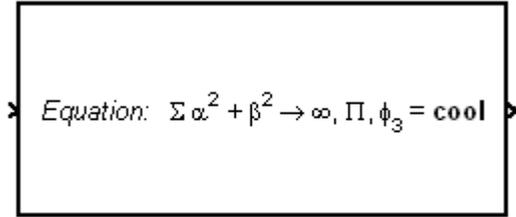
Equation in Mask Icon

Draw a left-aligned equation as the mask icon.

In the Icons & Ports dialog of the Mask Editor, set **Icon units** to Normalized.

In the **Icon drawing commands** text box, enter the following command.

```
text(.05, .5, '\{itEquation:} \Sigma \alpha^2 +  
\beta^2 \rightarrow \infty, \Pi, \phi_3 = {\bfcool}',  
'hor', 'left', 'texmode', 'on')
```



Equation: $\Sigma \alpha^2 + \beta^2 \rightarrow \infty, \Pi, \phi_3 = \text{cool}$

Equation

See Also

`disp` | `fprintf` | `port_label`

Introduced before R2006a

Simulink Debugger Commands

ashow	Show algebraic loop
atrace	Set algebraic loop trace level
bafter	Insert breakpoint after specified method
break	Insert breakpoint before specified method
bshow	Show specified block
clear	Clear breakpoints from model
continue	Continue simulation
disp	Display block's I/O when simulation stops
ebreak	Enable (or disable) breakpoint on solver errors
elist	List simulation methods in order in which they are executed during simulation
emode	Toggle model execution between accelerated and normal mode
etrace	Enable or disable method tracing
help	Display help for debugger commands
nanbreak	Set or clear nonfinite value break mode
next	Advance simulation to start of next method at current level in model's execution list
probe	I/O and state data for blocks
quit	Stop simulation debugger
rbreak	Break simulation before solver reset
run	Run simulation to completion
slist	Sorted list of model blocks
states	Current state values
status	Debugging options in effect
step	Advance simulation by one or more methods
stimes	Model sample times
stop	Stop simulation
strace	Set solver trace level
systems	List nonvirtual systems of model
tbreak	Set or clear time breakpoint
trace	Display block's I/O each time block executes
undisp	Remove block from debugger's list of display points
4-2 untrace	Remove block from debugger's list of trace points
where	Display current location of simulation in simulation loop
xbreak	Break when debugger encounters step-size-limiting state

ashow

Show algebraic loop

Syntax

```
ashow <gcb | s:b | s#n | clear>
```

```
as <gcb | s:b | s#n | clear>
```

Arguments

<code>gcb</code>	Current block.
<code>s:b</code>	The block whose system index is <code>s</code> and block index is <code>b</code> .
<code>s#n</code>	The algebraic loop numbered <code>n</code> in system <code>s</code> .
<code>clear</code>	Switch that clears loop coloring.

Description

`ashow` without any arguments lists all of a model's algebraic loops in the MATLAB Command Window. `ashow gcb` or `ashow s:b` highlights the algebraic loop that contains the specified block. `ashow s#n` highlights the `n`th algebraic loop in system `s`. The `ashow clear` command removes algebraic loop highlights from the model diagram.

See Also

`atrace` | `slist`

Introduced before R2006a

atrace

Set algebraic loop trace level

Syntax

```
atrace level
```

```
at level
```

Arguments

`level` Trace level (0 = none, 4 = everything).

Description

The `atrace` command sets the algebraic loop trace level for a simulation.

Command	Displays for Each Algebraic Loop
<code>atrace 0</code>	No information
<code>atrace 1</code>	The loop variable solution, the number of iterations required to solve the loop, and the estimated solution error
<code>atrace 2</code>	Same as level 1
<code>atrace 3</code>	Level 2 plus Jacobian matrix used to solve loop
<code>atrace 4</code>	Level 3 plus intermediate solutions of the loop variable

See Also

`states` | `systems`

Introduced before R2006a

bafter

Insert breakpoint after specified method

Syntax

```
bafter
```

```
ba
```

```
bafter m:mid
```

```
bafter <sysIdx:blkIdx | gcb> [meth] [tid:TID]
```

```
bafter <s:sysIdx | gcs> [meth] [tid:TID]
```

```
bafter model [meth] [tid:TID]
```

Arguments

<i>mid</i>	Method ID
<i>sysIdx:blkIdx</i>	Block ID
<i>x</i>	
gcb	Currently selected block
<i>sysIdx</i>	System ID
gcs	Currently selected system
model	Currently selected model
<i>meth</i>	A method name, e.g., <code>Outputs.Major</code>
TID	Task ID

Description

`bafter` inserts a breakpoint after the current method.

Instead of `bafter`, you can use the short form of `ba` with any of the syntaxes.

`bafter m:mid` inserts a breakpoint after the method specified by `mid` (see “Method ID”).

`bafter sysIdx:blkIdx` inserts a breakpoint after each invocation of the method of the block specified by `sysIdx:blkIdx` (see “Block ID”) in major time steps. `bafter gcb` inserts a breakpoint after each invocation of a method of the currently selected block (see `gcb`) in major times steps.

`bafter s:sysIdx` inserts a breakpoint after each method of the root system or nonvirtual subsystem specified by the system ID: `sysIdx`.

Note The `systems` command displays the system IDs for all nonvirtual systems in the currently selected model.

`bafter gcs` inserts a breakpoint after each method of the currently selected nonvirtual system.

`bafter model` inserts a breakpoint after each method of the currently selected model.

The optional `meth` parameter allow you to set a breakpoint after a particular block, system, or model method and task. For example, `bafter gcb Outputs` sets a breakpoint after the `Outputs` method of the currently selected block.

The optional TID parameter allows you to set a breakpoint after invocation of a method by a particular task. For example, suppose that the currently selected nonvirtual subsystem operates on task 2 and 3. Then `bafter gcs Outputs tid:2` sets a breakpoint after the invocation of the subsystem's `Outputs` method that occurs when task 2 is active.

See Also

`break` | `clear` | `ebreak` | `nanbreak` | `rbreak` | `slist` | `systems` | `tbreak` | `where`
| `xbreak` | `zcbreak`

Introduced before R2006a

break

Insert breakpoint before specified method

Syntax

```
break
```

```
b
```

```
break m:mid
```

```
break <sysIdx:blkIdx | gcb> [meth] [tid:TID]
```

```
break <s:sysIdx | gcs> [meth] [tid:TID]
```

```
break model [meth] [tid:TID]
```

Arguments

<i>mid</i>	Method ID
<i>sysIdx</i> : <i>blkIdx</i>	Block ID
<i>x</i>	
gcb	Currently selected block
<i>sysIdx</i>	System ID
<i>gcs</i>	Currently selected system
model	Currently selected model
<i>meth</i>	A method name, e.g., <code>Outputs.Major</code>
<i>TID</i>	task ID

Description

`break` inserts a breakpoint before the current method.

Instead of `break`, you can use the short form of `b` with any of the syntaxes.

`break m:mid` inserts a breakpoint before the method specified by *mid* (see “Method ID”).

`break sysIdx:blkIdx` inserts a breakpoint before each invocation of the method of the block specified by *sysIdx:blkIdx* (see “Block ID”) in major time steps. `break gcb` inserts a breakpoint before each invocation of a method of the currently selected block (see *gcb*) in major time steps.

`break s:sysIdx` inserts a breakpoint at each method of the root system or nonvirtual subsystem specified by the system ID: *sysIdx*.

Note The `systems` command displays the system IDs for all nonvirtual systems in the currently selected model.

`break gcs` inserts a breakpoint at each method of the currently selected nonvirtual system.

`break model` inserts a breakpoint at each method of the currently selected model.

The optional *meth* parameter allow you to set a breakpoint at a particular block, system, or model method. For example, `break gcb Outputs` sets a breakpoint at the `Outputs` method of the currently selected block.

The optional `TID` parameter allows you to set a breakpoint at the invocation of a method by a particular task. For example, suppose that the currently selected nonvirtual subsystem operates on task 2 and 3. Then `break gcs Outputs tid:2` sets a breakpoint at the invocation of the subsystem's `Outputs` method that occurs when task 2 is active.

See Also

`bafter` | `clear` | `ebreak` | `nanbreak` | `rbreak` | `slist` | `systems` | `tbreak` | `where` | `xbreak` | `zcbreak`

Introduced before R2006a

bshow

Show specified block

Syntax

```
bshow s:b
```

```
bs s:b
```

Arguments

`s:b` The block whose system index is `s` and block index is `b`.

Description

The `bshow` command opens the model window containing the specified block and selects the block.

See Also

`slist`

Introduced before R2006a

clear

Clear breakpoints from model

Syntax

```
clear  
cl  
clear m:mid  
clear id  
clear <sysIdx:blkIdx | gcb>
```

Arguments

<i>mid</i>	Method ID
<i>id</i>	Breakpoint ID
<i>sysIdx:blkIdx</i>	Block ID
<i>Idx</i>	
gcb	Currently selected block

Description

`clear` clears a breakpoint from the current method.

Instead of `clear`, you can use the short form of `cl` with any of the syntaxes.

`clear m:mid` clears a breakpoint from the method specified by *mid*.

`clear id` clears the breakpoint specified by the breakpoint ID *id*.

`clear sysIdx:blkIdx` clears any breakpoints set on the methods of the block specified by *sysIdx:blkIdx*.

clear **gcb** clears any breakpoints set on the methods of the currently selected block.

See Also

bafter | break | slist

Introduced before R2006a

continue

Continue simulation

Syntax

```
continue
```

```
c
```

Description

The `continue` command continues the simulation from the current breakpoint. If animation mode is not enabled, the simulation continues until it reaches another breakpoint or its final time step. If animation mode is enabled, the simulation continues in animation mode to the first method of the next major time step, ignoring breakpoints.

See Also

`quit` | `run` | `stop`

Introduced before R2006a

disp

Display block's I/O when simulation stops

Syntax

```
disp
```

```
d
```

```
disp gcb
```

```
disp s:b
```

Arguments

s:b The block whose system index is *s* and block index is *b*.

gcb Current block.

Description

The `disp` command registers a block as a display point. The debugger displays the inputs and outputs of all display points in the MATLAB Command Window whenever the simulation halts. Invoking `disp` without arguments shows a list of display points. Use `undisp` to unregister a block.

Instead of `disp`, you can use the short form of `d` with any of the syntaxes.

See Also

`probe` | `slist` | `trace` | `undisp`

Introduced before R2006a

ebreak

Enable (or disable) breakpoint on solver errors

Syntax

```
ebreak
```

```
eb
```

Description

This command causes the simulation to stop if the solver detects a recoverable error in the model. If you do not set or disable this breakpoint, the solver recovers from the error and proceeds with the simulation without notifying you.

See Also

```
bafter | break | clear | nanbreak | rbreak | slist | systems | tbreak | where  
| xbreak | zcbreak
```

Introduced before R2006a

elist

List simulation methods in order in which they are executed during simulation

Syntax

```
elist
```

```
e1
```

```
elist m:mid [tid:TID]
```

```
elist <gcs | s:sysIdx> [mth] [tid:TID]
```

```
elist <gcb | sysIdx:blkIdx> [mth] [tid:TID]
```

Description

Instead of `elist`, you can use the short form of `e1` with any of the syntaxes.

`elist m:mid` lists the methods invoked by the system or nonvirtual subsystem method corresponding to the method id *mid* (see the `where` command for information on method IDs), e.g.,

```
(sldebug @19): elist m:19

RootSystem.Outputs 'vdp' [tid=0] : ← Calling method
0:0 Integrator.Outputs 'vdp/x1' [tid=0]
0:1 Outport.Outputs 'vdp/Out1' [tid=0]
0:2 Integrator.Outputs 'vdp/x2' [tid=0]
...
↑      ↑      ↑      ↑
Block id Method Block Task id
```

The method list specifies the calling method followed by the methods that it calls in the order in which they are invoked. The entry for the calling method includes

- The name of the method

The name of the method is prefixed by the type of system that defines the method, e.g., `RootSystem`.

- The name of the model or subsystem instance on which the method is invoked
- The ID of the task that invokes the method

The entry for each called method includes

- The ID (*sysIdx:blkIdx*) of the block instance on which the method is invoked

The block ID is prefixed by a number specifying the system that contains the block (the *sysIdx*). This allows Simulink software to assign the same block ID to blocks residing in different subsystems.

- The name of the method

The method name is prefixed with the type of block that defines the method, e.g., `Integrator`.

- The name of the block instance on which the method is invoked
- The task that invokes the method

The optional task ID parameter (**tid**: *TID*) allows you to restrict the displayed lists to methods invoked for a specified task. You can specify this option only for system or atomic subsystem methods that invoke Outputs or Update methods.

`elist <gcs | s:sysIdx>` lists the methods executed for the currently selected system (specified by the `gcs` command) or the system or nonvirtual subsystem specified by the system ID *sysIdx*, e.g.,

```
(sldebug @19): elist gcs

RootSystem.Start 'vdp':
  0:0 Integrator.Start 'vdp/x1'
  0:2 Integrator.Start 'vdp/x2'
  0:4 Scope.Start 'vdp/Scope'
  0:5 Fcn.Start 'vdp/Fcn'
  0:6 Product.Start 'vdp/Product'
  0:7 Gain.Start 'vdp/Mu'
  0:8 Sum.Start 'vdp/Sum'

RootSystem.Initialize 'vdp':
  0:0 Integrator.Initialize 'vdp/x1'
  ...
```

The system ID of a model's root system is 0. You can use the debugger's `systems` command to determine the system IDs of a model's subsystems.

Note The `elist` and `where` commands use block IDs to identify subsystems in their output. The block ID for a subsystem is not the same as the system ID displayed by the `systems` command. Use the `elist sysIdx:blkIdx` form of the `elist` command to display the methods of a subsystem whose block ID appears in the output of a previous invocation of the `elist` or `where` command.

`elist <gcs | s:sysIdx> mth` lists methods of type `mth` to be executed for the system specified by the `gcs` command or the system ID `sysIdx`, e.g.,

```
(sldebug @19): elist gcs Start  
  
RootSystem.Start 'vdp':  
  0:0 Integrator.Start 'vdp/x1'  
  0:2 Integrator.Start 'vdp/x2'  
  0:4 Scope.Start 'vdp/Scope'  
  0:5 Fcn.Start 'vdp/Fcn'  
  0:6 Product.Start 'vdp/Product'  
  0:7 Gain.Start 'vdp/Mu'  
  0:8 Sum.Start 'vdp/Sum'  
  ...
```

Use `elist gcb` to list the methods invoked by the nonvirtual subsystem currently selected in the model.

See Also

`slist` | `systems` | `where`

Introduced before R2006a

emode

Toggle model execution between accelerated and normal mode

Syntax

emode

em

Description

Toggles the simulation between accelerated and normal mode when using the Accelerator mode in Simulink software. For more information, see “Run Accelerator Mode with the Simulink Debugger”.

Introduced before R2006a

etrace

Enable or disable method tracing

Syntax

```
etrace level level-number
```

```
et level level-number
```

Description

This command enables or disables method tracing, depending on the value of `level`:

Level	Description
0	Turn tracing off.
1	Trace model methods.
2	Trace model and system methods.
3	Trace model, system, and block methods.

When method tracing is on, the debugger prints a message at the command line every time a method of the specified level is entered or exited. The message specifies the current simulation time, whether the simulation is entering or exiting the method, the method id and name, and the name of the model, system, or block to which the method belongs.

See Also

`elist` | `trace` | `where`

Introduced before R2006a

help

Display help for debugger commands

Syntax

```
help
```

```
?
```

```
h
```

Description

The `help` command displays a list of debugger commands in the command window. The list includes the syntax and a brief description of each command.

Introduced before R2006a

nanbreak

Set or clear nonfinite value break mode

Syntax

```
nanbreak
```

```
na
```

Description

The `nanbreak` command causes the debugger to break whenever the simulation encounters a nonfinite (NaN or Inf) value. If nonfinite break mode is set, `nanbreak` clears it.

See Also

`bafter` | `break` | `rbreak` | `tbreak` | `xbreak` | `zcbreak`

Topics

`ebreak`

Introduced before R2006a

next

Advance simulation to start of next method at current level in model's execution list

Syntax

```
next
```

```
n
```

Description

The `next` command advances the simulation to the start of the next method at the current level in the model's method execution list.

Note The `next` command has the same effect as the `step over` command. See `step` for more information.

See Also

`step`

Introduced before R2006a

probe

I/O and state data for blocks

Syntax

```
probe
probe s:b
probe gcb
probe level level-type
p
```

Description

`probe` sets the Simulink debugger to interactive probe mode. In this mode, the debugger displays the I/O of a selected block. To exit interactive probe mode, enter a debugger command or press the **Enter** key.

`probe s:b` displays the I/O of the block whose system index is *s* and block index is *b*.

`probe gcb` displays the I/O of the currently selected block.

`probe level level-type` sets the verbosity level for `probe`, `trace`, and `dis`. If *level-type* is `io`, the debugger displays block I/O. If *level-type* is `all` (default), the debugger displays all information for the current state of a block, including inputs and outputs, states, and zero crossings.

`p` is the short form of the command.

Examples

Display I/O for the currently selected block `Out2` in the model `vdp` using the Simulink debugger.

1 In the MATLAB Command Window, enter:

```
sldebug 'vdp'
```

The MATLAB command prompt `>>` changes to the Simulink debugger prompt (sldebug @0): `>>`.

2 Enter:

```
probe gcb
```

The MATLAB Command Window displays:

```
probe: Data of 0:3 Outport block 'vdp/Out2':  
U1      = [0]
```

See Also

`disp` | `trace`

Introduced in R2007a

quit

Stop simulation debugger

Syntax

```
quit  
q
```

Description

`quit` stops the Simulink debugger and returns to the MATLAB command prompt.

`q` is the short form of the command.

Examples

Start the Simulink debugger for the model `vdp` and then stop it.

- 1 In the MATLAB Command Window, enter:

```
sldebug 'vdp'
```

The MATLAB command prompt `>>` changes to the Simulink debugger prompt (`sldebug @0`): `>>`.

- 2 Enter:

```
quit
```

See Also

`stop`

Introduced before R2006a

rbreak

Break simulation before solver reset

Syntax

```
rbreak  
rb
```

Description

`rbreak` enables (or disables) a solver reset breakpoint if the breakpoint is disabled (or enabled). The breakpoint causes the debugger to halt the simulation whenever an event requires a solver reset. The halt occurs before the solver resets.

`rb` is the short form of the command.

Examples

Start Simulink debugger for the model `vdp` and a set breakpoint before a solver reset.

- 1 In the MATLAB Command Window, enter:

```
sldebug 'vdp'
```

The MATLAB command prompt `>>` is replaced with the Simulink debugger prompt (`sldebug @0`): `>>`.

- 2 Enter:

```
rbreak
```

The MATLAB Command Window displays:

```
Break on solver reset request           : enabled
```

See Also

`bafter` | `break` | `ebreak` | `nanbreak` | `tbreak` | `xbreak` | `zcbreak`

Introduced before R2006a

run

Run simulation to completion

Syntax

```
run  
r
```

Description

`run` starts the simulation from the current breakpoint to its final time step. It ignores breakpoints and display points.

`r` is the short form of the command

Examples

Continue the simulation for the model `vdp` using the Simulink debugger.

- 1 In the MATLAB Command Window, enter:

```
sldebug 'vdp'
```

The MATLAB command prompt `>>` changes to the Simulink debugger prompt (`sldebug @0`): `>>`.

- 2 Enter:

```
run
```

See Also

`continue` | `quit` | `stop`

Introduced before R2006a

slist

Sorted list of model blocks

Syntax

```
slist  
sli
```

Description

`slist` displays a list of blocks for the root system and each nonvirtual subsystem sorted according to data dependencies and other criteria.

For each system (root or nonvirtual), `slist` displays:

- Title line specifying the name of the system, the number of nonvirtual blocks that the system contains, and the number of blocks in the system that have direct feedthrough ports.
- Entry for each block in the order in which the blocks appear in the sorted list.

For each block entry, `slist` displays the block ID and the name and type of the block. The block ID consists of a system index and a block index separated by a colon (`sysIdx:blkIdx`).

- Block index is the position of the block in the sorted list.
- System index is the order in which the Simulink software generated the system sorted list. The system index has no special significance. It simply allows blocks that appear in the same position in different sorted lists to have unique identifiers.

Simulink software uses sorted lists to create block method execution lists (see `elist`) for root system and nonvirtual subsystem methods. In general, root system and nonvirtual subsystem methods invoke the block methods in the same order as the blocks appear in the sorted list.

Exceptions occur in the execution order of block methods. For example, execution lists for multicast models group together all blocks operating at the same rate and in the same

task. Slower groups appear later than faster groups. The grouping of methods by task can result in a block method execution order that is different from the block sorted order. However, within groups, methods execute in the same order as the corresponding blocks appear in the sorted list.

`sli` is the short form of the command.

Examples

Display a sorted list of the root system in the `vdp` model using the Simulink debugger.

- 1 In the MATLAB Command Window, enter:

```
sldebug 'vdp'
```

The MATLAB command prompt `>>` changes to the Simulink debugger prompt (`sldebug @0`): `>>`.

- 2 Enter:

```
slist
```

The MATLAB Command Window displays:

```
---- Sorted list for 'vdp' [9 nonvirtual blocks, directFeed=0]
0:0   'vdp/x1' (Integrator)
0:1   'vdp/Out1' (Outport)
0:2   'vdp/x2' (Integrator)
0:3   'vdp/Out2' (Outport)
0:4   'vdp/Scope' (Scope)
0:5   'vdp/Fcn' (Fcn)
0:6   'vdp/Product' (Product)
0:7   'vdp/Mu' (Gain)
0:8   'vdp/Sum' (Sum)
```

See Also

`elist` | `systems`

Introduced before R2006a

states

Current state values

Syntax

```
states
```

Description

`states` displays a list of the current states of the model. The list includes the index, current value, `system:block:element ID`, state vector name, and block name for each state.

Examples

Display information about the states for the `vdp` model.

- 1 In the MATLAB Command Window, enter:

```
sldebug 'vdp'
```

The MATLAB command prompt `>>` changes to the Simulink debugger prompt (`sldebug @0`): `>>`.

- 2 Enter:

```
states
```

The MATLAB Command Window displays:

```
Continuous States:
Idx Value (system:block:element Name 'BlockName')
  0  0 (0:0:0 CSTATE 'vdp/x1')
  1  0 (0:2:0 CSTATE 'vdp/x2')
```

Introduced before R2006a

status

Debugging options in effect

Syntax

Description

Display a list of the debugging options in effect.

Examples

View Debugger Status for vdp

Start the debugger with `vdp`. The command prompt changes to the Simulink debugger prompt (`sldebug @0`): `>>`.

```
sldebug 'vdp'
```

Display the debugging status.

```
status
```

```
%-----%  
Current simulation time           : 0 (MajorTimeStep)  
Solver needs reset                 : no  
Solver derivatives cache needs reset : no  
Zero crossing signals cache needs reset : no  
Default command to execute on return/enter : ""  
Break at zero crossing events      : disabled  
Break on solver error              : disabled  
Break on failed integration step   : disabled  
Time break point                   : disabled  
Break on non-finite (NaN,Inf) values : disabled  
Break on solver reset request      : disabled  
Display level for disp, trace, probe : 1 (i/o, states)
```

```
Solver trace level           : 0
Algebraic loop tracing level : 0
Animation Mode               : off
Execution Mode               : Normal
Display level for etrace     : 0 (disabled)
Break points                 : none installed
Display points               : none installed
```

Introduced before R2006a

step

Advance simulation by one or more methods

Syntax

```
step  
step in  
step over  
step out  
step top  
step blockmth  
s
```

Description

`step` or `step in` advances the simulation to the next method in the current time step.

`step over` advances the simulation over the next method.

`step out` advances the simulation the end of the current simulation point hierarchy.

`step top` advances the simulation to the first method executed in the next time step.

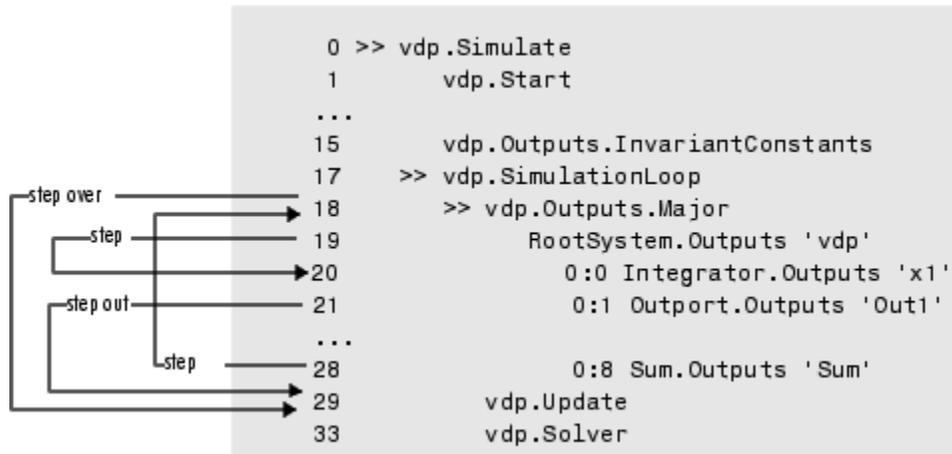
`step blockmth` advances the simulation to the next method that operates on a block.

`s` is the short form of the command.

If `step` advances the simulation to the start of a block method, the debugger points at the block on which the method operates.

Examples

The following diagram illustrates the effect of various forms of the `step` command for the model `vdp`.



See Also

`elist` | `next` | `where`

Introduced in R2007a

stimes

Model sample times

Syntax

```
stimes  
sti
```

Description

`stimes` displays information about the model sample times, including the sample time period, offset, and task ID.

`sti` is the short form of the command.

Examples

Display sample times for the model `vdp` using the Simulink debugger.

- 1 In the MATLAB Command Window, enter:

```
sldebug 'vdp'
```

The MATLAB command prompt `>>` changes to the Simulink debugger prompt (`sldebug @0`): `>>`.

- 2 Enter:

```
stimes
```

The MATLAB Command Window displays:

```
--- Sample times for 'vdp' [Number of sample times = 1]  
1. [0      , 0      ] tid=0 (continuous sample time)
```

Introduced before R2006a

stop

Stop simulation

Syntax

```
stop
```

Description

`stop` stops the simulation of the model you are debugging.

Examples

Start and stop a simulation for the model `vdp` using the Simulink debugger.

- 1 Start a debugger session. In the MATLAB Command Window, enter:

```
sldebug 'vdp'
```

The MATLAB command prompt `>>` changes to the Simulink debugger prompt (`sldebug @0`): `>>`.

- 2 Start a simulation of the model. Enter:

```
run
```

- 3 Stop the simulation. Enter:

```
stop
```

See Also

`continue` | `quit` | `run`

Introduced before R2006a

strace

Set solver trace level

Syntax

```
strace level  
i
```

Description

`strace level` causes the solver to display diagnostic information in the MATLAB Command Window, depending on the value of `level`. Values are 0 (no information) or 1 (maximum information about time steps, integration steps, zero crossings, and solver resets).

`i` is the short form of the command.

Examples

Display maximum information about a simulation for the model `vdp` using the Simulink debugger.

- 1 In the MATLAB Command Window, enter:

```
sldebug 'vdp'
```

The MATLAB command prompt `>>` changes to the Simulink debugger prompt (`sldebug @0`): `>>`.

- 2 Get information about the notation `.`. Enter:

```
help time
```

The MATLAB Command Window displays:

```
Time is displayed as:  
TM = <time while in MajorTimeStep>
```

```
Tm = <time while in MinorTimeStep>
Tr = <time while in solver reset>
Tz = <time at or just after zero crossing>
TzL = <time while in major step just before (at left post of) zero crossing>
TzR = <time while in major step at or just after (at right post of) zero crossing>
Ts = <time of successful integration step>
Tf = <time of failed integration step>
Tn = <time while in Newton iteration> (when using implicit solvers)
Tj = <time during Jacobian evaluation> (when using implicit solvers)
```

Step size is displayed as:

```
Hm = <step size at the start of solver phase>
Hs = <successful integration step size>
Hf = <failed integration step size>
Hn = <step size during Newton iteration> (when using implicit solvers)
Hz = <value of 'TM - TzL' during zero crossing search>
Iz = <value of 'Tz - TzL' during zero crossing search>
```

3 Set trace to display all information. Enter:

```
strace 1
```

When diagnostic tracing is on, the debugger displays the sizes of major and minor time steps.

```
[TM = 13.21072088374186 ] Start of Major Time Step
[Tm = 13.21072088374186 ] Start of Minor Time Step
```

The debugger displays integration information. This information includes the time step of the integration method, step size of the integration method, outcome of the integration step, normalized error, and index of the state.

```
[Tm = 13.21072088374186 ] [H = 0.2751116230148764 ] Begin Integration Step
[Tf = 13.48583250675674 ] [Hf = 0.2751116230148764 ] Fail [Er = 1.0404e+000]
  [Ix = 1]
[Tm = 13.21072088374186 ] [H = 0.2183536061326544 ] Retry
[Ts = 13.42907448987452 ] [Hs = 0.2183536061326539 ] Pass [Er = 2.8856e-001]
  [Ix = 1]
```

For zero crossings, the debugger displays information about the iterative search algorithm when the zero crossing occurred. This information includes the time step of the zero crossing, step size of the zero crossing detection algorithm, length of the time interval bracketing the zero crossing, and a flag denoting the rising or falling direction of the zero crossing.

```
[Tz = 3.615333333333301 ] Detected 1 Zero Crossing Event 0[F]
  Begin iterative search to bracket zero crossing event
[Tz = 3.621111157580072 ] [Hz = 0.0057778242426771424 ] [Iz = 4.2222e-003] 0[F]
[Tz = 3.621116982080098 ] [Hz = 0.005783648746797265 ] [Iz = 4.2164e-003] 0[F]
[Tz = 3.621116987943544 ] [Hz = 0.005783654610242994 ] [Iz = 4.2163e-003] 0[F]
[Tz = 3.621116987943544 ] [Hz = 0.005783654610242994 ] [Iz = 1.1804e-011] 0[F]
[Tz = 3.621116987949452 ] [Hz = 0.005783654616151157 ] [Iz = 5.8962e-012] 0[F]
[Tz = 3.621116987949452 ] [Hz = 0.005783654616151157 ] [Iz = 5.1514e-014] 0[F]
  End iterative search to bracket zero crossing event
```

When a solver resets occur, the debugger displays the time at which the solver was reset.

```
[Tr = 6.246905153573676 ] Process Solver Reset  
[Tr = 6.246905153573676 ] Reset Zero Crossing Cache  
[Tr = 6.246905153573676 ] Reset Derivative Cache
```

See Also

[atrace](#) | [etrace](#) | [states](#) | [trace](#) | [zclist](#)

Introduced before R2006a

systems

List nonvirtual systems of model

Syntax

```
systems  
sys
```

Description

`systems` displays the nonvirtual subsystems for a model in the MATLAB Command Window.

`sys` is the short form of the command.

Examples

Display the nonvirtual systems for the model `sldemo_enginewc` using the Simulink debugger.

- 1 In the MATLAB Command Window, enter:

```
sldebug 'sldemo_enginewc'
```

The MATLAB command prompt `>>` changes to the Simulink debugger prompt (`sldebug @0`): `>>`.

- 2 Enter:

```
systems
```

The MATLAB Command Window displays the nonvirtual subsystems.

```
0 'sldemo_enginewc'  
2 'sldemo_enginewc/Compression'  
3 'sldemo_enginewc/Controller'  
4 'sldemo_enginewc/Throttle & Manifold/Throttle/TmpAtomicSubsysAtSwitchInport1'  
5 'sldemo_enginewc/valve timing/positive edge to dual edge conversion'
```

See Also

slis

Introduced before R2006a

tbreak

Set or clear time breakpoint

Syntax

```
tbreak
```

```
tb
```

```
tbreak t
```

Description

The `tbreak` command sets a breakpoint at the specified time step. If a breakpoint already exists at the specified time, `tbreak` clears the breakpoint. If you do not specify a time, `tbreak` toggles a breakpoint at the current time step.

Instead of `tbreak`, you can use the short form of `tb`, with or without `t`.

See Also

`bafter` | `break` | `nanbreak` | `rbreak` | `xbreak` | `zcbreak`

Topics

`ebreak`

Introduced before R2006a

trace

Display block's I/O each time block executes

Syntax

```
trace gcb  
trace s:b
```

```
tr gcb  
trace s:b
```

Arguments

`s:b` The block whose system index is `s` and block index is `b`.
`gcb` Current block.

Description

The `trace` command registers a block as a trace point. The debugger displays the I/O of each registered block each time the block executes.

See Also

`disp` | `probe` | `slist` | `strace` | `untrace`

Introduced before R2006a

undisp

Remove block from debugger's list of display points

Syntax

```
undisp gcb
```

```
und gcb
```

```
undisp s:b
```

```
und s:b
```

Arguments

`s:b` The block whose system index is `s` and block index is `b`.

`gcb` Current block.

Description

The `undisp` command removes the specified block from the debugger's list of display points.

See Also

`disp` | `slist`

Introduced before R2006a

untrace

Remove block from debugger's list of trace points

Syntax

```
untrace gcb
```

```
unt gcb
```

```
untrace s:b
```

```
unt s:b
```

Arguments

`s:b` The block whose system index is `s` and block index is `b`.

`gcb` Current block.

Description

The `untrace` command removes the specified block from the debugger's list of trace points.

See Also

`slist` | `trace`

Introduced before R2006a

where

Display current location of simulation in simulation loop

Syntax

```
where [detail]
```

```
w [detail]
```

Description

The `where` command displays the current location of the simulation in the simulation loop, for example,

```
sldebug @7): where
  0 >> vdp.Simulate
  1   >> vdp.Start
  2     >> RootSystem.Start 'vdp'
  7       >| 0:8 Sum.Start 'Sum'
```



The display consists of a list of simulation nodes with the last entry being the node that is about to be entered or exited. Each entry contains the following information:

- Method ID

The method ID identifies a specific invocation of a method.

- A symbol specifying its state:
 - `>>` (active)
 - `>|` (about to be entered)
 - `<|` (about to be exited)

- Name of the method invoked (e.g., `RootSystem.Start`)
- Name of the block or system on which the method is invoked (e.g., `Sum`)
- System and block ID (`sysIdx:blkIdx`) of the block on which the method is invoked

For example, `0:8` indicates that the specified method operates on block 8 of system 0.

`where detail`, where `detail` is any nonnegative integer, includes inactive nodes in the display.

```
0 >> vdp.Simulate
  1   >> vdp.Start
  2     >> RootSystem.Start 'vdp'
  3       0:4 Scope.Start 'Scope'
  4         0:5 Fcn.Start 'Fcn'
  5         0:6 Product.Start
'Product'
  6         0:7 Gain.Start 'Mu'
  7       >| 0:8 Sum.Start 'Sum'
```

See Also

`step`

Introduced before R2006a

xbreak

Break when debugger encounters step-size-limiting state

Syntax

```
xbreak
```

```
x
```

Description

The `xbreak` command pauses execution of the model when the debugger encounters a state that limits the size of the steps that the solver takes. If `xbreak` mode is already on, `xbreak` turns the mode off.

See Also

`bafter` | `break` | `nanbreak` | `rbreak` | `tbreak` | `zcbreak`

Topics

`ebreak`

Introduced before R2006a

zcbreak

Toggle breaking at nonsampled zero-crossing events

Syntax

```
zcbreak
```

```
zcb
```

Description

The `zcbreak` command causes the debugger to break when a nonsampled zero-crossing event occurs. If zero-crossing break mode is already on, `zcbreak` turns the mode off.

See Also

```
bafter | break | nanbreak | tbreak | xbreak | zclist
```

Introduced before R2006a

zclist

List blocks containing nonsampled zero crossings

Syntax

```
zclist
```

```
zcl
```

Description

The `zclist` command displays a list of blocks in which nonsampled zero crossings can occur. The command displays the list in the MATLAB Command Window.

See Also

`zcbreak`

Introduced before R2006a

Simulink Classes

eventData	Provide information about block method execution events
Simulink.AliasType	Create alias for signal and parameter data type
Simulink.Annotation	Specify properties of model annotation
Simulink.BlockCompDworkData	Provide postcompilation information about block's DWork vector
Simulink.BlockCompInputPortData	Provide postcompilation information about block input port
Simulink.BlockCompOutputPortData	Provide postcompilation information about block output port
Simulink.BlockData	Provide run-time information about block-related data, such as block parameters
Simulink.BlockPath	Fully specified Simulink block path
Simulink.BlockPortData	Describe block input or output port
Simulink.BlockPreCompInputPortData	Provide precompilation information about block input port
Simulink.BlockPreCompOutputPortData	Provide precompilation information about block output port
Simulink.CoderInfo	Specify information needed to generate code for signal or parameter data
Simulink.ConfigSetRef	Link model to configuration set stored independently of any model
Simulink.GlobalDataTransfer	Configure concurrent execution data transfers
Simulink.MDLInfo	Extract model file information without loading block diagram into memory
getDescription	Extract model file description without loading block diagram into memory
getMetadata	Extract model file metadata without loading block diagram into memory
Simulink.ModelAdvisor	Run Model Advisor from MATLAB file
5-2 Simulink.ModelDataLogs	Container for signal data logs of a model
Simulink.SimState.ModelSimState	Access SimState snapshot data

Simulink.ModelWorkspace

Describe model workspace

matlab.io.datastore.sd datastore class

Package: matlab.io.datastore

Datastore for Simulation Data Inspector signals

Description

A `matlab.io.datastore.sd datastore` object enables the Simulation Data Inspector to interact with large sets of data that are too large to fit in memory. An `sd datastore` object references the data for one signal.

Construction

`ds_element = dsrObj.getAsDatastore(arg)` creates the `sd datastore`, `ds_element`, for the signal in the `Simulink.sdi.Datasetref` object selected by the search criterion `arg`.

`ds_element = matlab.io.datastore.sd datastore(signalID)` creates the `sd datastore`, `ds_element` for the signal specified by `signalID`.

Input Arguments

arg — Element selection criterion

character vector | integer

Search criterion used to retrieve the element from the `Simulink.sdi.DatasetRef` object. For name based searches, specify `arg` as a character vector. For index based searches, `arg` is an integer, representing the index of the desired element.

Example: 'MySignal'

Example: 3

signalID — Numeric signal identifier

integer

Unique number identifying a signal.

Properties

Name — Signal name

character vector

Name of the signal specified as a character vector.

Example: 'My Signal'

signal — Simulink.sdi.Signal object

Simulink.sdi.Signal object

Simulink.sdi.Signal object associated with the sdi datastore.

Methods

Inherited Methods

hasdata	Determine if data is available to read
preview	Return subset of data from datastore
progress	Return percentage of data that you have read from a datastore
read	Read data in datastore
readall	Read all data in datastore
reset	Reset datastore to initial state

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create an sd datastore Object for a Signal

This example shows how to create a sd datastore object for a signal in a Simulink.sdi.DatasetRef object.

```
% Simulate model sldemo_fuelsys to create a run of logged signals
sim('sldemo_fuelsys')

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
run = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = run.getDatasetRef;

% Get the names of the elements in the object
names = run_DSRef.getElementNames
```

```
names =
```

```
10x1 cell array
```

```
{'EGO Fault Switch:1'      }
{'air_fuel_ratio'         }
{'Engine Speed Fault Switch:1' }
{'speed'                  }
{'MAP Fault Switch:1'     }
{'map'                    }
{'ego'                    }
{'Throttle Angle Fault Switch:1'}
{'throttle'               }
{'fuel'                   }
```

```
% Get sdatastore object for fuel signal  
fuel_ds = run_DSRef.getAsDatastore(10);
```

See Also

`Simulink.sdi.DatasetRef.getAsDatastore` |
`matlab.io.datastore.SimulationDatastore`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

matlab.io.datastore.SimulationDatastore class

Package: matlab.io.datastore

Datastore for inputs and outputs of Simulink models

Description

A `matlab.io.datastore.SimulationDatastore` object enables a Simulink model to interact with big data. You can load big data as simulation input and log big output data from a simulation. To simulate models with big data, you store the data in a MAT-file and refer to the data through a `SimulationDatastore` object. See “Work with Big Data for Simulations”.

A `SimulationDatastore` object refers to big simulation data (which a MAT-file stores) for one signal. If the MAT-file stores simulation data for a bus signal, a `SimulationDatastore` object refers to the data for one leaf signal element in the bus. You can use the datastore object to inspect and access the data and, through a parent object such as `Simulink.SimulationData.Signal`, simulate a Simulink model with the data.

To analyze the datastore data, you can use the methods and properties of the `SimulationDatastore` object as well as MATLAB tools such as the `tail` function. For more information about the MATLAB tools, see “Getting Started with Datastore” (MATLAB).

Construction

After you store big simulation data in a `Simulink.SimulationData.Dataset` object in a MAT-file, a signal element in the `Dataset` object points to the big data. To create a `matlab.io.datastore.SimulationDatastore` object that refers to the big data:

- 1 At the command prompt or in a script, create a `Simulink.SimulationData.DatasetRef` object that refers to the `Dataset` object in the MAT-file.

2 Use one of these techniques:

- Use one-based, curly-brace indexing (for example, `{1}`) to return an object that represents the target signal element, such as `Simulink.SimulationData.Signal` or `Simulink.SimulationData.State`. For example, for a `DatasetRef` object named `logsout_ref`, to create a `Signal` object that refers to the second signal element, use this code:

```
myLoggedSig = logsout_ref{2}
```

- Use the `getAsDatastore` method of the `DatasetRef` object to return an object that represents the target signal element. For more information, see `Simulink.SimulationData.DatasetRef.getAsDatastore`.

The `SimulationDatastore` object resides in the `Values` property of the returned object.

Properties

FileName — Name and path of file that contains big data

character vector

Name and path of the file that contains the big data, returned as a character vector. This property is read-only.

Data Types: `char`

NumSamples — Total number of samples (time steps) in the datastore

integer

Total number of samples (time steps) in the datastore, returned as an integer. The `readall` method extracts this many samples from the big data. This property is read-only.

Data Types: `uint64`

ReadSize — Amount of data to read at a time

100 (default) | scalar double

Amount of data to read at a time, in number of samples (time steps), specified as a scalar double. The `read` method extracts this many samples from the big data.

Data Types: `double`

Methods

<code>hasdata</code>	Determine if data is available to read
<code>preview</code>	Return subset of data from datastore
<code>progress</code>	Return percentage of data that you have read from a datastore
<code>read</code>	Read data in datastore
<code>readall</code>	Read all data in datastore
<code>reset</code>	Reset datastore to initial state

Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Limitations

- `SimulationDatastore` does not support using a parallel pool with Parallel Computing Toolbox installed. To analyze data using tall arrays or run MapReduce algorithms, set the global execution environment to be the local MATLAB session using `mapreducer`. Enter this code:

```
mapreducer(0)
```

For information about controlling parallel resources, see “Run mapreduce on a Parallel Pool” (Parallel Computing Toolbox).

- You cannot use a tall variable as simulation input data.

Examples

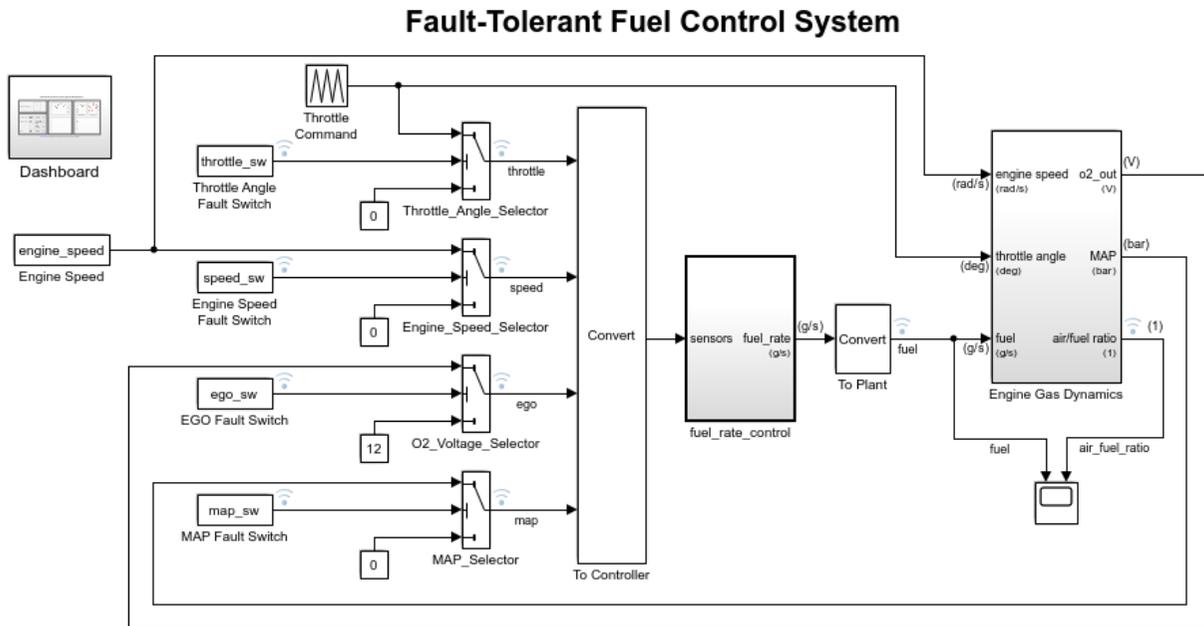
Inspect and Analyze Data in Simulation Dastore

This example shows how to log big data from a simulation and inspect and analyze portions of that data by interacting with a `matlab.io.datastore.SimulationDatastore` object.

Log Big Data from Model

Open the example model `sldemo_fuelsys`.

```
open_system('sldemo_fuelsys')
```



[Open the Dashboard](#) subsystem to simulate any combination of sensor failures.

Copyright 1990-2016 The MathWorks, Inc.

Select **Configuration Parameters > Data Import/Export > Log Dataset to file.**

```
set_param('sldemo_fuelsys', 'LoggingToFile', 'on')
```

Simulate the model.

```
sim('sldemo_fuelsys')
```

The MAT-file `out.mat` appears in your current folder. The file contains data for logged signals such as `fuel` (which is at the root level of the model).

At the command prompt, create a `DatasetRef` object that refers to the logging variable by name, `sldemo_fuelsys_output`.

```
DSRef = Simulink.SimulationData.DatasetRef('out.mat', 'sldemo_fuelsys_output');
```

Preview Big Data

Use curly braces (`{` and `}`) to extract the signal element `fuel`, which is the tenth element in `DSRef`, as a `Simulink.SimulationData.Signal` object that contains a `SimulationDatastore` object.

```
SimDataSig = DSRef{10};
```

To more easily interact with the `SimulationDatastore` object that resides in the `Values` property of the `Signal` object, store a handle in a variable named `DStore`.

```
DStore = SimDataSig.Values;
```

Use the `preview` method to inspect the first five samples of logged data for the `fuel` signal.

```
preview(DStore)
```

```
ans =
```

```
10x1 timetable
```

Time	Data
0 sec	1.209
0.00056199 sec	1.209
0.0033719 sec	1.209
0.01 sec	1.1729
0.02 sec	1.1409
0.03 sec	1.1124
0.04 sec	1.0873
0.05 sec	1.0652
0.055328 sec	1.0652

```
0.055328 sec    1.0652
```

Inspect Specific Sample

Inspect the 603rd sample of logged fuel data.

Set the `ReadSize` property of `DStore` to a number that, considering memory resources, your computer can tolerate. For example, set `ReadSize` to 200.

```
DStore.ReadSize = 200;
```

Read from the datastore three times. Each read operation advances the reading position by 200 samples.

```
read(DStore);  
read(DStore);  
read(DStore);
```

Now that you are very close to the 603rd sample, set `ReadSize` to a smaller number. For example, set `ReadSize` to 5.

```
DStore.ReadSize = 5;
```

Read from the datastore again.

```
read(DStore)
```

```
ans =
```

```
5x1 timetable
```

Time	Data
5.79 sec	1.6097
5.8 sec	1.6136
5.81 sec	1.6003
5.82 sec	1.5904
5.83 sec	1.5832

The third sample of read data is the 603rd sample in the datastore.

Inspect Earlier Sample

Inspect the 403rd sample of logged `fuel` data. Due to previous read operations, the datastore now reads starting from the 606th sample, so you must reset the datastore. Then, you can read from the first sample up to the 403rd sample.

Use the `reset` method to reset `DStore`.

```
reset(DStore);
```

Set `ReadSize` to 200 again.

```
DStore.ReadSize = 200;
```

Read from the datastore twice to advance the read position to the 401st sample.

```
read(DStore);
read(DStore);
```

Set `ReadSize` to 5 again.

```
DStore.ReadSize = 5;
```

Read from the datastore.

```
read(DStore)
```

```
ans =
```

```
5x1 timetable
```

Time	Data
3.85 sec	0.999
3.86 sec	0.99219
3.87 sec	0.98538
3.88 sec	0.97858
3.89 sec	0.97179

Extract Multiple Samples

Extract samples 1001 through 1020 (a total of 20 samples).

Reset the datastore.

```
reset(DStore)
```

Advance to sample 1001.

```
DStore.ReadSize = 200;
```

```
for i = 1:5  
    read(DStore);  
end
```

Prepare to extract 20 samples from the datastore.

```
DStore.ReadSize = 20;
```

Extract samples 1001 through 1020. Store the extracted data in a variable named `targetSamples`.

```
targetSamples = read(DStore)
```

```
targetSamples =
```

```
20x1 timetable
```

Time	Data
9.7 sec	1.5828
9.71 sec	1.5733
9.72 sec	1.5664
9.73 sec	1.5614
9.74 sec	1.5579
9.75 sec	1.5553
9.76 sec	1.5703
9.77 sec	1.582
9.78 sec	1.5913
9.79 sec	1.5988
9.8 sec	1.605
9.81 sec	1.6101
9.82 sec	1.6145
9.83 sec	1.6184
9.84 sec	1.6049
9.85 sec	1.595

```
9.86 sec    1.5877
9.87 sec    1.5824
9.88 sec    1.5785
9.89 sec    1.5757
```

Find Maximum Value of Data in Datastore

Reset the datastore.

```
reset(DStore)
```

Write a while loop, using the `hasdata` method, to incrementally analyze the data in chunks of 200 samples.

```
DStore.ReadSize = 200;
runningMax = [];
while hasdata(DStore)
    tt = read(DStore);
    rawChunk = tt.Data;
    runningMax = max([rawChunk; runningMax]);
end
```

Now, the variable `runningMax` stores the maximum value in the entire datastore.

```
runningMax
```

```
runningMax =
```

```
    1.6423
```

- “Work with Big Data for Simulations”

See Also

Topics

“Work with Big Data for Simulations”

Introduced in R2017a

hasdata

Class: matlab.io.datastore.SimulationDatastore

Package: matlab.io.datastore

Determine if data is available to read

Syntax

```
tf = hasdata(dst)
```

Description

`tf = hasdata(dst)` returns logical 1 (true) if there is data available to read from the datastore (matlab.io.datastore.SimulationDatastore object) specified by `dst`. Otherwise, it returns logical 0 (false).

Input Arguments

dst — Input datastore

matlab.io.datastore.SimulationDatastore object

Input datastore, specified as a matlab.io.datastore.SimulationDatastore object. To create a SimulationDatastore object, see matlab.io.datastore.SimulationDatastore.

Examples

Inspect and Analyze Data in Simulation Datastore

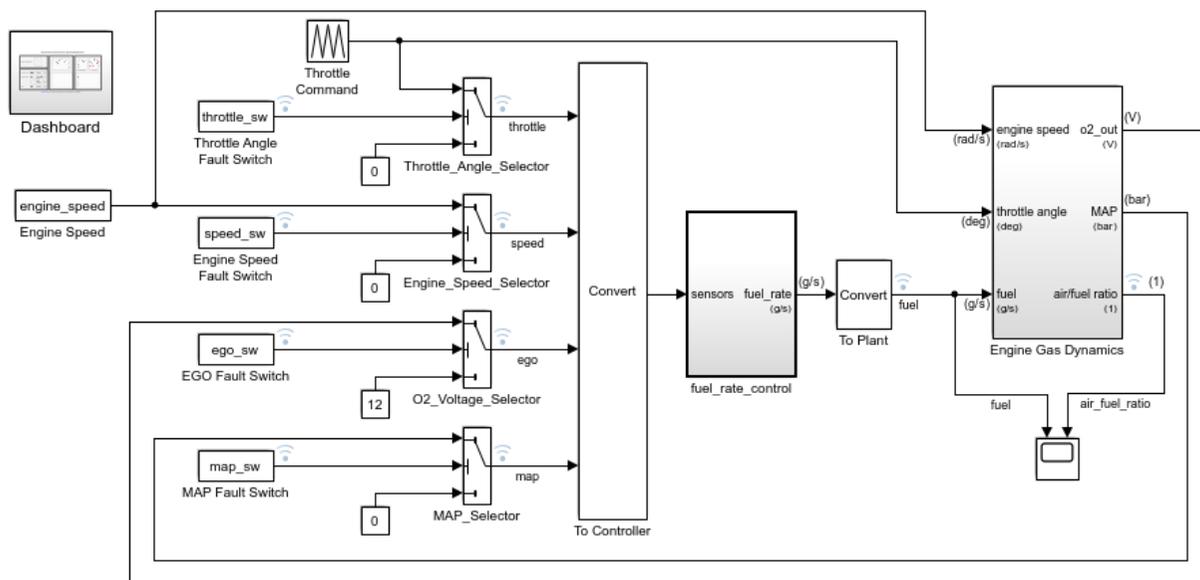
This example shows how to log big data from a simulation and inspect and analyze portions of that data by interacting with a matlab.io.datastore.SimulationDatastore object.

Log Big Data from Model

Open the example model `sldemo_fuelsys`.

```
open_system('sldemo_fuelsys')
```

Fault-Tolerant Fuel Control System



[Open the Dashboard](#) subsystem to simulate any combination of sensor failures.

Copyright 1990-2016 The MathWorks, Inc.

Select **Configuration Parameters > Data Import/Export > Log Dataset to file**.

```
set_param('sldemo_fuelsys', 'LoggingToFile', 'on')
```

Simulate the model.

```
sim('sldemo_fuelsys')
```

The MAT-file `out.mat` appears in your current folder. The file contains data for logged signals such as `fuel` (which is at the root level of the model).

At the command prompt, create a `DatasetRef` object that refers to the logging variable by name, `sldemo_fuelsys_output`.

```
DSRef = Simulink.SimulationData.DatasetRef('out.mat', 'sldemo_fuelsys_output');
```

Preview Big Data

Use curly braces (`{` and `}`) to extract the signal element `fuel`, which is the tenth element in `DSRef`, as a `Simulink.SimulationData.Signal` object that contains a `SimulationDatastore` object.

```
SimDataSig = DSRef{10};
```

To more easily interact with the `SimulationDatastore` object that resides in the `Values` property of the `Signal` object, store a handle in a variable named `DStore`.

```
DStore = SimDataSig.Values;
```

Use the `preview` method to inspect the first five samples of logged data for the `fuel` signal.

```
preview(DStore)
```

```
ans =
```

```
10x1 timetable
```

Time	Data
0 sec	1.209
0.00056199 sec	1.209
0.0033719 sec	1.209
0.01 sec	1.1729
0.02 sec	1.1409
0.03 sec	1.1124
0.04 sec	1.0873
0.05 sec	1.0652
0.055328 sec	1.0652
0.055328 sec	1.0652

Inspect Specific Sample

Inspect the 603rd sample of logged `fuel` data.

Set the `ReadSize` property of `DStore` to a number that, considering memory resources, your computer can tolerate. For example, set `ReadSize` to 200.

```
DStore.ReadSize = 200;
```

Read from the datastore three times. Each read operation advances the reading position by 200 samples.

```
read(DStore);
read(DStore);
read(DStore);
```

Now that you are very close to the 603rd sample, set `ReadSize` to a smaller number. For example, set `ReadSize` to 5.

```
DStore.ReadSize = 5;
```

Read from the datastore again.

```
read(DStore)
```

```
ans =
```

```
5x1 timetable
```

Time	Data
5.79 sec	1.6097
5.8 sec	1.6136
5.81 sec	1.6003
5.82 sec	1.5904
5.83 sec	1.5832

The third sample of read data is the 603rd sample in the datastore.

Inspect Earlier Sample

Inspect the 403rd sample of logged `fuel` data. Due to previous read operations, the datastore now reads starting from the 606th sample, so you must reset the datastore. Then, you can read from the first sample up to the 403rd sample.

Use the `reset` method to reset `DStore`.

```
reset(DStore);
```

Set ReadSize to 200 again.

```
DStore.ReadSize = 200;
```

Read from the datastore twice to advance the read position to the 401st sample.

```
read(DStore);  
read(DStore);
```

Set ReadSize to 5 again.

```
DStore.ReadSize = 5;
```

Read from the datastore.

```
read(DStore)
```

```
ans =
```

```
5x1 timetable
```

Time	Data
3.85 sec	0.999
3.86 sec	0.99219
3.87 sec	0.98538
3.88 sec	0.97858
3.89 sec	0.97179

Extract Multiple Samples

Extract samples 1001 through 1020 (a total of 20 samples).

Reset the datastore.

```
reset(DStore)
```

Advance to sample 1001.

```
DStore.ReadSize = 200;
```

```
for i = 1:5
    read(DStore);
end
```

Prepare to extract 20 samples from the datastore.

```
DStore.ReadSize = 20;
```

Extract samples 1001 through 1020. Store the extracted data in a variable named targetSamples.

```
targetSamples = read(DStore)
```

```
targetSamples =
```

```
20x1 timetable
```

Time	Data
9.7 sec	1.5828
9.71 sec	1.5733
9.72 sec	1.5664
9.73 sec	1.5614
9.74 sec	1.5579
9.75 sec	1.5553
9.76 sec	1.5703
9.77 sec	1.582
9.78 sec	1.5913
9.79 sec	1.5988
9.8 sec	1.605
9.81 sec	1.6101
9.82 sec	1.6145
9.83 sec	1.6184
9.84 sec	1.6049
9.85 sec	1.595
9.86 sec	1.5877
9.87 sec	1.5824
9.88 sec	1.5785
9.89 sec	1.5757

Find Maximum Value of Data in Datastore

Reset the datastore.

```
reset(DStore)
```

Write a `while` loop, using the `hasdata` method, to incrementally analyze the data in chunks of 200 samples.

```
DStore.ReadSize = 200;
runningMax = [];
while hasdata(DStore)
    tt = read(DStore);
    rawChunk = tt.Data;
    runningMax = max([rawChunk; runningMax]);
end
```

Now, the variable `runningMax` stores the maximum value in the entire datastore.

```
runningMax
```

```
runningMax =
```

```
    1.6423
```

- “Work with Big Data for Simulations”

See Also

Topics

“Work with Big Data for Simulations”

Introduced in R2017a

preview

Class: matlab.io.datastore.SimulationDatastore

Package: matlab.io.datastore

Return subset of data from datastore

Syntax

```
data = preview(dst)
```

Description

`data = preview(dst)` returns a subset of data from datastore (`matlab.io.datastore.SimulationDatastore` object) `dst` without changing its current read position. `preview` returns only the first ten samples (time steps) of data in the datastore. Use this method to quickly inspect and verify that the data appears as you expect.

Input Arguments

dst — Input datastore

`matlab.io.datastore.SimulationDatastore` object

Input datastore, specified as a `matlab.io.datastore.SimulationDatastore` object. To create a `SimulationDatastore` object, see `matlab.io.datastore.SimulationDatastore`.

Output Arguments

data — Subset of data

`timetable` object

Subset of data, returned as a timetable object. For information about timetable, see “Timetables” (MATLAB).

Examples

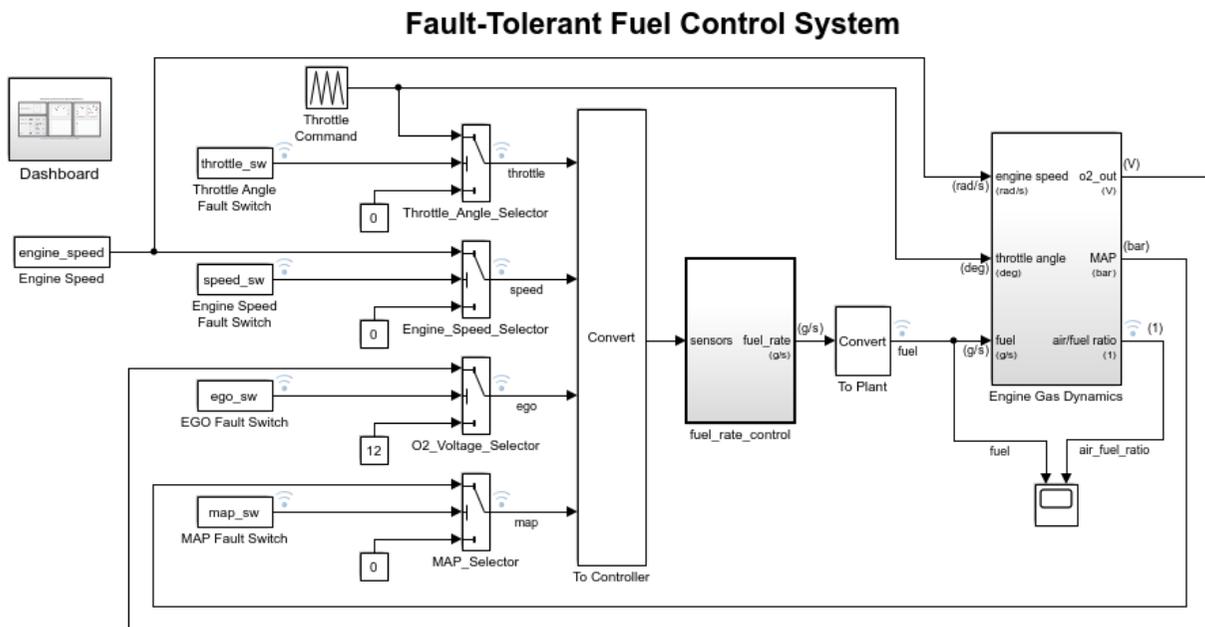
Inspect and Analyze Data in Simulation Datastore

This example shows how to log big data from a simulation and inspect and analyze portions of that data by interacting with a `matlab.io.datastore.SimulationDatastore` object.

Log Big Data from Model

Open the example model `sldemo_fuelsys`.

```
open_system('sldemo_fuelsys')
```



[Open the Dashboard](#) subsystem to simulate any combination of sensor failures.

Copyright 1990-2016 The MathWorks, Inc.

Select **Configuration Parameters > Data Import/Export > Log Dataset to file.**

```
set_param('sldemo_fuelsys','LoggingToFile','on')
```

Simulate the model.

```
sim('sldemo_fuelsys')
```

The MAT-file `out.mat` appears in your current folder. The file contains data for logged signals such as `fuel` (which is at the root level of the model).

At the command prompt, create a `DatasetRef` object that refers to the logging variable by name, `sldemo_fuelsys_output`.

```
DSRef = Simulink.SimulationData.DatasetRef('out.mat','sldemo_fuelsys_output');
```

Preview Big Data

Use curly braces (`{}` and `}`) to extract the signal element `fuel`, which is the tenth element in `DSRef`, as a `Simulink.SimulationData.Signal` object that contains a `SimulationDatastore` object.

```
SimDataSig = DSRef{10};
```

To more easily interact with the `SimulationDatastore` object that resides in the `Values` property of the `Signal` object, store a handle in a variable named `DStore`.

```
DStore = SimDataSig.Values;
```

Use the `preview` method to inspect the first five samples of logged data for the `fuel` signal.

```
preview(DStore)
```

```
ans =
```

```
10x1 timetable
```

Time	Data
0 sec	1.209
0.00056199 sec	1.209

```
0.0033719 sec      1.209
0.01 sec           1.1729
0.02 sec           1.1409
0.03 sec           1.1124
0.04 sec           1.0873
0.05 sec           1.0652
0.055328 sec      1.0652
0.055328 sec      1.0652
```

Inspect Specific Sample

Inspect the 603rd sample of logged fuel data.

Set the `ReadSize` property of `DStore` to a number that, considering memory resources, your computer can tolerate. For example, set `ReadSize` to 200.

```
DStore.ReadSize = 200;
```

Read from the datastore three times. Each read operation advances the reading position by 200 samples.

```
read(DStore);
read(DStore);
read(DStore);
```

Now that you are very close to the 603rd sample, set `ReadSize` to a smaller number. For example, set `ReadSize` to 5.

```
DStore.ReadSize = 5;
```

Read from the datastore again.

```
read(DStore)
```

```
ans =
```

```
5x1 timetable
```

Time	Data
5.79 sec	1.6097
5.8 sec	1.6136

```

5.81 sec    1.6003
5.82 sec    1.5904
5.83 sec    1.5832

```

The third sample of read data is the 603rd sample in the datastore.

Inspect Earlier Sample

Inspect the 403rd sample of logged `fuel` data. Due to previous read operations, the datastore now reads starting from the 606th sample, so you must reset the datastore. Then, you can read from the first sample up to the 403rd sample.

Use the `reset` method to reset `DStore`.

```
reset(DStore);
```

Set `ReadSize` to 200 again.

```
DStore.ReadSize = 200;
```

Read from the datastore twice to advance the read position to the 401st sample.

```
read(DStore);
read(DStore);
```

Set `ReadSize` to 5 again.

```
DStore.ReadSize = 5;
```

Read from the datastore.

```
read(DStore)
```

```
ans =
```

```
5x1 timetable
```

Time	Data
3.85 sec	0.999
3.86 sec	0.99219
3.87 sec	0.98538

```
3.88 sec    0.97858
3.89 sec    0.97179
```

Extract Multiple Samples

Extract samples 1001 through 1020 (a total of 20 samples).

Reset the datastore.

```
reset(DStore)
```

Advance to sample 1001.

```
DStore.ReadSize = 200;
```

```
for i = 1:5
    read(DStore);
end
```

Prepare to extract 20 samples from the datastore.

```
DStore.ReadSize = 20;
```

Extract samples 1001 through 1020. Store the extracted data in a variable named `targetSamples`.

```
targetSamples = read(DStore)
```

```
targetSamples =
```

```
20x1 timetable
```

Time	Data
9.7 sec	1.5828
9.71 sec	1.5733
9.72 sec	1.5664
9.73 sec	1.5614
9.74 sec	1.5579
9.75 sec	1.5553
9.76 sec	1.5703
9.77 sec	1.582

```
9.78 sec    1.5913
9.79 sec    1.5988
9.8 sec     1.605
9.81 sec    1.6101
9.82 sec    1.6145
9.83 sec    1.6184
9.84 sec    1.6049
9.85 sec    1.595
9.86 sec    1.5877
9.87 sec    1.5824
9.88 sec    1.5785
9.89 sec    1.5757
```

Find Maximum Value of Data in Datastore

Reset the datastore.

```
reset(DStore)
```

Write a while loop, using the `hasdata` method, to incrementally analyze the data in chunks of 200 samples.

```
DStore.ReadSize = 200;
runningMax = [];
while hasdata(DStore)
    tt = read(DStore);
    rawChunk = tt.Data;
    runningMax = max([rawChunk; runningMax]);
end
```

Now, the variable `runningMax` stores the maximum value in the entire datastore.

```
runningMax
```

```
runningMax =
```

```
1.6423
```

- “Work with Big Data for Simulations”

See Also

Topics

“Work with Big Data for Simulations”

Introduced in R2017a

progress

Class: matlab.io.datastore.SimulationDatastore

Package: matlab.io.datastore

Return percentage of data that you have read from a datastore

Syntax

```
p = progress(dst)
```

Description

`p = progress(dst)` returns the percentage, as a number between 0 and 1, of the data that you have read from a datastore (`matlab.io.datastore.SimulationDatastore` object). For example, a return value of 0.55 means you have read 55% of the data. Use the `progress` method and the `NumSamples` property to determine the current read position.

You read data from a `SimulationDatastore` object by using the `read` method.

Input Arguments

dst — Input datastore

`matlab.io.datastore.SimulationDatastore` object

Input datastore, specified as a `matlab.io.datastore.SimulationDatastore` object.

To create a `SimulationDatastore` object, see

`matlab.io.datastore.SimulationDatastore`.

Output Arguments

p — Percentage of data that you have read from the datastore

scalar double

Percentage of data that you have read from the datastore, returned as a scalar double.

Data Types: `double`

See Also

Topics

“Work with Big Data for Simulations”

Introduced in R2017a

read

Class: `matlab.io.datastore.SimulationDatastore`

Package: `matlab.io.datastore`

Read data in datastore

Syntax

```
data = read(dst)
[data,info] = read(dst)
```

Description

`data = read(dst)` returns data from a datastore (`matlab.io.datastore.SimulationDatastore` object). Subsequent calls to the `read` function continue reading from the endpoint of the previous call. Use the `ReadSize` property of the `SimulationDatastore` object to specify the amount of data, in samples (time steps), to read at a time. Use the `progress` method and the `NumSamples` property to determine the current read position.

`[data,info] = read(dst)` also returns information about the extracted data in `info`.

Input Arguments

dst — Input datastore

`matlab.io.datastore.SimulationDatastore` object

Input datastore, specified as a `matlab.io.datastore.SimulationDatastore` object. To create a `SimulationDatastore` object, see `matlab.io.datastore.SimulationDatastore`.

Output Arguments

data — Output data

`timetable` object

Output data, returned as a `timetable` object. For information about `timetable`, see “Timetables” (MATLAB).

info — Information about read data

structure array

Information about read data, returned as a structure. The structure has one field, `FileName`, which is a fully resolved path containing the path string, the name of the file, and the file extension.

Examples

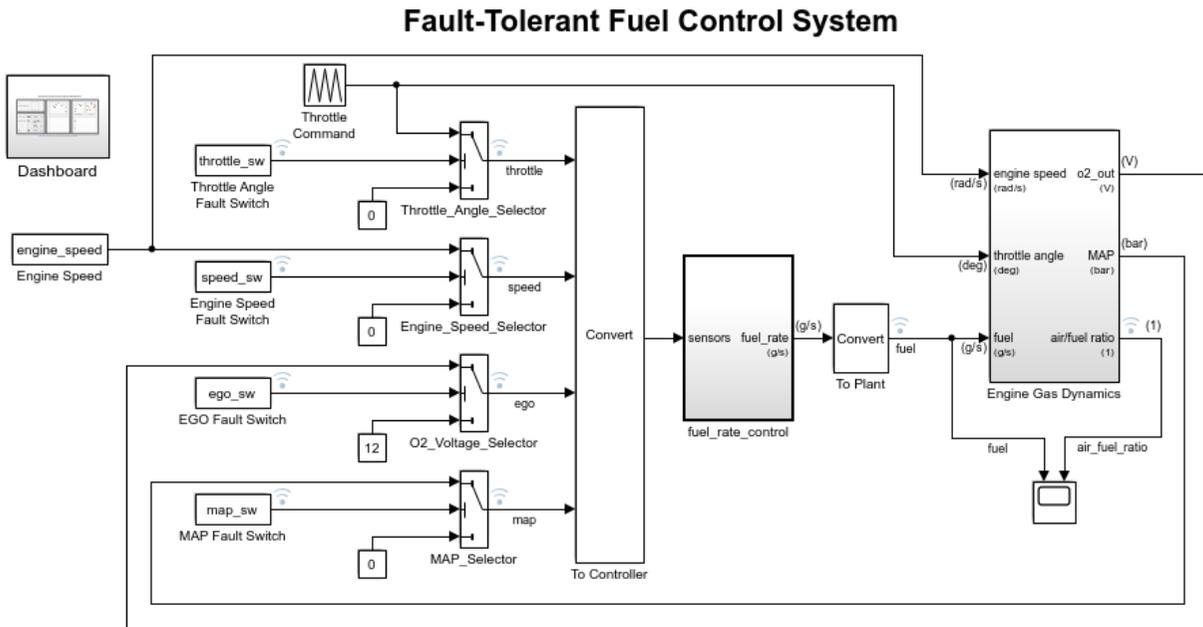
Inspect and Analyze Data in Simulation Datastore

This example shows how to log big data from a simulation and inspect and analyze portions of that data by interacting with a `matlab.io.datastore.SimulationDatastore` object.

Log Big Data from Model

Open the example model `sldemo_fuelsys`.

```
open_system('sldemo_fuelsys')
```



[Open the Dashboard](#) subsystem to simulate any combination of sensor failures.

Copyright 1990-2016 The MathWorks, Inc.

Select **Configuration Parameters > Data Import/Export > Log Dataset to file.**

```
set_param('sldemo_fuelsys', 'LoggingToFile', 'on')
```

Simulate the model.

```
sim('sldemo_fuelsys')
```

The MAT-file `out.mat` appears in your current folder. The file contains data for logged signals such as `fuel` (which is at the root level of the model).

At the command prompt, create a `DatasetRef` object that refers to the logging variable by name, `sldemo_fuelsys_output`.

```
DSRef = Simulink.SimulationData.DatasetRef('out.mat', 'sldemo_fuelsys_output');
```

Preview Big Data

Use curly braces (`{` and `}`) to extract the signal element `fuel`, which is the tenth element in `DSRef`, as a `Simulink.SimulationData.Signal` object that contains a `SimulationDatastore` object.

```
SimDataSig = DSRef{10};
```

To more easily interact with the `SimulationDatastore` object that resides in the `Values` property of the `Signal` object, store a handle in a variable named `DStore`.

```
DStore = SimDataSig.Values;
```

Use the `preview` method to inspect the first five samples of logged data for the `fuel` signal.

```
preview(DStore)
```

```
ans =
```

```
10x1 timetable
```

Time	Data
0 sec	1.209
0.00056199 sec	1.209
0.0033719 sec	1.209
0.01 sec	1.1729
0.02 sec	1.1409
0.03 sec	1.1124
0.04 sec	1.0873
0.05 sec	1.0652
0.055328 sec	1.0652
0.055328 sec	1.0652

Inspect Specific Sample

Inspect the 603rd sample of logged `fuel` data.

Set the `ReadSize` property of `DStore` to a number that, considering memory resources, your computer can tolerate. For example, set `ReadSize` to 200.

```
DStore.ReadSize = 200;
```

Read from the datastore three times. Each read operation advances the reading position by 200 samples.

```
read(DStore);
read(DStore);
read(DStore);
```

Now that you are very close to the 603rd sample, set `ReadSize` to a smaller number. For example, set `ReadSize` to 5.

```
DStore.ReadSize = 5;
```

Read from the datastore again.

```
read(DStore)
```

```
ans =
```

```
5x1 timetable
```

Time	Data
5.79 sec	1.6097
5.8 sec	1.6136
5.81 sec	1.6003
5.82 sec	1.5904
5.83 sec	1.5832

The third sample of read data is the 603rd sample in the datastore.

Inspect Earlier Sample

Inspect the 403rd sample of logged `fuel` data. Due to previous read operations, the datastore now reads starting from the 606th sample, so you must reset the datastore. Then, you can read from the first sample up to the 403rd sample.

Use the `reset` method to reset `DStore`.

```
reset(DStore);
```

Set ReadSize to 200 again.

```
DStore.ReadSize = 200;
```

Read from the datastore twice to advance the read position to the 401st sample.

```
read(DStore);  
read(DStore);
```

Set ReadSize to 5 again.

```
DStore.ReadSize = 5;
```

Read from the datastore.

```
read(DStore)
```

```
ans =
```

```
5x1 timetable
```

Time	Data
3.85 sec	0.999
3.86 sec	0.99219
3.87 sec	0.98538
3.88 sec	0.97858
3.89 sec	0.97179

Extract Multiple Samples

Extract samples 1001 through 1020 (a total of 20 samples).

Reset the datastore.

```
reset(DStore)
```

Advance to sample 1001.

```
DStore.ReadSize = 200;
```

```
for i = 1:5
```

```

    read(DStore);
end

```

Prepare to extract 20 samples from the datastore.

```
DStore.ReadSize = 20;
```

Extract samples 1001 through 1020. Store the extracted data in a variable named targetSamples.

```
targetSamples = read(DStore)
```

```
targetSamples =
```

```
20x1 timetable
```

Time	Data
-----	-----
9.7 sec	1.5828
9.71 sec	1.5733
9.72 sec	1.5664
9.73 sec	1.5614
9.74 sec	1.5579
9.75 sec	1.5553
9.76 sec	1.5703
9.77 sec	1.582
9.78 sec	1.5913
9.79 sec	1.5988
9.8 sec	1.605
9.81 sec	1.6101
9.82 sec	1.6145
9.83 sec	1.6184
9.84 sec	1.6049
9.85 sec	1.595
9.86 sec	1.5877
9.87 sec	1.5824
9.88 sec	1.5785
9.89 sec	1.5757

Find Maximum Value of Data in Datastore

Reset the datastore.

```
reset(DStore)
```

Write a while loop, using the `hasdata` method, to incrementally analyze the data in chunks of 200 samples.

```
DStore.ReadSize = 200;
runningMax = [];
while hasdata(DStore)
    tt = read(DStore);
    rawChunk = tt.Data;
    runningMax = max([rawChunk; runningMax]);
end
```

Now, the variable `runningMax` stores the maximum value in the entire datastore.

```
runningMax
```

```
runningMax =
```

```
    1.6423
```

- “Work with Big Data for Simulations”

See Also

Topics

“Work with Big Data for Simulations”

Introduced in R2017a

readall

Class: matlab.io.datastore.SimulationDatastore

Package: matlab.io.datastore

Read all data in datastore

Syntax

```
data = readall(dst)
```

Description

`data = readall(dst)` returns all the data in the datastore (matlab.io.datastore.SimulationDatastore object) specified by `dst`.

If all the data in the datastore does not fit in memory, `readall` returns an error. To determine how many samples (time steps) a datastore holds, inspect the `NumSamples` property of the `SimulationDatastore` object.

Input Arguments

dst — Input datastore

matlab.io.datastore.SimulationDatastore object

Input datastore, specified as a matlab.io.datastore.SimulationDatastore object. To create a SimulationDatastore object, see matlab.io.datastore.SimulationDatastore.

Output Arguments

data — All data in the datastore

timetable object

All data in the datastore, returned as a `timetable` object. For information about `timetable`, see “Timetables” (MATLAB).

See Also

Topics

“Work with Big Data for Simulations”

Introduced in R2017a

reset

Class: `matlab.io.datastore.SimulationDatastore`

Package: `matlab.io.datastore`

Reset datastore to initial state

Syntax

```
reset(dst)
```

Description

`reset(dst)` sets the read position of the datastore (`matlab.io.datastore.SimulationDatastore` object) specified by `dst` to the first sample in the datastore. Use `reset` to reread data from a datastore. You read from a datastore by using the `read` method.

Input Arguments

dst — Input datastore

`matlab.io.datastore.SimulationDatastore` object

Input datastore, specified as a `matlab.io.datastore.SimulationDatastore` object. To create a `SimulationDatastore` object, see `matlab.io.datastore.SimulationDatastore`.

Examples

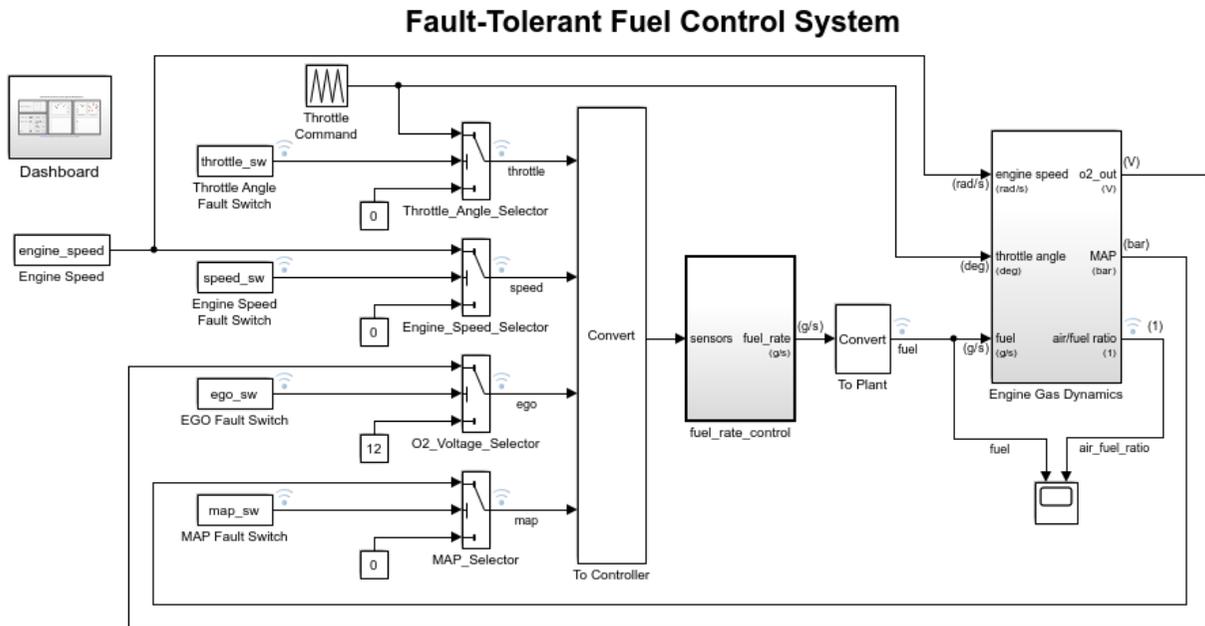
Inspect and Analyze Data in Simulation Datastore

This example shows how to log big data from a simulation and inspect and analyze portions of that data by interacting with a `matlab.io.datastore.SimulationDatastore` object.

Log Big Data from Model

Open the example model `sldemo_fuelsys`.

```
open_system('sldemo_fuelsys')
```



[Open the Dashboard](#) subsystem to simulate any combination of sensor failures.

Copyright 1990-2016 The MathWorks, Inc.

Select **Configuration Parameters > Data Import/Export > Log Dataset to file**.

```
set_param('sldemo_fuelsys', 'LoggingToFile', 'on')
```

Simulate the model.

```
sim('sldemo_fuelsys')
```

The MAT-file `out.mat` appears in your current folder. The file contains data for logged signals such as `fuel` (which is at the root level of the model).

At the command prompt, create a `DatasetRef` object that refers to the logging variable by name, `sldemo_fuelsys_output`.

```
DSRef = Simulink.SimulationData.DatasetRef('out.mat', 'sldemo_fuelsys_output');
```

Preview Big Data

Use curly braces (`{` and `}`) to extract the signal element `fuel`, which is the tenth element in `DSRef`, as a `Simulink.SimulationData.Signal` object that contains a `SimulationDatastore` object.

```
SimDataSig = DSRef{10};
```

To more easily interact with the `SimulationDatastore` object that resides in the `Values` property of the `Signal` object, store a handle in a variable named `DStore`.

```
DStore = SimDataSig.Values;
```

Use the `preview` method to inspect the first five samples of logged data for the `fuel` signal.

```
preview(DStore)
```

```
ans =
```

```
10x1 timetable
```

Time	Data
-----	-----
0 sec	1.209
0.00056199 sec	1.209
0.0033719 sec	1.209
0.01 sec	1.1729
0.02 sec	1.1409
0.03 sec	1.1124
0.04 sec	1.0873
0.05 sec	1.0652
0.055328 sec	1.0652

```
0.055328 sec    1.0652
```

Inspect Specific Sample

Inspect the 603rd sample of logged fuel data.

Set the `ReadSize` property of `DStore` to a number that, considering memory resources, your computer can tolerate. For example, set `ReadSize` to 200.

```
DStore.ReadSize = 200;
```

Read from the datastore three times. Each read operation advances the reading position by 200 samples.

```
read(DStore);  
read(DStore);  
read(DStore);
```

Now that you are very close to the 603rd sample, set `ReadSize` to a smaller number. For example, set `ReadSize` to 5.

```
DStore.ReadSize = 5;
```

Read from the datastore again.

```
read(DStore)
```

```
ans =
```

```
5x1 timetable
```

Time	Data
5.79 sec	1.6097
5.8 sec	1.6136
5.81 sec	1.6003
5.82 sec	1.5904
5.83 sec	1.5832

The third sample of read data is the 603rd sample in the datastore.

Inspect Earlier Sample

Inspect the 403rd sample of logged `fuel` data. Due to previous read operations, the datastore now reads starting from the 606th sample, so you must reset the datastore. Then, you can read from the first sample up to the 403rd sample.

Use the `reset` method to reset `DStore`.

```
reset(DStore);
```

Set `ReadSize` to 200 again.

```
DStore.ReadSize = 200;
```

Read from the datastore twice to advance the read position to the 401st sample.

```
read(DStore);
read(DStore);
```

Set `ReadSize` to 5 again.

```
DStore.ReadSize = 5;
```

Read from the datastore.

```
read(DStore)
```

```
ans =
```

```
5x1 timetable
```

Time	Data
3.85 sec	0.999
3.86 sec	0.99219
3.87 sec	0.98538
3.88 sec	0.97858
3.89 sec	0.97179

Extract Multiple Samples

Extract samples 1001 through 1020 (a total of 20 samples).

Reset the datastore.

```
reset(DStore)
```

Advance to sample 1001.

```
DStore.ReadSize = 200;
```

```
for i = 1:5  
    read(DStore);  
end
```

Prepare to extract 20 samples from the datastore.

```
DStore.ReadSize = 20;
```

Extract samples 1001 through 1020. Store the extracted data in a variable named `targetSamples`.

```
targetSamples = read(DStore)
```

```
targetSamples =
```

```
20x1 timetable
```

Time	Data
9.7 sec	1.5828
9.71 sec	1.5733
9.72 sec	1.5664
9.73 sec	1.5614
9.74 sec	1.5579
9.75 sec	1.5553
9.76 sec	1.5703
9.77 sec	1.582
9.78 sec	1.5913
9.79 sec	1.5988
9.8 sec	1.605
9.81 sec	1.6101
9.82 sec	1.6145
9.83 sec	1.6184
9.84 sec	1.6049
9.85 sec	1.595

```
9.86 sec    1.5877
9.87 sec    1.5824
9.88 sec    1.5785
9.89 sec    1.5757
```

Find Maximum Value of Data in Datastore

Reset the datastore.

```
reset(DStore)
```

Write a while loop, using the `hasdata` method, to incrementally analyze the data in chunks of 200 samples.

```
DStore.ReadSize = 200;
runningMax = [];
while hasdata(DStore)
    tt = read(DStore);
    rawChunk = tt.Data;
    runningMax = max([rawChunk; runningMax]);
end
```

Now, the variable `runningMax` stores the maximum value in the entire datastore.

```
runningMax
```

```
runningMax =
```

```
    1.6423
```

- “Work with Big Data for Simulations”

See Also

Topics

“Work with Big Data for Simulations”

Introduced in R2017a

eventData

Provide information about block method execution events

Description

Simulink software creates an instance of this class when a block method execution event occurs during simulation and passes it to any listeners registered for the event (see `add_exec_event_listener`). The instance specifies the type of event that occurred and the block whose method execution triggered the event. See “Access Block Data During Simulation” for more information.

Parent

None

Children

None

Property Summary

Name	Description
“Type” on page 5-51	Type of method execution event that occurred.
“Source” on page 5-51	Block that triggered the event.

Properties

Type

Type of method execution event that occurred. Possible values are:

event	Occurs...
'PreOutputs'	Before a block's Outputs method executes.
'PostOutputs'	After a block's Outputs method executes.
'PreUpdate'	Before a block's Update method executes.
'PostUpdate'	After a block's Update method executes.
'PreDerivatives'	Before a block's Derivatives method executes.
'PostDerivatives'	After a block's Derivatives method executes.

character vector

RO

Source

Block that triggered the event

`Simulink.RunTimeBlock`

RO

Introduced in R2009b

LibraryBrowser.LibraryBrowser2 class

Package: LibraryBrowser

Simulink Library Browser

Description

Programmatically display, hide, size, and position the Simulink Library Browser.

Construction

```
lb = LibraryBrowser.LibraryBrowser2
```

Properties

IsOnTop — Always put library window on top

0 (default) | 1

Always put library window on top of other Simulink Editor windows, specified as 1 for always on top.

Example: `lb.IsOnTop = 1`

Methods

Method	Meaning	Example
refresh	Update the library browser display with changes that affect the library browser. Examples include adding a library to the library browser, removing a library from the library browser, and changes to your custom libraries, slblocks function, or sl_customization.m file. In general, refresh the library browser when you have made any changes that affect libraries on your MATLAB path that are registered in the library browser. .	<pre>lb = LibraryBrowser.LibraryBrowser2; refresh(lb)</pre>
show	Display the library browser.	<pre>lb = LibraryBrowser.LibraryBrowser2; show(lb)</pre>
hide	Hide the library browser.	<pre>lb = LibraryBrowser.LibraryBrowser2; hide(lb)</pre>
getPosition	Get the position of the library browser. Returned as four integers, in pixels: upper-left x coordinate, upper-left y coordinate, width, and depth.	<pre>lb = LibraryBrowser.LibraryBrowser2; getPosition(lb) ans = 50 279 600 600</pre>
setPosition	Set the position of the library browser. Use an array of integers, in pixels: upper-left x coordinate, upper-left y coordinate, width, and depth.	<pre>lb = LibraryBrowser.LibraryBrowser2; setPosition(lb,[70 250 500 500])</pre>

See Also

Topics

“Customize Library Browser Appearance”

“Registering Customizations”

Introduced in R2016b

allowModelReferenceDiscreteSampleTimeInheritanceImpl

Model reference sample time inheritance status for discrete sample times

Syntax

```
flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)
```

Description

`flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)` specifies whether a System object in a reference model is allowed to inherit the sample time of the parent model. Use this method only for System objects that use discrete sample time and are intended for inclusion in Simulink via the MATLAB System block.

During model compilation, Simulink sets the model reference sample time inheritance before the System object `setupImpl` method is called.

Note You must set `Access = protected` for this method.

Input Arguments

obj

System object handle

Output Arguments

flag

Flag indicating whether model reference discrete sample time inheritance is allowed for the MATLAB System block containing the System object, returned as a logical value.

The default value for this argument depends on the number of inputs to the System object. To use the default value, you do not need to include this method in your System object class definition file.

Number of System object Inputs	Default Value and Override Effects
No inputs	<p>Default: <code>false</code> — Model reference discrete sample time inheritance is not allowed.</p> <p>If your System object uses discrete sample time in its algorithm, override the default by returning <code>true</code> from <code>allowModelReferenceDiscreteSampleTimeInheritanceImpl</code>.</p>
One or more inputs	<p>Default: <code>true</code> — If no other Simulink constraint prevents it, model reference sample time inheritance is allowed.</p> <p>If your System object does not use sample time in its algorithm, override the default by returning <code>false</code> from <code>allowModelReferenceDiscreteSampleTimeInheritanceImpl</code>.</p>

Examples

Set Sample Time Inheritance for System Object

For a System object that has one or more inputs, to disallow model reference discrete sample time inheritance for that object, set the sample time inheritance to `false`. Include this code in your class definition file for the object.

```
methods (Access = protected)
    function flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(~)
        flag = false;
    end
end
```

- “Set Model Reference Discrete Sample Time Inheritance”
- “Overview of Model Referencing”
- “Inherit Sample Times for Model Referencing”

See Also

`matlab.System`

Topics

“Set Model Reference Discrete Sample Time Inheritance”

“Overview of Model Referencing”

“Inherit Sample Times for Model Referencing”

getInputNamesImpl

Names of MATLAB System block input ports

Syntax

```
[name1,name2,...] = getInputNamesImpl(obj)
```

Description

[name1,name2,...] = getInputNamesImpl(obj) specifies the names of the input ports of the System object on a MATLAB System block. The number of returned input names matches the number of inputs returned by the `getNumInputs` method. If you change a property value that changes the number of inputs, the names of those inputs also change.

`getInputNamesImpl` is called by the `getInputNames` method by the MATLAB System block.

Note You must set `Access = protected` for this method.

Input Arguments

obj

System object

Output Arguments

name1,name2,...

Names of the inputs for the specified object, returned as character vectors

Default: empty character vector

Examples

Specify Input Port Name

Specify in your class definition file the names of two input ports as 'upper' and 'lower'.

```
methods (Access = protected)
    function varargout = getInputNamesImpl(obj)
        numInputs = getNumInputs(obj);
        varargout = cell(1,numInputs);
        varargout{1} = 'upper';
        if numInputs > 1
            varargout{2} = 'lower';
        end
    end
end
end
```

- “Specify Input and Output Names”

See Also

[getNumInputsImpl](#) | [getOutputNamesImpl](#)

Topics

“Specify Input and Output Names”

getOutputNamesImpl

Names of MATLAB System block output ports

Syntax

```
[name1,name2,...] = getOutputNamesImpl(obj)
```

Description

[name1,name2,...] = `getOutputNamesImpl(obj)` returns the names of the output ports from System object, `obj` implemented in a MATLAB System block. The number of returned output names matches the number of outputs returned by the `getNumOutputs` method. If you change a property value that affects the number of outputs, the names of those outputs also change.

`getOutputNamesImpl` is called by the MATLAB System block.

Note You must set `Access = protected` for this method.

Input Arguments

`obj`

System object

Output Arguments

`name1,name2,...`

Names of the outputs for the specified object, returned as character vectors.

Default: empty character vector

Examples

Specify Output Port Name

Specify the name of an output port as 'count'.

```
methods (Access = protected)
    function outputName = getOutputNamesImpl(~)
        outputName = 'count';
    end
end
```

- “Specify Input and Output Names”

See Also

[getInputNamesImpl](#) | [getNumOutputsImpl](#)

Topics

“Specify Input and Output Names”

getPropertyGroupsImpl

Property groups for System object display

Syntax

```
group = getPropertyGroupsImpl
```

Description

`group = getPropertyGroupsImpl` specifies how to group properties for display. You specify property sections (`matlab.system.display.Section`) and section groups (`matlab.system.display.SectionGroup`) within this method. Sections arrange properties into groups. Section groups arrange sections and properties into groups. If a System object, included through the MATLAB System block, has a section, but that section is not in a section group, its properties appear above the block dialog tab panels.

If you do not include a `getPropertyGroupsImpl` method in your code, all public properties are included in the dialog box by default. If you include a `getPropertyGroupsImpl` method but do not list a property, that property does not appear in the dialog box.

When the System object is displayed at the MATLAB command line, the properties are grouped as defined in `getPropertyGroupsImpl`. If your `getPropertyGroupsImpl` defines multiple section groups, only properties from the first section group are displayed at the command line. To display properties in other sections, a link is provided at the end of a System object property display. Group titles are also displayed at the command line. To omit the "Main" title for the first group of properties, set `TitleSource` to 'Auto' in `matlab.system.display.SectionGroup`.

`getPropertyGroupsImpl` is called by the MATLAB System block and when displaying the object at the command line.

Note You must set `Access = protected` and `Static` for this method.

Output Arguments

group

Property group or groups

Examples

Define Block Dialog Tabs

Define two block dialog tabs, each containing specific properties. For this example, you use the `getPropertyGroupsImpl`, `matlab.system.display.SectionGroup`, and `matlab.system.display.Section` methods in your class definition file.

```
methods (Static, Access = protected)
    function groups = getPropertyGroupsImpl
        valueGroup = matlab.system.display.Section(...
            'Title','Value parameters',...
            'PropertyList',{'StartValue','EndValue'});

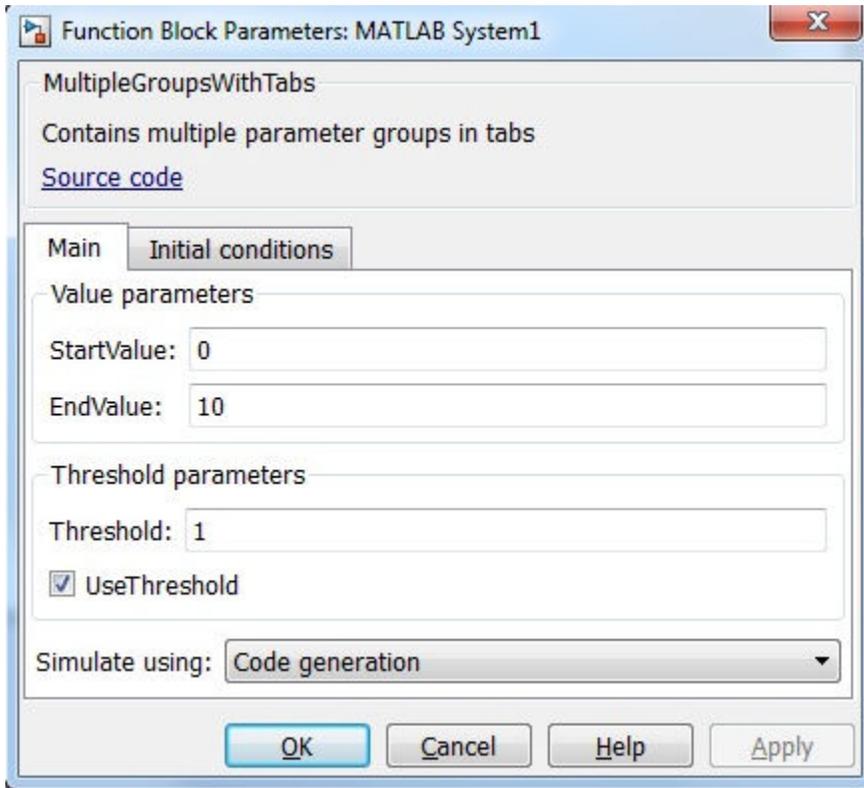
        thresholdGroup = matlab.system.display.Section(...
            'Title','Threshold parameters',...
            'PropertyList',{'Threshold','UseThreshold'});

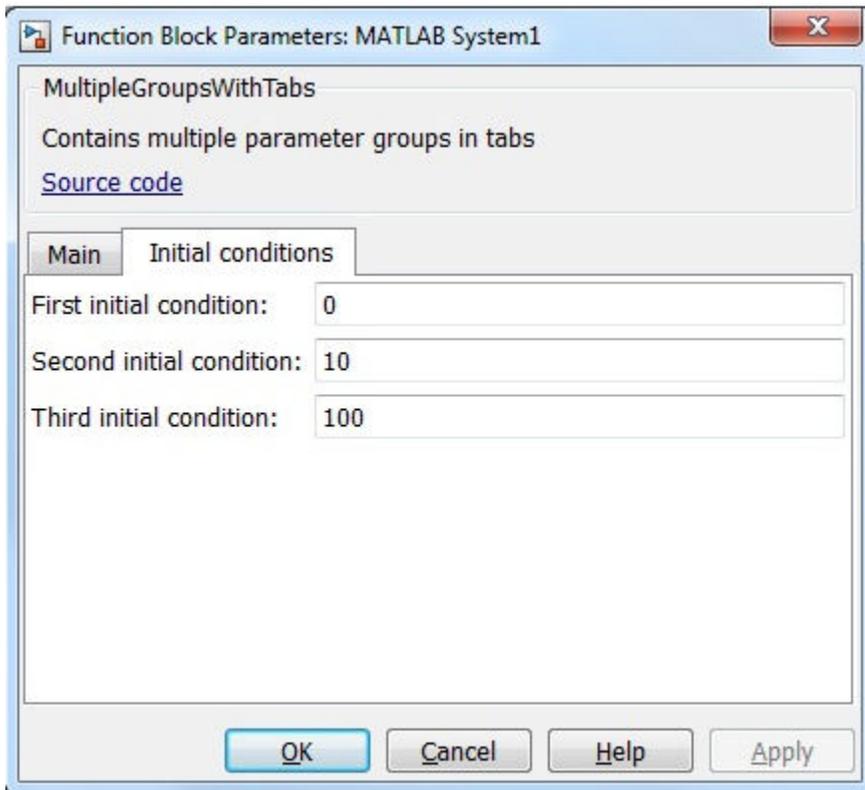
        mainGroup = matlab.system.display.SectionGroup(...
            'Title','Main', ...
            'Sections',[valueGroup,thresholdGroup]);

        initGroup = matlab.system.display.SectionGroup(...
            'Title','Initial conditions', ...
            'PropertyList',{'IC1','IC2','IC3'});

        groups = [mainGroup,initGroup];
    end
end
```

The resulting dialog box appears as follows.





- “Add Property Groups to System Object and MATLAB System Block”

See Also

`matlab.system.display.Header` | `matlab.system.display.Section` |
`matlab.system.display.SectionGroup`

Topics

“Add Property Groups to System Object and MATLAB System Block”

getSimulateUsingImpl

Specify value for Simulate using parameter

Syntax

```
simmode = getSimulateUsingImpl
```

Description

`simmode = getSimulateUsingImpl` specifies the simulation mode of the System object implemented in a MATLAB System block. The simulation mode restricts your System object to simulation using either code generation or interpreted execution. The associated `showSimulateUsingImpl` method controls whether the **Simulate using** option is displayed on the dialog box.

`getSimulateUsingImpl` is called by the MATLAB System block.

Note You must set `Access = protected` and `Static` for this method.

Output Arguments

simmode

Simulation mode, returned as the character vector 'Code generation' or 'Interpreted execution'. If you do not include the `getSimulateUsingImpl` method in your class definition file, the simulation mode is unrestricted. Depending on the value returned by the associated `showSimulateUsingImpl` method, the simulation mode is displayed as either a dropdown list on the dialog box or not at all.

Examples

Specify the Simulation Mode

In the class definition file of your System object, define the simulation mode to display in the MATLAB System block. To prevent **Simulate using** from displaying, see `showSimulateUsingImpl`.

```
methods (Static, Access = protected)
    function simMode = getSimulateUsingImpl
        simMode = 'Interpreted execution';
    end
end
```

- “Control Simulation Type in MATLAB System Block”

See Also

`showSimulateUsingImpl`

Topics

“Control Simulation Type in MATLAB System Block”

showFiSettingsImpl

Fixed point data type tab visibility for System objects

Syntax

```
flag = showFiSettingsImpl
```

Description

`flag = showFiSettingsImpl` specifies whether the Data Types tab appears on the MATLAB System block dialog box. The Data Types tab includes parameters to control processing of fixed point data the MATLAB System block. You cannot specify which parameters appear on the tab. If you implement `showFiSettingsImpl`, the simulation mode is set code generation.

`showFiSettingsImpl` is called by the MATLAB System block.

The parameters that appear on the Data Types tab, which cannot be customized, are

- **Saturate on integer overflow** is a check box to control the action to take on integer overflow for built-in integer types. The default is that the box is checked, which indicates to saturate. This is also the default for when **Same as MATLAB** is selected as the **MATLAB System fimath** option.
- **Treat these inherited Simulink signal types as fi objects** is a pull down that indicates which inherited data types to treat as fi data types. Valid options are `Fixed point` and `Fixed point & integer`. The default value is `Fixed point`.
- **MATLAB System fimath** has two radio button options: **Same as MATLAB** and **Specify Other**. The default, **Same as MATLAB**, uses the current MATLAB fixed-point math settings. **Specify Other** enables the edit box for specifying the desired fixed-point math settings. For information on setting fixed-point math, see `fimath`, in the Fixed-Point Designer documentation.

Note If you do not want to display the tab, you do not need to implement this method in your class definition file.

You must set `Access = protected` and `Static` for this method.

Output Arguments

flag

Flag indicating whether to display the Data Types tab on the MATLAB System block mask, returned as a logical scalar value. Returning a `true` value displays the tab. A `false` value does not display the tab.

Default: `false`

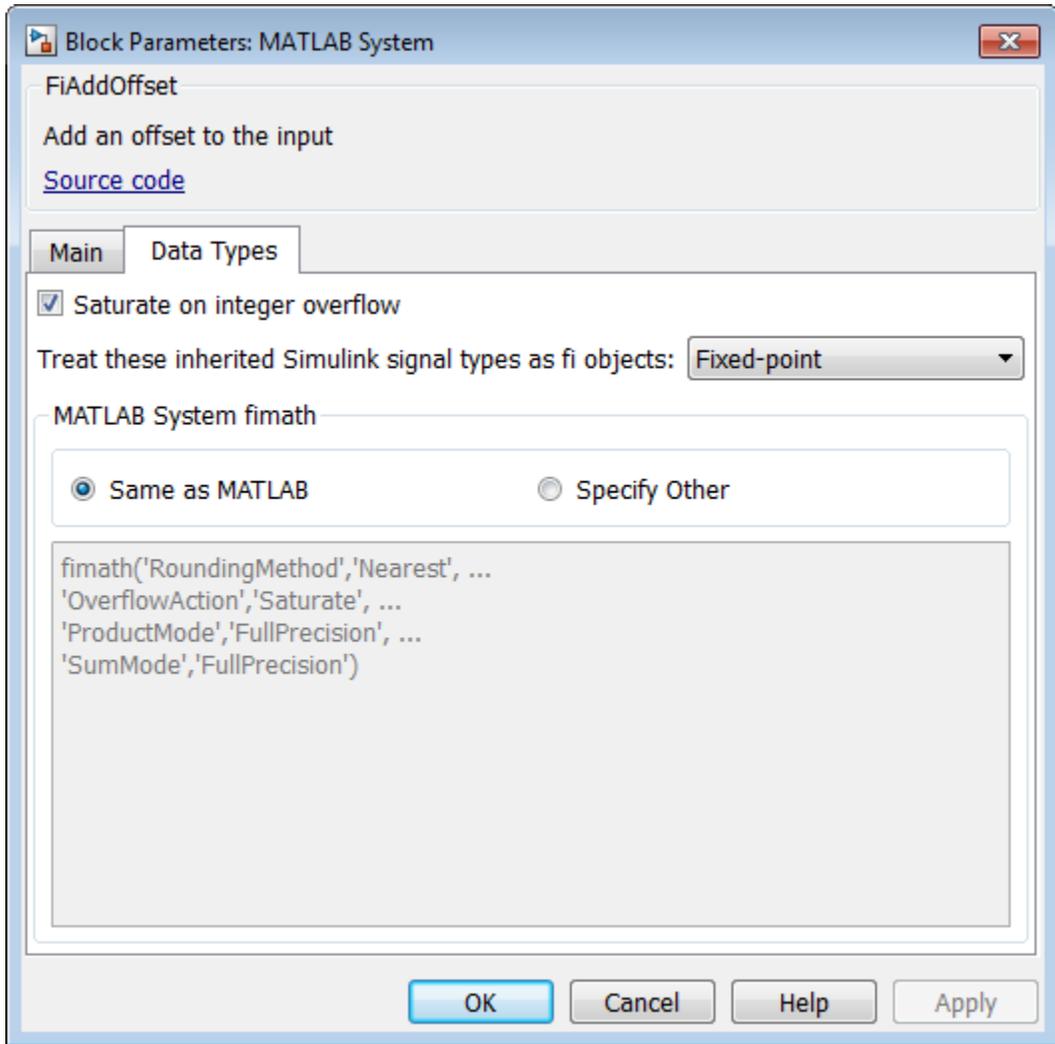
Examples

Show the Data Types Tab

Show the Data Types tab on the MATLAB System block dialog box.

```
methods (Static, Access = protected)
    function isVisible = showFiSettingsImpl
        isVisible = true;
    end
end
```

If you set the flag, `isVisible`, to `true`, the tab appears as follows when you add the object to Simulink with the MATLAB System block.



- “Add Data Types Tab to MATLAB System Block”

See Also

Topics

“Add Data Types Tab to MATLAB System Block”

showSimulateUsingImpl

Visibility of `Simulate` using parameter

Syntax

```
flag = showSimulateUsingImpl
```

Description

`flag = showSimulateUsingImpl` specifies whether **Simulation mode** appears on the MATLAB System block dialog box.

`showSimulateUsingImpl` is called by the MATLAB System block.

Note You must set `Access = protected` and `Static` for this method.

Output Arguments

flag

Flag indicating whether to display the **Simulate using** parameter and dropdown list on the MATLAB System block mask, returned as a logical scalar value. A `true` value displays the parameter and dropdown list. A `false` value hides the parameter and dropdown list.

Default: `true`

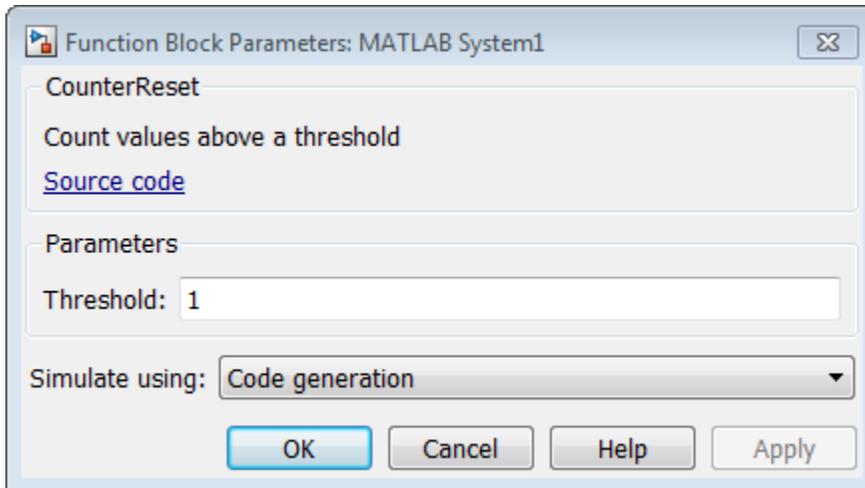
Examples

Hide the Simulate using Parameter

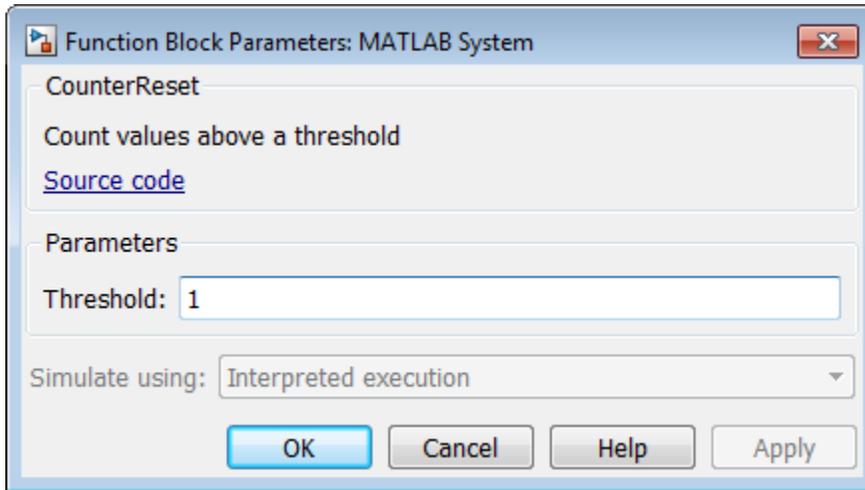
Hide the **Simulate using** parameter on the MATLAB System block dialog box.

```
methods (Static, Access = protected)
    function flag = showSimulateUsingImpl
        flag = false;
    end
end
```

If you set the flag to `true` or omit the `showSimulateUsingImpl` method, which defaults to `true`, the dialog appears as follows when you add the object to Simulink with the MATLAB System block.



If you also specify a single value for `getSimulateUsingImpl`, the dialog appears as follows when you add the object to Simulink with the MATLAB System block.



- “Control Simulation Type in MATLAB System Block”

See Also

`getSimulateUsingImpl`

Topics

“Control Simulation Type in MATLAB System Block”

getGlobalNamesImpl

Global variable names for MATLAB System block

Syntax

```
name = getGlobalNamesImpl(obj)
```

Description

`name = getGlobalNamesImpl(obj)` specifies the names of global variables that are declared in a System object for use in a Simulink P-code file. For P-code files, in addition to declaring your global variables in `stepImpl`, `outputImpl`, or `updateImpl`, you must include the `getGlobalNamesImpl` method. You declare global variables in a cell array in the `getGlobalNamesImpl` method. System objects that contain these global variables are included in Simulink using a MATLAB System block. To enable a global variable in Simulink, your model also must include a Data Store Memory block with a **Data Store Name** that matches the global variable name.

`getGlobalNamesImpl` is called by the MATLAB System block.

Note You must set `Access = protected` for this method.

Input Arguments

`obj`

System object

Output Arguments

name

Name of the cell array containing the global variable names. The elements of the cell array are character vectors.

Examples

Specify Global Names

Specify two global names in your class definition file.

```
methods(Access = protected)
    function glnames = getGlobalNamesImpl(obj)
        glnames = {'FEE', 'OTHERFEE'};
    end

    function y = stepImpl(obj,u)
        global FEE
        global OTHERFEE
        y = u - FEE * obj.lastData + OTHERFEE;
        obj.lastData = u;
    end
end
```

- “System Object Global Variables in Simulink”

See Also

`outputImpl` | `stepImpl` | `updateImpl`

Topics

“System Object Global Variables in Simulink”

Introduced in R2016b

getHeaderImpl

Header for System object display

Syntax

```
header = getHeaderImpl
```

Description

`header = getHeaderImpl` specifies the dialog header to display on the MATLAB System block dialog box. If you do not specify the `getHeaderImpl` method, no title or text appears for the header in the block dialog box.

`getHeaderImpl` is called by the MATLAB System block.

Note You must set `Access = protected` and `Static` for this method.

Output Arguments

header

Header text

Examples

Define Header for System Block Dialog Box

Define a header in your class definition file for the `EnhancedCounter` System object.

```
methods (Static, Access = protected)
    function header = getHeaderImpl
```

```
        header = matlab.system.display.Header('EnhancedCounter', ...  
            'Title', 'Enhanced Counter');  
    end  
end
```

- “Add Header to MATLAB System Block”

See Also

`getPropertyGroupsImpl`

Topics

“Add Header to MATLAB System Block”

getDiscreteStateImpl

Discrete state property values

Syntax

```
s = getDiscreteStateImpl(obj)
```

Description

`s = getDiscreteStateImpl(obj)` returns a struct `s` of internal state value properties, which have the `DiscreteState` attribute. The field names of the struct are the object's `DiscreteState` property names. To restrict or change the values returned by `getDiscreteState` method, you can override this `getDiscreteStateImpl` method.

`getDiscreteStatesImpl` is called by the `getDiscreteState` method, which is called by the `setup` method.

Note You must set `Access = protected` for this method.

You cannot modify any properties in this method.

Input Arguments

`obj`

System object handle

Output Arguments

`s`

State values, returned as a struct

Examples

Get Discrete State Values

Use the `getDiscreteStateImpl` method in your class definition file to get the discrete states of the object.

```
methods (Access = protected)
    function s = getDiscreteStateImpl(obj)
    end
end
```

See Also

`setupImpl`

Topics

“Define Property Attributes” (MATLAB)

supportsMultipleInstanceImpl

Support System object in Simulink For Each subsystem

Syntax

```
flag = supportsMultipleInstanceImpl(obj)
```

Description

`flag = supportsMultipleInstanceImpl(obj)` specifies whether the System object can be used in a Simulink For Each subsystem via the MATLAB System block. To enable For Each support, you must include the `supportsMultipleInstanceImpl` in your class definition file and have it return `true`. Do not enable For Each support if your System object allocates exclusive resources that may conflict with other System objects, such as allocating file handles, memory by address, or hardware resources.

During Simulink model compilation and propagation, the MATLAB System block calls the `supportMultipleInstance` method, which then calls the `supportsMultipleInstanceImpl` method to determine For Each support.

Note You must set `Access = protected` for this method.

You cannot modify any properties in this method.

Input Arguments

obj

System object handle

Output Arguments

flag

Boolean value indicating whether the System object can be used in a For Each subsystem. The default value, if you do not include the `supportMultipleInstance` method, is `false`.

Examples

Enable For-Each Support for System Object

Specify in your class definition file that the System object can be used in a Simulink For Each subsystem.

```
methods (Access = protected)
    function flag = supportsMultipleInstanceImpl(obj)
        flag = true;
    end
end
```

- “Enable For Each Subsystem Support”

See Also

`matlab.System`

Topics

“Enable For Each Subsystem Support”

processTunedPropertiesImpl

Action when tunable properties change

Syntax

```
processTunedPropertiesImpl(obj)
```

Description

`processTunedPropertiesImpl(obj)` specifies the algorithm to perform when one or more tunable property values change. This method is called as part of the next call to the System object after a tunable property value changes. A property is tunable only if its `Nontunable` attribute is `false`, which is the default.

`processTunedPropertiesImpl` is called when you run the System object.

Note You must set `Access = protected` for this method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink MATLAB System block.

Input Arguments

`obj`

System object

Examples

Specify Action When Tunable Property Changes

Use `processTunedPropertiesImpl` to recalculate the lookup table if the value of either the `NumNotes` or `MiddleC` property changes before the next call to the System object. `propChange` indicates if either property has changed.

```
methods (Access = protected)
    function processTunedPropertiesImpl(obj)
        propChange = isChangedProperty(obj, 'NumNotes') ||...
                    isChangedProperty(obj, 'MiddleC')
        if propChange
            obj.pLookupTable = obj.MiddleC * (1+log(1:obj.NumNotes)/log(12));
        end
    end
end
```

- “Validate Property and Input Values” (MATLAB)
- “Define Property Attributes” (MATLAB)

Tips

Use this method when a tunable property affects the value of a different property.

To check if a property has changed since `stepImpl` was last called, use `isChangedProperty` within `processTunedPropertiesImpl`. See “Specify Action When Tunable Property Changes” on page 5-83 for an example.

In MATLAB when multiple tunable properties are changed before running the System object, `processTunedPropertiesImpl` is called only once for all the changes. `isChangedProperty` returns `true` for all the changed properties.

In Simulink, when a parameter is changed in a MATLAB System block dialog, the next simulation step calls `processTunedPropertiesImpl` before calling `stepImpl`. All tunable parameters are considered changed and `processTunedPropertiesImpl` method is called for each of them. `isChangedProperty` returns `true` for all the dialog properties.

See Also

`setProperty` | `validatePropertiesImpl`

Topics

“Validate Property and Input Values” (MATLAB)

“Define Property Attributes” (MATLAB)

matlab.system.mixin.CustomIcon class

Package: matlab.system.mixin

Custom icon mixin class

Description

`matlab.system.mixin.CustomIcon` is a class that specifies the `getIcon` method. This method customizes the name of the icon used for the `System` object implemented through a MATLAB System block.

To use this method, you must subclass from this class in addition to the `matlab.System` base class. Type the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.system &...  
    matlab.system.mixin.CustomIcon
```

Methods

<code>getIconImpl</code>	Name to display as block icon
--------------------------	-------------------------------

See Also

`matlab.System` | `matlab.system.display.Icon`

Topics

“Add Text to Block Icon”

getIconImpl

Class: matlab.system.mixin.CustomIcon

Package: matlab.system.mixin

Name to display as block icon

Syntax

```
icon = getIconImpl(obj)
```

Description

`icon = getIconImpl(obj)` specifies the text or image to display on the block icon of the MATLAB System block. If you do not specify the `getIconImpl` method, the block displays the class name of the System object as the block icon. For example, if you specify `pkg.MyObject` in the MATLAB System block, the default icon is labeled `MyObject`.

`getIconImpl` is called by the MATLAB System block during Simulink model compilation.

Note You must set `Access = protected` for this method.

Input Arguments

obj

System object handle

Output Arguments

icon

The text or image to display as the block icon. Each cell is displayed as a separate line.

Examples

Add System Block Icon Name

Specify in your class definition file the name of the block icon as 'Enhanced Counter' using two lines.

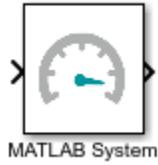
```
methods (Access = protected)
    function icon = getIconImpl(~)
        icon = {'Enhanced', 'Counter'};
    end
end
```

Add Image to MATLAB System Block

Define an image in your class definition file.

```
methods(Access = protected)
    function icon = getIconImpl(~)
        % Define icon for System block
        icon = matlab.system.display.Icon('my_icon.png');
    end
end
```

The image now appears on the System block icon.



- “Customize System Block Appearance”

See Also

`matlab.system.display.Icon` | `matlab.system.mixin.CustomIcon`

Topics

“Customize System Block Appearance”

matlab.system.display.Header class

Package: matlab.system.display

Header for System objects properties

Syntax

```
matlab.system.display.Header(N1,V1,...Nn,Vn)
matlab.system.display.Header(Obj,...)
```

Description

`matlab.system.display.Header(N1,V1,...Nn,Vn)` specifies a header for the System object, with the header properties defined in Name-Value (N,V) pairs. You use `matlab.system.display.Header` within the `getHeaderImpl` method. The available header properties are

- `Title` — Header title. The default value is an empty character vector.
- `Text` — Header description. The default value is an empty character vector.
- `ShowSourceLink` — Show link to source code for the object.

`matlab.system.display.Header(Obj,...)` creates a header for the specified System object (`Obj`) and sets the following property values:

- `Title` — Set to the `Obj` class name.
- `Text` — Set to help summary for `Obj`.
- `ShowSourceLink` — Set to `true` if `Obj` is MATLAB code. In this case, the **Source Code** link is displayed. If `Obj` is P-coded and the source code is not available, set this property to `false`.

You can use `mfilename('class')` from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

Examples

Define System Block Header

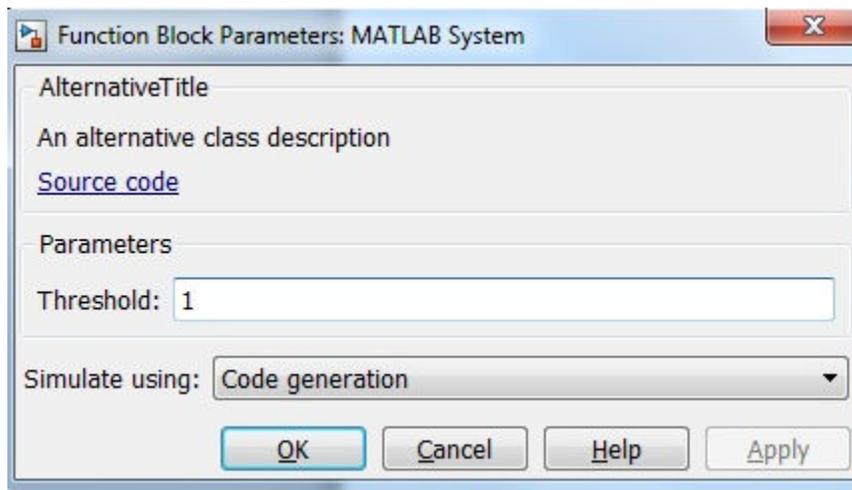
Define a header in your class definition file.

```

methods (Static, Access = protected)
    function header = getHeaderImpl
        header = matlab.system.display.Header(mfilename('class'), ...
            'Title', 'AlternativeTitle', ...
            'Text', 'An alternative class description');
    end
end

```

The resulting output appears as follows. In this case, **Source code** appears because the ShowSourceLink property was set to true.



- “Object-Oriented Programming” (MATLAB)
- “Add Header to MATLAB System Block”

See Also

getHeaderImpl | matlab.system.display.Section |
 matlab.system.display.SectionGroup

Topics

“Object-Oriented Programming” (MATLAB)

“Add Header to MATLAB System Block”

matlab.system.display.Section class

Package: matlab.system.display

Property group section for System objects

Syntax

```
matlab.system.display.Section(N1,V1,...Nn,Vn)
matlab.system.display.Section(Obj,...)
```

Description

`matlab.system.display.Section(N1,V1,...Nn,Vn)` creates a property group section for displaying System object properties, which you define using property Name-Value pairs (N,V). You use `matlab.system.display.Section` to define property groups using the `getPropertyGroupsImpl` method. The available Section properties are

- **Title** — Section title. The default value is an empty character vector.
- **TitleSource** — Source of section title. Valid values are 'Property' and 'Auto'. The default value is 'Property', which uses the character vector from the **Title** property. If the **Obj** name is given, the default value is `Auto`, which uses the **Obj** name.
- **Description** — Section description. The default value is an empty character vector.
- **PropertyList** — Section property list as a cell array of property names. The default value is an empty array. If the **Obj** name is given, the default value is all eligible display properties.

Note Certain properties are not eligible for display either in a dialog box or in the System object summary on the command-line. Property types that cannot be displayed are: hidden, abstract, private or protected access, discrete state, and continuous state. Dependent properties do not display in a dialog box, but do display in the command-line summary.

`matlab.system.display.Section(Obj, ...)` creates a property group section for the specified System object (`Obj`) and sets the following property values:

- `TitleSource` — Set to 'Auto', which uses the `Obj` name.
- `PropertyList` — Set to all publicly-available properties in the `Obj`.

You can use `mfilename('class')` from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

Methods

Examples

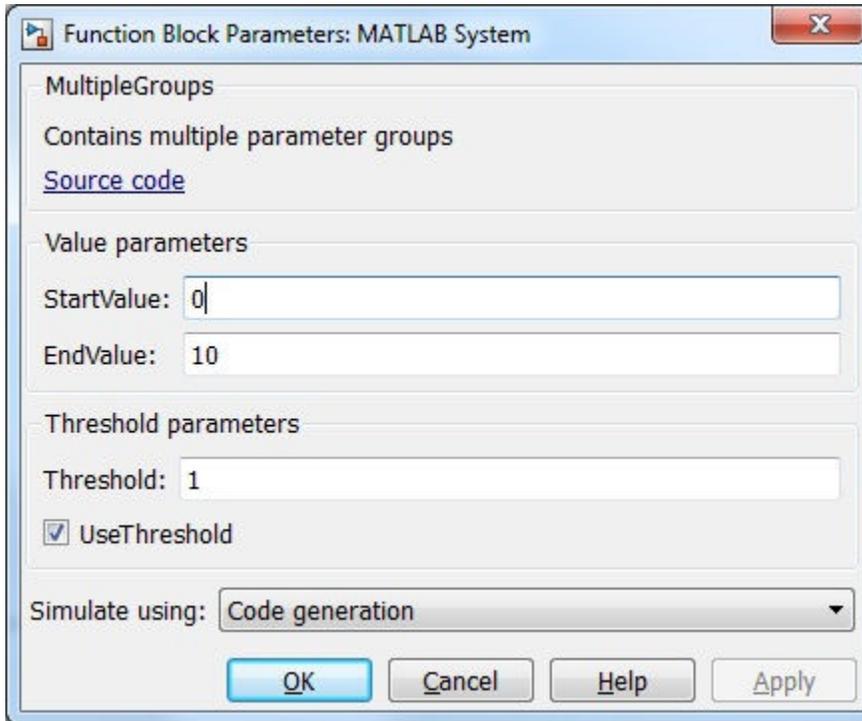
Define Property Groups

Define two property groups in your class definition file by specifying their titles and property lists.

```
methods (Static, Access = protected)
    function groups = getPropertyGroupsImpl
        valueGroup = matlab.system.display.Section(...
            'Title', 'Value parameters', ...
            'PropertyList', {'StartValue', 'EndValue'});

        thresholdGroup = matlab.system.display.Section(...
            'Title', 'Threshold parameters', ...
            'PropertyList', {'Threshold', 'UseThreshold'});
        groups = [valueGroup, thresholdGroup];
    end
end
```

When you specify the System object in the MATLAB System block, the resulting dialog box appears as follows.



- “Add Property Groups to System Object and MATLAB System Block”

See Also

`getPropertyGroupsImpl` | `matlab.system.display.Header` |
`matlab.system.display.SectionGroup`

Topics

“Add Property Groups to System Object and MATLAB System Block”

matlab.system.display.Action class

Package: matlab.system.display

Custom button

Syntax

```
matlab.system.display.Action(action)
matlab.system.display.Action(action,Name,Value)
```

Description

`matlab.system.display.Action(action)` specifies a button to display on the MATLAB System block. This button executes a function by launching a System object method or invoking any MATLAB function or code.

A typical button function launches a figure. The launched figure is decoupled from the block dialog box. Changes to the block are not synced to the displayed figure.

You define `matlab.system.display.Action` within the `getPropertyGroupsImpl` method in your class definition file. You can define multiple buttons using separate instances of `matlab.system.display.Action` in your class definition file.

`matlab.system.display.Action(action,Name,Value)` includes `Name,Value` pair arguments, which you can use to specify any properties.

Input Arguments

action

Action taken when the user presses the specified button on the MATLAB System block dialog. The action is defined as a function handle or as a MATLAB command. If you define the action as a function handle, the function definition must define two inputs. These inputs are a `matlab.system.display.ActionData` object and a System object instance, which can be used to invoke a method.

A `matlab.system.display.ActionData` object is the callback object for a display action. You use the `UserData` property of `matlab.system.display.ActionData` to store persistent data, such as a figure handle.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Properties

You specify these properties as part of the input using `Name`, `Value` pair arguments. Optionally, you can define them using `object.property` syntax.

- `ActionCalledFcn` — Action to take when the button is pressed. You cannot specify this property using a Name-Value pair argument.
- `Label` — Text to display on the button. The default value is an empty character vector.
- `Description` — Text for the button tooltip. The default value is an empty character vector.
- `Placement` — Character vector indicating where on a separate row in the property group to place the button. Valid values are 'first', 'last', or a property name. If you specify a property name, the button is placed above that property. The default value is 'last'.
- `Alignment` — Character vector indicating how to align the button. Valid values are 'left' and 'right'. The default value is 'left'.

Examples

Define Button on MATLAB System Block

Define a **Visualize** button and its associated function to open a figure that plots a ramp using the parameter values in the block dialog.

```

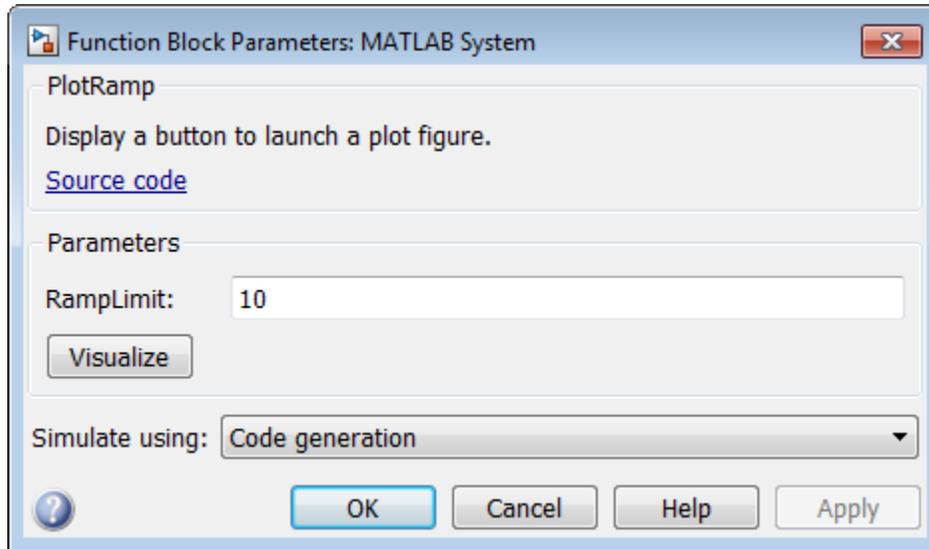
methods(Static,Access = protected)
    function group = getPropertyGroupsImpl
        group = matlab.system.display.Section(mfilename('class'));
        group.Actions = matlab.system.display.Action(@(~,obj) ...
            visualize(obj), 'Label', 'Visualize');
    end
end

methods
    function obj = PlotRamp(varargin)
        setProperties(obj,nargin,varargin{:});
    end

    function visualize(obj)
        figure;
        d = 1:obj.RampLimit;
        plot(d);
    end
end

```

When you specify the System object in the MATLAB System block, the resulting block dialog box appears as follows.



To open the same figure, rather than multiple figures, when the button is pressed more than once, use this code instead.

```

methods(Static,Access = protected)
    function group = getPropertyGroupsImpl
        group = matlab.system.display.Section(mfilename('class'));
        group.Actions = matlab.system.display.Action(@(actionData,obj) ...
            visualize(obj,actionData),'Label','Visualize');
    end
end

methods
    function obj = ActionDemo(varargin)
        setProperties(obj,nargin,varargin{:});
    end

    function visualize(obj,actionData)
        f = actionData.UserData;
        if isempty(f) || ~ishandle(f)
            f = figure;
            actionData.UserData = f;
        else
            figure(f); % Make figure current
        end

        d = 1:obj.RampLimit;
        plot(d);
    end
end

```

- “Object-Oriented Programming” (MATLAB)
- Class Attributes (MATLAB)
- Property Attributes (MATLAB)
- “Add Button to MATLAB System Block”

See Also

matlab.System.getPropertyGroupsImpl | matlab.system.display.Section |
matlab.system.display.SectionGroup

Topics

“Object-Oriented Programming” (MATLAB)
Class Attributes (MATLAB)
Property Attributes (MATLAB)

“Add Button to MATLAB System Block”

matlab.system.display.SectionGroup class

Package: matlab.system.display

Section group for System objects

Syntax

```
matlab.system.display.SectionGroup(N1,V1,...Nn,Vn)
matlab.system.display.SectionGroup(Obj,...)
```

Description

`matlab.system.display.SectionGroup(N1,V1,...Nn,Vn)` creates a group for displaying System object properties and display sections created with `matlab.system.display.Section`. You define such sections or properties using property Name-Value pairs (N,V). A section group can contain both properties and sections. You use `matlab.system.display.SectionGroup` to define section groups using the `getPropertyGroupsImpl` method. Section groups display as separate tabs in the MATLAB System block. The available Section properties are

- **Title** — Group title. The default value is an empty character vector.
- **TitleSource** — Source of group title. Valid values are 'Property' and 'Auto'. The default value is 'Property', which uses the character vector from the Title property. If the Obj name is given, the default value is Auto, which uses the Obj name. In the System object property display at the MATLAB command line, you can omit the default "Main" title for the first group of properties by setting TitleSource to 'Auto'.
- **Description** — Group or tab description that appears above any properties or panels. The default value is an empty character vector.
- **PropertyList** — Group or tab property list as a cell array of property names. The default value is an empty array. If the Obj name is given, the default value is all eligible display properties.
- **Sections** — Group sections as an array of section objects. If the Obj name is given, the default value is the default section for the Obj.

`matlab.system.display.SectionGroup(Obj,...)` creates a section group for the specified System object (`Obj`) and sets the following property values:

- `TitleSource` — Set to 'Auto'.
- `Sections` — Set to `matlab.system.display.Section` object for `Obj`.

You can use `mfilename('class')` from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

Examples

Define Block Dialog Tabs

Define in your class definition file two tabs, each containing specific properties. For this example, you use the `matlab.system.display.SectionGroup`, `matlab.system.display.Section`, and `getPropertyGroupsImpl` methods.

```
methods (Static, Access = protected)
    function groups = getPropertyGroupsImpl
        valueGroup = matlab.system.display.Section(...
            'Title','Value parameters',...
            'PropertyList',{'StartValue','EndValue'});

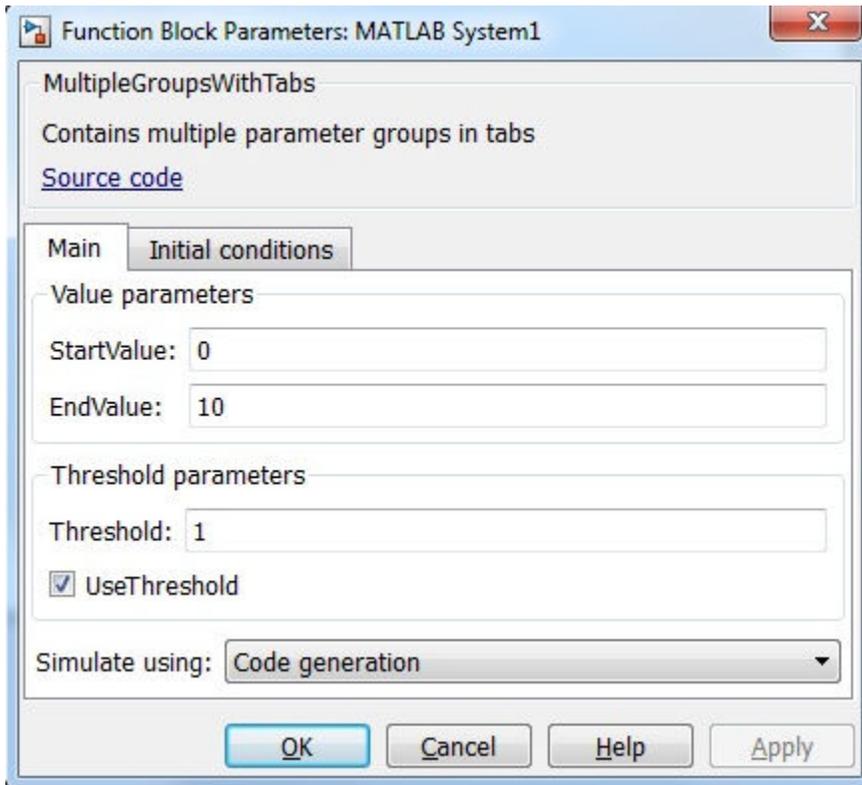
        thresholdGroup = matlab.system.display.Section(...
            'Title','Threshold parameters',...
            'PropertyList',{'Threshold','UseThreshold'});

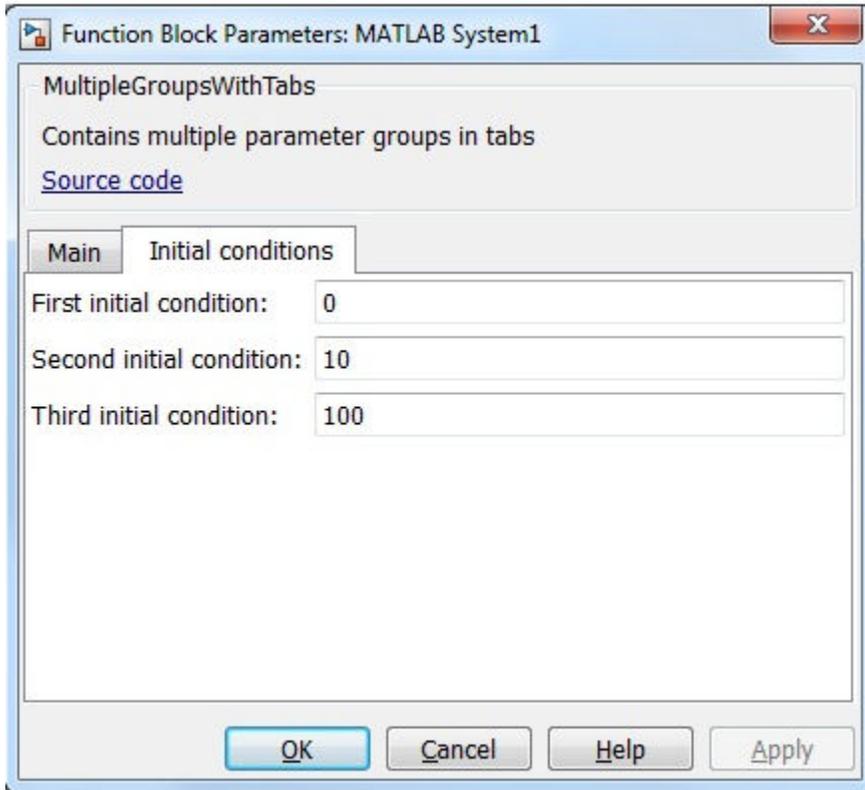
        mainGroup = matlab.system.display.SectionGroup(...
            'Title','Main', ...
            'Sections',[valueGroup,thresholdGroup]);

        initGroup = matlab.system.display.SectionGroup(...
            'Title','Initial conditions', ...
            'PropertyList',{'IC1','IC2','IC3'});

        groups = [mainGroup,initGroup];
    end
end
```

The resulting dialog appears as follows when you add the object to Simulink with the MATLAB System block.





- “Add Property Groups to System Object and MATLAB System Block”

See Also

`getPropertyGroupsImpl` | `matlab.system.display.Header` | `matlab.system.display.Section`

Topics

“Add Property Groups to System Object and MATLAB System Block”

matlab.system.display.Icon class

Package: matlab.system.display

Custom icon image

Syntax

```
icon = matlab.system.display.Icon(imageFile)
```

Description

`icon = matlab.system.display.Icon(imageFile)` sets the `imageFile` image as the MATLAB System block icon. To set the icon image, use the `icon` output argument from `getIconImpl`.

Input Arguments

imageFile — Image file

character array

Image file to display on the block icon, specified as a character array. If the image is not on the path, use the full path to your image file.

The image file must be in a file format supported for block masks. See “Draw Static Icon”.

Example: 'image.png'

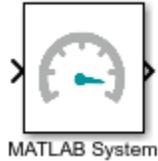
Examples

Add Image to MATLAB System Block

Define an image in your class definition file.

```
methods(Access = protected)
    function icon = getIconImpl(~)
        % Define icon for MATLAB System block
        icon = matlab.system.display.Icon('my_icon.png');
    end
end
```

The image now appears on the MATLAB System block icon.



- Class Attributes (MATLAB)
- Property Attributes (MATLAB)
- “Customize System Block Appearance”

See Also

`getIconImpl` | `matlab.system.mixin.CustomIcon`

Topics

Class Attributes (MATLAB)
Property Attributes (MATLAB)
“Customize System Block Appearance”

Introduced in R2017a

matlab.system.mixin.Propagates class

Package: matlab.system.mixin

Signal characteristics propagation mixin class

Description

`matlab.system.mixin.Propagates` specifies the output size, data type, and complexity of a `System` object. Use this mixin class and its methods when you will include your `System` object in Simulink via the MATLAB System block. This mixin is called by the MATLAB System block during Simulink model compilation.

Implement the methods of this class when Simulink cannot infer the output specifications directly from the inputs or when you want bus support. If you do not include this mixin, Simulink cannot propagate the output or bus data type, an error occurs.

To use this mixin, subclass from this `matlab.system.mixin.Propagates` in addition to subclassing from the `matlab.System` base class. Type the following syntax as the first line of your class definition file. `ObjectName` is the name of your `System` object.

```
classdef ObjectName < matlab.System &...  
    matlab.system.mixin.Propagates
```

Methods

<code>getDiscreteStateSpecificationImpl</code>	Discrete state size, data type, and complexity
<code>getOutputDataTypeImpl</code>	Data types of output ports
<code>getOutputSizeImpl</code>	Sizes of output ports
<code>isOutputComplexImpl</code>	Complexity of output ports
<code>isOutputFixedSizeImpl</code>	Fixed- or variable-size output ports
<code>propagatedInputComplexity</code>	Complexity of input during Simulink propagation
<code>propagatedInputDataType</code>	Data type of input during Simulink propagation
<code>propagatedInputFixedSize</code>	Fixed-size status of input during Simulink propagation
<code>propagatedInputSize</code>	Size of input during Simulink propagation

Note If your `System` object has exactly one input and one output and no discrete property states, or if you do not need bus support, you do not have to implement any of these methods. The `matlab.system.mixin.Propagates` provides default values in these cases.

See Also

`matlab.System`

Topics

“Set Output Data Type”

“Set Output Size”

“Set Output Complexity”

“Specify Whether Output Is Fixed-Size or Variable-Size”

“Specify Discrete State Output Specification”

getDiscreteStateSpecificationImpl

Class: matlab.system.mixin.Propagates

Package: matlab.system.mixin

Discrete state size, data type, and complexity

Syntax

```
[sz,dt,cp] = getDiscreteStateSpecificationImpl(obj,propertyname)
```

Description

`[sz,dt,cp] = getDiscreteStateSpecificationImpl(obj,propertyname)` returns the size, data type, and complexity of the discrete state property. This property must be a discrete state property. You must define this method if your System object has discrete state properties and is used in the MATLAB System block. If you define this method for a property that is not discrete state, an error occurs during model compilation.

You always set the `getDiscreteStateSpecificationImpl` method access to `protected` because it is an internal method that users do not directly call or run.

`getDiscreteStateSpecificationImpl` is called by the MATLAB System block during Simulink model compilation.

Note You must set `Access = protected` for this method.

You cannot modify any properties in this method.

Input Arguments

obj

System object handle

propertyname

Name of discrete state property of the System object

Output Arguments

sz

Vector containing the length of each dimension of the property.

Default: [1 1]

dt

Data type of the property. For built-in data types, `dt` is a character vector. For fixed-point data types, `dt` is a `numericType` object.

Default: `double`

cp

Complexity of the property as a scalar, logical value, where `true` = complex and `false` = real.

Default: `false`

Examples

Specify Discrete State Property Size, Data Type, and Complexity

Specify in your class definition file the size, data type, and complexity of a discrete state property.

```
methods (Access = protected)
    function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
        sz = [1 1];
        dt = 'double';
        cp = false;
```

```
end  
end
```

- “Specify Discrete State Output Specification”

See Also

`matlab.system.mixin.Propagates`

Topics

“Specify Discrete State Output Specification”

getOutputDataTypeImpl

Class: matlab.system.mixin.Propagates

Package: matlab.system.mixin

Data types of output ports

Syntax

```
[dt_1,dt_2,...,dt_n] = getOutputDataTypeImpl(obj)
```

Description

[dt_1,dt_2,...,dt_n] = getOutputDataTypeImpl(obj) returns the data type of each output port as a character vector for built-in data types or as a numeric object for fixed-point data types. The number of outputs must match the value returned from the getNumOutputs method or the number of output arguments listed in the stepImpl method.

For System objects with one input and one output and where you want the input and output data types to be the same, you do not need to implement this method. In this case, getOutputDataTypeImpl assumes the input and output data types are the same and returns the data type of the input.

If your System object has more than one input or output, and you subclass from matlab.system.mixin.Propagates, you must set the output data types in the getOutputDataTypeImpl method. For Simulink, if the input and output data types are different, you might have to cast the output value to the data type of the appropriate dt_n output argument. You specify this casting in the stepImpl method. For bus output, you must specify the name of the output bus in getOutputDataTypeImpl.

If needed to determine the output data type, you can use propagatedInputDataType within the getOutputDataTypeImpl method to obtain the input type.

Note You must set Access = protected for this method.

You cannot modify any properties in this method.

If you are debugging your code and examine the data types before Simulink completes propagation, you might see outputs with empty, [], data types. This occurs because Simulink has not completed setting the output data types.

Input Arguments

`obj`

System object

Output Arguments

`dt_1, dt_2, ...`

Data type of the property. For built-in data types, `dt` is a character vector. For fixed-point data types, `dt` is a numeric type object.

Examples

Specify Output Data Type

Specify, in your class definition file how to control the output data type from a MATLAB System block. This example shows how to use the `getOutputDataTypeImpl` method to change the output data type from single to double, or propagate the input as a double. It also shows how to cast the data type to change the output data type in the `stepImpl` method.

```
classdef DataTypeChange < matlab.System & ...
    matlab.system.mixin.Propagates

    properties(Nontunable)
        Quantize = false
    end

    methods(Access = protected)
```

```
function y = stepImpl(obj,u)
    if obj.Quantize == true
        % Cast for output data type to differ from input.
        y = single(u);
    else
        % Propagate output data type.
        y = u;
    end
end

function out = getOutputDataTypeImpl(obj)
    if obj.Quantize == true
        out = 'single';
    else
        out = propagatedInputDataType(obj,1);
    end
end
end
end
```

Specify Bus Output

Specify, in your class definition file, that the System object data type is a bus. You must also include a property to specify the bus name.

```
properties(Nontunable)
    OutputBusName = 'myBus'
end

methods (Access = protected)
    function out = getOutputDataTypeImpl(obj)
        out = obj.OutputBusName;
    end
end
```

- “Set Output Data Type”

See Also

`matlab.system.mixin.Propagates` | `propagatedInputDataType`

Topics

“Set Output Data Type”

getOutputSizeImpl

Class: matlab.system.mixin.Propagates

Package: matlab.system.mixin

Sizes of output ports

Syntax

```
[sz_1,sz_2,...,sz_n] = getOutputSizeImpl(obj)
```

Description

[sz_1,sz_2,...,sz_n] = getOutputSizeImpl(obj) returns the size of each output port. The number of outputs must match the value returned from the getNumOutputs method or the number of output arguments listed in the stepImpl method.

If your System object has only one input and one output and you want the input and output sizes to be the same, you do not need to implement this method. In this case getOutputSizeImpl assumes that the input and output sizes are the same and returns the size of the input. For variable-size inputs in MATLAB, the size varies each time you run your object. For variable-size inputs in Simulink, the output size is the maximum input size.

You must implement the getOutputSizeImpl method to define the output size, if:

- Your System object has more than one input or output
- You need the output and input sizes to be different.

If the output size differs from the input size, you must also use the propagatedInputSize method

During Simulink model compilation and propagation, the MATLAB System block calls the getOutputSize method, which then calls the getOutputSizeImpl method to determine the output size.

All inputs default to variable-size inputs. For these inputs, the output size is the maximum input size.

Note You must set `Access = protected` for this method.

You cannot modify any properties in this method.

Input Arguments

`obj`

System object handle

Output Arguments

`sz_1, sz_2, ...`

Vector containing the size of each output port.

Examples

Specify Output Size

Specify in your class definition file the size of a System object output.

```
methods (Access = protected)
    function sz_1 = getOutputSizeImpl(obj)
        sz_1 = [1 1];
    end
end
```

Specify Multiple Output Ports

Specify in your class definition file the sizes of multiple System object outputs.

```
methods (Access = protected)
    function [sz_1,sz_2] = getOutputSizeImpl(obj)
        sz_1 = propagatedInputSize(obj,1);
        sz_2 = [1 1];
    end
end
```

Specify Output When Using Propagated Input Size

Specify in your class definition file the size of System object output when it depends on the propagated input size.

```
methods (Access = protected)
    function varargout = getOutputSizeImpl(obj)
        varargout{1} = propagatedInputSize(obj,1);
        if obj.HasSecondOutput
            varargout{2} = [1 1];
        end
    end
end
```

- “Set Output Size”

See Also

[matlab.system.mixin.Propagates](#) | [propagatedInputSize](#)

Topics

“Set Output Size”

isOutputComplexImpl

Class: matlab.system.mixin.Propagates

Package: matlab.system.mixin

Complexity of output ports

Syntax

```
[cp_1, cp_2, ..., cp_n] = isOutputComplexImpl(obj)
```

Description

`[cp_1, cp_2, ..., cp_n] = isOutputComplexImpl(obj)` returns whether each output port has complex data. The number of outputs must match the value returned from the `getNumOutputs` method or the number of output arguments listed in the `stepImpl` method.

For System objects with one input and one output and where you want the input and output complexities to be the same, you do not need to implement this method. In this case `isOutputComplexImpl` assumes the input and output complexities are the same and returns the complexity of the input.

If your System object has more than one input or output or you need the output and input complexities to be different, you must implement the `isOutputComplexImpl` method to define the output complexity. You also must use the `propagatedInputComplexity` method if the output complexity differs from the input complexity.

During Simulink model compilation and propagation, the MATLAB System block calls the `isOutputComplex` method, which then calls the `isOutputComplexImpl` method to determine the output complexity.

Note You must set `Access = protected` for this method.

You cannot modify any properties in this method.

Input Arguments

`obj`

System object handle

Output Arguments

`cp_1, cp_2, ...`

Logical, scalar value indicating whether the specific output port is complex (`true`) or real (`false`).

Examples

Specify Output as Real-Valued

Specify in your class definition file that the output from a System object is a real value.

```
methods (Access = protected)
    function c1 = isOutputComplexImpl(obj)
        c1 = false;
    end
end
```

- “Set Output Complexity”

See Also

`matlab.system.mixin.Propagates` | `propagatedInputComplexity`

Topics

“Set Output Complexity”

isOutputFixedSizeImpl

Class: matlab.system.mixin.Propagates

Package: matlab.system.mixin

Fixed- or variable-size output ports

Syntax

```
[flag_1,flag_2,...flag_n] = isOutputFixedSizeImpl(obj)
```

Description

[flag_1,flag_2,...flag_n] = isOutputFixedSizeImpl(obj) returns whether each output port is fixed size. The number of outputs must match the value returned from the getNumOutputs method, which is the number of output arguments listed in the stepImpl method.

For System objects with one input and one output and where you want the input and output fixed sizes to be the same, you do not need to implement this method. In this case isOutputFixedSizeImpl assumes the input and output fixed sizes are the same and returns the fixed size of the input.

If your System object has more than one input or output or you need the output and input fixed sizes to be different, you must implement the isOutputFixedSizeImpl method to define the output fixed size. You also must use the propagatedInputFixedSize method if the output fixed size status differs from the input fixed size status.

During Simulink model compilation and propagation, the MATLAB System block calls the isOutputFixedSize method, which then calls the isOutputFixedSizeImpl method to determine the output fixed size.

All inputs default to variable-size inputs For these inputs, the output size is the maximum input size.

Note You must set `Access = protected` for this method.

You cannot modify any properties in this method.

Input Arguments

`obj`

System object handle

Output Arguments

`flag_1, flag2, ...`

Logical, scalar value indicating whether the specific output port is fixed size (`true`) or variable size (`false`).

Examples

Specify Output as Fixed Size

Specify in your class definition file that the output from a System object is of fixed size.

```
methods (Access = protected)
    function c1 = isOutputFixedSizeImpl(obj)
        c1 = true;
    end
end
```

- “Specify Whether Output Is Fixed-Size or Variable-Size”

See Also

`matlab.system.mixin.Propagates` | `propagatedInputFixedSize`

Topics

“Specify Whether Output Is Fixed-Size or Variable-Size”

propagatedInputComplexity

Class: matlab.system.mixin.Propagates

Package: matlab.system.mixin

Complexity of input during Simulink propagation

Syntax

```
flag = propagatedInputComplexity(obj, index)
```

Description

`flag = propagatedInputComplexity(obj, index)` returns true or false to indicate whether the input argument for the indicated System object is complex. `index` specifies the input for which to return the complexity flag.

You can use `propagatedInputComplexity` only from within the `isOutputComplexImpl` method in your class definition file. Use `isOutputComplexImpl` when:

- Your System object has more than one input or output.
- The input complexity determines the output complexity.
- The output complexity must differ from the input complexity.

Input Arguments

obj

System object

index

Index of the specified input. Do not count the `obj` in the `index`. The first input is always `obj`.

Output Arguments

flag

Complexity of the specified input, returned as `true` or `false`

Examples

Match Input and Output Complexity

Get the complexity of the second input when you run the object and set the output to match it. Assume that the first input has no impact on the output complexity.

```
methods (Access = protected)
    function outcomplx = isOutputComplexImpl(obj)
        outcomplx = propagatedInputComplexity(obj,2);
    end
end
```

- “Set Output Complexity”

See Also

`isOutputComplexImpl` | `matlab.system.mixin.Propagates`

Topics

“Set Output Complexity”

propagatedInputDataType

Class: matlab.system.mixin.Propagates

Package: matlab.system.mixin

Data type of input during Simulink propagation

Syntax

```
dt = propagatedInputDataType(obj, index)
```

Description

`dt = propagatedInputDataType(obj, index)` returns the data type of an input argument for a System object. `index` specifies the input for which to return the data type.

You can use `propagatedInputDataType` only from within `getOutputDataTypeImpl`. Use `getOutputDataTypeImpl` when:

- Your System object has more than one input or output.
- The input data type status determines the output data type.
- The output data type must differ from the input data type.

Input Arguments

obj

System object

index

Index of the specified input. Do not count the `obj` in the `index`. The first input is always `obj`.

Output Arguments

dt

Data type of the specified input, returned as a character vector for floating-point input or as a numeric type for fixed-point input.

Examples

Match Input and Output Data Type

Get the data type of the second input. If the second input data type is `double`, then the output data type is `int32`. For all other cases, the output data type matches the second input data type. Assume that the first input has no impact on the output.

```
methods (Access = protected)
    function dt = getOutputDataTypeImpl(obj)
        if strcmpi(propagatedInputDataType(obj,2), 'double')
            dt = 'int32';
        else
            dt = propagatedInputDataType(obj,2);
        end
    end
end
```

- “Set Output Data Type”
- “Data Type Propagation”

See Also

`getOutputDataTypeImpl` | `matlab.system.mixin.Propagates`

Topics

- “Set Output Data Type”
- “Data Type Propagation”

propagatedInputFixedSize

Class: matlab.system.mixin.Propagates

Package: matlab.system.mixin

Fixed-size status of input during Simulink propagation

Syntax

```
flag = propagatedInputFixedSize(obj, index)
```

Description

`flag = propagatedInputFixedSize(obj, index)` returns true or false to indicate whether an input argument of a System object is fixed size. `index` specifies the input for which to return the fixed-size flag.

You can use `propagatedInputFixedSize` only from within `isOutputFixedSizeImpl`. Use `isOutputFixedSizeImpl` when:

- Your System object has more than one input or output.
- The input fixed-size status determines the output fixed-size status.
- The output fixed-size status must differ from the input fixed-size status.

Input Arguments

obj

System object

index

Index of the specified input. Do not count the `obj` in the `index`. The first input is always `obj`.

Output Arguments

flag

Fixed-size status of the specified input, returned as `true` or `false`.

Examples

Match Fixed-Size Status of Input and Output

Get the fixed-size status of the third input and set the output to match it. Assume that the first and second inputs have no impact on the output.

```
methods (Access = protected)
    function outtype = isOutputFixedSizeImpl(obj)
        outtype = propagatedInputFixedSize(obj,3)
    end
end
```

- “Specify Whether Output Is Fixed-Size or Variable-Size”

See Also

`isOutputFixedSizeImpl` | `matlab.system.mixin.Propagates`

Topics

“Specify Whether Output Is Fixed-Size or Variable-Size”

propagatedInputSize

Class: matlab.system.mixin.Propagates

Package: matlab.system.mixin

Size of input during Simulink propagation

Syntax

```
sz = propagatedInputSize(obj, index)
```

Description

`sz = propagatedInputSize(obj, index)` returns, as a vector, the input size of the specified System object. The `index` specifies the input for which to return the size information. (Do not count the `obj` in the `index`. The first input is always `obj`.)

You can use `propagatedInputSize` only from within the `getOutputSizeImpl` method in your class definition file. Use `getOutputSizeImpl` when:

- Your System object has more than one input or output.
- The input size determines the output size.
- The output size must differ from the input size.

Note For variable-size inputs, the propagated input size from `propagatedInputSize` differs depending on the environment.

- MATLAB — `propagatedInputSize` returns the size of the inputs used when you run the object for the first time.
 - Simulink — `propagatedInputSize` returns the upper bound of the input sizes.
-

Input Arguments

obj

System object

index

Index of the specified input

Output Arguments

sz

Size of the specified input, returned as a vector

Examples

Match Size of Input and Output

Get the size of the second input. If the first dimension of the second input has a size greater than 1, then set the output size to a 1 x 2 vector. For all other cases, the output is a 2 x 1 matrix. Assume that the first input has no impact on the output size.

```
methods (Access = protected)
    function outsz = getOutputSizeImpl(obj)
        sz = propagatedInputSize(obj,2);
        if sz(1) == 1
            outsz = [1,2];
        else
            outsz = [2,1];
        end
    end
end
```

- “Set Output Size”

See Also

`getOutputSizeImpl` | `matlab.system.mixin.Propagates`

Topics

“Set Output Size”

matlab.system.mixin.Nondirect class

Package: matlab.system.mixin

Nondirect feedthrough mixin class

Description

`matlab.system.mixin.Nondirect` is a class that uses the `output` and `update` methods to process nondirect feedthrough data through a `System` object.

For `System` objects that use direct feedthrough, the object's input is needed to generate the output at that time. For these direct feedthrough objects, running the `System` object calculates the output and updates the state values. For nondirect feedthrough, however, the object's output depends only on the internal states at that time. The inputs are used to update the object states. For these objects, calculating the output with `outputImpl` is separated from updating the state values with `updateImpl`. If you use the `matlab.system.mixin.Nondirect` mixin and include the `stepImpl` method in your class definition file, an error occurs. In this case, you must include the `updateImpl` and `outputImpl` methods instead.

The following cases describe when `System` objects in Simulink use direct or nondirect feedthrough.

- System object supports code generation and does not inherit from the `Propagates` mixin — Simulink automatically infers the direct feedthrough settings from the `System` object code.
- System object supports code generation and inherits from the `Propagates` mixin — Simulink does not automatically infer the direct feedthrough settings. Instead, it uses the value returned by the `isInputDirectFeedthroughImpl` method.
- System object does not support code generation — Default `isInputDirectFeedthroughImpl` method returns `false`, indicating that direct feedthrough is not enabled. To override the default behavior, implement the `isInputDirectFeedthroughImpl` method in your class definition file.

Use the `Nondirect` mixin to allow a `System` object to be used in a Simulink feedback loop. A delay object is an example of a nondirect feedthrough object.

To use this mixin, you must subclass from this class in addition to subclassing from the `matlab.System` base class. Type the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.system & matlab.system.mixin.Nondirect
```

Methods

<code>isInputDirectFeedthroughImpl</code>	Direct feedthrough status of input
<code>outputImpl</code>	Output calculation from input or internal state of System object
<code>updateImpl</code>	Update object states based on inputs

See Also

`matlab.System`

Topics

“Use Update and Output for Nondirect Feedthrough”

isInputDirectFeedthroughImpl

Class: matlab.system.mixin.Nondirect

Package: matlab.system.mixin

Direct feedthrough status of input

Syntax

```
[flag1, ..., flagN] = isInputDirectFeedthroughImpl(obj, u1, u2, ..., uN)
```

Description

[flag1, ..., flagN] = isInputDirectFeedthroughImpl(obj, u1, u2, ..., uN) specifies whether each input is a direct feedthrough input. If direct feedthrough is true, the output depends on the input at each time instant.

Note You must set `Access = protected` for this method.

You cannot modify any properties or implement or access tunable properties in this method.

If you do not include the `isInputDirectFeedthroughImpl` method in your System object class definition file, all inputs are assumed to be direct feedthrough.

The following cases describe when System objects in Simulink code generation use direct or nondirect feedthrough.

- System object supports code generation and does not inherit from the `Propagates` mixin — Simulink automatically infers the direct feedthrough settings from the System object code.
- System object supports code generation and inherits from the `Propagates` mixin — Simulink does not automatically infer the direct feedthrough settings. Instead, it uses the value returned by the `isInputDirectFeedthroughImpl` method.

- System object does not support code generation — Default `isInputDirectFeedthroughImpl` method returns false, indicating that direct feedthrough is not enabled. To override the default behavior, implement the `isInputDirectFeedthroughImpl` method in your class definition file.

`isInputDirectFeedthroughImpl` is called by the MATLAB System block.

Input Arguments

`obj`

System object handle

`u1, u2, . . . , uN`

Specifications of the inputs to the algorithm.

Output Arguments

`flag1, . . . , flagN`

Logical value or either `true` or `false`. This value indicates whether the corresponding input is direct feedthrough or not, respectively. The number of outputs must match the number of outputs returned by the `getNumOutputs` method.

Examples

Specify Input as Nondirect Feedthrough

Use `isInputDirectFeedthroughImpl` in your class definition file to mark the inputs as nondirect feedthrough.

```
methods (Access = protected)
    function flag = isInputDirectFeedthroughImpl(~,~)
        flag = false;
```

```
end  
end
```

- “Use Update and Output for Nondirect Feedthrough”

See Also

`matlab.system.mixin.Nondirect`

Topics

“Use Update and Output for Nondirect Feedthrough”

outputImpl

Class: matlab.system.mixin.Nondirect

Package: matlab.system.mixin

Output calculation from input or internal state of System object

Syntax

```
[y1,y2,...,yN] = outputImpl(obj,u1,u2,...,uN)
```

Description

`[y1,y2,...,yN] = outputImpl(obj,u1,u2,...,uN)` specifies the algorithm to output the System object states. . The output values are calculated from the states and property values. Any inputs that you set to nondirect feedthrough are ignored during output calculation.

`outputImpl` is called by the `output` method. It is also called before the `updateImpl` method. For sink objects, calling `updateImpl` before `outputImpl` locks the object. For all other types of objects, calling `updateImpl` before `outputImpl` causes an error.

Note You must set `Access = protected` for this method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink MATLAB System block.

Input Arguments

obj

System object handle

u1, u2, . . . uN

Inputs from the algorithm. The number of inputs must match the number of inputs returned by the `getNumInputs` method. Nondirect feedthrough inputs are ignored during normal execution of the System object. However, for code generation, you must provide these inputs even if they are empty.

Output Arguments

y1, y2, . . . yN

Outputs calculated from the specified algorithm. The number of outputs must match the number of outputs returned by the `getNumOutputs` method.

Examples

Set Up Output that Does Not Depend on Input

Specify in your class definition file that the output does not directly depend on the current input with the `outputImpl` method. `PreviousInput` is a property of the `obj`.

```
methods (Access = protected)
    function [y] = outputImpl(obj, ~)
        y = obj.PreviousInput(end);
    end
end
```

- “Use Update and Output for Nondirect Feedthrough”

See Also

`matlab.system.mixin.Nondirect` | `matlab.system.mixin.Propagates`

Topics

“Use Update and Output for Nondirect Feedthrough”

updateImpl

Class: matlab.system.mixin.Nondirect

Package: matlab.system.mixin

Update object states based on inputs

Syntax

```
updateImpl(obj, u1, u2, ..., uN)
```

Description

`updateImpl(obj, u1, u2, ..., uN)` specifies the algorithm to update the System object states. You use this method when your algorithm outputs depend only on the object's internal state and internal properties. Do not use this method to update the outputs from the inputs.

`updateImpl` is called by the `update` method and after the `outputImpl` method. For sink objects, calling `updateImpl` before `outputImpl` locks the object. For all other types of objects, calling `updateImpl` before `outputImpl` causes an error.

Note You must set `Access = protected` for this method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink MATLAB System block.

Input Arguments

`obj`

System object handle

u1, u2, . . . uN

Inputs to the algorithm. The number of inputs must match the number of inputs returned by the `getNumInputs` method.

Examples

Set Up Output that Does Not Depend on Current Input

Update the object with previous inputs. Use `updateImpl` in your class definition file. This example saves the `u` input and shifts the previous inputs.

```
methods (Access = protected)
    function updateImpl(obj,u)
        obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
    end
end
```

- “Use Update and Output for Nondirect Feedthrough”

See Also

`matlab.system.mixin.Nondirect`

Topics

“Use Update and Output for Nondirect Feedthrough”

matlab.system.mixin.SampleTime class

Control sample time for System objects in Simulink

Description

`matlab.system.mixin.SampleTime` specifies the sample time specifications for a System object when it is included in a MATLAB System block. Inherit from this mixin class and its methods to control the sample time of your System object in Simulink, via the MATLAB System block.

With this mixin, you can:

- Specify the sample time type
- Specify the sample time
- Specify the offset time
- Get the current simulation time

System objects that inherit from this mixin class must also inherit from `matlab.System`. For example:

```
classdef MySystemObject < matlab.System & matlab.system.mixin.SampleTime
```

Methods

<code>getSampleTimeImpl</code>	Specify sample time type, offset time, and sample time
<code>getSampleTime</code>	Query sample time
<code>getCurrentTime</code>	Current simulation time in MATLAB System block
<code>createSampleTime</code>	Create sample time specification object

See Also

`createSampleTime` | `getCurrentTime` | `getSampleTime` | `getSampleTimeImpl` | `matlab.System`

Topics

“Specify Sample Time for MATLAB System Block System Objects”

Introduced in R2017b

getSampleTimeImpl

Class: matlab.system.mixin.SampleTime

Specify sample time type, offset time, and sample time

Syntax

```
sts = getSampleTimeImpl(obj)
```

Description

`sts = getSampleTimeImpl(obj)` returns the sample time specification created within the method body, `sts`, for the System object `obj`. Specify the sample time specification within the body of `getSampleTimeImpl` by calling `createSampleTime`. The sample time specification affects the simulation time when the System object is included in a MATLAB System block.

This method is called during setup by `setupImpl`.

Input Arguments

obj — System object

system object

System object for which you want to specify the sample time.

Output Arguments

sts — Sample time specification object

sample time specification object

An object defining the sample time specification values. You create this object with the `createSampleTime` function.

Examples

Specify Inherited Sample Time

Specify that the MATLAB System block should inherit the sample from upstream blocks.

```
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj, 'Type', 'Inherited');
end
```

Specify Discrete Periodic Sample Time

Specify a discrete periodic sample time for the MATLAB System block.

```
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj, 'Type', 'Discrete Periodic', ...
        'SampleTime', 10.2, 'OffsetTime', 0.5);
end
```

- “Specify Sample Time for MATLAB System Block System Objects”

See Also

`createSampleTime` | `getCurrentTime` | `getSampleTime` |
`matlab.system.mixin.SampleTime`

Topics

“Specify Sample Time for MATLAB System Block System Objects”

Introduced in R2017b

getSampleTime

Class: matlab.system.mixin.SampleTime

Query sample time

Syntax

```
sts = getSampleTime(obj)
```

Description

`sts = getSampleTime(obj)` returns the sample time specification for the `System` object `obj` when the `System` object is included in a MATLAB System block. You can call `getSampleTime` in the `stepImpl` method to change the algorithm based on the sample time.

Before sample time has propagated throughout the MATLAB System block model, `getSampleTime` returns the `getSampleTimeImpl` sample time specification. If your system object does not override `getSampleTimeImpl`, the default `Inherited` sample time specification is returned.

After sample time has propagated, `getSampleTime` returns the sample time specification populated with the actual MATLAB System block sample time type, sample time, and offset time.

Input Arguments

obj — **System object**
system object

System object included in a MATLAB System block that you want to query.

Output Arguments

sts — Sample time specification object

sample time specification object

The sample time specification for the System object. For more details about sample time specification objects, see `createSampleTime`.

Examples

Return Sample Time

This example of `stepImpl` returns a count value `y`, the current simulation time `ct`, and the sample time `st`. The sample time is obtained by calling `getSampleTime`.

```
function [y,ct,st] = stepImpl(obj,u)
    y = obj.Count + u;
    obj.Count = y;
    ct = getCurrentTime(obj);
    sts = getSampleTime(obj);
    st = sts.SampleTime;
end
```

For a complete class definition, see “Specify Sample Time for MATLAB System Block System Objects”.

- “Specify Sample Time for MATLAB System Block System Objects”

See Also

`createSampleTime` | `getCurrentTime` | `getSampleTimeImpl` | `matlab.system.mixin.SampleTime`

Topics

“Specify Sample Time for MATLAB System Block System Objects”

Introduced in R2017b

getCurrentTime

Class: matlab.system.mixin.SampleTime

Current simulation time in MATLAB System block

Syntax

```
t = getCurrentTime(obj)
```

Description

`t = getCurrentTime(obj)` returns the current simulation time in the MATLAB System block. Call this method in the `stepImpl` method of your System object.

Input Arguments

obj — System object

system object

System object included in a MATLAB System block that you want to query.

Output Arguments

t — Current simulation time

double

The current simulation time of the MATLAB System block that contains the System object.

Examples

Return Current Simulation Time

This example of `stepImpl` returns a count value `y` and the current simulation time `ct`. The simulation time is obtained by calling `getCurrentTime`.

```
function [y,ct] = stepImpl(obj,u)
    y = obj.Count + u;
    obj.Count = y;
    ct = getCurrentTime(obj);
end
```

For a complete class definition, see “Specify Sample Time for MATLAB System Block System Objects”.

- “Specify Sample Time for MATLAB System Block System Objects”

See Also

`createSampleTime` | `getSampleTime` | `getSampleTimeImpl` |
`matlab.system.mixin.SampleTime`

Topics

“Specify Sample Time for MATLAB System Block System Objects”

Introduced in R2017b

createSampleTime

Class: matlab.system.mixin.SampleTime

Create sample time specification object

Syntax

```
sts = createSampleTime(obj)
sts = createSampleTime(obj, 'Type', Type)
sts = createSampleTime(obj, Name, Value)
```

Description

`sts = createSampleTime(obj)` creates a sample time object for inherited sample time for the System object `obj`. Use this sample time object in the `getSampleTimeImpl` method of your System object. The sample time specification affects the simulation time when the System object is included in a MATLAB System block.

`sts = createSampleTime(obj, 'Type', Type)` creates a sample time object with the specified sample time type.

`sts = createSampleTime(obj, Name, Value)` creates a sample time object with additional options specified by one or more `Name, Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Input Arguments

obj — System object

system object

System object that you want to specify the sample time.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'SampleTime', 1`

Type — Sample time type

'Inherited' (default) | 'Discrete Periodic' | 'Fixed In Minor Step'

Type of sample time you want the System object to use in Simulink. For descriptions of the different types of sample times, see:

- 'Inherited' - “Inherited Sample Time”
- 'Discrete Periodic' - “Discrete Sample Time”
- 'Fixed In Minor Step' - “Fixed-in-Minor-Step”

Data Types: character

SampleTime — Time between samples

1 (default) | positive number

For `Discrete Periodic` sample time only.

Specify the time between sample hits in Simulink.

Data Types: single | double

offsetTime — Offset from sample time

0 (default) | nonnegative number less than `SampleTime`

For `Discrete Periodic` sample time only.

Specify the offset time for the sample hit. The offset is a time interval indicating an update delay. The block is updated later in the sample interval than other blocks operating at the same sample rate.

The offset time must be nonnegative and less than `SampleTime`.

Data Types: single | double

Output Arguments

sts — Sample time specification object

sample time specification object

The sample time specification object. This object has the following properties:

- **Type** — Type of sample time
- **SampleTime** — Time between samples
- **OffsetTime** — Offset from sample time

Use this object as the return value of `getSampleTimeImpl`.

Examples

Create Inherited Sample Time Objects

Specify that the MATLAB System block should inherit the sample from upstream blocks. Inherited sample time is the default, so no additional arguments are needed.

```
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj);
end
```

Create Discrete Periodic Sample Time Object

Specify a discrete periodic sample time specification for the MATLAB System block, including offset time and the sample time.

```
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj, 'Type', 'Discrete Periodic', ...
        'SampleTime', 10.2, 'OffsetTime', 0.5);
end
```

- “Specify Sample Time for MATLAB System Block System Objects”

See Also

`getCurrentTime` | `getSampleTime` | `getSampleTimeImpl` |
`matlab.system.mixin.SampleTime`

Topics

“Specify Sample Time for MATLAB System Block System Objects”

Introduced in R2017b

ModelAdvisor.Preferences class

Package: ModelAdvisor

Set Model Advisor window preferences by specifying which folders and tabs to display

Description

Use instances of this class to set Model Advisor preferences.

Construction

The constructor `ModelAdvisor.Preferences` creates an instance of this class with default property values.

Create an instance `modelPreferences` of the `ModelAdvisor.Preferences` class.

```
modelPreferences = ModelAdvisor.Preferences;
```

Properties

DeselectByProduct — Deselect the By Product folder

(default) | `true`

Selection of the **By Product** folder in the Model Advisor window. The default value is `true`.

Example: `true`

Data Types: `logical`

ShowAccordion — Display advisors

(default) | `false`

Display of the **Code Generation Advisor**, **Upgrade Advisor**, and **Performance Advisor** in the Model Advisor window. The default value is `false`. You can use these

advisors to help configure your model for code generation, upgrade your model for the current release, or improve performance.

Example: `true`

Data Types: `logical`

ShowByProduct — Display the By Product folder

(default) | `true`

Display of the **By Product** folder in the Model Advisor window. The default value is `true`.

Example: `true`

Data Types: `logical`

ShowByTask — Display the By Task folder

(default) | `true`

Display of the **By Task** folder in the Model Advisor window. The default value is `true`.

Example: `true`

Data Types: `logical`

ShowExclusionsInRpt — Include exclusions in report

(default) | `true`

Include exclusions in the Model Advisor report. The default value is `true`.

Example: `true`

Data Types: `logical`

ShowExclusionTab — Display the Exclusions tab

(default) | `false`

Display of the **Exclusions** tab in the Model Advisor window. The default value is `false`. When you click the **Exclusions** tab, the Model Advisor window displays checks that are excluded from the Model Advisor analysis.

Example: `true`

Data Types: `logical`

ShowSourceTab — Display the Source tab

(default) | `false`

Display of the **Source** tab in the Model Advisor window. The default value is `false`. When you click the **Source** tab, the Model Advisor window displays the check Title, TitleID, and location of the MATLAB source code for the check.

Example: `true`

Data Types: `logical`

Examples

Turn Off Display Of By Product Folder

This example shows how to not display the **By Product** folder in the Model Advisor window:

```
mp = ModelAdvisor.Preferences;  
mp.load;  
mp.ShowByProduct = false;  
mp.save
```

Alternatives

You can set the Model Advisor preferences by using the Model Advisor Preferences dialog box:

- On the Model Advisor menu, select **Settings > Preferences**.
- From the Model Editor, select **Analysis > Model Advisor > Preferences**.

See Also

“Run Model Checks”

Introduced in R2014b

Simulink.AliasType

Create alias for signal and parameter data type

Description

This class allows you to designate MATLAB variables as aliases for signal and parameter data types. You designate variables as aliases by creating instances of this class and assigning them to variables in the MATLAB base workspace or a data dictionary (see “Creating a Data Type Alias” on page 5-158). The MATLAB variable to which a `Simulink.AliasType` object is assigned is called a data type alias. The data type to which an alias refers is called its base type. Simulink software allows you to set the `BaseType` property of the object that the variable references, thereby designating the data type for which it is an alias.

Simulink software lets you use aliases instead of actual type names in dialog boxes and `set_param` commands to specify the data types of Simulink block outputs and parameters. Using aliases to specify signal and parameter data types can greatly simplify global changes to the signal and parameter data types that a model specifies. In particular, changing the data type of all signals and parameters whose data type is specified by an alias requires changing only the base type of the alias. By contrast, changing the data types of signals and parameters whose data types are specified by an actual type name requires respecifying the data type of each signal and parameter individually.

You can use objects of this class to create an alias for Simulink built-in data types, fixed-point data types, enumerated data types, `Simulink.NumericType` objects, and other `Simulink.AliasType` objects. The generated code uses the alias only if you use an ERT-based system target file (Embedded Coder).

To generate `Simulink.AliasType` objects that correspond to `typedef` statements in your external C code, consider using the `Simulink.importExternalCTypes` function.

Alternatively, to define and name a numeric data type, you can use an object of the class `Simulink.NumericType`.

Creating a Data Type Alias

You can use either the Model Explorer or MATLAB commands to create a data type alias. See “MATLAB Commands for Creating Data Type Aliases” on page 5-158.

To use the Model Explorer to create an alias:

- 1 On the Model Explorer **Model Hierarchy** pane, select `Base Workspace`.

You must create data type aliases in the MATLAB workspace or in a data dictionary. If you attempt to create an alias in a model workspace, Simulink software displays an error.

- 2 From the Model Explorer **Add** menu, select **Simulink.AliasType**.

Simulink software creates an instance of a `Simulink.AliasType` object and assigns it to a variable named `Alias` in the MATLAB workspace.

- 3 Rename the variable to a more appropriate name, for example, a name that reflects its intended usage.

To change the name, edit the name displayed in the **Name** field on the Model Explorer **Contents** pane.

- 4 On the Model Explorer **Dialog** pane, in the **Base type** field, enter the name of the data type that this alias represents.

You can specify the name of any existing standard or user-defined data type in this field. Skip this step if the base type is `double` (the default).

MATLAB Commands for Creating Data Type Aliases

Use the following syntax to create a data type alias at the command prompt or in a MATLAB script.

```
ALIAS = Simulink.AliasType;
```

`ALIAS` is the name of the variable that you want to serve as the alias. For example, the following line creates an alias named `MyFloat`.

```
MyFloat = Simulink.AliasType;
```

The following notations get and set the properties of a data type alias, respectively:

```
PROPVALUE = ALIAS.PROPNAME;  
ALIAS.PROPNAME = PROPVALUE;
```

ALIAS is the name of the alias object, PROPNAME is the name of the alias object properties, and PROPVALUE is the property value. For example, the following code saves the current value of MyFloat's BaseType property and assigns it a new value.

```
old = MyFloat.BaseType;  
MyFloat.BaseType = 'single';
```

For information on the names, permitted values, and usage of the properties of data type alias objects, see “Properties” on page 5-164.

Create Alias for Enumerated Data Type

You can use a Simulink.AliasType object to create an alias for an enumerated data type. For example, you can create an alias for an enumerated type called SLDemoSign.

```
myEnumAlias = Simulink.AliasType;  
myEnumAlias.BaseType = 'Enum: SLDemoSign';
```

You can use the expression myEnumAlias to specify the data type of signals or parameters as the enumerated type SLDemoSign.

Create Alias for Fixed-Point Data Type

To create an alias for a fixed-point data type by using a Simulink.AliasType object, set the BaseType property of the object by using a call to the fixdt function. Because the value of BaseType must be a character vector, wrap the function call in a character vector. For example, this code creates an alias for an unsigned fixed-point data type with word length 16 and fraction length 7.

```
myFixptAlias = Simulink.AliasType;  
myFixptAlias.BaseType = 'fixdt(0,16,7)';
```

Alternatively, to specify the characteristics of the fixed-point data type interactively, in the property dialog box for myFixptAlias, you can use the Data Type Assistant. To open the property dialog box, at the command prompt, enter:

```
open('myFixptAlias')
```

For information about using the Data Type Assistant, see “Specify Data Types Using Data Type Assistant”.

Data Type Aliases in Generated Code

If you have a Simulink Coder license, you can cause data type aliases to appear in the code generated for a model using any of the following methods:

- Specifying the signal data type of a block in the model as a `Simulink.AliasType` via the **Block Parameters** dialog box.
- Creating a `Simulink.Signal` object that uses the `Simulink.AliasType` as its data type. Use this signal object as the name of a signal in the model and specify that the signal name must resolve to an object in the MATLAB workspace. See “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder) in the Simulink Coder User's Guide.
- Creating a `Simulink.Parameter` object that uses the `Simulink.AliasType` as its data type. Use this parameter object as a block parameter in the model. See “Parameter Data Types in the Generated Code” (Simulink Coder) in the Simulink Coder User's Guide.

Notes

- If you assign a data type in a block dialog box and if you use a `Simulink.Signal` object on the signal feeding into the block, the code is always generated using the data type in the dialog box.
- The Simulink Coder code generator tries to preserve the names of alias types in the generated code. However, in some cases, an alias type name can revert to its underlying equivalent built-in data type. If you have an Embedded Coder license, you can specify that the code generator use the alias type name in the generated code, by using replacement types (see “Model Configuration Parameters: Code Generation Data Type Replacement” (Embedded Coder)).
- The `Simulink.AliasType` class does not support multiword data types for code generation.
- You can specify the data type of a complex signal using the `Simulink.AliasType` class. In this case, if the **DataScope** property of the `Simulink.AliasType` class is set to `Imported` (or `Auto` with a header file specified), provide a definition for the complex type. As shown in the following example, the alias type definition, `IAT_int32`, must contain the name of the complex type prefixed by `c`.

```
#ifndef myAliasTypes_H_
#define myAliasTypes_H_

#include "rtwtypes.h"

typedef int32_T IAT_int32;
typedef cint32_T cIAT_int32;

#endif
```

In the preceding example, while you must define `IAT_int32` in the base workspace, you do not need to define `cIAT_int32` in the base workspace.

- If you define two nested alias types, Simulink Coder generates an error if the **DataScope** property of the alias type is set to `Imported` or if either of the alias types specifies a header file.
-

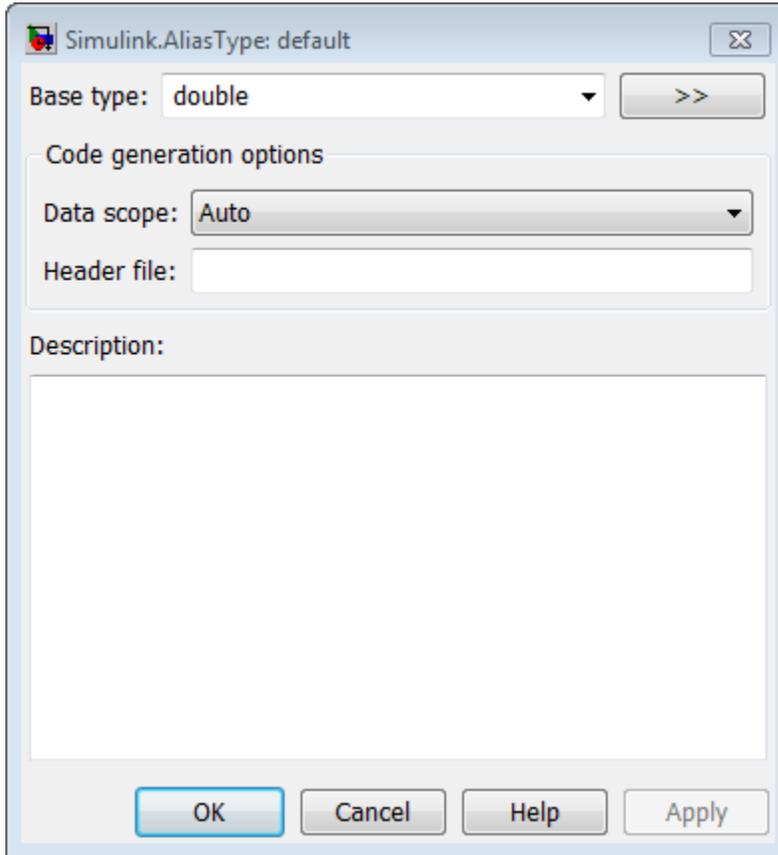
Parent

None

Children

None

Property Dialog Box



Base type

The data type to which this alias refers. The default is `double`. To specify another data type, select the data type from the adjacent drop-down list of standard data types or enter the data type name in the edit field.

To specify a fixed-point data type, you can use a call to the `fixdt` function, such as `fixdt(0,16,7)`. To specify the characteristics of the type interactively, expand the Data Type Assistant and set **Mode** to `Fixed point`. For information about using the Data Type Assistant, see “Specify Data Types Using Data Type Assistant”.

You can, with one exception, specify a nonstandard data type, e.g., a data type defined by a `Simulink.NumericType` object, by entering the data type name in the edit field. The exception is a `Simulink.NumericType` whose `Category` is `Fixed-point: unspecified scaling`.

Note `Fixed-point: unspecified scaling` is a partially specified type whose definition is completed by the block that uses the `Simulink.NumericType`. Forbidding its use in alias types avoids creating aliases that have different base types depending on where they are used.

Data scope

Specifies whether the data type definition is imported from, or exported to, a header file during code generation. The possible values are:

Value	Action
Auto (default)	If no value is specified for Header file , export the type definition to <code>model_types.h</code> , where <code>model</code> is the model name. If you have an Embedded Coder license, and you have specified a data type replacement, then export the type definition to <code>rtwtypes.h</code> . If a value is specified for Header file , import the data type definition from the specified header file.
Exported	Export the data type definition to a header file, which can be specified in the Header file field. If no value is specified for Header file , the header file name defaults to <code>type.h</code> . <code>type</code> is the data type name.
Imported	Import the data type definition from a header file, which can be specified in the Header file field. If no value is specified for Header file , the header file name defaults to <code>type.h</code> . <code>type</code> is the data type name.

Header file

Name of a C header file from which a data type definition is imported, or to which a data type definition is exported, based on the value of **Data scope**. If this field is specified, the specified name is used during code generation for importing or exporting. If this field is empty, the value defaults to `type.h` if **Data scope** equals `Imported` or `Exported`, or defaults to `model_types.h` if **Data scope** equals `Auto`.

By default, the generated `#include` directive uses the preprocessor delimiter `"` instead of `<` and `>`. To generate the directive `#include <myTypes.h>`, specify **Header file** as `<myTypes.h>`.

Description

Describes the usage of the data type referenced by this alias.

Properties

Name	Description
BaseType	A character vector specifying the name of a standard or custom data type. (Base type)
DataScope	A character vector specifying whether the data type definition is imported from, or exported to, a header file during code generation. (Data scope)
Description	A character vector that describes the usage of the data type. Can be a null character vector. (Description)
HeaderFile	A character vector that specifies the name of a C header file from which a data type definition is imported, or to which a data type definition is exported, during code generation. (Header file)

See Also

`Simulink.NumericType`

Topics

“Control Signal Data Types”

“Control Data Type Names in Generated Code” (Embedded Coder)

“About Data Types in Simulink”

Introduced before R2006a

Simulink.Annotation

Specify properties of model annotation

Description

Instances of this class specify the properties of annotations. You can use `getCallbackAnnotation` in an annotation callback function to get the `Simulink.Annotation` instance for the annotation associated with the callback function. You can use `find_system` and `get_param` to get the `Simulink.Annotation` instance associated with any annotation in a model. For example, this code gets the annotation object for the first annotation in the currently selected model and turns on its drop shadow

```
ah = find_system(gcs,'FindAll','on','type','annotation');
ao = get_param(ah(1),'Object');
ao.DropShadow = 'on';
```

Children

None.

Property Summary

Property	Description	Values
<code>AnnotationType</code>	Type of annotation—text (note), area, or image. This property is read only.	<code>note_annotation</code> <code>area_annotation</code> <code>image_annotation</code>

Property	Description	Values
BackgroundColor	Background color of this annotation.	<p>RGB value array [r, g, b, a] where r, g, b, and a are the red, green, blue, and alpha values of the color normalized to the range 0.0 to 1.0, delineated with commas. The alpha value is optional and ignored.</p> <p>Annotation background color can also be 'black', 'white', 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'gray', 'lightBlue', 'orange', 'darkGreen'.</p>
ClickFcn	Specifies MATLAB code to execute when you click this annotation. See “Associate a Click Function with an Annotation” for more information.	character vector
DeleteFcn	MATLAB code to execute before deleting this annotation. See “Annotation Callback Functions”.	character vector
Description	Description of this annotation.	character vector
DropShadow	Turn drop shadow display on or off.	'on' {'off'}
FixedHeight	Specify whether the bottom border of the annotation resizes as you add content	'on' {'off'}, where 'off' means that the bottom border resizes as you add content
FixedWidth	Specify whether to use word wrap or to have the width of the annotation expand to accommodate text	'on' {'off'}, where 'off' means to use word wrap

Property	Description	Values
FontAngle	Angle of the annotation font. The default value, 'auto', uses of the default font angle specified for lines in the Font Styles dialog box.	'normal' 'italic' 'oblique' {'auto'}
FontName	Name of annotation font. The default value, 'auto', uses the default font specified for lines in the Font Styles dialog box.	character vector
FontSize	Size of annotation font in points. The default value, -1, uses the default text size for lines specified in the Font Styles dialog box.	decimal number {'-1'}
FontWeight	Weight of the annotation font. The default value, 'auto', use of the default weight for lines specified in the Font Styles dialog box.	'light' 'normal' 'demi' 'bold' {'auto'}
ForegroundColor	Foreground color of this annotation.	<p>RGB value array [r, g, b, a] where r, g, b, and a are the red, green, blue, and alpha values of the color normalized to the range 0.0 to 1.0, delineated with commas. The alpha value is optional and ignored.</p> <p>Annotation background color can also be 'black', 'white', 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'gray', 'lightBlue', 'orange', 'darkGreen'.</p>
Handle	Annotation handle.	real
HiliteAncestors	For internal use.	

Property	Description	Values
Horizontal-Alignment	Horizontal alignment of this annotation.	'center' {'left'} 'right'
Interpreter	Specifies whether the annotation is interpreted as rich or contains LaTeX commands	'rich' 'tex' {'off'}
IsImage	Specifies whether the annotation is an image-only annotation.	'on' {'off'}
InternalMargins	Space from the bounding box of text to the borders of the annotation.	1x4 array [left top right bottom]. The default is [1 1 1 1]. The maximum value for a coordinate is 32767.
LoadFcn	MATLAB code to execute when the model containing this annotation is loaded. See “Annotation Callback Functions”.	character vector
Name	Text of annotation. Same as Text.	character vector
Parent	Parent name of annotation object.	character vector
Path	Path to the annotation.	character vector
PlainText	Read-only display of the text in the annotation, without formatting	vector
Position	Location of the annotation	1x4 array [left top right bottom]. The maximum value for a coordinate is 32767.
RequirementInfo	For internal use.	character vector
Selected	Specifies whether this annotation is currently selected.	'on' 'off'

Property	Description	Values
Tag	Text to assign to the annotation Tag parameter and save with the annotation.	character vector
TeXMode	Specifies whether to render TeX markup.	'on' {'off'}
Text	Text of annotation. Same as Name.	character vector
Type	Annotation type. This is always 'annotation'.	'annotation'
UseDisplayText-AsClickCallback	<p>Specifies whether to use the contents of the Text property as the click function for this annotation.</p> <p>If set to 'on', the text of the annotation is interpreted as a valid MATLAB expression and run. If set to 'off', clicking the annotation runs the click function, if there is one. If there is no click function, clicking the annotation has no effect.</p> <p>See “Associate a Click Function with an Annotation” for more information.</p>	'on' {'off'}
UserData	Any data that you want to associate with this annotation.	vector
VerticalAlignment	Vertical alignment of this annotation.	'middle' {'top'} 'cap' 'baseline' 'bottom'

Method Summary

Method	Description
delete	Delete this annotation from the Simulink model.
dialog	Display the Annotation properties dialog box.
disp	Display the property names and their settings for this Annotation object.
fitToView	Zoom in on this annotation and highlight it in the model.
get	Return the specified property settings for this annotation.
help	Display a list of properties for this Annotation object with short descriptions.
methods	Display all nonglobal methods of this Annotation object.
set	Set the specified property of this Annotation object with the specified value.
setImage	Set the annotation contents to the specified image file. The resulting annotation is an image-only annotation.
struct	Return and display a MATLAB structure containing the property settings of this Annotation object.
view	Display this annotation in the Simulink Editor with this annotation highlighted.

Introduced before R2006a

Simulink.BlockCompDworkData

Provide postcompilation information about block's DWork vector

Description

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the “Dwork” on page 5-377 method of a block's runtime object after the model containing the block has been compiled.

Parent

`Simulink.BlockData`

Children

None

Property Summary

Name	Description
“Usage” on page 5-171	Usage type of this DWork vector.
“UsedAsDiscState” on page 5-172	True if this DWork vector is being used to store the values of a block's discrete states.

Properties

Usage

Returns a character vector indicating how this DWork vector is used. Permissible values are:

- DWork
- DState
- Scratch
- Mode

character vector

RW for MATLAB S-function blocks, RO for other blocks.

UsedAsDiscState

True if this DWork vector is being used to store the values of a block's discrete states.

Boolean

RW for MATLAB S-Function blocks, RO for other blocks.

Introduced before R2006a

Simulink.BlockComplInputPortData

Provide postcompilation information about block input port

Description

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the “InputPort” on page 5-378 method of a block's run-time object after the model containing the block has been compiled.

Parent

`Simulink.BlockPortData`

Children

None

Property Summary

Name	Description
“DirectFeedthrough” on page 5-173	True if this port has direct feedthrough.
“Overwritable” on page 5-174	True if this port is overwritable.

Properties

DirectFeedthrough

True if this input port has direct feedthrough.

Boolean

RW for MATLAB S functions, RO for other blocks.

Overwritable

True if this input port is overwritable.

Boolean

RW for MATLAB S functions, RO for other blocks.

Introduced before R2006a

Simulink.BlockCompOutputPortData

Provide postcompilation information about block output port

Description

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the “OutputPort” on page 5-379 method of a block's run-time object after the model containing the block has been compiled.

Parent

`Simulink.BlockPortData`

Children

None

Property Summary

Name	Description
“Reusable” on page 5-196	Specifies whether an output port's memory is reusable.

Properties

Reusable

Specifies whether an output port's memory is reusable. Options are: `NotReusableAndGlobal` and `ReusableAndLocal`.

character vector

RW for MATLAB S functions, RO for other blocks.

Introduced before R2006a

Simulink.BlockData

Provide run-time information about block-related data, such as block parameters

Description

This class defines properties that are common to objects that provide run-time information about a block's ports and work vectors.

Parent

None

Children

`Simulink.BlockPortData`, `Simulink.BlockCompDworkData`

Property Summary

Name	Description
“AliasedThroughDataType” on page 5-178	Fundamental base data type.
“AliasedThroughDataTypeID” on page 5-179	Fundamental base data type ID.
“Complexity” on page 5-179	Numeric type (real or complex) of the block data.
“Data” on page 5-179	The block data.
“DataAsDouble” on page 5-179	The block data in <code>double</code> form.
“Datatype” on page 5-180	Data type of the block data.
“DatatypeID” on page 5-180	Index of the data type of the block data.

Name	Description
“Dimensions” on page 5-181	Dimensions of the block data.
“Name” on page 5-181	Name of the block data.
“Type” on page 5-181	Type of block data (e.g., a parameter).

Properties

AliasedThroughDataType

Data type aliases allow a data type (B) to be recursively aliased to another alias type or BaseType (A). If alias type A is aliased to another alias type that is aliased to another alias type and so forth, this property allows the alias type to be iteratively searched (aliased through) until the type is no longer an alias type and that final result is the value of the property returned. For example, assume that you have created the Simulink Alias types A and B as follows:

```
A=Simulink.AliasType('double')
```

```
A =  
Simulink.AliasType  
    Description: ''  
    HeaderFile: ''  
    BaseType: 'double'  
B=Simulink.AliasType('A')
```

```
B =  
Simulink.AliasType  
    Description: ''  
    HeaderFile: ''  
    BaseType: 'A'
```

If the data type of an item of block data is B, this property returns the base type A instead of B.

character vector

RO

AliasedThroughDataTypeID

Index of the data type alias returned by the `AliasedThroughDataType` property.

integer

RO

Complexity

Numeric type (real or complex) of the block data.

character vector

RW for MATLAB S functions, RO for other blocks.

Data

The block data.

The data type specified by the “Datatype” on page 5-180 or “DatatypeID” on page 5-180 properties of this object.

RW

DataAsDouble

The block data's in `double` form.

`double`

RO

Datatype

Data type of the values of the block-related object.

character vector

RO

DatatypeID

Index of the data type of the values of the block-related object. enter the numeric value for the desired data type, as follows:

Data Type	Value
'inherited'	-1
'double'	0
'single'	1
'int8'	2
'uint8'	3
'int16'	4
'uint16'	5
'int32'	6
'uint32'	7
'boolean' or fixed-point data types	8

integer

RW for MATLAB S functions, RO for other blocks

Dimensions

Dimensions of the block-related object, e.g., parameter or DWork vector.

array

RW for MATLAB S functions, RO for other blocks

Name

Name of block-related object, e.g., a block parameter or DWork vector.

character vector

RW for MATLAB S functions, RO for other blocks

Type

Type of block data. Possible values are:

Type	Description
'BlockPreCompInputPortData '	This object contains data for an input port before the model is compiled.
'BlockPreCompOutputPortData '	This object contains data for an output port before the model is compiled.
'BlockCompInputPortData '	This object contains data for an input port after the model is compiled.
'BlockCompOutputPortData '	This object contains data for an output port after the model is compiled.
'BlockPreCompDworkData '	This object contains data for a DWork vector before the model is compiled.

Type	Description
'BlockCompDworkData '	This object contains data for a DWork vector after the model is compiled.
'BlockDialogPrmData '	This object describes a dialog box parameter of a Level-2 MATLAB S-function.
'BlockRuntimePrmData '	This object describes a run-time parameter of a Level-2 MATLAB S-function.
'BlockCompContStatesData '	This object describes the continuous states of the block at the current time step.
'BlockDerivativesData '	This object describes the derivatives of the block's continuous states at the current time step.

character vector

RO

Introduced before R2006a

Simulink.BlockPath

Fully specified Simulink block path

Description

A `Simulink.BlockPath` object represents a fully specified block path that uniquely identifies a block within a model hierarchy, including model reference hierarchies that involve multiple instances of a referenced model. Simulink uses block path objects in a variety of contexts. For example, when you specify normal mode visibility, Simulink uses block path objects to identify the models with Normal mode visibility. For details, see “Normal Mode Visibility”.

The `Simulink.BlockPath` class is very similar to the `Simulink.SimulationData.BlockPath` class.

You must have Simulink installed to use the `Simulink.BlockPath` class. However, you do not have to have Simulink installed to use the `Simulink.SimulationData.BlockPath` class. If you have Simulink installed, consider using `Simulink.BlockPath` instead of `Simulink.SimulationData.BlockPath`, because the `Simulink.BlockPath` class includes a method for checking the validity of block path objects without you having to update the model diagram.

Property Summary

Name	Description
SubPath on page 5-184	Individual component within the block specified by the block path

Method Summary

Name	Description
BlockPath on page 5-184	Create a block path.

Name	Description
convertToCell on page 5-187	Convert a block path to a cell array of character vectors.
getBlock on page 5-188	Get a single block path in the model reference hierarchy.
getLength on page 5-189	Get the length of the block path.
validate on page 5-189	Determine whether the block path represents a valid block hierarchy.

Properties

SubPath

Represents an individual component within the block specified by the block path.

For example, if the block path refers to a Stateflow chart, you can use SubPath to indicate the chart signals. For example:

```
Block Path:  
    'sf_car/shift_logic'  
  
SubPath:  
    'gear_state.first'
```

character vector

RW

Methods

BlockPath

Create block path

```
blockpath_object = Simulink.BlockPath()  
blockpath_object = Simulink.BlockPath(blockpath)  
blockpath_object = Simulink.BlockPath(paths)  
blockpath_object = Simulink.BlockPath(paths, subpath)
```

blockpath

Block path object that you want to copy.

paths

A character vector or cell array of character vectors that Simulink uses to build the block path.

Specify each character vector in order, from the top model to the specific block for which you are creating a block path.

Each character vector must be a path to a block within the Simulink model. The block must be:

- A block in a single model
- A Model block (except for the last character vector, which may be a block other than a Model block)
- A block that is in a model that is referenced by a Model block that is specified in the previous character vector

When you create a block path for specifying Normal mode visibility:

- The first character vector must represent a block that is in the top model in the model reference hierarchy.
- Character vectors must represent Model blocks that are in Normal mode.
- Character vectors that represent variant models or variant subsystems must refer to an active variant.

You can use `gcb` in the cell array to specify the currently selected block.

subpath

Character vector that represents an individual component within a block.

blockpath_object

Block path that you create.

`blockpath_object = Simulink.BlockPath()` creates an empty block path.

`blockpath_object = Simulink.BlockPath(blockpath)` creates a copy of the block path of the block path object that you specify with the `source_blockpath` argument.

`blockpath = Simulink.BlockPath(paths)` creates a block path from the cell array of character vectors that you specify with the `paths` argument. Each character vector represents a path at a level of model hierarchy. Simulink builds the full block path based on the character vectors.

`blockpath = Simulink.BlockPath(paths, subpath)` creates a block path from the character vector or cell array of character vectors that you specify with the `paths` argument and creates a path for the individual component (for example, a signal) of the block.

Create a block path object called `bp1`, using `gcb` to get the current block.

```
sldemo_mdref_depgraph
bp1 = Simulink.BlockPath(gcb)
```

The resulting block path is the top-level Model block called `thermostat` (the top-left Model block).

```
bp1 =

  Simulink.BlockPath
  Package: Simulink

  Block Path:
    'sldemo_mdref_depgraph/thermostat'
```

Create a block path object called `bp2`, using a cell array of character vectors representing elements of the block path.

```
sldemo_mdref_depgraph
bp2 = Simulink.BlockPath({'sldemo_mdref_depgraph/thermostat', ...
'sldemo_mdref_heater/Fahrenheit to Celsius', ...
'sldemo_mdref_F2C/Gain1'})
```

The resulting block path reflects the model reference hierarchy for the block path

```

bp2 =

    Simulink.BlockPath
    Package: Simulink

    Block Path:
        'sldemo_mdllref_depgraph/thermostat'
        'sldemo_mdllref_heater/Fahrenheit to Celsius'
        'sldemo_mdllref_F2C/Gain1'

```

convertToCell

Convert block path to cell array of character vectors

```
cellarray = Simulink.BlockPath.convertToCell()
```

```
cellarray
```

Cell array of character vectors representing elements of block path.

`cellarray = Simulink.BlockPath.convertToCell()` converts a block path to a cell array of character vectors.

```

sldemo_mdllref_depgraph
bp2 = Simulink.BlockPath({'sldemo_mdllref_depgraph/thermostat', ...
'sldemo_mdllref_heater/Fahrenheit to Celsius', ...
'sldemo_mdllref_F2C/Gain1'})
cellarray_for_bp2 = bp2.convertToCell()

```

The result is a cell array representing the elements of the block path.

```

cellarray_for_bp2 =

    'sldemo_mdllref_depgraph/thermostat'
    'sldemo_mdllref_heater/Fahrenheit to Celsius'
    'sldemo_mdllref_F2C/Gain1'

```

getBlock

Get block path in model reference hierarchy

```
block = Simulink.BlockPath.getBlock(index)
```

index

The index of the block for which you want to get the block path. The index reflects the level in the model reference hierarchy. An index of 1 represents a block in the top-level model, an index of 2 represents a block in a model referenced by the block of index 1, and an index of n represents a block that the block with index $n-1$ references.

block

The block representing the level in the model reference hierarchy specified by the index argument.

`blockpath = Simulink.BlockPath.getBlock(index)` returns the block path of the block specified by the index argument.

Get the block for the second level in the model reference hierarchy.

```
sldemo_mdhref_depgraph
bp2 = Simulink.BlockPath({'sldemo_mdhref_depgraph/thermostat', ...
'sldemo_mdhref_heater/Fahrenheit to Celsius', ...
'sldemo_mdhref_F2C/Gain1'})
blockpath = bp2.getBlock(2)
```

The result is the thermostat block, which is at the second level in the block path hierarchy.

```
blockpath =
```

```
sldemo_mdhref_heater/Fahrenheit to Celsius
```

getLength

Get length of block path

```
length = Simulink.BlockPath.getLength()
```

length

The length of the block path. The length is the number of levels in the model reference hierarchy.

`length = Simulink.BlockPath.getLength()` returns a numeric value that corresponds to the number of levels in the model reference hierarchy for the block path.

Get the length of block path bp2.

```
sldemo_mdldref_depgraph
bp2 = Simulink.BlockPath({'sldemo_mdldref_depgraph/thermostat', ...
'sldemo_mdldref_heater/Fahrenheit to Celsius', ...
'sldemo_mdldref_F2C/Gain1'})
length_bp2 = bp2.getLength()
```

The result reflects that the block path has three elements.

```
length_bp2 =
```

```
3
```

validate

Determine whether block path represents valid block hierarchy

```
Simulink.BlockPath.validate()
Simulink.BlockPath.validate(AllowInactiveVariant)
```

`Simulink.BlockPath.validate()` determines whether the block path represents a valid block hierarchy. If there are any validity issues, messages appear in the MATLAB command window. The method checks that:

- All elements in the block path represent valid blocks.
- Each element except for the last element:
 - Is a valid Model block
 - References the model of the next element

See Also

`Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.Dataset`

Simulink.BlockPortData

Describe block input or output port

Description

This class defines properties that are common to objects that provide run-time information about a block's ports.

Parent

`Simulink.BlockData`

Children

`Simulink.BlockPreCompInputPortData`,
`Simulink.BlockPreCompOutputPortData`, `Simulink.BlockCompInputPortData`,
`Simulink.BlockCompOutputPortData`

Property Summary

Name	Description
“IsBus” on page 5-192	True if this port is connected to a bus.
“IsSampleHit” on page 5-192	True if this port produces output or accepts input at the current simulation time step.
“SampleTime” on page 5-192	Sample time of this port.
“SampleTimeIndex” on page 5-192	Sample time index of this port.

Properties

IsBus

True if this port is connected to a bus.

Boolean

RO

IsSampleHit

True if this port produces output or accepts input at the current simulation time step.

Boolean

RO

SampleTime

Sample time of this port.

[period offset] where `period` and `offset` are values of type `double`. See “Specify Sample Time” for more information.

RW for MATLAB S functions, RO for other blocks

SampleTimeIndex

Sample time index of this port.

integer

RO

Introduced before R2006a

Simulink.BlockPreCompInputPortData

Provide precompilation information about block input port

Description

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the “InputPort” on page 5-378 method of a block's run-time object before the model containing the block has been compiled.

Parent

`Simulink.BlockPortData`

Children

None

Property Summary

Name	Description
“DirectFeedthrough” on page 5-194	True if this port has direct feedthrough.
“Overwritable” on page 5-195	True if this port is overwritable.

Properties

DirectFeedthrough

True if this input port has direct feedthrough.

Boolean

RW for MATLAB S functions, RO for other blocks

Overwritable

True if this input port is overwritable.

Boolean

RW for MATLAB S functions, RO for other blocks

Introduced before R2006a

Simulink.BlockPreCompOutputPortData

Provide precompilation information about block output port

Description

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the “OutputPort” on page 5-379 method of a block's run-time object before the model containing the block has been compiled.

Parent

`Simulink.BlockPortData`

Children

none

Property Summary

Name	Description
“Reusable” on page 5-196	Specifies whether an output port's memory is reusable.

Properties

Reusable

Specifies whether an output port's memory is reusable. Options are: `NotReusableAndGlobal` and `ReusableAndLocal`.

character vector

RW for MATLAB S functions, RO for other blocks

Introduced before R2006a

Simulink.Breakpoint class

Package: Simulink

Store and share data for a breakpoint set, configure the data for ASAP2 and AUTOSAR code generation

Description

An object of the `Simulink.Breakpoint` class stores breakpoint set data for a lookup table. You can use that data in one or more Prelookup blocks. With the object, you can specify a data type and code generation settings for the breakpoint set and share the set between multiple lookup tables. Use `Simulink.Breakpoint` objects and `Simulink.LookupTable` objects to configure `COM_AXIS` code generation for calibration.

The code generated for a `Simulink.Breakpoint` object is an array or a structure with two fields. If you configure the object to appear as a structure, one field stores the specified breakpoint set data and one scalar field stores the number of elements in the breakpoint set data. You can configure the structure type name, the field name, and other characteristics by using the properties of the object.

To package lookup table and breakpoint set data into a single structure in the generated code, for example, for `STD_AXIS` code generation, use a `Simulink.LookupTable` object to store all of the data. See “Package Shared Breakpoint and Table Data for Lookup Tables”.

Construction

`BpSet = Simulink.Breakpoint` returns a `Simulink.Breakpoint` object named `BpSet` with default property values.

To create a `Simulink.Breakpoint` object by using the Model Explorer, use the button



on the toolbar. The default name for the object is `Object`.

Property Dialog Box

Simulink.Breakpoint: myBreakpointSet

Breakpoints

Support tunable size

Value	Data type	Dimensions	Min	Max	Unit	Field name	Description
[]	auto	[0 0]	[]	[]	BP		

Code generation options

Data definition:

Storage class: Auto

Struct Type definition

Name:

Data scope: Auto

Header file:

OK Cancel Help Apply

Breakpoints

The breakpoint set information. You can configure these characteristics:

Support tunable size

Specification to enable tuning the effective size of the table in the generated code. If you select this option, in the generated code, the `Simulink.Breakpoint` object appears as a structure variable. The structure has one field to store the breakpoint vector data and one field to store the number of elements in the breakpoint vector. You can change the value of the second field to adjust the effective size of the table.

If you clear this option, the `Simulink.Breakpoint` object appears in the generated code as a separate array variable instead of a structure.

Value

Breakpoint set data. Specify a vector with at least two elements.

You can also use an expression with mathematical operators such as `sin(1:0.5:30)` as long as the expression returns a numeric vector. When you click **Apply** or **OK**, the object executes the expression and uses the result to set the value of this property.

When you set **Data type** to `auto`, to set **Value**, use a typed expression such as `single([1 2 3])` or use the `fi` constructor to embed an `fi` object.

You can edit this data by using a more intuitive interface in a lookup table block. See “Import Lookup Table Data from MATLAB”.

Data type

Data type of the breakpoint set. The default setting is `auto`, which means that the breakpoint set acquires a data type from the value that you specify in **Value**. If you use an untyped expression such as `[1 2 3]` to set **Value**, the breakpoint data use the data type `double`. If you specify a typed expression such as `single([1 2 3])` or an `fi` object, the breakpoint data use the data type specified by the expression or object.

You can explicitly specify an integer data type, a floating-point data type, a fixed-point data type, or a data type expression such as the name of a `Simulink.AliasType` object.

For more information about data types in Simulink, see “Data Types Supported by Simulink”. To decide how to control the data types of table and breakpoint data in `Simulink.LookupTable` and `Simulink.Breakpoint` objects, see “Control Data Types of Lookup Table Objects” (Simulink Coder).

Dimensions

Dimension lengths of the breakpoint set.

To use symbolic dimensions, specify a character vector. See “Implement Dimension Variants for Array Sizes in Generated Code” (Embedded Coder).

Min

Minimum value of the elements in the breakpoint set. The default value is empty, `[]`. You can specify a numeric, real value.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”.

Max

Maximum value of the elements in the breakpoint set. The default value is empty, `[]`. You can specify a numeric, real value.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”.

Unit

Physical unit of the elements in the breakpoint set. You can specify text such as degC. See “Unit Specification in Simulink Models”.

Field name

Name of a structure field in the generated code. This field stores the breakpoint set data. The default value is `BP`. To change the field name, specify text.

This column appears only if you select **Support tunable size**.

Tunable size name

Name of a structure field in the generated code. This scalar field stores the length of the breakpoint set (the number of elements), which the generated code algorithm uses to determine the size of the table. To tune the effective size of the table during code execution, change the value of this structure field in memory. The default name is `N`. To change the field name, specify text.

This column appears only if you select **Support tunable size**.

Description

Description of the breakpoint set. You can specify text such as `This breakpoint set represents the pressure input.`

Data definition: Storage class

Storage class of the structure variable (if you select **Support tunable size**) or array variable in the generated code. The variable stores the breakpoint set data. The default setting is `Auto`.

For more information about storage classes, see “Override Default Parameter Behavior by Creating Global Variables in the Generated Code” (Simulink Coder)

If you have Embedded Coder, you can choose a custom storage class. For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Data definition: Alias

Alternative name for the variable in the generated code. The default value is empty, in which case the generated code uses the name of the `Simulink.Breakpoint` object as the name of the variable. To set the alias, specify text.

To enable this property, set **Data definition: Storage class** to a setting other than `Auto`.

Data definition: Alignment

Data alignment boundary in the generated code. The starting memory address for the data allocated for the structure or array variable is a multiple of the value that you specify. The default value is -1 , which allows the code generator to determine an optimal alignment based on usage.

Specify a positive integer that is a power of 2, not exceeding 128. For more information about using data alignment for code replacement, see “Data Alignment for Code Replacement” (Embedded Coder).

Struct Type definition: Name

Name of the structure type that the structure variable uses in the generated code. The default value is empty. Specify text.

This property appears only if you select **Support tunable size**.

Struct Type definition: Data scope

Scope of the structure type definition (imported from your handwritten code or exported from the generated code). The default value is `Auto`. When you select `Auto`:

- If you do not specify a value in the **Struct Type definition: Header file** box, the generated code exports the structure type definition to the file `model_types.h`. `model` is the name of the model.
- If you specify a value in the **Struct Type definition: Header file** box, such as `myHdr.h`, the generated code imports the structure type definition from `myHdr.h`.

To explicitly specify the data scope:

- To import the structure type definition into the generated code from your custom code, select `Imported`.
- To export the structure type definition from the generated code, select `Exported`.

If you do not specify a value in the **Struct Type definition: Header file** box, the generated code imports or exports the type definition from or to `StructName.h`. `StructName` is the name that you specify with the property **Struct Type definition: Name**.

This property appears only if you select **Support tunable size**.

Struct Type definition: Header file

Name of the header file that contains the structure type definition. You can import the definition from a header file that you create, or export the definition into a

generated header file. To control the scope of the structure type, adjust the setting for the **Struct Type definition: Data scope** property.

This property appears only if you select **Support tunable size**.

Properties

Breakpoints — Breakpoint set data

`Simulink.lookuptable.Breakpoint` object

Breakpoint set data, specified as a `Simulink.lookuptable.Breakpoint` object. Use this embedded object to configure the structure field names and characteristics of the breakpoint set data such as breakpoint values, data type, and dimensions.

CoderInfo — Code generation settings for variable

`Simulink.CoderInfo` object

Code generation settings for the structure variable (if you set `SupportTunableSize` to `true`) or array variable (`false`) that stores the breakpoint set data, specified as a `Simulink.CoderInfo` object. You can specify a storage class or custom storage class by using this embedded object. For more information, see `Simulink.CoderInfo`.

StructTypeInfo — Settings for structure type in the generated code

`Simulink.lookuptable.StructTypeInfo` object

Settings for the structure type that the structure variable uses in the generated code, specified as a `Simulink.lookuptable.StructTypeInfo` object.

If you set `SupportTunableSize` to `false`, the `Simulink.Breakpoint` object does not appear in the generated code as a structure. The code generator ignores this property.

SupportTunableSize — Option to generate code that enables tunability of table size

`false` (default) | `true`

Option to generate code that enables tunability of the effective size of the table, specified as `true` or `false`. See “Support tunable size” on page 5-0 .

Data Types: `logical`

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Share Breakpoint Data Between One-Dimensional Lookup Tables

- 1 Create a `Simulink.Breakpoint` object named `myBpSet`.

```
myBpSet = Simulink.Breakpoint
```

- 2 Specify the breakpoint data.

```
myBpSet.Breakpoints.Value = [-2 -1 0 1 2];
```

- 3 Create a `Simulink.LookupTable` object named `FirstLUTObj`.

```
FirstLUTObj = Simulink.LookupTable;
```

- 4 Specify the table data.

```
FirstLUTObj.Table.Value = [1.1 2.2 3.3 4.4 5.5];
```

- 5 Configure the lookup table object to refer to the breakpoint set object.

```
FirstLUTObj.Breakpoints = {'myBpSet'};
```

- 6 Create another `Simulink.LookupTable` object to store a different set of table data. Configure the lookup table object to refer to the same breakpoint set object.

```
SecondLUTObj = Simulink.LookupTable;  
SecondLUTObj.Table.Value = [1.2 2.3 3.4 4.5 5.6];  
SecondLUTObj.Breakpoints = {'myBpSet'};
```

You can use `FirstLUTObj` and `SecondLUTObj` to specify the table data in two different Interpolation Using Prelookup blocks. Use `myBpSet` to specify the breakpoint set data in one or two Prelookup blocks that provide the inputs for the Interpolation Using Prelookup blocks.

Limitations

- You cannot subclass `Simulink.Breakpoint` or `Simulink.LookupTable`. For this reason, you cannot apply custom storage classes other than those in the built-in `Simulink` package.
- You cannot use `Simulink.Breakpoint` objects or `Simulink.LookupTable` objects that refer to `Simulink.Breakpoint` objects as instance-specific parameter data for reusable components. For example, you cannot use one of these objects as:
 - A model argument in a model workspace or a model argument value in a Model block.
 - The value of a mask parameter on a CodeReuse Subsystem block.
 - The value of a mask parameter on a subsystem that you reuse by creating a custom library.

However, you can use standalone `Simulink.LookupTable` objects, which do not refer to `Simulink.Breakpoint` objects, in these ways.

- You cannot generate code according to the `FIX_AXIS` style.
- When blocks in a subsystem use `Simulink.LookupTable` or `Simulink.Breakpoint` objects, you cannot set data type override (see “Control Fixed-Point Instrumentation and Data Type Override”) only on the subsystem. Instead, set data type override on the entire model.

See Also

`Simulink.LookupTable` | `Simulink.Parameter` |
`Simulink.lookuptable.Breakpoint` | `Simulink.lookuptable.StructTypeInfo`
 | `Simulink.lookuptable.Table`

Topics

“Configure `STD_AXIS` and `COM_AXIS` Lookup Tables for AUTOSAR Measurement and Calibration” (Embedded Coder)

“About Lookup Table Blocks”

“Package Shared Breakpoint and Table Data for Lookup Tables”

“Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder)

Introduced in R2016b

Simulink.Bus class

Package: Simulink

Specify properties of bus signal

Description

Objects of the `Simulink.Bus` class, used with objects of the `Simulink.BusElement` class, specify the properties of a bus signal. Bus objects validate the properties of bus signals. When you simulate a model or update diagram, Simulink checks whether the buses connected to the blocks have the properties specified by the bus objects. If not, Simulink halts and displays an error message. For a complete list of blocks that support using a bus object as a data type, see “When to Use Bus Objects”.

You can use the Simulink Bus editor or MATLAB commands to create and modify bus objects in the base MATLAB workspace. You cannot store a bus object in a model workspace.

When you use the Bus Editor, you create `Simulink.Bus` and `Simulink.BusElement` objects in the base workspace or the associated Simulink data dictionary.

Also, you can use a bus object to specify the attributes of a signal (for example, at the root level of a model or in a Data Store Memory block).

Construction

`busObj = Simulink.Bus` returns a bus object with these property values:

```
Description: ''
DataScope: 'Auto'
HeaderFile: ''
Alignment: -1
Elements: [0x0 Simulink.BusElement]
```

The name of the bus object is the name of the MATLAB variable to which you assign the bus object. You can set individual properties after you construct the bus object.

Output Arguments

busObject — Bus object

`Simulink.Bus` object

Bus object, returned as a `Simulink.Bus` object.

Properties

Description — Bus object description

character vector

Bus object description, specified as a character vector. Use the description to document information about the bus object, such as the kind of signal it applies to or where the bus object is used. This information does not affect Simulink processing.

Elements — Bus elements

array of `Simulink.BusElement` objects

Bus elements, specified as an array of `Simulink.BusElement` objects. Each bus element object defines the name, data type, dimensions, and other properties of the signal within a bus.

DataScope — Data type definition mode in generated code

'Auto' (default) | 'Exported' | 'Imported'

Data type definition mode in generated code, specified as 'Auto', 'Exported', or 'Imported'. This property specifies whether during code generation the data type definition is imported from, or exported to, the header file specified with the `HeaderFile` property.

Value	Action
'Auto' (default)	Import the data type definition from the specified header file. If you do not specify the header file, export the data type definition to the default header file.
'Exported'	Export the data type definition to the specified header file or to the default header file.

Value	Action
'Imported'	Import the data type definition from the specified header file specified or from the default header file.

HeaderFile — C header file used with data type definition

character vector

C header file used with data type definition, specified as a character vector. The header file is the file to import the data type definition from or export the data type definition to (based on the value of the `DataScope` property). The Simulink Coder software uses this property for code generation. Simulink software ignores this property.

By default, the generated `#include` directive uses the preprocessor delimiter `"` instead of `<` and `>`. To generate the directive `#include <myTypes.h>`, specify `HeaderFile` as `<myTypes.h>`.

Alignment — Data alignment boundary

-1 (default) | integer

Data alignment boundary, specified as an integer, in number of bytes. The Simulink Coder software uses this property for code generation. Simulink software ignores this property.

The starting memory address for the data allocated for the bus is a multiple of the `Alignment` setting. If the object occurs in a context that requires alignment, you must specify an `Alignment` value with a positive integer that is a power of 2, not exceeding 128.

Methods

Method	Purpose
<code>Simulink.Bus.cellToObject</code>	Convert cell array containing bus information to bus objects
<code>Simulink.Bus.createMATLABStruct</code>	Create MATLAB structures using same hierarchy and attributes as bus signals
<code>Simulink.Bus.createObject</code>	Create bus objects from blocks or MATLAB structures

Method	Purpose
<code>Simulink.Bus.objectToCell</code>	Use bus objects to create cell array containing bus information
<code>Simulink.Bus.save</code>	Save bus objects in MATLAB file

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Create Bus Objects and Bus Elements

Create the `CONTROL` bus object and its bus elements. The bus objects are stored in the base workspace.

```
clear elems;
elems(1) = Simulink.BusElement;
elems(1).Name = 'VALVE1';
elems(1).Dimensions = 1;
elems(1).DimensionsMode = 'Fixed';
elems(1).DataType = 'double';
elems(1).SampleTime = -1;
elems(1).Complexity = 'real';

elems(2) = Simulink.BusElement;
elems(2).Name = 'VALVE2';
elems(2).Dimensions = 1;
elems(2).DimensionsMode = 'Fixed';
elems(2).DataType = 'double';
elems(2).SampleTime = -1;
elems(2).Complexity = 'real';

CONTROL = Simulink.Bus;
CONTROL.Elements = elems;
```

This script is similar to the file that you get by saving a bus object to a MATLAB file and choosing the `Object` format.

- “Create Bus Objects Programmatically”
- “Group Signals into Structures in the Generated Code Using Buses” (Simulink Coder)
- “Data Alignment for Code Replacement” (Embedded Coder)
- “Create Bus Objects with the Bus Editor”
- “Save and Import Bus Objects”
- “When to Use Bus Objects”

Alternatives

You can use the Bus Editor to create interactively a bus object and its bus elements. For details, see “Create Bus Objects with the Bus Editor”.

Programmatically, you can create bus objects from:

- Blocks in a model
- MATLAB data
- External C code. See `Simulink.importExternalCTypes`.

See Also

`Simulink.Bus.cellToObject` | `Simulink.Bus.createObject` |
`Simulink.BusElement`

Topics

- “Create Bus Objects Programmatically”
- “Group Signals into Structures in the Generated Code Using Buses” (Simulink Coder)
- “Data Alignment for Code Replacement” (Embedded Coder)
- “Create Bus Objects with the Bus Editor”
- “Save and Import Bus Objects”
- “When to Use Bus Objects”

Introduced before R2006a

Simulink.BusElement class

Package: Simulink

Describe element of bus signal

Description

An object of this class specifies the properties of a signal in a bus. Bus element objects validate the properties of signals in a bus. A `Simulink.Bus` object contains bus elements. A bus element exists only within a bus object. You can specify a bus object, but not a bus element, as a block parameter value. When you simulate a model or update diagram, Simulink checks whether the signals in a bus connected to blocks have the properties specified by the bus elements. If not, Simulink halts and displays an error message.

You can use the Simulink Bus editor (see “Create Bus Objects with the Bus Editor”), or MATLAB commands (see “Create Bus Objects Programmatically”) to create and modify bus objects and bus elements in the base MATLAB workspace.

Construction

`busElementName = Simulink.BusElement` returns a bus element with these property values:

```
Name: 'a'  
Complexity: 'real'  
Dimensions: 1  
    DataType: 'double'  
        Min: []  
        Max: []  
DimensionsMode: 'Fixed'  
    SampleTime: -1  
        Unit: ''  
    Description: ''
```

Output Arguments

busElement — Bus element

`Simulink.BusElement` object

Bus element, returned as a `Simulink.BusElement` object.

Properties

Name — Name of bus element

character vector

Name of bus element, specified as a character vector.

Complexity — Numeric type of bus element

'real' (default) | 'complex'

Numeric type of the bus element, specified as 'real' or 'complex'.

Dimensions — Dimensions of bus element

array

Dimensions of bus element, specified as an array.

DataType — Data type of bus element

built-in Simulink data type | a `Simulink.NumericType` object | a `Simulink.Bus` object

Data type of bus element, specified as a built-in Simulink data type or `Simulink.NumericType` object. Examples of built-in data types include `double` and `uint8`. You can specify a `Simulink.NumericType` object whose `DataTypeMode` property is set to a value other than 'Fixed-point: unspecified scaling'. Specifying a bus object allows you to create bus objects that specify hierarchical buses (that is, buses that contain other buses).

Min — Minimum value of bus element

double | []

Minimum value of the bus element, specified as a double. This value must be a finite real double scalar or, if the element is a bus, the value must be empty, [].

Max — Maximum value of bus element

double | []

Maximum value of the bus element, specified as a double. This value must be a finite real double scalar or, if the element is a bus, the value must be empty, [].

DimensionsMode — How to handle size of bus element

'Fixed' (default) | 'Variable'

How to handle size of bus element, specified as 'Fixed' or 'Variable'.

SampleTime — Sample time of bus element

-1 (default) | double

Sample time of bus element, specified as a double. The sample time is the size of the interval between times when this signal value must be recomputed. If these conditions apply, use the default value of -1:

- The bus element is a bus.
- The bus that includes this element passes through a block that changes the bus sample time, such as a Rate Transition block.

Unit — Physical unit for expressing bus element

character vector

Physical unit for expressing bus element, specified as a character vector (for example, 'inches').

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create Bus Objects and Bus Elements

Create the CONTROL bus object and its bus elements. The bus objects are stored in the base workspace.

```
clear elems;
elems(1) = Simulink.BusElement;
elems(1).Name = 'VALVE1';
elems(1).Dimensions = 1;
elems(1).DimensionsMode = 'Fixed';
elems(1).DataType = 'double';
elems(1).SampleTime = -1;
elems(1).Complexity = 'real';

elems(2) = Simulink.BusElement;
elems(2).Name = 'VALVE2';
elems(2).Dimensions = 1;
elems(2).DimensionsMode = 'Fixed';
elems(2).DataType = 'double';
elems(2).SampleTime = -1;
elems(2).Complexity = 'real';

CONTROL = Simulink.Bus;
CONTROL.Elements = elems;
```

This script is similar to the file that you get by saving a bus object to a MATLAB file and choosing the Object format. For information about saving bus objects, see “Save and Import Bus Objects”.

- “Create Bus Objects Programmatically”
- “Save and Import Bus Objects”
- “Signal Names and Labels”
- “Specify Sample Time”
- “Variable-Size Signal Basics”

Alternatives

You can use the Bus Editor to create interactively a bus object and its bus elements. For details, see “Create Bus Objects with the Bus Editor”.

Programmatically, you can create bus objects and elements from:

- Blocks in a model
- MATLAB data

See Also

`Simulink.Bus` | `Simulink.Bus.cellToObject` | `Simulink.Bus.createObject`

Topics

“Create Bus Objects Programmatically”

“Save and Import Bus Objects”

“Signal Names and Labels”

“Specify Sample Time”

“Variable-Size Signal Basics”

Introduced before R2006a

Simulink.CoderInfo

Specify information needed to generate code for signal or parameter data

Description

Simulink creates a `Simulink.CoderInfo` object for each data object that you create. Data objects represent signal, state, or parameter data. The `Simulink.CoderInfo` object exists in the `CoderInfo` property of each data object.

Data objects include objects of these classes:

- `Simulink.Parameter`
- `Simulink.Signal`
- `Simulink.LookupTable`
- `Simulink.Breakpoint`

Use the properties of the `Simulink.CoderInfo` object to configure the representation of the parent data object in the generated code.

You can set the properties of a `Simulink.CoderInfo` object through the `CoderInfo` property or the property dialog box of the parent data object. For example, the following MATLAB expression sets the `StorageClass` property of a `Simulink.CoderInfo` object used by a signal object named `mysignal`.

```
mysignal.CoderInfo.StorageClass = 'ExportedGlobal';
```

Property Dialog Box

Use a data object property dialog box to set the properties of a `Simulink.CoderInfo` object.

Properties

Name	Description
Alias	Alternative name for this data in the generated code.
Alignment	Data alignment boundary for this data. See “Data Alignment for Code Replacement” (Embedded Coder) for more information.
CustomAttributes	Custom storage class attributes of this data. You must set the property <code>StorageClass</code> to 'Custom' to enable this property. See “Introduction to Custom Storage Classes” (Embedded Coder) for more information.
CustomStorageClass	Custom storage class of this data. You must set the property <code>StorageClass</code> to 'Custom' to enable this property.
StorageClass	Storage class of this data. For more information, see “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder) or “Block Parameter Representation in the Generated Code” (Simulink Coder).

See Also

Topics

“Control Signals and States in Code by Applying Storage Classes” (Simulink Coder)

“Block Parameter Representation in the Generated Code” (Simulink Coder)

“Data Objects”

“Introduction to Custom Storage Classes” (Embedded Coder)

Introduced in R2015a

ConfigSet

Model configuration set

Description

Use the `ConfigSet` object to access a model configuration set. Get information about your configuration set, manage configuration parameters, and manage components.

Creation

Use the `getActiveConfigSet` function to get the active configuration set for a model. Use the `getConfigSet` function to get a model configuration set by name.

Properties

Components — Components of the configuration set

cell array of `Simulink.ConfigComponent` objects

Components of the configuration set, specified by a cell array of `Simulink.ConfigComponent` objects.

Description — Description of the configuration set

character vector

Description of the configuration set, specified as a character vector. Use the description to provide additional information about a configuration set, such as its purpose.

Name — Name of the configuration set

character vector

Name of the configuration set, specified by a character vector. This name represents the configuration set in the Model Explorer.

Object Functions

<code>attachComponent</code>	Attach a component to a configuration set
<code>copy</code>	Copy a configuration set
<code>getComponent</code>	Get a configuration set component
<code>getFullName</code>	
<code>getModel</code>	
<code>get_param</code>	Get parameter names and values
<code>isActive</code>	
<code>isValidParam</code>	
<code>saveAs</code>	
<code>setPropEnabled</code>	
<code>set_param</code>	Set system and block parameter values

Examples

Sample title

See Also

`getActiveConfigSet` | `getConfigSet` | `getConfigSets`

Topics

“About Configuration Sets”

“Manage a Configuration Set”

Introduced in R2006a

Simulink.ConfigSetRef

Link model to configuration set stored independently of any model

Description

Instances of this handle class allow a model to reference configuration sets that exist outside any model. See “Manage a Configuration Set”, “Overview”, and “Manage a Configuration Reference” for more information.

Property Summary

Name	Description
“Description” on page 5-223	Description of the configuration reference.
“Name” on page 5-223	Name of the configuration reference.
“SourceName” on page 5-224	Name of the variable in the workspace or the data dictionary that contains the referenced configuration set.

Note You can use the **Configuration Reference** dialog box to set the Name, Description, and SourceName properties of a configuration reference. See “Create and Attach a Configuration Reference” for details.

Method Summary

Name	Description
“copy” on page 5-224	Create a copy of a configuration reference.
“getFullName” on page 5-224	Get the full pathname of a configuration reference.
“getModel” on page 5-225	Get the handle of the model that owns a configuration reference.

Name	Description
“get_param” on page 5-225	Get the value of a configuration set parameter indirectly through a configuration reference.
“getRefConfigSet” on page 5-226	Get the configuration set specified by a configuration reference.
“isActive” on page 5-226	Determine whether a configuration reference is the active configuration object of the model.
“refresh” on page 5-226	Update configuration reference after any change to properties or configuration set availability.

Properties

Description

Description of the configuration reference. You can use this property to provide additional information about a configuration reference, such as its purpose. This field can remain blank.

character vector

RW

Name

Name of the configuration reference. This name represents the configuration reference in the GUI.

character vector

RW

SourceName

Name of the variable in the workspace or the data dictionary that contains the referenced configuration set.

character vector

RW

Methods

copy

Create a copy of this configuration reference.

copy

This method creates a copy of this configuration set.

Note You must use this method to create copies of configuration references. This is because `Simulink.ConfigSetRef` is a handle class. See “Handle Versus Value Classes” for more information.

getFullName

Get the full pathname of a configuration reference.

getFullName

This method returns a character vector specifying the full pathname of a configuration reference, e.g., `'vdp/Configuration'`.

getModel

Get the model that owns this configuration reference.

```
getModel
```

Returns a handle to the model that owns this configuration reference.

The following command opens the block diagram of the model that owns the configuration set referenced by the MATLAB workspace variable `hCr`.

```
open_system(hCr.getModel);
```

get_param

Get the value of a configuration set parameter indirectly through a configuration reference.

```
get_param(paramName)
```

```
paramName
```

Character vector specifying the name of the parameter whose value is to be returned.

This method returns the value of the specified parameter from the configuration set to which the configuration reference points. To obtain this value, the method uses the value of `SourceName` to retrieve the configuration set, then retrieves the value of `paramName` from that configuration set. Specifying `paramName` as `'ObjectParameters'` returns the names of all valid parameters in the configuration set. If a valid configuration set is not attached to the configuration reference, the method returns unreliable values.

The inverse method, `set_param`, is not defined for configuration references. To obtain a parameter value through a configuration reference, you must first use the `getRefConfigSet` method to retrieve the configuration set from the reference, then use `set_param` directly on the configuration set itself.

You can also use the `get_param` model construction command to get the values of parameters of a model's active configuration set, e.g., `get_param(bdroot, 'SolverName')` gets the solver name of the currently selected model.

The following command gets the name of the solver used by the selected model's active configuration.

```
hAcs = getActiveConfigSet(bdroot);  
hAcs.get_param('SolverName');
```

getRefConfigSet

Get the configuration set specified by a configuration reference

```
getRefConfigSet
```

Returns a handle to the configuration set specified by the `SourceName` property of a configuration reference.

isActive

Determine whether this configuration set is its model's active configuration set.

```
isActive
```

Returns `true` if this configuration set is the active configuration set of the model that owns this configuration set.

refresh

Update configuration reference after any change to properties or configuration set availability

`refresh`

Updates a configuration reference after using the API to change any property of the reference, or after providing a configuration set that did not exist at the time the set was originally specified in `SourceName`. If you omit executing `refresh` after any such change, the configuration reference handle will be stale, and using it will give incorrect results.

Introduced in R2007a

Simulink.GlobalDataTransfer class

Package: Simulink

Configure concurrent execution data transfers

Description

The `Simulink.GlobalDataTransfer` object contains the data transfer information for the concurrent execution of a model. To access the properties of this class, use the `get_param` function to get the handle for this class, and then use dot notation to access the properties. For example:

```
dt=get_param(gcs,'DataTransfer');  
dt.DefaultTransitionBetweenContTasks  
  
ans =  
  
Ensure deterministic transfer (minimum delay)
```

Properties

DefaultTransitionBetweenSyncTasks

Global setting for data transfer handling option when the source and destination of a signal are in two different and periodic tasks.

Data Type: Enumeration. Can be one of:

- 'Ensure data integrity only'
- 'Ensure deterministic transfer (maximum delay)'
- 'Ensure deterministic transfer (minimum delay)'

Access: Read/write

DefaultTransitionBetweenContTasks

Global setting for the data transfer handling option for signals that have a continuous sample time.

Data Type: Enumeration. Can be one of:

- 'Ensure data integrity only'
- 'Ensure deterministic transfer (maximum delay)'
- 'Ensure deterministic transfer (minimum delay)'

Access: Read/write

DefaultExtrapolationMethodBetweenContTasks

Global setting for the data transfer extrapolation method for signals that have a continuous sample time.

Data Type: Enumeration. Can be one of:

- 'None'
- 'Zero Order Hold'
- 'Linear'
- 'Quadratic'

Access: Read/write

AutoInsertRateTranBlk

Setting for whether or not Simulink software automatically inserts hidden Rate Transition blocks between blocks that have different sample rates to ensure the integrity of data transfers between tasks; and optional determinism of data transfers for periodic tasks.

Data Type: Boolean. Can be one of:

- 0
- 1

Access: Read/write

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Access the properties of this class.

```
dt=get_param(gcs,'DataTransfer');  
dt.DefaultTransitionBetweenContTasks  
  
ans =  
  
Ensure deterministic transfer (minimum delay)
```

See Also

[Simulink.architecture.add](#) | [Simulink.architecture.delete](#) |
[Simulink.architecture.find_system](#) | [Simulink.architecture.get_param](#) |
[Simulink.architecture.importAndSelect](#) | [Simulink.architecture.profile](#)
| [Simulink.architecture.register](#) | [Simulink.architecture.set_param](#)

Topics

“Configure Data Transfer Settings Between Concurrent Tasks”

Simulink.HMI.InstrumentedSignals class

Package: Simulink.HMI

Access logged signals in model

Description

`Simulink.HMI.InstrumentedSignals` objects contain a list of all logged signals in a model, including signals from all subsystems, library instances, and Stateflow charts. The list does not include signals inside reference models. You can access the list of logged signals in a reference model by creating a `Simulink.HMI.InstrumentedSignals` object for the reference model.

The `Simulink.HMI.InstrumentedSignals` object provides access to `Simulink.HMI.SignalSpecification` objects using the `get` method.

Construction

`instSigs = get_param(model, 'InstrumentedSignals')` returns `instSigs`, a `Simulink.HMI.InstrumentedSignals` object containing a list of all of the logged signals in the model, `model`.

Input Arguments

model — Model name

character vector

Model name or full path to model.

Example: `'sldemo_fuelsys'`

Example: `fullpath(matlabroot, 'examples', 'simulink', 'ex_sldemo_absbrake.slx')`

'InstrumentedSignals' — Parameter selection

'InstrumentedSignals'

Desired return from `get_param`, specified as a character vector. Using a value of `'InstrumentedSignals'`, `get_param` returns a `Simulink.HMI.InstrumentedSignals` object with a list of all the logged signals.

Example: `'InstrumentedSignals'`

Properties

Model — Model name

character vector

Name of the model the aggregation of logged signals corresponds to.

Example: `'sldemo_fuelsys'`

Count — Number of logged signals

integer

Number of logged signals in the model.

Example: 10

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Evaluate and Modify Logged Signals in Model

This example shows how to obtain the block paths for logged signals and remove the logging badge for a signal using `Simulink.HMI.InstrumentedSignals` and `Simulink.HMI.SignalSpecification` objects.

Get InstrumentedSignals Object

```
% Load the model sldemo_absbrake
load_system('sldemo_absbrake')
```

```

% Get logged signals with Simulink.HMI.InstrumentedSignals object
instSigs = get_param('sldemo_absbrake', 'InstrumentedSignals');

% Check logged signals count
instSigs.Count

ans =

    uint32

     2

```

Inspect Block Paths with SignalSpecifications Objects

Use the `get` method to get the `Simulink.HMI.SignalSpecification` objects for each of the signals in the `Simulink.HMI.InstrumentedSignals` object

```

% Get Simulink.HMI.SignalSpecification objects
sig1 = instSigs.get(1);
sig2 = instSigs.get(2);

% Inspect block paths for signals
blockPath1 = sig1.BlockPath
blockPath2 = sig2.BlockPath

```

```
blockPath1 =
```

```

    Simulink.BlockPath
    Package: Simulink

    Block Path:
        'sldemo_absbrake/Bus Creator'

```

Use the `getBlock` method to access block path character vectors from this object.

```
blockPath2 =
```

```

    Simulink.BlockPath
    Package: Simulink

    Block Path:
        'sldemo_absbrake/Relative Slip'

```

Use the `getBlock` method to access block path character vectors from this object.

Remove Logging Badge for Bus Signal

Remove the logging badge for the signal from the Bus Creator block.

```
% Get block path string and port index for the Bus Creator signal
blockPath_str = blockPath1.getBlock(1);
portIndex = sig1.OutputPortIndex;

% Clear the logging badge for the Bus Creator signal
Simulink.sdi.markSignalForStreaming(blockPath_str, portIndex, 'off')
```

Save and Restore a Set of Logged Signals

This example shows the capability of using the `Simulink.HMI.InstrumentedSignals` object to save a set of logged signals to restore after running a simulation with a different set of signals.

Load Model and Save Initial Configuration

Load the `sldemo_fuelsys` model, and save the initial set of logged signals.

```
% Load model
load_system sldemo_fuelsys

% Get Simulink.HMI.InstrumentedSignals object
initSigs = get_param('sldemo_fuelsys', 'InstrumentedSignals');

% Save logging configuration to file for future use
save initial_instSigs.mat initSigs
```

Remove All Logging Badges

Return to a baseline of no logged signals so you can easily select a different configuration of signals to log.

```
% Clear all logging signals
set_param('sldemo_fuelsys', 'InstrumentedSignals', [])
```

Restore Saved Logging Configuration

After working with a different set of logged signals, you can easily restore a saved configuration with the `Simulink.HMI.InstrumentedSignals` object.

```
% Load the saved configuration
load initial_instSigs.mat

% Restore logging configuration
set_param('sldemo_fuelsys', 'InstrumentedSignals', initSigs)
```

See Also

[Simulink.HMI.SignalSpecification](#) | [Simulink.sdi.markSignalForStreaming](#)

Topics

“Populate the Simulation Data Inspector with Your Data”

“Tune and Visualize Your Model with Dashboard Blocks”

Introduced in R2015b

Simulink.HMI.SignalSpecification class

Package: Simulink.HMI

Information for logging a signal

Description

The `Simulink.HMI.SignalSpecification` object contains the block path and port index required by `Simulink.sdi.markSignalForStreaming` to turn logging on or off for a signal.

Construction

`sigSpec = instSigs.get(index)` returns the `Simulink.HMI.SignalSpecification` object `sigSpec` for the signal at the specified index in the `Simulink.HMI.InstrumentedSignals` object `instSigs`.

Input Arguments

index — Index of signal

integer

Numeric index of the signal within the `Simulink.HMI.InstrumentedSignals` object.

Example: 1

Properties

BlockPath — Block path for signal

`Simulink.BlockPath` object

`Simulink.BlockPath` object with the block path for the signal.

OutputPortIndex — Signal index on block port

integer

Index of the signal on the output port of its block. For Stateflow signals, the OutputPortIndex is set to 1.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Evaluate and Modify Logged Signals in Model

This example shows how to obtain the block paths for logged signals and remove the logging badge for a signal using `Simulink.HMI.InstrumentedSignals` and `Simulink.HMI.SignalSpecification` objects.

Get InstrumentedSignals Object

```
% Load the model sldemo_absbrake
load_system('sldemo_absbrake')

% Get logged signals with Simulink.HMI.InstrumentedSignals object
instSigs = get_param('sldemo_absbrake', 'InstrumentedSignals');

% Check logged signals count
instSigs.Count
```

```
ans =
```

```
uint32
```

```
2
```

Inspect Block Paths with SignalSpecifications Objects

Use the `get` method to get the `Simulink.HMI.SignalSpecification` objects for each of the signals in the `Simulink.HMI.InstrumentedSignals` object

```
% Get Simulink.HMI.SignalSpecification objects
sig1 = instSigs.get(1);
sig2 = instSigs.get(2);

% Inspect block paths for signals
blockPath1 = sig1.BlockPath
blockPath2 = sig2.BlockPath

blockPath1 =

    Simulink.BlockPath
    Package: Simulink

    Block Path:
        'sldemo_absbrake/Bus Creator'

    Use the getBlock method to access block path character vectors from this object.

blockPath2 =

    Simulink.BlockPath
    Package: Simulink

    Block Path:
        'sldemo_absbrake/Relative Slip'

    Use the getBlock method to access block path character vectors from this object.
```

Remove Logging Badge for Bus Signal

Remove the logging badge for the signal from the Bus Creator block.

```
% Get block path string and port index for the Bus Creator signal
blockPath_str = blockPath1.getBlock(1);
portIndex = sig1.OutputPortIndex;

% Clear the logging badge for the Bus Creator signal
Simulink.sdi.markSignalForStreaming(blockPath_str, portIndex, 'off')
```

See Also

[Simulink.HMI.InstrumentedSignals](#) | [Simulink.sdi.markSignalForStreaming](#)

Topics

“Populate the Simulation Data Inspector with Your Data”

“Tune and Visualize Your Model with Dashboard Blocks”

Introduced in R2015b

Simulink.LookupTable class

Package: Simulink

Store and share lookup table and breakpoint data, configure the data for ASAP2 and AUTOSAR code generation

Description

An object of the `Simulink.LookupTable` class stores lookup table and breakpoint data. You can use that data in a lookup table block such as the n-D Lookup Table block. With the object, you can specify data types and code generation settings for the table and the breakpoint sets.

When you store all of the table and breakpoint set data in a single `Simulink.LookupTable` object, all of the data appears in a single structure in the generated code. To configure STD_AXIS code generation for calibration, use this technique.

To share a breakpoint set between multiple lookup tables, for example for COM_AXIS code generation, use a `Simulink.Breakpoint` object in one or more Prelookup blocks. Use `Simulink.LookupTable` objects in Interpolation Using Prelookup blocks. Then, configure the lookup table objects to refer to the breakpoint object. For more information, see “Package Shared Breakpoint and Table Data for Lookup Tables”.

Construction

`LUTObj = Simulink.LookupTable` returns a `Simulink.LookupTable` object `LUTObj` with default property values.

To create a `Simulink.LookupTable` object by using the Model Explorer, use the button



on the toolbar. The default name for the object is `Object`.

Property Dialog Box

Simulink.LookupTable: LUTObj

Number of table dimensions: 1

Table

Value	Data type	Dimensions	Min	Max	Unit	Field name	Description
[]	auto	[0 0]	[]	[]		Table	

Breakpoints

Specification: Explicit values Support tunable size

Value	Data type	Dimensions	Min	Max	Unit	Field name	Description
1 []	auto	[0 0]	[]	[]		BP1	

Code generation options

Data definition:

Storage class: Auto

Struct Type definition

Name:

Data scope: Auto

Header file:

OK Cancel Help Apply

Number of table dimensions

Number of dimensions of the lookup table. Specify an integer value up to 30 (inclusive). For example, to represent a three-dimensional lookup table, specify the integer 3.

Table

Information for the table data. You can configure these characteristics:

Value

Table data. Specify a numeric vector or multidimensional array with at least two elements.

You can also use an expression with mathematical operators such as `sin(1:0.5:30)` as long as the expression returns a numeric vector or multidimensional array. When you click **Apply** or **OK**, the object executes the expression and uses the result to set the value of this property.

When you set **Data type** to `auto`, to set **Value**, use a typed expression such as `single([1 2 3])` or use the `fi` constructor to embed an `fi` object.

When you specify table data with three or more dimensions, **Value** displays the data as an expression that contains a call to the `reshape` function. To edit the values in the data, modify the first argument of the `reshape` call, which contains all of the values in a serialized vector. When you add or remove elements along a dimension, you must also correct the argument that represents the length of the modified dimension.

You can edit this data by using a more intuitive interface in a lookup table block. See “Import Lookup Table Data from MATLAB”.

Data type

Data type of the table data. The default setting is `auto`, which means that the table data acquire a data type from the value that you specify in **Value**. If you use an untyped expression such as `[1 2 3]` to set **Value**, the table data use the data type `double`. If you specify a typed expression such as `single([1 2 3])` or an `fi` object, the table data use the data type specified by the expression or object.

You can explicitly specify an integer data type, a floating-point data type, a fixed-point data type, or a data type expression such as the name of a `Simulink.AliasType` object.

For more information about data types in Simulink, see “Data Types Supported by Simulink”. To decide how to control the data types of table and breakpoint data in `Simulink.LookupTable` and `Simulink.Breakpoint` objects, see “Control Data Types of Lookup Table Objects” (Simulink Coder).

Dimensions

Dimension lengths of the lookup table data.

To use symbolic dimensions, specify a character vector. See “Implement Dimension Variants for Array Sizes in Generated Code” (Embedded Coder).

Min

Minimum value of the elements in the table data. The default value is empty, []. You can specify a numeric, real value.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”.

Max

Maximum value of the elements in the table data. The default value is empty, []. You can specify a numeric, real value.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”.

Unit

Physical units of the elements in the lookup table. You can specify text such as degC. See “Unit Specification in Simulink Models”.

Field name

Name of a structure field in the generated code. This field stores the table data if you configure the `Simulink.LookupTable` object to appear in the generated code as a structure. The default value is `Table`. To change the field name, specify text.

Description

Description of the lookup table. You can specify text such as `This lookup table describes the action of a pump.`

Breakpoints

Breakpoint set information. Each row is one breakpoint set. To configure additional breakpoint sets, specify an integer value in the **Number of table dimensions** box.

For the breakpoint sets, you can configure these characteristics:

Specification

Source for the information of the breakpoint sets, specified as `Explicit` values (default), `Reference`, or `Even spacing`.

- To store all of the table and breakpoint set data in the `Simulink.LookupTable` object, set **Specification** to `Explicit` values.

The `Simulink.LookupTable` object appears in the generated code as a single structure variable.

- To store the table data in the `Simulink.LookupTable` object and store the breakpoint set data in `Simulink.Breakpoint` objects, set **Specification** to `Reference`.

The `Simulink.LookupTable` object appears in the generated code as a separate array variable that contains the table data. Each `Simulink.Breakpoint` object appears as a separate array or structure variable that contains the breakpoint set data.

- To store the table data and evenly spaced breakpoints in the `Simulink.LookupTable` object, set **Specification** to `Even spacing`. Use the **First point** and **Spacing** parameters to generate a set of evenly spaced breakpoints.

Note When **Specification** is set to `Explicit values` or `Even spacing`, you can change the order of the tunable size, breakpoint, and table entries in a lookup table object-generated structure.

Support tunable size

Specification to enable tuning the effective size of the table in the generated code. In the code, the structure that corresponds to the object has an extra field for each breakpoint vector. Each extra field stores the length of the corresponding breakpoint vector. You can change the value of each field to adjust the effective size of the table.

This property appears only if you set **Specification** to `Explicit values` or `Even spacing`.

Note If you store breakpoint data in `Simulink.Breakpoint` objects by setting **Specification** to `Reference`, to enable tuning of the table size in the generated code, use the **Support tunable size** property of each `Simulink.Breakpoint` object.

Value

Data for the breakpoint set. Specify a numeric vector with at least two elements.

You can also use an expression with mathematical operators such as `sin(1:0.5:30)` as long as the expression returns a numeric vector. When you click **Apply** or **OK**, the object executes the expression and uses the result to set the value of this property.

When you set **Data type** to `auto`, to set **Value**, use a typed expression such as `single([1 2 3])` or use the `fi` constructor to embed an `fi` object.

You can edit this data by using a more intuitive interface in a lookup table block. See “Import Lookup Table Data from MATLAB”.

Data type

Data type of the breakpoint set. The default setting is `auto`, which means that the breakpoint set acquires a data type from the value that you specify in **Value**. If you use an untyped expression such as `[1 2 3]` to set **Value**, the breakpoint data use the data type `double`. If you specify a typed expression such as `single([1 2 3])` or an `fi` object, the breakpoint data use the data type specified by the expression or object.

You can explicitly specify an integer data type, a floating-point data type, a fixed-point data type, or a data type expression such as the name of a `Simulink.AliasType` object.

For more information about data types in Simulink, see “Data Types Supported by Simulink”. To decide how to control the data types of table and breakpoint data in `Simulink.LookupTable` and `Simulink.Breakpoint` objects, see “Control Data Types of Lookup Table Objects” (Simulink Coder).

Dimensions

Dimension lengths of the breakpoint set.

To use symbolic dimensions, specify a character vector. See “Implement Dimension Variants for Array Sizes in Generated Code” (Embedded Coder).

Min

Minimum value of the elements in the breakpoint set. The default value is empty, `[]`. You can specify a numeric, real value.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”.

Max

Maximum value of the elements in the breakpoint set. The default value is empty, []. You can specify a numeric, real value.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”.

Unit

Physical unit of the elements in the breakpoint set. You can specify text such as degF. See “Unit Specification in Simulink Models”.

Field name

Name of a structure field in the generated code. This field stores the breakpoint set data. The default value is BP1 for the first breakpoint set and BP2 for the second set. To change the field name, specify text.

Tunable size name

Name of a structure field in the generated code. This field stores the length (number of elements) of the breakpoint set, which the generated code algorithm uses to determine the size of the table. To tune the effective size of the table during code execution, change the value of this structure field in memory. The default name is N1 for the first breakpoint set and N2 for the second set. To change the field name, specify text.

This column appears only if you select **Support tunable size**.

Description

Description of the breakpoint set. You can specify text such as `This breakpoint set represents the pressure input.`

First point

First point in evenly spaced breakpoint data. This parameter is available when **Specification** is set to `Even spacing`.

Spacing

Spacing between points in evenly spaced breakpoint data. This parameter is available when **Specification** is set to `Even spacing`.

Name

Name of the `Simulink.Breakpoint` object that stores the information for this breakpoint set.

This column appears only if you set **Specification** to `Reference`.

First point name

Name of the `Simulink.Breakpoint` object that stores the information for the first point. This parameter is available when **Specification** is set to `Even spacing`.

Spacing name

Name of the `Simulink.Breakpoint` object that stores the information for the spacing. This parameter is available when **Specification** is set to `Even spacing`.

Data definition: Storage class

Storage class of the structure variable (if you set **Specification** to `Explicit values` or `Even spacing`) or array variable (`Reference`) in the generated code. The variable stores the table data and, if the variable is a structure, the breakpoint set data. The default setting is `Auto`.

For more information about storage classes, see “Override Default Parameter Behavior by Creating Global Variables in the Generated Code” (Simulink Coder)

If you have Embedded Coder, you can choose a custom storage class. For information about custom storage classes, see “Control Data Representation by Applying Custom Storage Classes” (Embedded Coder).

Data definition: Alias

Alternative name for the structure variable (if you set **Specification** to `Explicit values` or `Even spacing`) or array variable (`Reference`) in the generated code. The default value is empty, in which case the generated code uses the name of the `Simulink.LookupTable` object as the name of the structure or array variable. To set the alias, specify text.

To enable this property, set **Data definition: Storage class** to a setting other than `Auto`.

Data definition: Alignment

Data alignment boundary in the generated code. The starting memory address for the data allocated for the structure or array variable is a multiple of the value that you specify. The default value is `-1`, which allows the code generator to determine an optimal alignment based on usage.

Specify a positive integer that is a power of 2, not exceeding 128. For more information about using data alignment for code replacement, see “Data Alignment for Code Replacement” (Embedded Coder).

Struct Type definition: Name

Name of the structure type that the structure variable uses in the generated code. The default value is empty. Specify text.

This property appears only if you set **Specification** to `Explicit` values or `Even spacing`.

Struct Type definition: Data scope

Scope of the structure type definition (imported from your custom code or exported from the generated code). The default value is `Auto`. When you select `Auto`:

- If you do not specify a value in the **Struct Type definition: Header file** box, the generated code exports the structure type definition to the file `model_types.h`. `model` is the name of the model.
- If you specify a value in the **Struct Type definition: Header file** box, such as `myHdr.h`, the generated code imports the structure type definition from `myHdr.h`.

To explicitly specify the data scope:

- To import the structure type definition into the generated code from your custom code, select `Imported`.
- To export the structure type definition from the generated code, select `Exported`.

If you do not specify a value in the **Struct Type definition: Header file** box, the generated code imports or exports the type definition from or to `StructName.h`. `StructName` is the name that you specify by using the property **Struct Type definition: Name**.

This property appears only if you set **Specification** to `Explicit` values or `Even spacing`.

Struct Type definition: Header file

Name of the header file that contains the structure type definition. You can import the definition from a header file that you create, or export the definition into a generated header file. To control the scope of the structure type, adjust the setting for the **Struct Type definition: Data scope** property.

This property appears only if you set **Specification** to `Explicit` values or `Even spacing`.

Properties

Breakpoints — Breakpoint set information

vector of `Simulink.lookupable.Breakpoint` objects | cell array of character vectors

Breakpoint set information, specified as a vector of `Simulink.lookupable.Breakpoint` objects, a cell array of character vectors, or a vector of `Simulink.lookupable.Evenspacing` objects.

If you use a vector of `Simulink.lookupable.Breakpoint` objects, each object represents a breakpoint set. Using a vector of `Simulink.lookupable.Breakpoint` objects sets the property `BreakpointsSpecification` to 'Explicit values'.

If you use a cell array of character vectors, each character vector represents the name of a `Simulink.Breakpoint` object. Using a cell array of character vectors sets the property `BreakpointsSpecification` to 'Reference'.

If you use a vector of `Simulink.lookupable.Evenspacing` objects, each object represents a breakpoint set. Using a vector of `Simulink.lookupable.Evenspacing` objects sets the property `BreakpointsSpecification` to 'Even Spacing'.

BreakpointsSpecification — Source of breakpoint set information

'Explicit values' (default) | 'Reference' | 'Even spacing'

Source of the breakpoint set information, specified as 'Explicit values' (default), 'Even spacing', or 'Reference'. See “Specification” on page 5-0

Data Types: char

CoderInfo — Code generation settings for structure or array variable

`Simulink.CoderInfo` object

Code generation settings for the structure variable (if you set `BreakpointsSpecification` to 'Explicit values' or 'Even spacing') or array variable ('Reference') that stores the lookup table and breakpoint sets, specified as a `Simulink.CoderInfo` object. You can specify a storage class or custom storage class by using this embedded object. See `Simulink.CoderInfo`.

StructTypeInfo — Settings for structure type in the generated code

`Simulink.lookupable.StructTypeInfo` object

Settings for the structure type that the structure variable uses in the generated code, specified as a `Simulink.lookuptable.StructTypeInfo` object.

If you set `BreakpointsSpecification` to 'Reference', the `Simulink.LookupTable` object does not appear in the generated code as a structure. The code generator ignores this property.

SupportTunableSize — Option to generate code that enables tunability of table size

`false` (default) | `true`

Option to generate code that enables tunability of the effective size of the table, specified as `true` or `false`. See “Support tunable size” on page 5-0 .

Data Types: `logical`

Table — Information for table data

`Simulink.lookuptable.Table` object

Information for the table data, specified as a `Simulink.lookuptable.Table` object.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Represent a One-Dimensional Lookup Table

- 1 Create a `Simulink.LookupTable` object named `LUTObj`.

```
LUTObj = Simulink.LookupTable;
```

- 2 Specify the table data.

```
LUTObj.Table.Value = [1.1 2.2 3.3 4.4 5.5];
```

- 3 Specify the breakpoint set data.

```
LUTObj.Breakpoints(1).Value = [-2 -1 0 1 2];
```

- 4 Specify a name for the structure type in the generated code.

```
LUTObj.StructTypeInfo.Name = 'myLUTStruct';
```

You can use LUTObj in a 1-D Lookup Table block dialog box. In the block, set **Data specification** to `Lookup table object` and **Name** to LUTObj.

Represent a Two-Dimensional Lookup Table

- 1 Create a Simulink.LookupTable object named LUTObj.

```
LUTObj = Simulink.LookupTable;
```

- 2 Specify the table data.

```
LUTObj.Table.Value = [1.1 2.2 3.3 4.4 5.5; ...
                    6.6 7.7 8.8 9.9 10.1];
```

- 3 Specify the breakpoint set data. In the Breakpoints property, use the vector index 2 to set the values in the second breakpoint set.

```
LUTObj.Breakpoints(1).Value = [-1 1];
```

```
LUTObj.Breakpoints(2).Value = [-2 -1 0 1 2];
```

LUTObj creates a Simulink.lookuptable.Breakpoint object as the second vector element in the value of the Breakpoints property. Except for the Value property, the new object has default property values.

- 4 Specify a name for the structure type in the generated code.

```
LUTObj.StructTypeInfo.Name = 'myLUTStruct';
```

You can use LUTObj in a 2-D Lookup Table block dialog box.

Evenly Space Every Second Value Starting from 1

To evenly space every second value starting from 1, use the Breakpoint object.

- 1 Create a Simulink.LookupTable object named LUTObj.

```
LUTObj=Simulink.LookupTable
```

```
LUTObj =
```

```
LookupTable with properties:
```

```
        Table: [1x1 Simulink.lookuptable.Table]
BreakpointsSpecification: 'Explicit values'
        Breakpoints: [1x1 Simulink.lookuptable.Breakpoint]
SupportTunableSize: 0
        CoderInfo: [1x1 Simulink.CoderInfo]
        StructTypeInfo: [1x1 Simulink.lookuptable.StructTypeInfo]
```

2 Set up the breakpoint property to even spacing.

```
LUTObj.BreakpointsSpecification='Even spacing'
```

```
LUTObj =
```

```
LookupTable with properties:
```

```
        Table: [1x1 Simulink.lookuptable.Table]
BreakpointsSpecification: 'Even spacing'
        Breakpoints: [1x1 Simulink.lookuptable.Evenspacing]
SupportTunableSize: 0
        CoderInfo: [1x1 Simulink.CoderInfo]
        StructTypeInfo: [1x1 Simulink.lookuptable.StructTypeInfo]
```

3 Get the properties of the breakpoint.

```
LUTObj.Breakpoints(1)
```

```
ans =
```

```
Evenspacing with properties:
```

```
    FirstPoint: []
      Spacing: []
    DataType: 'auto'
        Min: []
        Max: []
        Unit: ''
FirstPointName: 'BPFirstPoint1'
  SpacingName: 'BPSpacing1'
TunableSizeName: 'N1'
  Description: ''
```

- 4 To set the first point property, use the Breakpoint object `FirstPoint` property.

```
LUTObj.Breakpoints(1).FirstPoint=1
```

- 5 To set the spacing property, use the Breakpoint object.

```
LUTObj.Breakpoints(1).Spacing=2
```

- 6 Get the properties of the breakpoint.

```
LUTObj.Breakpoints(1)
```

```
ans =
```

```
    Evenspacing with properties:
```

```

    FirstPoint: 1
      Spacing: 2
    DataType: 'auto'
         Min: []
         Max: []
         Unit: ''
    FirstPointName: 'BPFirstPoint1'
      SpacingName: 'BPSpacing1'
    TunableSizeName: 'N1'
      Description: ''

```

Control Code Generation for Lookup Table and Breakpoint Sets

Create a `Simulink.LookupTable` object named `LUTObj`.

```
LUTObj = Simulink.LookupTable;
```

Specify the table data.

```
LUTObj.Table.Value = [1.00 2.25 3.50 4.75 6.00; ...
                    7.25 8.50 9.75 11.00 12.25];
```

Specify the breakpoint set data. In the `Breakpoints` property, use the array index 2 to create an additional `Simulink.lookuptable.BreakpointInfo` object, which represents the second breakpoint set.

```
LUTObj.Breakpoints(1).Value = [-1 1];
```

```
LUTObj.Breakpoints(2).Value = [-2 -1 0 1 2];
```

Specify data types for the lookup table and each breakpoint set.

```
LUTObj.Table.DataType = 'fixdt(1,16,2)';  
LUTObj.Breakpoints(1).DataType = 'int16';  
LUTObj.Breakpoints(2).DataType = 'int16';
```

Specify unique names for the structure fields that store the table data and breakpoint sets in the generated code.

```
LUTObj.Table.FieldName = 'myTable';  
LUTObj.Breakpoints(1).FieldName = 'myBPSet1';  
LUTObj.Breakpoints(2).FieldName = 'myBPSet2';
```

Export the structure variable definition from the generated code by using the storage class `ExportedGlobal`.

```
LUTObj.CoderInfo.StorageClass = 'ExportedGlobal';
```

Name the structure type in the generated code `LUTStructType`. Export the structure type definition to a generated header file named `myLUTHdr.h`.

```
LUTObj.StructTypeInfo.Name = 'LUTStructType';  
LUTObj.StructTypeInfo.DataScope = 'Exported';  
LUTObj.StructTypeInfo.HeaderFileName = 'myLUTHdr.h';
```

In an n-D Lookup Table block in a model, set **Data specification** to `Lookup table object` and **Name** to `LUTObj`.

```
load_system('myModel_LUTObj')  
set_param('myModel_LUTObj/Lookup Table', 'DataSpecification', 'Lookup table object', ...  
         'LookupTableObject', 'LUTObj')
```

Generate code from the model.

```
rtwbuild('myModel_LUTObj')  
  
### Starting build procedure for model: myModel_LUTObj  
### Successful completion of code generation for model: myModel_LUTObj
```

The generated code defines the structure type `LUTStructType` in the generated header file `myLUTHdr.h`.

```
file = fullfile('myModel_LUTObj_ert_rtw', 'myLUTHdr.h');
rtwdemodbtype(file, 'typedef struct {' LUTStructType;', 1, 1)
```

```
typedef struct {
    int16_T myBPSet1[2];
    int16_T myBPSet2[5];
    int16_T myTable[10];
} LUTStructType;
```

The code uses the global structure variable LUTObj to store the table and breakpoint set data. The table data is scaled based on the specified fixed-point data type.

```
file = fullfile('myModel_LUTObj_ert_rtw', 'myModel_LUTObj.c');
rtwdemodbtype(file, 'LUTStructType LUTObj = {' /* Variable: LUTObj', 1, 1)
```

```
LUTStructType LUTObj = {
    { -1, 1 },

    { -2, -1, 0, 1, 2 },

    { 4, 29, 9, 34, 14, 39, 19, 44, 24, 49 }
} ; /* Variable: LUTObj
```

Generate Code That Uses Conditionally Compiled Dimension Lengths

Suppose your handwritten code conditionally allocates memory and initializes a lookup table based on dimension lengths that you specify as #define macros. This example shows how to generate code that uses your external table and breakpoint data.

Symbolic dimensions require that you use an ERT-based system target file, which requires Embedded Coder®.

Explore External Code

In your current folder, copy these macro definitions into a header file named ex_myHdr_LUT.h.

```
#include "rtwtypes.h"

#ifndef _HEADER_MYHDR_H_
#define _HEADER_MYHDR_H_
```

```
#define bp1Len 2
#define bp2Len 2

typedef struct {
    real_T BP1[bp1Len];
    real_T BP2[bp2Len];
    real_T Table[bp1Len * bp2Len];
} LUTObj_Type;

extern LUTObj_Type LUTObj;

#endif
```

Copy this static initialization code into a source file named `ex_mySrc_LUT.c`.

```
#include "ex_myHdr_LUT.h"

#if bp1Len == 2 && bp2Len == 2
LUTObj_Type LUTObj = {
    { 1.0, 2.0 },

    { 3.0, 4.0 },

    { 3.0, 2.0, 4.0, 1.0 }
};
#endif

#if bp1Len == 3 && bp2Len == 3
LUTObj_Type LUTObj = {
    { 1.0, 2.0, 3.0 },

    { 4.0, 5.0, 6.0 },

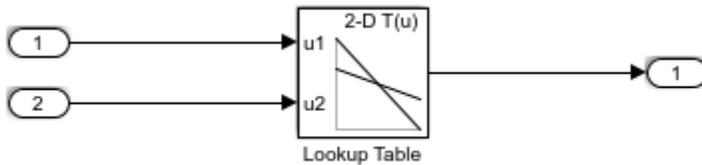
    { 1.0, 6.0, 2.0, 3.0, 8.0, 9.0, 5.0, 4.0, 7.0 }
};
#endif
```

To generate code that imports this data, create `bp1Len` and `bp2Len` as `Simulink.Parameter` objects in MATLAB. Create `LUTObj` as a `Simulink.LookupTable` object. Use the parameter objects to specify the dimension lengths for the table and breakpoint set data in the `Simulink.LookupTable` object.

Create Example Model

Create the example model `ex_LUTObj` by using an n-D Lookup Table block. In the Lookup Table block dialog box, on the **Table and Breakpoints** tab, set **Number of table dimensions** to 2.

```
open_system('ex_LUTObj')
```



Create Simulink.LookupTable Object

In the Model Explorer **Model Hierarchy** pane, select **Base Workspace**.

On the toolbar, click the **Add Simulink LookupTable** button. A `Simulink.LookupTable` object named `Object` appears in the base workspace.

In the **Contents** pane (the middle pane), rename the object as `LUTObj`.

Alternatively, create the object at the command prompt:

```
LUTObj = Simulink.LookupTable;
```

Configure Simulink.LookupTable Object

In the **Contents** pane, select the new object `LUTObj`. The property dialog box appears in the **Dialog** pane (the right pane).

Set **Number of table dimensions** to 2.

Under **Table**, set **Value** to `[3 4; 2 1]`.

In the first row under **Breakpoints**, set **Value** to `[1 2]`.

In the second row under **Breakpoints**, set **Value** to `[3 4]`. Click **Apply**.

Under **Struct Type definition**, set **Data scope** to `Imported`. Set **Header file** to `ex_myHdr_LUT.h`. Set **Name** to `LUTObj_Type`.

In the Lookup Table block dialog box, set **Data specification** to Lookup table object. Set **Name** to LUTObj. Click **Apply**.

Alternatively, to configure the object and the blocks, use these commands:

```
LUTObj.Breakpoints(1).Value = [1 2];
LUTObj.Breakpoints(2).Value = [3 4];
LUTObj.Table.Value = [3 4; 2 1];
LUTObj.StructTypeInfo.DataScope = 'Imported';
LUTObj.StructTypeInfo.HeaderFileName = 'ex_myHdr_LUT.h';
LUTObj.StructTypeInfo.Name = 'LUTObj_Type';
set_param('ex_LUTObj/Lookup Table', 'LookupTableObject', 'LUTObj')
set_param('ex_LUTObj/Lookup Table', ...
    'DataSpecification', 'Lookup table object')
```

Enable the code generator to use Simulink.Parameter objects as macros that specify dimension lengths. Select the configuration parameter **Allow symbolic dimension specification**.

```
set_param('ex_LUTObj', 'AllowSymbolicDim', 'on')
```

Create the Simulink.Parameter objects that represent the macros bp1Len and bp2Len. To generate code that imports the macros from your header file ex_myHdr_LUT.h, apply the custom storage class ImportedDefine.

```
bp1Len = Simulink.Parameter(2);
bp1Len.Min = 2;
bp1Len.Max = 3;
bp1Len.DataType = 'int32';
bp1Len.CoderInfo.StorageClass = 'Custom';
bp1Len.CoderInfo.CustomStorageClass = 'ImportedDefine';
bp1Len.CoderInfo.CustomAttributes.HeaderFile = 'ex_myHdr_LUT.h';

bp2Len = Simulink.Parameter(2);
bp2Len.Min = 2;
bp2Len.Max = 3;
bp2Len.DataType = 'int32';
bp2Len.CoderInfo.StorageClass = 'Custom';
bp2Len.CoderInfo.CustomStorageClass = 'ImportedDefine';
bp2Len.CoderInfo.CustomAttributes.HeaderFile = 'ex_myHdr_LUT.h';
```

Configure the existing Simulink.LookupTable object LUTObj to use the Simulink.Parameter objects. Set the dimension lengths of the breakpoint set data and the table data by using the names of the parameter objects.

```
LUTObj.Breakpoints(1).Dimensions = '[1 bp1Len]';
LUTObj.Breakpoints(2).Dimensions = '[1 bp2Len]';
LUTObj.Table.Dimensions = '[bp1Len bp2Len]';
```

Configure LUTObj as imported data by applying the custom storage class ImportFromFile. To import your definition of LUTObj, add the name of the file ex_mySrc_LUT.c to the model configuration parameter **Configuration Parameters > Code Generation > Custom Code > Additional Build Information > Source files**.

```
LUTObj.CoderInfo.StorageClass = 'Custom';
LUTObj.CoderInfo.CustomStorageClass = 'ImportFromFile';
LUTObj.CoderInfo.CustomAttributes.HeaderFile = 'ex_myHdr_LUT.h';
```

```
set_param('ex_LUTObj', 'CustomSource', 'ex_mySrc_LUT.c')
```

Generate and Inspect Code

Configure the model to compile an executable from the generated code.

```
set_param('ex_LUTObj', 'GenCodeOnly', 'off')
```

Generate code from the model.

```
rtwbuild('ex_LUTObj')

### Starting build procedure for model: ex_LUTObj
### Successful completion of build procedure for model: ex_LUTObj
```

In the code generation report, view the generated file ex_LUTObj.h. The file imports the macro definitions and the structure type definition by including your header file ex_myHdr_LUT.h.

```
file = fullfile('ex_LUTObj_ert_rtw', 'ex_LUTObj.h');
rtwdemodbtype(file, '#include "ex_myHdr_LUT.h"', '#include "ex_myHdr_LUT.h"', 1, 1)
```

```
#include "ex_myHdr_LUT.h"
```

In the source file ex_LUTObj.c, the code algorithm in the model step function passes the breakpoint and table data to the function that performs the table lookup. The algorithm also passes bp1Len so the lookup function can traverse the rows and columns of the table data, which appear in the generated code as a serialized 1-D array.

```
file = fullfile('ex_LUTObj_ert_rtw', 'ex_LUTObj.c');
rtwdemodbtype(file, '/* Model step function */', '/* Model initialize function */', 1, 0)
```

```
/* Model step function */
void ex_LUTObj_step(void)
{
    /* Outport: '<Root>/Out1' incorporates:
     * Inport: '<Root>/In1'
     * Inport: '<Root>/In2'
     * Lookup_n-D: '<Root>/Lookup Table'
     */
    ex_LUTObj_Y.Out1 = look2_binlcapw(ex_LUTObj_U.In1, ex_LUTObj_U.In2,
        (&(LUTObj.BP1[0])), (&(LUTObj.BP2[0])), (&(LUTObj.Table[0])),
        ex_LUTObj_ConstP.LookupTable_maxIndex, (uint32_T)bp1Len);
}
```

Limitations

- You cannot subclass `Simulink.Breakpoint` or `Simulink.LookupTable`. For this reason, you cannot apply custom storage classes other than those in the built-in Simulink package.
- You cannot use `Simulink.Breakpoint` objects or `Simulink.LookupTable` objects that refer to `Simulink.Breakpoint` objects as instance-specific parameter data for reusable components. For example, you cannot use one of these objects as:
 - A model argument in a model workspace or a model argument value in a Model block.
 - The value of a mask parameter on a CodeReuse Subsystem block.
 - The value of a mask parameter on a subsystem that you reuse by creating a custom library.

However, you can use standalone `Simulink.LookupTable` objects, which do not refer to `Simulink.Breakpoint` objects, in these ways.

- When blocks in a subsystem use `Simulink.LookupTable` or `Simulink.Breakpoint` objects, you cannot set data type override only on the subsystem. Instead, set data type override on the entire model.

See Also

`Simulink.Breakpoint` | `Simulink.Parameter` |
`Simulink.lookuptable.Breakpoint` | `Simulink.lookuptable.Evenspacing` |
`Simulink.lookuptable.StructTypeInfo` | `Simulink.lookuptable.Table`

Topics

“Configure STD_AXIS and COM_AXIS Lookup Tables for AUTOSAR Measurement and Calibration” (Embedded Coder)

“About Lookup Table Blocks”

“Package Shared Breakpoint and Table Data for Lookup Tables”

“Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder)

Introduced in R2016b

Simulink.lookuptable.Breakpoint class

Package: Simulink.lookuptable

Configure breakpoint set data for lookup table object

Description

An object of the `Simulink.lookuptable.Breakpoint` class stores breakpoint set information for a lookup table. The object resides in the `Breakpoints` property of a `Simulink.LookupTable` object or `Simulink.Breakpoint` object.

You can use `Simulink.LookupTable` and `Simulink.Breakpoint` objects to store and configure a lookup table for ASAP2 and AUTOSAR code generation.

To represent multiple breakpoint sets for a multidimensional lookup table, store a vector of `Simulink.lookuptable.Breakpoint` objects in the `Breakpoints` property of a `Simulink.LookupTable` object.

To share a breakpoint set between multiple lookup tables, use a `Simulink.Breakpoint` object to store and configure the breakpoint set information. Use the object in a Prelookup block and create `Simulink.LookupTable` objects to use in Interpolation Using Prelookup blocks.

Construction

When you create a `Simulink.LookupTable` object or `Simulink.Breakpoint` object, a `Simulink.lookuptable.Breakpoint` object appears as the value of the `Breakpoints` property.

To create more `Simulink.lookuptable.Breakpoint` objects for a `Simulink.LookupTable` object, use this technique:

Access the `Breakpoints` property by specifying a vector index.

To create a `Simulink.lookuptable.Breakpoint` object, you can set the value of any of the object properties. The `Simulink.LookupTable` object creates the

Simulink.lookupable.Breakpoint object with default property values, and sets the property that you specified.

The value of the `Breakpoints` property is an array of Simulink.lookupable.Breakpoint objects. Each embedded object represents one breakpoint set.

For example, suppose that you create a Simulink.LookupTable object named LUTObj. To create more breakpoint sets, access the `Breakpoints` property by specifying vector indices:

```
LUTObj.Breakpoints(1).Value = [-1 1];
LUTObj.Breakpoints(2).Value = [-2 -1 0 1 2];
LUTObj.Breakpoints(3).Value = [-5 -3 0 3 5];
```

The object LUTObj creates additional Simulink.lookupable.Breakpoint objects and sets the `Value` property of each object. LUTObj now stores information for three breakpoint sets.

Properties

Data Type — Data type of breakpoint set elements

'auto' (default) | character vector

Data type of breakpoint set elements, specified as a character vector. You can explicitly specify an integer, a floating-point, a fixed-point data type, or a data type expression such as the name of a Simulink.AliasType object.

The default value, 'auto', means that the breakpoint set acquires a data type from the value that you specify in the `Value` property. If you use an untyped expression such as `[1 2 3]` to set `Value`, the breakpoint data use the data type `double`. If you specify a typed expression such as `single([1 2 3])` or an `fi` object, the breakpoint data use the data type specified by the expression or object.

For more information about data types in Simulink, see “Data Types Supported by Simulink”. To decide how to control the data types of table and breakpoint data in Simulink.LookupTable and Simulink.Breakpoint objects, see “Control Data Types of Lookup Table Objects” (Simulink Coder).

Example: 'int16'

Example: 'myTypeAlias'

Data Types: char

Description — Description of breakpoint set

' ' (default) | character vector

Description of the breakpoint set, specified as a character vector.

Example: 'This breakpoint set represents the pressure input.'

Data Types: char

Dimensions — Dimension lengths of breakpoint set

[0 0] (default) | numeric vector

Dimension lengths of the breakpoint set, returned as a numeric vector or specified as a character vector.

To use symbolic dimensions, specify a character vector.

FieldName — Name of structure field that stores breakpoint set data

'BP' (default) | character vector

Name of a structure field in the generated code, specified as a character vector. This field stores the breakpoint set data.

The code generator uses this property only under these circumstances, which cause the breakpoint data to appear in the generated code as a structure field:

- The `Simulink.lookuptable.Breakpoint` object exists in a `Simulink.LookupTable` object and in the `Simulink.LookupTable` object you set `BreakpointsSpecification` to 'Explicit values'.
- The `Simulink.lookuptable.Breakpoint` object exists in a `Simulink.Breakpoint` object and in the `Simulink.Breakpoint` object you set `SupportTunableSize` to true.

Example: 'MyBkptSet1'

Data Types: char

Max — Maximum value of breakpoint set elements

[] (default) | numeric double value

Maximum value of the elements of the breakpoint set, specified as a numeric, real value of the data type `double`.

Example: `17.23`

Data Types: `double`

min — Minimum value of breakpoint set elements

`[]` (default) | numeric `double` value

Minimum value of the elements of the breakpoint set, specified as a numeric, real value of the data type `double`.

Example: `-52.6`

Data Types: `double`

TunableSizeName — Name of structure field that stores length of breakpoint set

`'N'` (default) | character vector

Name of a structure field in the generated code, specified as a character vector. This field stores the length of the breakpoint set, which the generated code algorithm uses to determine the size of the table. To tune the effective size of the table during code execution, change the value of this structure field in memory.

The code generator uses this property only under these circumstances, which enable a tunable table size in the generated code:

- The `Simulink.lookupable.Breakpoint` object exists in a `Simulink.LookupTable` object and in the `Simulink.LookupTable` object you set:
 - `BreakpointsSpecification` to `'Explicit values'`.
 - `SupportTunableSize` to `true`.
- The `Simulink.lookupable.Breakpoint` object exists in a `Simulink.Breakpoint` object and in the `Simulink.Breakpoint` object you set `SupportTunableSize` to `true`.

Example: `'LengthForDim1'`

Data Types: `char`

unit — Physical unit of breakpoint set

`''` (default) | character vector

Physical unit of the elements of the breakpoint set, specified as a character vector.

Example: 'inches'

Data Types: char

value — Breakpoint set data

[] (default) | numeric vector

The breakpoint set data, specified as a numeric vector with at least two elements. To control the data type of the breakpoint set, use the `DataType` property of the `Simulink.lookupable.Breakpoint` object.

When you set `DataType` to 'auto', to set the `Value` property, use a typed expression such as `single([1 2 3])` or use the `fi` constructor to embed an `fi` object.

Example: `[10 20 30]`

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fi

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

See Also

`Simulink.Breakpoint` | `Simulink.LookupTable`

Introduced in R2016b

Simulink.lookupable.Evenspacing class

Package: Simulink.lookupable

Configure even spacing set data for lookup table object

Description

An object of the `Simulink.lookupable.Evenspacing` class stores event spacing set information for a lookup table. The object resides in the `Evenspacing` property of a `Simulink.LookupTable` object.

You can use `Simulink.LookupTable` objects to store and configure a lookup table for ASAP2 and AUTOSAR code generation.

To represent multiple breakpoint sets for a multidimensional lookup table, store a vector of `Simulink.lookupable.Evenspacing` objects in the `Evenspacing` property of a `Simulink.LookupTable` object.

Construction

When you create a `Simulink.LookupTable` object and set `BreakpointSpecification` to 'Even spacing', a `Simulink.lookupable.Evenspacing` object appears as the value of the `Breakpoints` property.

To create more `Simulink.lookupable.Evenspacing` objects for a `Simulink.LookupTable` object, use this technique:

Access the `Breakpoints` property by specifying a vector index.

To create a `Simulink.lookupable.Evenspacing` object, you can set the value of any of the object properties. The `Simulink.LookupTable` object creates the `Simulink.lookupable.Evenspacing` object with default property values, and sets the property that you specified.

The value of the `Breakpoints` property is an array of `Simulink.lookupable.Evenspacing` objects. Each embedded object represents one breakpoint set.

For example, suppose that you create a `Simulink.LookupTable` object named `myLUTObj`. To create more breakpoint sets, access the `Breakpoints` property by specifying scalar indices for `FirstPoint` and `Spacing` properties. To create more even spacing breakpoint sets, update the object with this pair of properties:

```
LUTObj.Breakpoints(1).FirstPoint=-1;
LUTObj.Breakpoints(1).Spacing=2;
LUTObj.Breakpoints(1).FirstPoint=-2;
LUTObj.Breakpoints(1).Spacing=1;
LUTObj.Breakpoints(1).FirstPoint=-5;
LUTObj.Breakpoints(1).Spacing=2;
```

The object `myLUTObj` creates additional `Simulink.lookupable.Evenspacing` objects and sets the `FirstPoint` and `Spacing` properties of each object. `LUTObj` now stores information for three breakpoint sets.

Properties

FirstPoint — First point in evenly spaced breakpoint data

[] (default) | numeric scalar

First point in evenly spaced breakpoint data, specified as a numeric scalar. To control the data type of the breakpoint set, use the `DataType` property of the `Simulink.lookupable.Evenspacing` object.

When you set `DataType` to `'auto'`, to set the `FirstPoint` property, use a typed expression such as `single(1)` or use the `fi` constructor to embed a `fi` object.

Example: `-1`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

Spacing — Spacing between points in evenly spaced breakpoint data

[] (default) | numeric, positive, monotonically increasing scalar

Spacing between points in evenly spaced breakpoint data, specified as a numeric scalar. To control the data type of the breakpoint set, use the `DataType` property of the `Simulink.lookupTable.Evenspacing` object.

When you set `DataType` to `'auto'`, to set the `Spacing` property, use a typed expression such as `single(1)` or use the `fi` constructor to embed an `fi` object.

Example: `-1`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

`DataType` — Data type of breakpoint set elements

`'auto'` (default) | character vector

Data type of breakpoint set elements, specified as a character vector. You can explicitly specify an integer, a floating-point, a fixed-point data type, or a data type expression such as the name of a `Simulink.AliasType` object.

The default value, `'auto'`, means that the breakpoint set acquires a data type from the value that you specify in the `Value` property. If you use an untyped expression such as `[1 2 3]` to set `Value`, the breakpoint data use the data type `double`. If you specify a typed expression such as `single([1 2 3])` or an `fi` object, the breakpoint data use the data type specified by the expression or object.

For more information about data types in Simulink, see “Data Types Supported by Simulink”. To decide how to control the data types of table and breakpoint data in `Simulink.LookupTable` and `Simulink.Breakpoint` objects, see “Control Data Types of Lookup Table Objects” (Simulink Coder).

Example: `'int16'`

Example: `'myTypeAlias'`

Data Types: `char`

`Min` — Minimum value of breakpoint set elements

`[]` (default) | numeric `double` value

Minimum value of the elements of the breakpoint set, specified as a numeric, real value of the data type `double`.

Example: `-52.6`

Data Types: `double`

Max — Maximum value of breakpoint set elements

`[]` (default) | numeric `double` value

Maximum value of the elements of the breakpoint set, specified as a numeric, real value of the data type `double`.

Example: `17.23`

Data Types: `double`

Unit — Physical unit of breakpoint set

`' '` (default) | character vector

Physical unit of the elements of the breakpoint set, specified as a character vector.

Example: `'inches'`

Data Types: `char`

FirstPointName — Name of the `Simulink.lookuptable.Evenspacing` object that stores the information for the first point

`'BPFirstPoint1'` (default) | character vector

Name of the `Simulink.Breakpoint` object that stores the information for the first point, specified as a character vector. Generated code uses this name to display the first point.

Example: `'myFirstPointName'`

Data Types: `char`

SpacingName — Name of the `Simulink.lookuptable.Evenspacing` object that stores the information for the spacing

`'auto'` (default) | character vector

Name of the `Simulink.Breakpoint` object that stores the information for the spacing, specified as a character vector. Generated code uses this name to display the spacing.

Example: `'mySpacing'`

Data Types: `char`

TunableSizeName — Name of structure field that stores length of breakpoint set`'N'` (default) | character vector

Name of a structure field in the generated code, specified as a character vector. This field stores the length of the breakpoint set, which the generated code algorithm uses to determine the size of the table. To tune the effective size of the table during code execution, change the value of this structure field in memory.

The code generator uses the property only under these circumstances, which enable a tunable table size in the generated code:

- The `Simulink.lookuptable.Evenspacing` object exists in a `Simulink.LookupTable` object, in which you set `BreakpointsSpecification` to `'Even spacing'` and `SupportTunableSize` to `true`.

Example: `'LengthForDim1'`

Data Types: `char`

Description — Description of breakpoint set`''` (default) | character vector

Description of the breakpoint set, specified as a character vector.

Example: `'This breakpoint set represents the pressure input.'`

Data Types: `char`

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

See Also

`Simulink.LookupTable` | `Simulink.lookuptable.Breakpoint`

Introduced in R2017b

Simulink.lookupable.StructTypeInfo class

Package: Simulink.lookupable

Configure settings for structure type that lookup table object uses in the generated code

Description

An object of the `Simulink.lookupable.StructTypeInfo` class controls the structure type that the generated code creates to store data for lookup table objects. The `Simulink.lookupable.StructTypeInfo` object resides in the `StructTypeInfo` property of a `Simulink.LookupTable` object or `Simulink.Breakpoint` object. Use these parent objects to store and configure a lookup table for ASAP2 and AUTOSAR code generation.

A `Simulink.LookupTable` object appears as a structure in the generated code when you set the `Specification` property to 'Explicit values'. A `Simulink.Breakpoint` object appears as a structure in the generated code when you set the `SupportTunableSize` property to true.

Construction

When you create a `Simulink.LookupTable` or `Simulink.Breakpoint` object, a `Simulink.lookupable.StructTypeInfo` object appears as the value of the `StructTypeInfo` property.

Properties

DataScope — Scope of structure type definition

'Auto' (default) | 'Exported' | 'Imported'

Scope of structure type definition, specified as a character vector.

Data Types: char

HeaderFileName — Name of header file that contains structure type definition`' '` (default) | character vector

Name of the header file that contains the structure type definition, specified as a character vector.

Example: `'myHdr.h'`

Data Types: `char`

Name — Name of structure type`' '` (default) | character vector

Name of the structure type, specified as a character vector.

Data Types: `char`

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

See Also

`Simulink.Breakpoint` | `Simulink.LookupTable`

Introduced in R2016b

Simulink.lookuptable.Table class

Package: Simulink.lookuptable

Configure table data for lookup table object

Description

An object of the `Simulink.lookuptable.Table` class stores table information for a lookup table. The object resides in the `Table` property of a `Simulink.LookupTable` object. You can use the `Simulink.LookupTable` object to store and configure a lookup table for ASAP2 and AUTOSAR code generation.

Construction

When you create a `Simulink.LookupTable` object, a `Simulink.lookuptable.Table` object appears as the value of the `Table` property.

Properties

Data Type — Data type of table data elements

'auto' (default) | character vector

Data type of the table data elements, specified as a character vector. You can explicitly specify an integer data type, a floating-point data type, a fixed-point data type, or a data type expression such as the name of a `Simulink.AliasType` object.

The default value, 'auto', means that the table data acquire a data type from the value that you specify in the `Value` property. If you use an untyped expression such as `[1 2 3]` to set `Value`, the table data use the data type `double`. If you specify a typed expression such as `single([1 2 3])` or an `fi` object, the table data use the data type specified by the expression or object.

For more information about data types in Simulink, see “Data Types Supported by Simulink”. To decide how to control the data types of table and breakpoint data in

Simulink.LookupTable and Simulink.Breakpoint objects, see “Control Data Types of Lookup Table Objects” (Simulink Coder).

Example: 'int16'

Example: 'myTypeAlias'

Data Types: char

Description — Description of table data

' ' (default) | character vector

Description of the table data, specified as a character vector.

Example: 'This lookup table describes the action of a pump.'

Data Types: char

Dimensions — Dimension lengths of table data

[0 0] (default) | numeric vector

Dimension lengths of the table data, returned as a numeric vector or specified as a character vector.

To use symbolic dimensions, specify a character vector.

FieldName — Name of a structure field in the generated code

'Table' (default) | character vector

Name of a structure field in the generated code, specified as a character vector. This field stores the table data if you configure the Simulink.LookupTable object to appear in the generated code as a structure.

Example: 'MyPumpTable'

Data Types: char

Max — Maximum value of table data elements

[] (default) | numeric double value

Maximum value of the elements of the table data, specified as a numeric, real value of the data type double.

Example: 17.23

Data Types: `double`

Min — Minimum value of table data elements

`[]` (default) | numeric `double` value

Minimum value of the elements of the table data, specified as a numeric, real value of the data type `double`.

Example: `-52.6`

Data Types: `double`

Unit — Physical unit of table elements

`' '` (default) | character vector

Physical unit of the elements of the table data, specified as a character vector.

Example: `'degC'`

Data Types: `char`

Value — Table data

`[]` (default) | numeric vector or multidimensional array

The table data, specified as a numeric vector or multidimensional array with at least two elements. To control the data type of the table data, use the `DataType` property of the `Simulink.lookuptable.Table` object.

When you set `DataType` to `'auto'`, to set the `Value` property, use a typed expression such as `single([1 2 3])` or use the `fi` constructor to embed an `fi` object.

Example: `[10 20 30; 40 50 60]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

See Also

`Simulink.Breakpoint` | `Simulink.LookupTable`

Introduced in R2016b

Simulink.MDLInfo class

Package: Simulink

Extract model file information without loading block diagram into memory

Description

The class `Simulink.MDLInfo` extracts information from a model file without loading the block diagram into memory.

You can create an `MdlInfo` object containing all the model information properties, or you can use the static methods for convenient access to individual properties without creating the class first. For example, to get the description only:

```
description = Simulink.MDLInfo.getDescription('mymodel')
```

To get the metadata only:

```
metadata = Simulink.MDLInfo.getMetadata('mymodel')
```

All model information properties are read only.

Construction

`info = Simulink.MDLInfo('mymodel')` creates an instance of the `MdlInfo` class `info` and populates the properties with the information from the model file `'mymodel'`.

`mymodel` can be:

- A block diagram name (for example, `vdp`)
- The file name for a file on the MATLAB path (for example, `mymodel.slx`)
- A file name relative to the current folder (for example, `mydir/mymodel.slx`)
- A fully qualified file name (for example, `C:\mydir\mymodel.slx`)

`Simulink.MDLInfo` resolves the supplied name by looking at files on the MATLAB path, and ignores any block diagrams in memory. This may cause unexpected results if

you supply the name of a loaded model, but its file is shadowed by another file on the MATLAB path. If a file is shadowed, you see a warning in the command window. To avoid any confusion, supply a fully-qualified file name to `Simulink.MDLInfo`.

Properties

BlockDiagramName

Name of block diagram.

Description

Description of model.

FileName

Name of model file.

Interface

Names and attributes of the block diagram's root inports, outports, model references, etc., describing the graphical interface if you created a Model Reference block from this model.

Structure.

IsLibrary

Whether the block diagram is a library.

LastSavedArchitecture

Platform architecture when saved, for example, 'glnxa64'.

Metadata

Names and attributes of arbitrary data associated with the model.

Structure. The structure fields can be character vectors, numeric matrices of type "double", or more structures. Use the method `getMetadata` to extract this metadata structure without loading the model.

ModelVersion

Model version number.

ReleaseName

Name of release, for example, 'R2016a'.

SavedCharacterEncoding

Character encoding when saved, for example, 'UTF-8'.

SimulinkVersion

Version number of Simulink software that was used to save the model file.

Methods

`getDescription` Extract model file description without loading block diagram into memory

`getMetadata` Extract model file metadata without loading block diagram into memory

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

Examples

Construct and view a model information object:

```
info = Simulink.MDLInfo('mymodel')
% Get the Version when the model was saved
simulink_version = info.SimulinkVersion;
% Get model metadata
metadata = info.metadata
```

To add metadata to a model, create a metadata structure containing the information you require and use `set_param` to attach it to the model. For example:

```
metadata.TestStatus = 'untested';
metadata.ExpectedCompletionDate
    = '01/01/2011';
load_system(mymodelname);
set_param(mymodelname, 'Metadata', ...
    metadata) % must be a struct
save_system(mymodelname);
close_system(mymodelname);
```

Construct a model information object for a model named `mpowertrain`, in order to find the names of referenced models without loading the model into memory:

```
info = Simulink.MDLInfo('mpowertrain')
% Get the Interface property
info.Interface
```

Output:

```
ans =
           Inports: [0x1 struct]
           Outports: [0x1 struct]
           Trigports: [0x1 struct]
           Connports: [0x1 struct]
           ModelVersion: '1.122'
           ModelReferences: {2x1 cell}
           ParameterArgumentNames: ''
           TestPointedSignals: [0x1 struct]
```

Get the referenced models:

```
info.Interface.ModelReferences
```

Output is in the form *model name / block path | referenced model name*:

```
ans =
    'mpowertrain/Model Variants|manual_transmission'
    'mpowertrain/engine model|menginemodel'
```

See Also

`Simulink.MDLInfo.getDescription`; `Simulink.MDLInfo.getMetadata`

Simulink.MDLInfo.getDescription

Class: Simulink.MDLInfo

Package: Simulink

Extract model file description without loading block diagram into memory

Syntax

```
description = Simulink.MDLInfo.getDescription('mymodel')
description = info.getDescription
```

Description

`description = Simulink.MDLInfo.getDescription('mymodel')` returns the description associated with the file *mymodel*, without loading the model.

mymodel can be:

- A block diagram name (for example, `vdp`)
- The file name for a file on the MATLAB path (for example, `mymodel.slx`)
- A file name relative to the current folder (for example, `mydir/mymodel.slx`)
- A fully qualified file name (for example, `C:\mydir\mymodel.slx`)

`description = info.getDescription` returns the description property of the Simulink.MDLInfo object `info`.

Examples

Get the description without loading the model or creating a Simulink.MDLInfo object:

```
description = Simulink.MDLInfo.getDescription('mymodel')
```

Create a Simulink.MDLInfo object containing all the model information properties, and get the description property:

```
info = Simulink.MDLInfo('mymodel')  
description = info.getDescription
```

See Also

`Simulink.MDLInfo`; `Simulink.MDLInfo.getMetadata`

Simulink.MDLInfo.getMetadata

Class: Simulink.MDLInfo

Package: Simulink

Extract model file metadata without loading block diagram into memory

Syntax

```
metadata = Simulink.MDLInfo.getMetadata('mymodel')  
metadata = info.getMetadata
```

Description

`metadata = Simulink.MDLInfo.getMetadata('mymodel')` extracts the structure metadata associated with the file `mymodel`, without loading the model.

`mymodel` can be:

- A block diagram name (for example, `vdp`)
- The file name for a file on the MATLAB path (for example, `mymodel.slx`)
- A file name relative to the current folder (for example, `mydir/mymodel.slx`)
- A fully qualified file name (for example, `C:\mydir\mymodel.slx`)

`metadata = info.getMetadata` returns the metadata property of the Simulink.MDLInfo object `info`.

`metadata` is a structure containing the names and attributes of arbitrary data associated with the model. The structure fields can be character vectors, numeric matrices of type "double", or more structures.

To add metadata to a model, create a metadata structure containing the information you require and use `set_param` to attach it to the model. If it is important to extract the information without loading the model, use `metadata` instead of adding custom user data with `add_param`.

Examples

Create a metadata structure and use `set_param` to attach it to the model:

```
metadata.TestStatus = 'untested';  
metadata.ExpectedCompletionDate = '01/01/2011';  
load_system('mymodel');  
set_param('mymodel','Metadata',metadata) % must be a struct  
save_system('mymodel');  
close_system('mymodel');
```

Get the metadata without loading the model or creating a `Simulink.MDLInfo` object:

```
metadata = Simulink.MDLInfo.getMetadata('mymodel')
```

Create a `Simulink.MDLInfo` object containing all the model information properties, and get the metadata property:

```
info = Simulink.MDLInfo('mymodel')  
metadata = info.getMetadata
```

See Also

`Simulink.MDLInfo`; `Simulink.MDLInfo.getDescription`

Simulink.ModelAdvisor

Run Model Advisor from MATLAB file

Description

Use instances of this class in MATLAB programs to run the Model Advisor, for example, to perform a standard set of checks. MATLAB software creates an instance of this object for each model that you open in the current MATLAB session. To get a handle to a model's Model Advisor object, execute the following command

```
ma = Simulink.ModelAdvisor.getModelAdvisor(model);
```

where *model* is the name of the model or subsystem that you want to check. Your program can then use the Model Advisor object's methods to initialize and run the Model Advisor's checks.

About IDs

Many `Simulink.ModelAdvisor` object methods require or return IDs. An *ID* is a unique identifier for a Model Advisor check, task, or group. ID must remain constant. A `Simulink.ModelAdvisor` object includes methods that enable you to retrieve the ID or IDs for all checks, tasks, and groups, checks belonging to groups and tasks, the active check, and selected checks, tasks and groups.

You find check IDs in the Model Advisor, using check context menus.

To Find	Do This
Check Title, ID, or location of the MATLAB source code	<ol style="list-style-type: none"> 1 On the model window toolbar, select Settings > Preferences. 2 In the Model Advisor Preferences dialog box, select Show Source Tab. 3 In the right pane of the Model Advisor window, click the Source tab. The Model Advisor window displays the check Title, TitleId, and location of the MATLAB source code for the check.

To Find	Do This
Check ID	<ol style="list-style-type: none"> 1 In the left pane of the Model Advisor, select the check. 2 Right-click the check name and select Send Check ID to Workspace. The ID is displayed in the Command Window and sent to the base workspace.
Check IDs for selected checks in a folder	<ol style="list-style-type: none"> 1 In the left pane of the Model Advisor, select the checks for which you want IDs. Clear the other checks in the folder. 2 Right-click the folder and select Send Check ID to Workspace. An array of the selected check IDs are sent to the base workspace.

Syntax

```
ma = Simulink.ModelAdvisor
```

Arguments

ma

A variable representing the `Simulink.ModelAdvisor` object you create.

Properties

EmitInputParametersToReport

The `EmitInputParametersToReport` property specifies the display of check input parameters in the Model Advisor report.

Value	Description
'true' (default)	Display check input parameters in the Model Advisor report.
'false'	Do not display check input parameters in the Model Advisor report.

Method Summary

Name	Description
“closeReport” on page 5-291	Close Model Advisor report.
“deselectCheck” on page 5-291	Clear checks.
“deselectCheckAll” on page 5-292	Clear all checks.
“deselectCheckForGroup” on page 5-292	Clear a group of checks.
“deselectCheckForTask” on page 5-293	Clear checks that belong to a specified task or set of tasks.
“deselectTask” on page 5-293	Clear tasks.
“deselectTaskAll” on page 5-294	Clear all tasks.
“displayReport” on page 5-294	Display Model Advisor report.
“exportReport” on page 5-295	Copy report to a specified location.
“filterResultWithExclusion” on page 5-295	Filter objects that have been excluded by user-defined exclusions.
“getBaselineMode” on page 5-296	Get baseline mode setting for the Model Advisor.
“getCheckAll” on page 5-297	Get the IDs of the checks performed by the Model Advisor.
“getCheckForGroup” on page 5-297	Get checks belonging to a check group.
“getCheckForTask” on page 5-298	Get checks belonging to a task.
“getCheckResult” on page 5-298	Get check results.
“getCheckResultData” on page 5-299	Get check result data.
“getCheckResultStatus” on page 5-299	Get pass/fail status of a check or set of checks.
“getGroupAll” on page 5-300	Get the IDs of the groups of tasks performed by the Model Advisor.
“getInputParameters” on page 5-300	Get input parameters of a check.
“getListviewParameters” on page 5-301	Get list view parameters of a check.
“getModelAdvisor” on page 5-302	Get the Model Advisor for a model or subsystem.

Name	Description
"getSelectedCheck" on page 5-303	Get selected checks.
"getSelectedSystem" on page 5-303	Get path of system currently targeted by the Model Advisor.
"getSelectedTask" on page 5-303	Get selected tasks.
"getTaskAll" on page 5-304	Get the IDs of the tasks performed by the Model Advisor.
"Simulink.ModelAdvisor.openConfigUI" on page 5-304	Start the Model Advisor Configuration editor.
"Simulink.ModelAdvisor.reportExists" on page 5-305	Determine whether a report exists for a system or subsystem.
"runCheck" on page 5-305	Run selected checks.
"runTask" on page 5-306	Run checks for selected tasks.
"selectCheck" on page 5-306	Select checks.
"selectCheckAll" on page 5-307	Select all checks.
"selectCheckForGroup" on page 5-307	Select a group of checks.
"selectCheckForTask" on page 5-308	Select checks that belong to a specified task.
"selectTask" on page 5-308	Select tasks.
"selectTaskAll" on page 5-309	Select all tasks.
"setActionEnable" on page 5-309	Set enable/disable status for a check action.
"setBaselineMode" on page 5-310	Set baseline mode for the Model Advisor.
"setCheckErrorSeverity" on page 5-311	Set severity of a check failure.
"setCheckResult" on page 5-311	Set result for the currently running check.
"setCheckResultData" on page 5-312	Set result data for the currently running check.
"setCheckResultStatus" on page 5-313	Set pass/fail status for the currently running check.

Name	Description
“setListViewParameters” on page 5-314	Set list view parameters for a check.
“verifyCheckRan” on page 5-314	Verify that checks have run.
“verifyCheckResult” on page 5-315	Generate a baseline set of check results or compare the current set of results to the baseline results.
“verifyCheckResultStatus” on page 5-316	Verify that a model has passed or failed a set of checks.
“verifyHTML” on page 5-317	Generate a baseline report or compare the current report to a baseline report.

Methods

closeReport

Close Model Advisor report

```
closeReport
```

Closes the report associated with this Model Advisor object, which closes the Model Advisor window.

“displayReport” on page 5-294

deselectCheck

Clear check

```
success = deselectCheck(ID)
```

ID

Character vector or cell array that specifies the IDs of the checks to be cleared.

`success`

True (1) if the check is cleared.

This method clears the checks specified by `ID`.

Note This method cannot clear disabled checks.

“`getCheckAll`” on page 5-297, “`deselectCheckForGroup`” on page 5-292, “`selectCheck`” on page 5-306

deselectCheckAll

Clear all checks

```
success = deselectCheckAll
```

`success`

True (1) if all checks are cleared.

Clears all checks that are not disabled.

“`selectCheckAll`” on page 5-307

deselectCheckForGroup

Clear group of checks

```
success = deselectCheckForGroup(groupName)
```

groupName

Character vector or cell array that specifies the names of the groups to be cleared.

success

True (1) if the method succeeds in clearing the specified group.

Clears a specified group of checks.

“selectCheckForGroup” on page 5-307

deselectCheckForTask

Clear checks that belong to specified task or set of tasks

```
success = deselectCheckForTask(ID)
```

ID

Character vector or cell array of character vectors that specify the IDs of tasks whose checks are to be cleared.

success

True (1) if the specified tasks are cleared.

Clears checks belonging to the tasks specified by the *ID* argument.

“getTaskAll” on page 5-304, “selectCheckForTask” on page 5-308

deselectTask

Clear task

```
success = deselectTask(ID)
```

ID

Character vector or cell array that specifies the ID of tasks to be cleared

`success`

True (1) if the method succeeded in clearing the specified tasks.

Clears the tasks specified by *ID*.

“selectTask” on page 5-308, “getTaskAll” on page 5-304

deselectTaskAll

Clears all tasks

```
success = deselectTaskAll
```

`success`

True (1) if this method succeeds in clearing all tasks.

Clears all tasks.

“selectTaskAll” on page 5-309

displayReport

Display report in Model Advisor window

```
displayReport
```

Displays the report associated with this Model Advisor object in the Model Advisor window. The report includes the most recent results of running checks on the system

associated with this Model Advisor object and the current selection status of checks, groups, and tasks for the system.

“Simulink.ModelAdvisor.reportExists” on page 5-305

exportReport

Create copy of report generated by Model Advisor

```
[success message] = exportReport(destination)
```

destination

Path name of copy to be made of the report file.

success

True (1) if this method succeeded in creating a copy of the report at the specified location.

message

Empty if the copy was successful; otherwise, the reason the copy did not succeed.

This method creates a copy of the last report generated by the Model Advisor and stores the copy at the specified location.

“Simulink.ModelAdvisor.reportExists” on page 5-305

filterResultWithExclusion

Filter objects that have been excluded by user-defined exclusions.

```
filteredResultHandles = obj.filterResultWithExclusion(ResultHandles)
```

filteredResultHandles

An array of objects causing exclusion enabled checks to warn or fail.

obj

A variable representing the `Simulink.ModelAdvisor.getModelAdvisor` object.

ResultHandles

An array of objects causing a check warning or failure.

This method filters objects that cause a check warning or failure with checks that have exclusions enabled.

Note This method is intended for excluding objects from custom checks created with the Model Advisor's customization API, a feature available with Simulink Check™.

“`getModelAdvisor`” on page 5-302

getBaselineMode

Determine whether Model Advisor is in baseline data generation mode

```
mode = getBaselineMode
```

mode

Boolean value indicating baseline mode.

The `mode` output variable returns true if the Model Advisor is in baseline data mode. Baseline data mode causes the verification methods of the Model Advisor, for example, “`verifyHTML`” on page 5-317, to generate baseline data.

“`setBaselineMode`” on page 5-310, “`verifyHTML`” on page 5-317, “`verifyCheckResult`” on page 5-315, “`verifyCheckResultStatus`” on page 5-316

getCheckAll

Get IDs of all checks

```
IDs = getCheckAll
```

IDs

Cell array of character vectors specifying the IDs of all checks performed by the Model Advisor.

Returns a cell array of character vectors specifying the IDs of all checks performed by the Model Advisor.

“getTaskAll” on page 5-304, “getGroupAll” on page 5-300

getCheckForGroup

Get checks that belong to check group

```
IDs = getCheckForGroup(groupName)
```

groupName

Character vector specifying the name of a group.

IDs

Cell array of IDs.

Returns a cell array of IDs of the tasks and checks belonging to the group specified by *groupName*.

“getCheckForTask” on page 5-298

getCheckForTask

Get checks that belong to task

```
checkIDs = getCheckForTask(taskID)
```

taskID

ID of a task.

checkIDs

Cell array of IDs of checks belonging to the specified task.

Returns a cell array of IDs of the checks belonging to the task specified by *taskID*.

“getCheckForGroup” on page 5-297

getCheckResult

Get results of running check or set of checks

```
result = getCheckResult(ID)
```

ID

ID of a check or cell array of check IDs.

result

A check result or cell array of check results.

Gets results for the specified checks. The format of the results depends on the checks that generated the data.

Note This method is intended for accessing results generated by custom checks created with the Model Advisor's customization API, an optional feature available with Simulink

Check software. For more information, see “Define Custom Checks” (Simulink Check) in the Simulink Check documentation.

“getCheckResultData” on page 5-299, “getCheckResultStatus” on page 5-299

getCheckResultData

Get data resulting from running check or set of checks

```
result = getCheckResultData(ID)
```

ID

Check ID or cell array of check IDs.

result

Data from a check result or cell array of data from check results.

Gets the check result data for the specified checks. The format of the data depends on the checks that generated the data.

Note This method is intended for accessing check result data generated by custom checks created with the Model Advisor's customization API, an optional feature available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check) in the Simulink Check documentation.

“getCheckResult” on page 5-298, “getCheckResultStatus” on page 5-299

getCheckResultStatus

Get status of check or set of checks

```
result = getCheckResultStatus(ID)
```

ID

Check ID or cell array of check IDs.

result

Boolean or a cell array of Boolean values indication the pass or fail status of a check or set of checks.

Invoke this method after running a set of checks to determine whether the checks passed or failed.

“getCheckResult” on page 5-298, “getCheckResultData” on page 5-299

getGroupAll

Get all groups of checks run by Model Advisor

```
IDs = getGroupAll
```

IDs

Cell array of IDs of all groups of checks run by the Model Advisor.

Returns a cell array of IDs of all groups of checks run by the Model Advisor.

“getCheckAll” on page 5-297, “getTaskAll” on page 5-304

getInputParameters

Get input parameters of check

```
params = obj.getInputParameters(check_ID)
```

params

A cell array of `ModelAdvisor.InputParameter` objects.

obj

A variable representing the `Simulink.ModelAdvisor` object.

check_ID

A character vector that uniquely identifies the check.

You can omit the *check_ID* if you use the method inside a check callback function.

Returns the input parameters associated with a check.

Note This method is intended for accessing custom checks created with the Model Advisor's customization API, an optional feature available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check) in the Simulink Check documentation.

`ModelAdvisor.InputParameter`

getListViewParameters

Get list view parameters of check

```
params = obj.getListViewParameters(check_ID)
```

params

A cell array of `ModelAdvisor.ListViewParameter` objects.

obj

A variable representing the `Simulink.ModelAdvisor` object.

check_ID

A character vector that uniquely identifies the check.

You can omit the *check_ID* if you use the method inside a check callback function.

Returns the list view parameters associated with a check.

Note This method is intended for accessing custom checks created with the Model Advisor's customization API, an optional feature available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check) in the Simulink Check documentation.

“setListViewParameters” on page 5-314, `ModelAdvisor.ListViewParameter`

getModelAdvisor

Get Model Advisor object for system or subsystem

```
obj = Simulink.ModelAdvisor.getModelAdvisor(system)
obj = Simulink.ModelAdvisor.getModelAdvisor(system, 'new')
```

system

Name of model or subsystem.

'new'

Required when changing Model Advisor working scope from one system to another without closing the previous session. Alternatively, you can close the previous session before invoking `getModelAdvisor`, in which case 'new' can be omitted.

obj

Model Advisor object.

This static method (see “Static Methods”) creates and returns an instance of `Simulink.ModelAdvisor` class for the model or subsystem specified by *system*.

getSelectedCheck

Get currently selected checks

```
IDs = getSelectedCheck
```

IDs

Cell array of IDs of currently selected checks.

Returns the IDs of the currently selected checks in the Model Advisor.

“getSelectedTask” on page 5-303

getSelectedSystem

Get system currently targeted by Model Advisor

```
path = getSelectedSystem
```

path

Path of the selected system.

Gets the path of the system currently targeted by the Model Advisor. That is, the system or subsystem most recently selected for checking either interactively by the user or programmatically via `Simulink.ModelAdvisor.getModelAdvisor`.

“getModelAdvisor” on page 5-302

getSelectedTask

Get selected tasks

```
IDs = getSelectedTask
```

IDs

Cell array of IDs of currently selected tasks.

Returns the IDs of the currently selected tasks in the Model Advisor.

“getSelectedCheck” on page 5-303

getTaskAll

Get tasks run by Model Advisor

```
IDs = getTaskAll
```

IDs

Cell array of IDs of tasks run by the Model Advisor.

Returns a cell array of IDs of tasks run by the Model Advisor.

“getCheckAll” on page 5-297, “getGroupAll” on page 5-300

Simulink.ModelAdvisor.openConfigUI

Starts Model Advisor Configuration editor

```
Simulink.ModelAdvisor.openConfigUI
```

This static method starts the Model Advisor Configuration editor. Use the Model Advisor Configuration editor to create customized configurations for the Model Advisor.

Note The Model Advisor Configuration editor is an optional feature available with Simulink Check software (see “Organize Checks and Folders Using the Model Advisor Configuration Editor” (Simulink Check) for more information).

- Before starting the Model Advisor Configuration editor, ensure that the current folder is writable. If the folder is not writable, you see an error message when you start the Model Advisor Configuration editor.
 - The Model Advisor Configuration editor uses the `slprj` folder in the code generation folder to store reports and other information. If the `slprj` folder does not exist in the code generation folder, the Model Advisor Configuration editor creates it. For more information, see “Model Reference Simulation Targets”.
-

Simulink.ModelAdvisor.reportExists

Determine whether report exists for model or subsystem

```
exists = reportexists('system')
```

system

Character vector specifying path of a system or subsystem.

`exists`

True (1) if a report exists for *system*.

This method returns true (1) if a report file exists for the model (system) or subsystem specified by *system* in the `slprj/modeladvisor` subfolder of the MATLAB working folder.

“exportReport” on page 5-295

runCheck

Run currently selected checks

```
success = runCheck(ID)
```

ID

ID or cell array of IDs of checks to run.

success

True (1) if the checks were run.

Runs the checks currently selected in the Model Advisor. Invoking this method is equivalent to selecting the **Run Selected Checks** button on the Model Advisor window.

“selectCheck” on page 5-306

runTask

Run currently selected tasks

```
success = runTask
```

success

True (1) if the tasks were run.

Runs the tasks currently selected in the Model Advisor. Invoking this method is equivalent to selecting the **Run Selected Checks** button on the Model Advisor window.

“selectTask” on page 5-308

selectCheck

Select check

```
success = selectCheck(ID)
```

ID

ID or cell array of IDs of checks to be selected.

success

True (1) if this method succeeded in selecting the specified checks.

Select the check specified by *ID*. This method cannot select a check that is disabled.

“selectCheckAll” on page 5-307, “selectCheckForGroup” on page 5-307, “deselectCheck” on page 5-291

selectCheckAll

Select all checks

```
success = selectCheckAll
```

success

True (1) if this method succeeded in selecting all checks.

Selects all checks that are not disabled.

“selectCheck” on page 5-306, “selectCheckForGroup” on page 5-307, “deselectCheck” on page 5-291

selectCheckForGroup

Select group of checks

```
success = selectCheckForGroup(ID)
```

ID

ID or cell array of group IDs.

success

True (1) if this method succeeded in selecting the specified groups

Selects the groups specified by *ID*.

“deselectCheckForGroup” on page 5-292

selectCheckForTask

Select checks that belong to specified task or set of tasks

```
success = selectCheckForTask(ID)
```

ID

ID or cell array of IDs of tasks whose checks are to be selected.

success

True (1) if this method succeeded in selecting the checks for the specified tasks

Selects checks belonging to the tasks specified by the *ID* argument.

“deselectCheckForTask” on page 5-293

selectTask

Select task

```
success = selectTask(ID)
```

ID

ID or cell array of IDs of the task to be selected.

success

True (1) if this method succeeds in selecting the specified tasks.

Selects a task.

“deselectTask” on page 5-293

selectTaskAll

Select all tasks

```
success = selectTaskAll
```

success

True (1) if this method succeeds in selecting all tasks.

Selects all tasks.

“deselectTaskAll” on page 5-294

setActionEnable

Set status for check action

```
obj.setActionEnable(value)
```

obj

A variable representing the Simulink.ModelAdvisor object.

value

Boolean value indicating whether the Action box is enabled or disabled.

- `true` — enable the Action box.
- `false` — Disable the Action box.

The `setActionEnable` method specifies the enables or disables the Action box. Only a check callback function can invoke this method.

Note This method is intended for accessing custom checks created with the Model Advisor's customization API, an optional feature available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check) in the Simulink Check documentation.

`ModelAdvisor.Action`

setBaselineMode

Set baseline data generation mode for Model Advisor

`setBaselineMode(mode)`

mode

Boolean value indicating setting of Model Advisor's baseline mode, either on (`true`) or off (`false`).

Sets the Model Advisor's baseline mode to *mode*. Baseline mode causes the Model Advisor's verify methods to generate baseline comparison data for verifying the results of a Model Advisor run.

“`getBaselineMode`” on page 5-296, “`verifyCheckResult`” on page 5-315, “`verifyHTML`” on page 5-317

setCheckErrorSeverity

Set severity of check failure

```
obj.setCheckErrorSeverity(value)
```

obj

A variable representing the `Simulink.ModelAdvisor` object.

value

Integer indicating severity of failure.

- 0 — Check Result = Warning
- 1 — Check Result = Failed

Sets result status for a currently running check that fails to *value*. Only a check callback function can invoke this method.

Note This method is intended for accessing custom checks created with the Model Advisor's customization API, an optional feature available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check) in the Simulink Check documentation.

“setCheckResultStatus” on page 5-313

setCheckResult

Set result for currently running check

```
success = setCheckResult(result)
```

result

Character vector or cell array that specifies the result of the currently running task.

success

True (1) if this method succeeds in setting the check result.

Sets the check result for the currently running check. Only the callback function of a check can invoke this method.

Note This method is intended for use with custom checks created with the Model Advisor's customization API, an optional feature available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check) in the Simulink Check documentation.

“getCheckResult” on page 5-298, “setCheckResultData” on page 5-312,
“setCheckResultStatus” on page 5-313

setCheckResultData

Set result data for currently running check

```
success = setCheckResultData(data)
```

data

Result data to be set.

success

True (1) if this method succeeds in setting the result data for the current check

Sets the check result data for the currently running check. Only the callback function of a check can invoke this method.

Note This method is intended for use with custom checks created with the Model Advisor's customization API, an optional feature available with Simulink Check

software. For more information, see “Define Custom Checks” (Simulink Check) in the Simulink Check documentation.

“getCheckResultData” on page 5-299, “setCheckResult” on page 5-311,
“setCheckResultStatus” on page 5-313

setCheckResultStatus

Set status for currently running check

```
success = setCheckResultStatus(status)
```

status

Boolean value that indicates the status of the check that just ran, either pass (`true`) or fail (`false`)

success

True (1) if the status was set.

Sets the pass or fail status for the currently running check to *status*. Only the callback function of the check can invoke this method.

Note This method is intended for use with custom checks created with the Model Advisor's customization API, an optional feature available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check) in the Simulink Check documentation.

“getCheckResultStatus” on page 5-299, “setCheckResult” on page 5-311,
“setCheckResultData” on page 5-312, “setCheckErrorSeverity” on page 5-311

setListViewParameters

Specify list view parameters for check

```
obj.setListViewParameters(check_ID, params)
```

obj

A variable representing the `Simulink.ModelAdvisor` object.

check_ID

A character vector that uniquely identifies the check.

You can omit the *check_ID* if you use the method inside a check callback function.

params

A cell array of `ModelAdvisor.ListViewParameter` objects.

Set the list view parameters for the check.

Note This method is intended for accessing custom checks created with the Model Advisor's customization API, an optional feature available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check) in the Simulink Check documentation.

“getListViewParameters” on page 5-301, `ModelAdvisor.ListViewParameter`

verifyCheckRan

Verify that Model Advisor has run set of checks

```
[success, missingChecks, additionalChecks] = verifyCheckRan(IDs)
```

IDs

Cell array of IDs of checks to verify.

success

Boolean value specifying whether the checks ran.

missingChecks

Cell array of IDs for specified checks that did not run.

additionalChecks

Cell array of IDs for unspecified checks that ran.

The output variable `success` returns `true` if both:

- All the checks specified by *IDs* ran.
- Only the checks specified by *IDs* ran.

The `missingChecks` argument provides the specified checks that did not run. The `additionalChecks` argument lists unspecified checks that ran.

“`verifyCheckResultStatus`” on page 5-316

verifyCheckResult

Generate baseline Model Advisor check results file or compare current check results to baseline check results

```
[success message] = verifyCheckResult(baseline, checkIDs)
```

baseline

Path of the baseline check results MAT-file.

checkIDs

Cell array of check IDs.

success

Boolean value specifying whether the method succeeded.

message

Character vector specifying an error message.

If the Model Advisor is in baseline mode (see “setBaselineMode” on page 5-310), this method stores the most recent results of running the checks specified by `checkIDs` in a MAT-file at the location specified by `baseline`. If the method is unable to store the check results at the specified location, it returns `false` in the output variable `success` and the reason for the failure in the output variable `message`. If the Model Advisor is not in baseline mode, this method compares the most recent results of running the checks specified by `checkIDs` with the report specified by `baseline`. If the current results match the baseline results, this method returns `true` as the value of the `success` output variable.

Note You must run the checks specified by `checkIDs` (see “runCheck” on page 5-305) before invoking `verifyCheckResult`.

This method enables you to compare the most recent check results generated by the Model Advisor with a baseline set of check results. You can use the method to generate the baseline report as well as perform current-to-baseline result comparisons. To generate a baseline report, put the Model Advisor in baseline mode, using “setBaselineMode” on page 5-310. Then invoke this method with the `baseline` argument set to the location where you want to store the baseline results. To perform a current-to-baseline report comparison, first ensure that the Model Advisor is not in baseline mode (see “getBaselineMode” on page 5-296). Then invoke this method with the path of the baseline report as the value of the `baseline` input argument.

“setBaselineMode” on page 5-310, “getBaselineMode” on page 5-296, “runCheck” on page 5-305, “verifyCheckResultStatus” on page 5-316

verifyCheckResultStatus

Verify that model has passed or failed set of checks

```
[success message] = verifyCheckResultStatus(baseline, checkIDs)
```

baseline

Array of Boolean variables.

checkIDs

Cell array of check IDs.

success

Boolean value specifying whether the method succeeded.

message

Character vector specifying an error message.

This method compares the pass or fail (*true* or *false*) statuses from the most recent running of the checks specified by *checkIDs* with the Boolean values specified by *baseline*. If the statuses match the baseline, this method returns *true* as the value of the *success* output variable.

Note You must run the checks specified by *checkIDs* (see “runCheck” on page 5-305) before invoking *verifyCheckResultStatus*.

“runCheck” on page 5-305

verifyHTML

Generate baseline Model Advisor report or compare current report to baseline report

```
[success message] = verifyHTML(baseline)
```

baseline

Path of a Model Advisor report.

success

Boolean value specifying whether the method succeeded.

message

Character vector specifying an error message.

If the Model Advisor is in baseline mode (see “setBaselineMode” on page 5-310), this method stores the report most recently generated by the Model Advisor at the location specified by *baseline*. If the method is unable to store a copy of the report at the specified location, it returns *false* in the output variable *success* and the reason for the failure in the output variable *message*. If the Model Advisor is not in baseline mode, this method compares the report most recently generated by the Model Advisor with the report specified by *baseline*. If the current report has exactly the same content as the baseline report, this method returns *true* as the value of the *success* output variable.

This method enables you to compare a report generated by the Model Advisor with a baseline report to determine if they differ. You can use the method to generate the baseline report as well as perform current-to-baseline report comparisons. To generate a baseline report, put the Model Advisor in baseline mode. Then invoke this method with the baseline argument set to the location where you want to store the baseline report. To perform a current-to-baseline report comparison, first ensure that the Model Advisor is not in baseline mode (see “getBaselineMode” on page 5-296). Then invoke this method with the path of the baseline report as the value of the *baseline* input argument.

“setBaselineMode” on page 5-310, “getBaselineMode” on page 5-296, “verifyCheckResult” on page 5-315

Introduced in R2006a

Simulink.ModelDataLogs

Container for signal data logs of a model

Description

Note The `ModelDataLogs` class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy `ModelDataLogs` API”.

In releases before R2016a, if you set **Configuration Parameters > Data Import/Export > Signal logging format** to `ModelDataLogs`, Simulink software created instances of the `Simulink.ModelDataLogs` class to contain signal logs while simulating a model. Logging created an instance of this class for a top model and for each model referenced by the top model that contains signals to be logged. Logging assigned the `ModelDataLogs` object for the top model to a variable in the base workspace. The name of the variable is the name specified in the **Configuration Parameters > Data Import/export > Signal logging name** parameter. The default value is `logout`.

A `ModelDataLogs` object has a variable number of properties. The first property, named `Name`, specifies the name of the model whose signal data the object contains or, if the model is a referenced model, the name of the Model block that references the model. The remaining properties reference objects that contain signal data logged during simulation of the model. The objects may be instances of any of the following types of objects:

- `Simulink.ModelDataLogs`

Container for the data logs of a model

- `Simulink.SubsysDataLogs`

Container for the data logs of a subsystem

- `Simulink.Timeseries`

Data log for any signal except a mux or bus signal

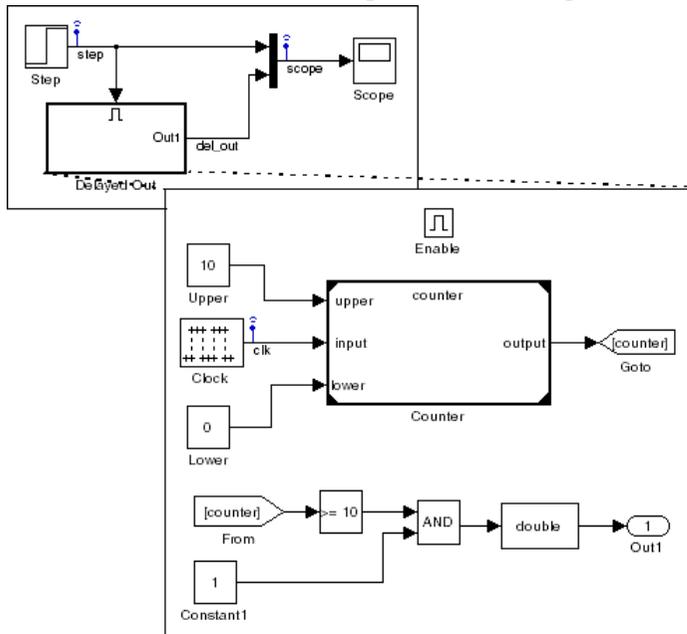
- `Simulink.TsArray`

Data log for a mux or bus signal

The names of the properties identify the data being logged as follows:

- For signal data logs, the name of the signal
- For a subsystem or model log container, the name of the subsystem or model, respectively

Consider, for example, the following model.



As indicated by the testpoint icons, this model specifies that Simulink software should log the signals named `step` and `scope` in the root system and the signal named `clk` in the subsystem named `Delayed Out`. After you simulate this model in a release earlier than R2016a, the MATLAB workspace contains the following variable:

```
Simulink.ModelDataLogs (siglgex):
  Name                elements  Simulink Class
  scope                2        TsArray
  step                 1        Timeseries
  ('Delayed Out')     2        SubsysDataLogs
```

You can use fully qualified object names or the Simulink `unpack` command to access the signal data. For example, to access the amplitudes of the `clk` signal in the `Delayed Out` subsystem in a `logsout` object, enter

```
data = logsout('Delayed Out').clk.Data;
```

or

```
>> logsout.unpack('all');
>> data = clk.Data;
```

Access Logged Signal Data Saved in ModelDataLogs Format

The `Simulink.ModelDataLogs` object contains signal data objects to capture signal logging information for specific model elements.

Model Element	Signal Data Object
Top-level or referenced model	<code>Simulink.ModelDataLogs</code>
Subsystem in a model	<code>Simulink.SubsysDataLogs</code>
Signal other than a bus or Mux signal	<code>Simulink.Timeseries</code>
Bus signal or Mux signal	<code>Simulink.TsArray</code>

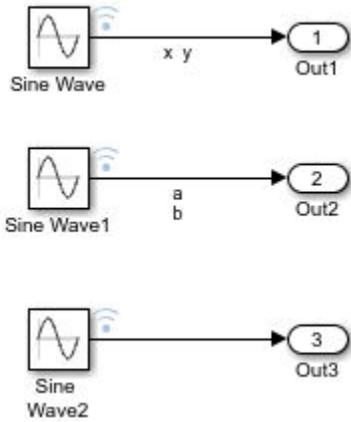
Handling Spaces and Newlines in Logged Names

Signal names in data logs can have spaces or newlines in their names when the signal:

- Is named and the name includes a space or newline character.
- Is unnamed and originates in a block whose name includes a space or newline character.

- Exists in a subsystem or referenced model, and the name of the subsystem, Model block, or of any superior block includes a space or newline character.

The following model shows a signal whose name contains a space, a signal whose name contains a newline, and an unnamed signal that originates in a block whose name contains a newline:



The following example shows how to handle spaces or new lines in logged names, if a model uses ModelDataLogs for the signal logging format.

```
logout
```

```
logout =
```

```
Simulink.ModelDataLogs (model_name):
  Name                Elements  Simulink Class
  ('x y')              1        Timeseries
  ('a
b')                   1        Timeseries
  ('SL_Sine
Wave1')               1        Timeseries
```

You cannot access any of the `Simulink.Timeseries` objects in this log using TAB name completion or by typing the name to MATLAB. This syntax is not recognized because the space or newline in each name appears to the MATLAB parser as a separator between identifiers. For example:

```

logout.x y

??? logout.x y
      |
Error: Unexpected MATLAB expression.

```

To reference a `Simulink.Timeseries` object whose name contains a space, enclose the element containing the space in single quotes:

```

logout.('x y')

    Name: 'x y'
  BlockPath: 'model_name/Sine'
  PortIndex: 1
 SignalName: 'x y'
 ParentName: 'x y'
   TimeInfo: [1x1 Simulink.TimeInfo]
         Time: [51x1 double]
         Data: [51x1 double]

```

To reference a `Simulink.Timeseries` object whose name contains a newline, concatenate to construct the element containing the newline:

```

cr=sprintf('\n')
logout.(['a' cr 'b'])

```

The same techniques work when a space or newline in a data log derives from the name of:

- An unnamed logged signal's originating block
- A subsystem or Model block that contains any logged signal
- Any block that is superior to such a block in the model hierarchy

This code can reference logged data for the signal:

```

logout.(['SL_Sine' cr 'Wave1'])

```

For names with multiple spaces, newlines, or both, repeat and combine the two techniques as needed to specify the intended name to MATLAB.

Bus Signals

ModelDataLogs format stores each logged bus signal data in a separate `Simulink.TsArray` object.

The hierarchy of a bus signal is preserved in the logged signal data. The logged name of a signal in a virtual bus derives from the name of the source signal. The logged name of a signal in a nonvirtual bus derives from the applicable bus object, and can differ from the name of the source signal. See “Composite Signals” for information about those capabilities.

See Also

“Convert Logged Data to Dataset Format”, “Migrate Scripts That Use Legacy ModelDataLogs API”, `Simulink.SubsysDataLogs`, `Simulink.Timeseries`, `Simulink.TsArray`, `who`, `whos`, `unpack`

Introduced before R2006a

Simulink.SimState.ModelSimState class

Package: Simulink.SimState

Access SimState snapshot data

Description

The `Simulink.SimState.ModelSimState` class contains all of the information associated with a “snapshot” of a simulation, including the logged states, the time of the snapshot, and the start time of the simulation. To access these data for a block, use the `getBlockSimState` method or the `loggedStates` property.

Properties

description

Specify a description. By default, Simulink generates a character vector based on your model name.

loggedStates

The logged states are the continuous and discrete states of the blocks in a model. These states represent a subset of the complete simulation state (`SimState`) of the model.

If `loggedStates` is in `Dataset` format, you cannot assign a structure or a `Simulink.SimulationData.Dataset` object with a different number of elements than that of the `Dataset` object used for `loggedStates`.

If the `loggedStates` is in `Structure` format, you cannot assign a `Dataset` object.

Attributes:

dependent

`loggedStates` is obtained from the saved states of the block. `loggedStates` depends on the full state being saved in the `SimState` object, unlike, properties like `description`, which are independent of the save states.

snapshotTime

Time at which Simulink takes a “snapshot” of the complete simulation states. This data is read only.

startTime

Time at which the simulation starts. This data is read only.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

Simulink.ModelWorkspace

Describe model workspace

Description

Instances of this class describe model workspaces. Simulink software creates an instance of this class for each model that you open during a Simulink session. See “Model Workspaces” for more information.

Property Summary

Name	Access	Description
DataSource	RW	Specifies the source used to initialize this workspace. Valid values are <ul style="list-style-type: none"> • 'Model File' • 'MAT-File' • 'MATLAB Code' • 'MATLAB File'
FileName	RW	Specifies the name of the MAT-file used to initialize this workspace. Simulink software ignores this property if DataSource is neither 'MAT-File' nor 'MATLAB Code'.
MATLABCode	RW	A character vector specifying MATLAB code used to initialize this workspace. Simulink software ignores this property if DataSource is not 'MATLAB Code'.

Method Summary

Name	Description
“assignin” on page 5-328	Assign a value to a variable in the model's workspace.

Name	Description
“clear” on page 5-329	Clear the model's workspace.
“evalin” on page 5-329	Evaluate an expression in the model's workspace.
“reload” on page 5-330	Reload the model workspace from the workspace's data source.
“save” on page 5-331	Save the model's workspace to a specified MAT-file.
“saveToSource” on page 5-332	Save the workspace to the MAT-file that the workspace designates as its data source.
“whos” on page 5-332	List the variables in the model workspace.
“getVariable” on page 5-329	Get value of variable from workspace.
“hasVariable” on page 5-330	Determine if variable exists in workspace.

Methods

assignin

Assign a value to a variable in the model's workspace.

```
assignin('varname', varvalue)
```

varname

 Name of the variable to be assigned a value.

varvalue

 Value to be assigned the variable.

This method assigns the value specified by `varvalue` to the variable whose name is `varname`.

“evalin” on page 5-329

clear

Clear the model's workspace.

```
clear
```

This method empties the workspace of its variables.

evalin

Evaluate an expression in the model's workspace.

```
evalin('expression')
```

```
expression
```

A MATLAB expression to be evaluated.

This method evaluates expression in the model workspace.

“assignin” on page 5-328

getVariable

Get value of variable from workspace.

```
variableValue = getVariable(workspaceHandle,variableName)
```

```
workspaceHandle
```

Handle to the workspace.

```
variableName
```

Name of the variable.

This method gets the value of a variable from a workspace.

If the value of the target variable is a handle to a handle object (such as `Simulink.Parameter`), `getVariable` returns a copy of the handle. Changes you make to the variable in the model workspace or to the returned variable (`variableValue`) affect both variables.

To return a deep copy of the handle object, use the `copy` method of the object. To modify a handle object that you store in a model workspace, it is a best practice to use both the `getVariable` and `assignin` methods (see “Modify Property Value of Handle Object” on page 2-337).

hasVariable

Determine if variable exists in workspace.

```
variableExists = hasVariable(workspaceHandle, variableName)
```

`workspaceHandle`

Handle to the workspace.

`variableName`

Name of the variable.

This method determines whether a variable exists in a workspace.

reload

Reload the model workspace from the workspace's data source.

```
reload
```

This method reloads the model workspace from the data source specified by its `DataSource` parameter. The data source must be 'MAT-File', 'MATLAB Code', or 'MATLAB File'.

“saveToSource” on page 5-332

save

Save the model's workspace to a specified MAT-file.

```
save('filename')
```

filename

Name of a MAT-file.

This method saves the model's workspace to the MAT-file specified by `filename`.

Note This method allows you to save the workspace to a file other than the file specified by the workspace's `FileName` property. If you want to save the model workspace to the file specified by the file's `FileName` property, it is simpler to use the workspace's `saveToSource` method.

```
hws = get_param('myModel','modelworkspace')
hws.DataSource = 'MAT-File';
hws.FileName = 'workspace';
hws.assignin('roll', 30);
hws.saveToSource;
hws.assignin('roll', 40);
hws.save('workspace_test.mat');
```

“reload” on page 5-330, “saveToSource” on page 5-332

saveToSource

Save the workspace to the MAT-file that it designates as its data source.

```
saveToSource
```

This method saves the model workspace designated by its `FileName` property.

```
hws = get_param('mymodel','modelworkspace')
hws.DataSource = 'MAT-File';
hws.FileName = 'params';
hws.assignin('roll', 30);
hws.saveToSource;
```

“save” on page 5-331, “reload” on page 5-330

whos

List the variables in the model workspace.

```
whos
```

This method lists the variables in the model's workspace. The listing includes the size and class of the variables.

```
>> hws = get_param('mymodel','modelworkspace');
>> hws.assignin('k', 2);
>> hws.whos
```

Name	Size	Bytes	Class
k	1x1	8	double array

See Also

Topics

“Model Workspaces”

“Variables”

Introduced before R2006a

Simulink.MSFcnRunTimeBlock

Get run-time information about Level-2 MATLAB S-function block

Description

This class allows a Level-2 MATLAB S-function or other MATLAB program to obtain information from Simulink software and provide information to Simulink software about a Level-2 MATLAB S-Function block. Simulink software creates an instance of this class for each Level-2 MATLAB S-Function block in a model. Simulink software passes the object to the callback methods of Level-2 MATLAB S-functions when it updates or simulates a model, allowing the callback methods to get and provide block-related information to Simulink software. See “Write Level-2 MATLAB S-Functions” for more information.

You can also use instances of this class in MATLAB programs to obtain information about Level-2 MATLAB S-Function blocks during a simulation. See “Access Block Data During Simulation” for more information.

The Level-2 MATLAB S-function template `matlabroot/toolbox/simulink/blocks/msfuntmpl.m` shows how to use a number of the following methods.

Parent Class

`Simulink.RunTimeBlock`

Derived Classes

None

Property Summary

Name	Description
“AllowSignalsWithMoreThan2D” on page 5-337	enable Level-2 MATLAB S-function to use multidimensional signals.
“DialogPrmsTunable” on page 5-337	Specifies which of the S-function's dialog parameters are tunable.
“NextTimeHit” on page 5-337	Time of the next sample hit for variable sample time S-functions.

Method Summary

Name	Description
“AutoRegRuntimePrms” on page 5-338	Register this block's dialog parameters as run-time parameters.
“AutoUpdateRuntimePrms” on page 5-338	Update this block's run-time parameters.
“IsDoingConstantOutput” on page 5-338	Determine whether the current simulation stage is the constant sample time stage.
“IsMajorTimeStep” on page 5-339	Determine whether the current simulation time step is a major time step.
“IsSampleHit” on page 5-339	Determine whether the current simulation time is one at which a task handled by this block is active.
“IsSpecialSampleHit” on page 5-340	Determine whether the current simulation time is one at which multiple tasks handled by this block are active.
“RegBlockMethod” on page 5-341	Register a callback method for this block.

Name	Description
"RegisterDataTypeFxpBinaryPoint" on page 5-341	Register fixed-point data type with binary point-only scaling.
"RegisterDataTypeFxpFSlopeFixexpBias" on page 5-342	Register fixed-point data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias.
"RegisterDataTypeFxpSlopeBias" on page 5-344	Register data type with [Slope Bias] scaling.
"SetAccelRunOnTLC" on page 5-345	Specify whether to use this block's TLC file to generate the simulation target for the model that uses it.
"SetPreCompInpPortInfoToDynamic" on page 5-345	Set precompiled attributes of this block's input ports to be inherited.
"SetPreCompOutPortInfoToDynamic" on page 5-346	Set precompiled attributes of this block's output ports to be inherited.
"SetPreCompPortInfoToDefaults" on page 5-346	Set precompiled attributes of this block's ports to the default values.
"SetSimViewingDevice" on page 5-346	Specify whether block is a viewer.
"SupportsMultipleExecInstances" on page 5-347	
"WriteRTWParam" on page 5-347	Write custom parameter information to Simulink Coder file.

Properties

AllowSignalsWithMoreThan2D

Allow Level-2 MATLAB S-functions to use multidimensional signals. You must set the `AllowSignalsWithMoreThan2D` property in the setup method.

Boolean

RW

DialogPrmsTunable

Specifies whether a dialog parameter of the S-function is tunable. Tunable parameters are registered as run-time parameters when you call the “`AutoRegRuntimePrms`” on page 5-338 method. Note that `SimOnlyTunable` parameters are not registered as run-time parameters. For example, the following lines initializes three dialog parameters where the first is tunable, the second in not tunable, and the third is tunable only during simulation.

```
block.NumDialogPrms      = 3;  
block.DialogPrmsTunable = {'Tunable', 'Nontunable', 'SimOnlyTunable'};
```

array

RW

NextTimeHit

Time of the next sample hit for variable sample-time S-functions.

double

RW

Methods

AutoRegRuntimePrms

Register a block's tunable dialog parameters as run-time parameters.

```
AutoRegRuntimePrms;
```

Use in the `PostPropagationSetup` method to register this block's tunable dialog parameters as run-time parameters.

AutoUpdateRuntimePrms

Update a block's run-time parameters.

```
AutoUpdateRuntimePrms;
```

Automatically update the values of the run-time parameters during a call to `ProcessParameters`.

See the S-function `matlabroot/toolbox/simulink/simdemos/simfeatures/adapt_lms.m` in the Simulink model `sldemo_msfcn_lms` for an example.

IsDoingConstantOutput

Determine whether this is in the constant sample time stage of a simulation.

```
bVal = IsDoingConstantOutput;
```

Returns true if this is the constant sample time stage of a simulation, i.e., the stage at the beginning of a simulation where Simulink software computes the values of block outputs that cannot change during the simulation (see “Constant Sample Time”). Use this method in the `Outputs` method of an S-function with port-based sample times to avoid unnecessarily computing the outputs of ports that have constant sample time, i.e., `[inf, 0]`.

```
function Outputs(block)
.
.
    if block.IsDoingConstantOutput
        ts = block.OutputPort(1).SampleTime;
        if ts(1) == Inf
            %% Compute port's output.
            end
        end
    end
.
.
%% end of Outputs
```

See “Specifying Port-Based Sample Times” for more information.

IsMajorTimeStep

Determine whether current time step is a major or a minor time step.

```
bVal = IsMajorTimeStep;
```

Returns true if the current time step is a major time step; false, if it is a minor time step. This method can be called only from the `Outputs` or `Update` methods.

IsSampleHit

Determine whether the current simulation time is one at which a task handled by this block is active.

```
bVal = IsSampleHit(stIdx);
```

`stIdx`

Global index of the sample time to be queried.

Use in `Outputs` or `Update` block methods when the MATLAB S-function has multiple sample times to determine whether a sample hit has occurred at `stIdx`. The sample time index `stIdx` is a global index for the Simulink model. For example, consider a model that contains three sample rates of 0.1, 0.2, and 0.5, and a MATLAB S-function block that contains two rates of 0.2 and 0.5. In the MATLAB S-function, `block.IsSampleHit(0)` returns true for the rate 0.1, not the rate 0.2.

This block method is similar to `ssIsSampleHit` for C-MeX S-functions, however `ssIsSampleHit` returns values based on only the sample times contained in the S-function. For example, if the model described above contained a C-MeX S-function with sample rates of 0.2 and 0.5, `ssIsSampleHit(S, 0, tid)` returns true for the rate of 0.2.

Use port-based sample times to avoid using the global sample time index for multi-rate systems (see `Simulink.BlockPortData`).

IsSpecialSampleHit

Determine whether the current simulation time is one at which multiple tasks implemented by this block are active.

```
bVal = IsSpecialSampleHit(stIdx1, stIdx1);
```

`stIdx1`

Index of sample time of first task to be queried.

`stIdx2`

Index of sample time of second task to be queried.

Use in `Outputs` or `Update` block methods to ensure the validity of data shared by multiple tasks running at different rates. Returns true if a sample hit has occurred at `stIdx1` and a sample hit has also occurred at `stIdx2` in the same time step (similar to `ssIsSpecialSampleHit` for C-Mex S-functions).

When using the `IsSpecialSampleHit` macro, the slower sample time must be an integer multiple of the faster sample time.

RegBlockMethod

Register a block callback method.

```
RegBlockMethod(methName, methHandle);
```

`methName`

Name of method to be registered.

`methHandle`

MATLAB function handle of the callback method to be registered.

Registers the block callback method specified by `methName` and `methHandle`. Use this method in the `setup` function of a Level-2 MATLAB S-function to specify the block callback methods that the S-function implements.

RegisterDataTypeFxpBinaryPoint

Register fixed-point data type with binary point-only scaling.

```
dtID = RegisterDataTypeFxpBinaryPoint(isSigned, wordLength,  
fractionalLength, obeyDataTypeOverride);
```

`isSigned`

true if the data type is signed.

false if the data type is unsigned.

`wordLength`

Total number of bits in the data type, including any sign bit.

`fractionalLength`

Number of bits in the data type to the right of the binary point.

`obeyDataTypeOverride`

`true` indicates that the **Data Type Override** setting for the subsystem is to be obeyed. Depending on the value of **Data Type Override**, the resulting data type could be `Double`, `Single`, `ScaledDouble`, or the fixed-point data type specified by the other arguments of the function.

`false` indicates that the **Data Type Override** setting is to be ignored.

This method registers a fixed-point data type with Simulink software and returns a data type ID. The data type ID can be used to specify the data types of input and output ports, run-time parameters, and DWork states. It can also be used with all the standard data type access methods defined for instances of this class, such as “DatatypeSize” on page 5-376.

Use this function if you want to register a fixed-point data type with binary point-only scaling. Alternatively, you can use one of the other fixed-point registration functions:

- Use “RegisterDataTypeFxpFSlopeFixexpBias” on page 5-342 to register a data type with [Slope Bias] scaling by specifying the word length, fractional slope, fixed exponent, and bias.
- Use “RegisterDataTypeFxpSlopeBias” on page 5-344 to register a data type with [Slope Bias] scaling.

If the registered data type is not one of the Simulink built-in data types, a Fixed-Point Designer license is checked out.

RegisterDataTypeFxpFSlopeFixexpBias

Register fixed-point data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias

```
dtID = RegisterDataTypeFxpFSlopeFixexpBias(isSigned, wordLength,  
fractionalSlope, fixedexponent, bias, obeyDataTypeOverride);
```

isSigned

true if the data type is signed.

false if the data type is unsigned.

wordLength

Total number of bits in the data type, including any sign bit.

fractionalSlope

Fractional slope of the data type.

fixedexponent

exponent of the slope of the data type.

bias

Bias of the scaling of the data type.

obeyDataTypeOverride

true indicates that the **Data Type Override** setting for the subsystem is to be obeyed. Depending on the value of **Data Type Override**, the resulting data type could be `True Doubles`, `True Singles`, `ScaledDouble`, or the fixed-point data type specified by the other arguments of the function.

false indicates that the **Data Type Override** setting is to be ignored.

This method registers a fixed-point data type with Simulink software and returns a data type ID. The data type ID can be used to specify the data types of input and output ports, run-time parameters, and DWork states. It can also be used with all the standard data type access methods defined for instances of this class, such as “DatatypeSize” on page 5-376.

Use this function if you want to register a fixed-point data type by specifying the word length, fractional slope, fixed exponent, and bias. Alternatively, you can use one of the other fixed-point registration functions:

- Use “RegisterDataTypeFxpBinaryPoint” on page 5-341 to register a data type with binary point-only scaling.
- Use “RegisterDataTypeFxpSlopeBias” on page 5-344 to register a data type with [Slope Bias] scaling.

If the registered data type is not one of the Simulink built-in data types, a Fixed-Point Designer license is checked out.

RegisterDataTypeFxpSlopeBias

Register data type with [Slope Bias] scaling.

```
dtID = RegisterDataTypeFxpSlopeBias(isSigned, wordLength,  
totalSlope, bias, obeyDataTypeOverride);
```

`isSigned`

`true` if the data type is signed.

`false` if the data type is unsigned.

`wordLength`

Total number of bits in the data type, including any sign bit.

`totalSlope`

Total slope of the scaling of the data type.

`bias`

Bias of the scaling of the data type.

`obeyDataTypeOverride`

`true` indicates that the **Data Type Override** setting for the subsystem is to be obeyed. Depending on the value of **Data Type Override**, the resulting data type could be `True Doubles`, `True Singles`, `ScaledDouble`, or the fixed-point data type specified by the other arguments of the function.

`false` indicates that the **Data Type Override** setting is to be ignored.

This method registers a fixed-point data type with Simulink software and returns a data type ID. The data type ID can be used to specify the data types of input and output ports, run-time parameters, and DWork states. It can also be used with all the standard data type access methods defined for instances of this class, such as “DatatypeSize” on page 5-376.

Use this function if you want to register a fixed-point data type with [Slope Bias] scaling. Alternatively, you can use one of the other fixed-point registration functions:

- Use “RegisterDataTypeFxpBinaryPoint” on page 5-341 to register a data type with binary point-only scaling.
- Use “RegisterDataTypeFxpFSlopeFixexpBias” on page 5-342 to register a data type by specifying the word length, fractional slope, fixed exponent, and bias

If the registered data type is not one of the Simulink built-in data types, a Fixed-Point Designer license is checked out.

SetAccelRunOnTLC

Specify whether to use block's TLC file to generate code for the Accelerator mode of Simulink software.

```
SetAccelRunOnTLC (bVal) ;
```

bVal

May be 'true' (use TLC file) or 'false' (run block in interpreted mode).

Specify if the block should use its TLC file to generate code that runs with the accelerator. If this option is 'false', the block runs in interpreted mode. See the S-function *matlabroot/toolbox/simulink/blocks/msfcn_times_two.m* in the Simulink model *msfcdemo_timestwo* for an example.

SetPreCompInpPortInfoToDynamic

Set precompiled attributes of this block's input ports to be inherited.

```
SetPreCompInpPortInfoToDynamic;
```

Initialize the compiled information (dimensions, data type, complexity, and sampling mode) of this block's input ports to be inherited. See the S-function *matlabroot/*

`toolbox/simulink/simdemos/simfeatures/adapt_lms.m` in the Simulink model `sldemo_msfcn_lms` for an example.

SetPreCompOutPortInfoToDynamic

Set precompiled attributes of this block's output ports to be inherited.

```
SetPreCompOutPortInfoToDynamic;
```

Initialize the compiled information (dimensions, data type, complexity, and sampling mode) of the block's output ports to be inherited. See the S-function `matlabroot/toolbox/simulink/simdemos/simfeatures/adapt_lms.m` in the Simulink model `sldemo_msfcn_lms` for an example.

SetPreCompPortInfoToDefaults

Set precompiled attributes of this block's ports to the default values.

```
SetPreCompPortInfoToDefaults;
```

Initialize the compiled information (dimensions, data type, complexity, and sampling mode) of the block's ports to the default values. By default, a port accepts a real scalar sampled signal with a data type of `double`.

SetSimViewingDevice

Specify whether this block is a viewer.

```
SetSimViewingDevice(bVal);
```

`bVal`

May be `'true'` (is a viewer) or `'false'` (is not a viewer).

Specify if the block is a viewer/scope. If this flag is specified, the block will be used only during simulation and automatically stubbed out in generated code.

SupportsMultipleExecInstances

Specify whether or not a For Each Subsystem supports an S-function inside of it.

```
SupportsMultipleExecInstances (bVal) ;
```

bVal

May be 'true' (S-function is supported) or 'false' (S-function is not supported).

Specify if an S-function can operate within a For Each Subsystem.

WriteRTWParam

Write a custom parameter to the Simulink Coder information file used for code generation.

```
WriteRTWParam(pType, pName, pVal)
```

pType

Type of the parameter to be written. Valid values are 'string' and 'matrix'.

pName

Name of the parameter to be written.

pVal

Value of the parameter to be written.

Use in the `writeRTW` method of the MATLAB S-function to write out custom parameters. These parameters are generally settings used to determine how code should be generated in the TLC file for the S-function. See the S-function `matlabroot/toolbox/simulink/`

`simdemos/simfeatures/adapt_lms.m` in the Simulink model `sldemo_msfcn_lms` for an example.

Introduced before R2006a

Simulink.NumericType

Specify floating point, integer, or fixed point data type

Description

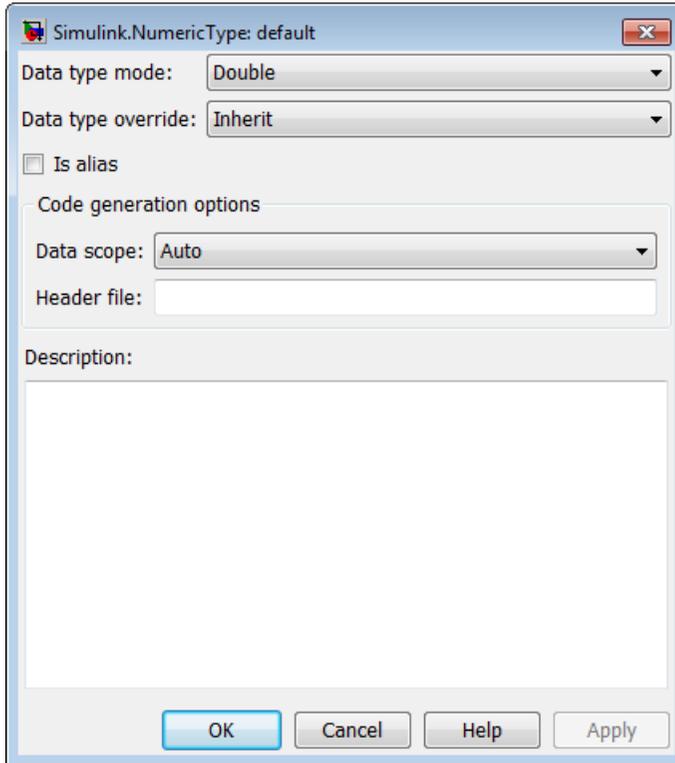
This class allows you to specify a numeric data type as follows:

- 1 Create an instance of this class in the MATLAB base workspace, a model workspace, or a data dictionary. To create a numeric type in a model workspace, you must disable the **Is alias** option.
- 2 Set the properties of the object to create a custom floating point, integer, or fixed point data type.
- 3 Assign the data type to all signals and parameters of your model that you want to conform to the data type.

Assigning a data type in this way allows you to change the data types of the signals and parameters in your model by changing the properties of the object that describe them. You do not have to change the model itself.

To rename a data type in a model and in the code that you generate from a model (typedef), you can use an object of the class `Simulink.AliasType`.

Property Dialog Box



Data type mode

Data type of this numeric type. The options are listed in this table.

Option	Description
Double	Same as the MATLAB double type.
Single	Same as the MATLAB single type.
Boolean	Same as the MATLAB boolean type.
Fixed-point: unspecified scaling	A fixed-point data type with unspecified scaling.
Fixed-point: binary point scaling	A fixed-point data type with binary-point scaling.

Option	Description
Fixed-point: slope and bias scaling	A fixed-point data type with slope and bias scaling.

Selecting a data type mode causes Simulink software to enable controls on the dialog box that apply to the mode and to disable other controls that do not apply. Selecting a fixed-point data type mode can, depending on the other dialog box options that you select, cause the model to run only on systems that have a Fixed-Point Designer option installed.

Data type override

Data type override setting for this numeric type. The options are listed in this table.

Option	Description
Inherit (default)	Data type override setting for the context in which this numeric type is used (block, signal, Stateflow chart in Simulink) applies to this numeric type.
Off	Data type override setting does not affect this numeric type.

Is alias

If you select this option for a workspace object of this type, Simulink software uses the name of the object as the data type for all objects that specify the object as its data type. Otherwise, Simulink software uses the data type mode of the data type as its name, or, if the data type mode is a fixed-point mode, Simulink software generates a name that encodes the type properties, using the encoding specified by Fixed-Point Designer.

Data scope

Specifies whether the data type definition is imported from, or exported to, a header file during code generation. The possible values are listed in this table.

Value	Action
Auto (default)	If no value is specified for Header file , export the type definition to <code>model_types.h</code> . <code>model</code> is the model name. If a value is specified for Header file , import the data type definition from the specified header file.

Value	Action
Exported	Export the data type definition to a header file, which can be specified in the Header file field. If no value is specified for Header file , the header file name defaults to <i>type.h</i> . <i>type</i> is the data type name.
Imported	Import the data type definition from a header file, which can be specified in the Header file field. If no value is specified for Header file , the header file name defaults to <i>type.h</i> . <i>type</i> is the data type name.

Header file

Name of a C header file from which a data type definition is imported, or to which a data type definition is exported, based on the value of **Data scope**. If this field is specified, the specified name is used during code generation for importing or exporting. If this field is empty, the value defaults to *type.h* if **Data scope** equals Imported or Exported, or defaults to *model_types.h* if **Data scope** equals Auto.

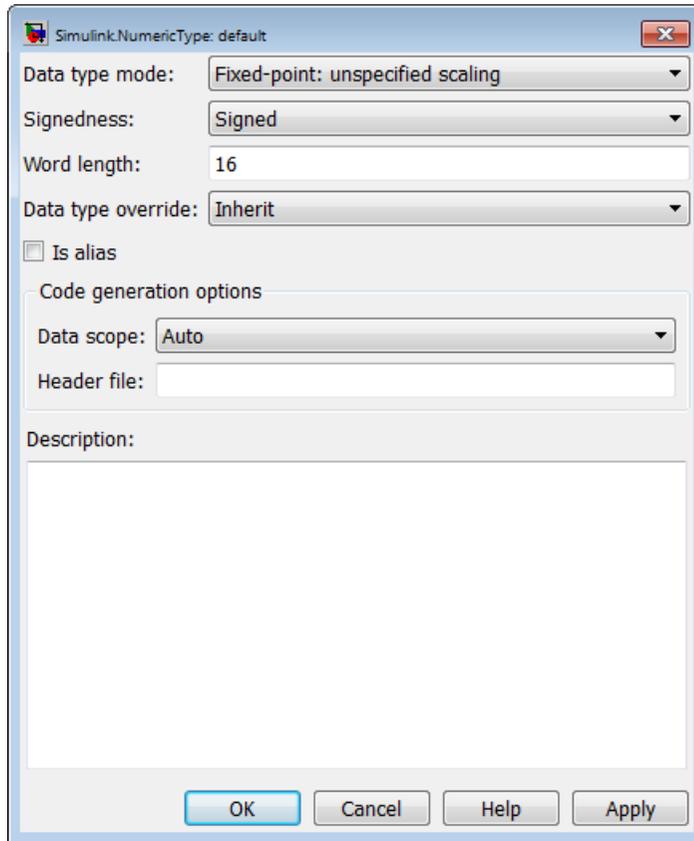
By default, the generated `#include` directive uses the preprocessor delimiter `"` instead of `<` and `>`. To generate the directive `#include <myTypes.h>`, specify **Header file** as `<myTypes.h>`.

Description

Description of this data type. This field is intended for use in documenting this data type. Simulink software ignores it.

Signedness

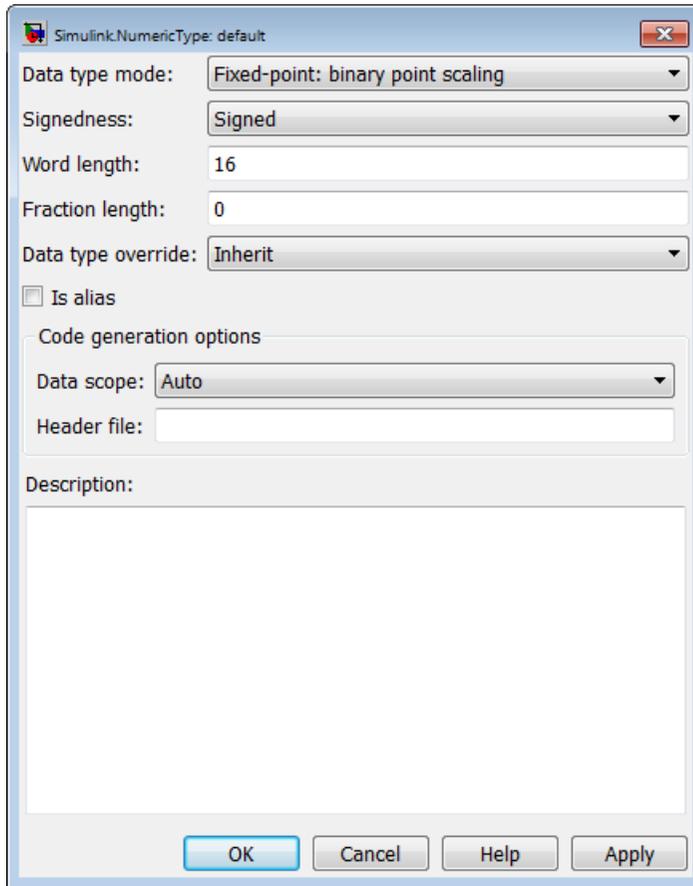
Specifies whether the data type is signed or unsigned, or inherits its signedness. Set the option to Signed, Unsigned, or Auto. This option is enabled only for fixed-point data type modes as shown.

**Word length**

Word length in bits of the fixed-point data type. This option is enabled only for fixed-point data type modes.

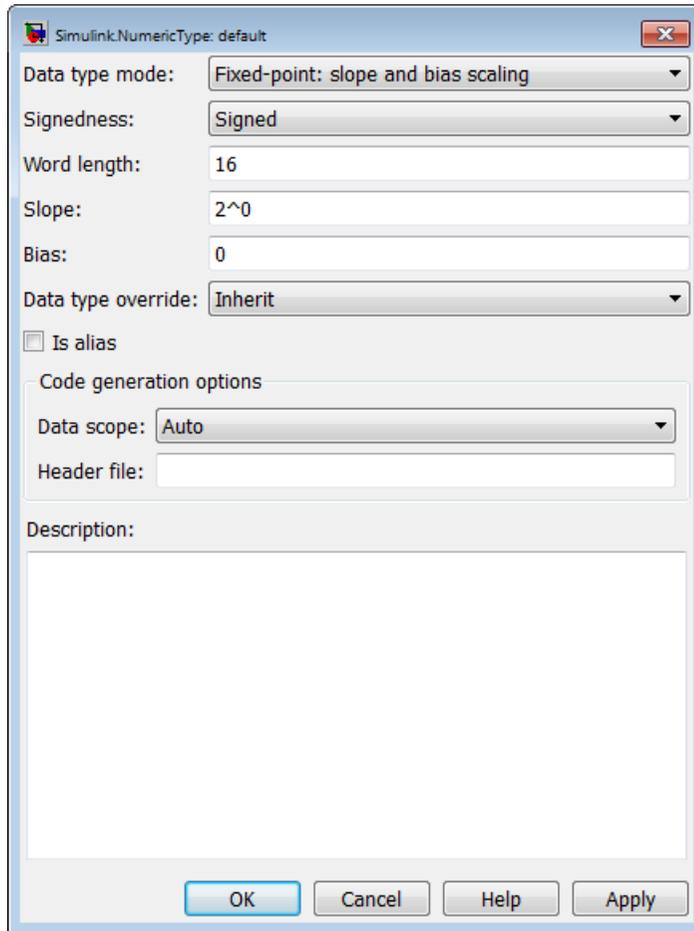
Fraction length

Number of bits to the right of the binary point. This option is enabled only if the data type mode is `Fixed-point: binary point scaling`.



Slope

Slope for slope and bias scaling. This option is enabled only if the data type mode is Fixed-point: slope and bias scaling.



Bias

Bias for slope and bias scaling. This option is enabled only if the data type mode is `Fixed-point: slope and bias scaling`. See the preceding figure.

Properties

Name	Access	Description
Bias	RW	Bias used for slope and bias scaling of a fixed-point data type. This field is intended for use by Fixed-Point Designer. (Bias)
DataScope	RW	A character vector specifying whether the data type definition is imported from, or exported to, a header file during code generation. (Data scope)
DataTypeMode	RW	Character vector that specifies the data type mode of this numeric type. Valid values are 'Double', 'Boolean', 'Single', 'Fixed-point: unspecified scaling', 'Fixed-point: binary point scaling', and 'Fixed-point: slope and bias scaling'. (Data type mode)
DataTypeOverride	RW	Character vector that specifies the data type override mode. Valid values are Inherit and Off. (Data type override)
Description	RW	Description of this data type. (Description)
Fixedexponent	RW	Exponent used for binary point scaling. This property equals $-\text{FractionLength}$. Setting this property causes Simulink software to set the FractionLength and Slope properties accordingly, and vice versa. This property applies only if the DataTypeMode is Fixed-point: binary point scaling or Fixed-point: slope and bias scaling. It does not appear in the object Property dialog box, but can be accessed at the command prompt.

Name	Access	Description
FractionLength	RW	Integer that specifies the size in bits of the fractional portion of the fixed-point number. This property equals <code>-Fixedexponent</code> . Setting this property causes Simulink software to set the <code>Fixedexponent</code> property accordingly, and vice versa. This field is intended for use by Fixed-Point Designer. (Fraction length)
HeaderFile	RW	A character vector that specifies the name of a C header file from which a data type definition is imported, or to which a data type definition is exported, during code generation. (Header file)
IsAlias	RW	Integer that specifies whether to use the name of this object as the name of the data type that it specifies. Valid values are 1 (yes) or 0 (no). (Is alias) The generated code uses the alias only if you use an ERT-based system target file (Embedded Coder).
Signedness	RW	Boolean that specifies whether this data type is signed, unsigned, or inherits its signedness. Valid values are 1 (signed), 0 (unsigned), or <code>Auto</code> (inherit signedness). (Signedness)

Name	Access	Description
Slope	RW	Slope for slope and bias scaling of fixed-point numbers. This property equals $\text{SlopeAdjustmentFactor} * 2^{\text{Fixedexponent}}$. If $\text{SlopeAdjustmentFactor}$ is 1.0, Simulink software displays the value of this field as $2^{\text{SlopeAdjustmentFactor}}$. Otherwise, it displays it as a numeric value. Setting this property causes Simulink software to set the Fixedexponent and $\text{SlopeAdjustmentFactor}$ properties accordingly, and vice versa. This property appears only if DataTypeMode is Fixed-point: slope and bias scaling. (Slope)
SlopeAdjustmentFactor	RW	Slope for slope and bias scaling of fixed-point numbers. Setting this property causes Simulink software to adjust the Slope property accordingly, and vice versa. This property applies only if DataTypeMode is Fixed-point: slope and bias scaling. It does not appear in the object Property dialog box, but can be accessed at the command prompt.
WordLength	RW	Integer that specifies the word size of this data type. This field is intended for use by Fixed-Point Designer. This property appears only if DataTypeMode is Fixed-point. (Word length)

Methods

Name	Description
isboolean	Determine whether data type is Boolean. Returns 1 when the DataTypeMode is 'Boolean', 0 otherwise.

Name	Description
isdouble	<p>Determine whether data type is double precision.</p> <p>Returns 1 when the <code>DataTypeMode</code> is 'Double', 0 otherwise.</p>
isfixed	<p>Determine whether data type is fixed point.</p> <p>Returns 1 when the <code>DataTypeMode</code> is any of the fixed-point options, 0 otherwise. The fixed-point options are:</p> <ul style="list-style-type: none"> • 'Fixed-point: unspecified scaling' • 'Fixed-point: binary point scaling' • 'Fixed-point: slope and bias scaling'
isfloat	<p>Determine whether data type is floating point.</p> <p>Returns 1 when the <code>DataTypeMode</code> is 'Double' or 'Single', 0 otherwise.</p>
isscalingbinarypoint	<p>Determine whether data type has binary point scaling.</p> <p>Returns 1 when the data type has binary point scaling or trivial slope and bias scaling, 0 otherwise. Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero.</p>
isscalingslopebias	<p>Determine whether data type has nontrivial slope and bias scaling.</p> <p>Returns 1 when the data type has nontrivial slope and bias scaling, 0 otherwise. Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero.</p>

Name	Description
isscalingunspecified	<p>Determine whether data type has unspecified scaling.</p> <p>Returns 1 when the data type is fixed point and its scaling has not been specified, 0 otherwise.</p> <p>DataTypeMode is 'Fixed-point: unspecified scaling'</p>
issingle	<p>Determine whether data type is single-precision.</p> <p>Returns 1 when the DataTypeMode is 'Single', 0 otherwise.</p>

See Also

`Simulink.AliasType`

Topics

“Validate a Floating-Point Embedded Model”

“Control Signal Data Types”

“Control Data Type Names in Generated Code” (Embedded Coder)

“Data Types Supported by Simulink”

“About Data Types in Simulink”

Introduced before R2006a

Simulink.Parameter

Store, share, and configure block parameter values

Description

Create a `Simulink.Parameter` object to set the value of one or more block parameters in a model, such as the **Gain** parameter of a Gain block. You create the object in a workspace or in a data dictionary. Set the parameter value in the object, not in the block.

Use a `Simulink.Parameter` object to:

- Share a value among multiple block parameters.
- Represent an engineering constant or a tunable calibration parameter.
- Separate a parameter value from its data type.
- Configure parameter data for code generation.

The `Value` property of the object stores the parameter value. To use the object in a model, set the value of a block parameter to an expression that involves the name of the object. Omit the `Value` property from the expression. For more information, see “Use Parameter Objects”.

For more information about block parameters, see “Set Block Parameter Values” and “Block Parameter Representation in the Generated Code” (Simulink Coder).

Creation

Create a `Simulink.Parameter` object:

- Directly from a block dialog box or the Property Inspector. See “Create, Edit, and Manage Workspace Variables”.
- By using the Model Data Editor. Inspect the **Parameters** tab.
- By using the Model Explorer. See “Create Data Objects from Built-In Data Class Package Simulink”.

- By using the `Simulink.Parameter` function, described below.

Syntax

```
paramObj = Simulink.Parameter  
paramObj = Simulink.Parameter(paramValue)
```

Description

`paramObj = Simulink.Parameter` returns a `Simulink.Parameter` object with default property values.

`paramObj = Simulink.Parameter(paramValue)` returns a `Simulink.Parameter` object and initializes the value of the `Value` property by using `paramValue`.

Properties

For information about properties in the property dialog box of a `Simulink.Parameter` object, see “`Simulink.Parameter` Property Dialog Box”.

CoderInfo — Specifications for generating code for parameter object

`Simulink.CoderInfo` object

Specifications for generating code for the parameter object, returned as a `Simulink.CoderInfo` object.

This property is read only. Instead, modify the properties of the `Simulink.CoderInfo` object that this property contains.

For example, the `StorageClass` property of the `Simulink.CoderInfo` object determines how Simulink code generation toolboxes allocate memory for the parameter object in the generated code. For more information, see “Override Default Parameter Behavior by Creating Global Variables in the Generated Code” (Simulink Coder) and “Simulink Package Custom Storage Classes” (Embedded Coder).

Complexity — Numeric complexity of parameter value

'real' (default) | 'complex'

Numeric complexity of the parameter value, returned as 'real' (if the value is real) or 'complex' (if the value is complex). Simulink determines the complexity from the parameter value that you specify in the `Value` property. This property is read only.

Data Types: `char`

Data Type — Data type of parameter value

'auto' (default) | character vector

Data type of the parameter value that you specify in the `Value` property. When you simulate the model or generate code, Simulink casts the value to the specified data type.

If you specify 'auto', the default setting, the parameter object uses the same data type as the block parameters that use the object. See “Reduce Maintenance Effort with Data Type Inheritance”.

When you set the `Value` property by using something other than a double number, the object typically sets the `Data Type` property based on the value of the `Value` property. For example, when you set the `Value` property to `int8(5)`, the object sets the value of the `Data Type` property to 'int8'.

To explicitly specify a built-in data type (see “Data Types Supported by Simulink”), specify one of these options:

- 'double'
- 'single'
- 'int8'
- 'uint8'
- 'int16'
- 'uint16'
- 'int32'
- 'uint32'
- 'boolean'

To specify a fixed-point data type, use the `fixdt` function. For example, specify `'fixdt(1,16,5)'`.

If you use a `Simulink.AliasType` or `Simulink.NumericType` object to create and share custom data types in your model, specify the name of the object.

To specify an enumerated data type, use the name of the type preceded by `Enum:`. For example, specify `'Enum: myEnumType'`.

When you store a structure or array of structures in the `Value` property of the object, the object sets the `DataType` property to `'struct'`. To specify a `Simulink.Bus` object as the data type, use the name of the bus object preceded by `Bus:`. For example, specify `'Bus: myBusObject'`.

Example: `'auto'`

Example: `'int8'`

Example: `'fixdt(1,16,5)'`

Example: `'myAliasTypeObject'`

Example: `'Enum: myEnumType'`

Example: `'Bus: myBusObject'`

Data Types: `char`

Description — Custom description of parameter object

`''` (empty character vector) (default) | character vector

Custom description of the parameter object, specified as a character vector. Use this property to document the significance that the parameter object has in your algorithm.

If you have Embedded Coder, you can configure this description to appear in the generated code as a comment. See “Simulink data object descriptions” (Simulink Coder).

Example: `'This parameter represents the maximum rotation speed of the engine.'`

Data Types: `char`

Dimensions — Dimensions of parameter value

`[0 0]` (default) | row vector | character vector

Dimensions of the value stored in the `Value` property, returned as a row vector or specified as a character vector.

When you set the `Value` property of the object, the object sets the value of the `Dimensions` property to a double row vector. The vector is the same vector that the `size` function returns.

To use symbolic dimensions, specify a character vector. See “Implement Dimension Variants for Array Sizes in Generated Code” (Embedded Coder).

Example: `[1 3]`

Example: `'[1 myDimParam]'`

Data Types: `double` | `char`

Max — Maximum value of parameter

`[]` (empty) (default) | real `double` scalar

Maximum value that the `Value` property of the object can store, specified as a real `double` scalar.

The default value is `[]` (empty), which means the parameter value does not have a maximum.

If you store a complex number in the `Value` property, the `Max` property applies separately to the real and imaginary parts.

If you store a structure in the `Value` property, the object ignores the `Max` property. Instead, use a `Simulink.Bus` object as the data type of the parameter object, and specify a maximum value for each field by using the elements of the bus object. See “Control Field Data Types and Characteristics by Creating Parameter Object”.

If the parameter value is greater than the maximum value or if the maximum value is outside the range of the object data type, Simulink generates a warning. When updating the diagram or starting a simulation, Simulink generates an error.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”.

Example: `5.32`

Data Types: `double`

Min — Minimum value of parameter

`[]` (empty) (default) | real `double` scalar

Minimum value that the `Value` property of the object can store, specified as a real `double` scalar.

The default value is `[]` (empty), which means the parameter value does not have a minimum.

If you store a complex number in the `Value` property, the `Min` property applies separately to the real and imaginary parts.

If you store a structure in the `Value` property, the object ignores the `Min` property. Instead, use a `Simulink.Bus` object as the data type of the parameter object, and specify a minimum value for each field by using the elements of the bus object. See “Control Field Data Types and Characteristics by Creating Parameter Object”.

If the parameter value is less than the minimum value or if the minimum value is outside the range of the object data type, Simulink generates a warning. When updating the diagram or starting a simulation, Simulink generates an error.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”

Example: `-0.92`

Data Types: `double`

unit — Physical unit of parameter value

`empty (default) | valid unit`

Physical unit of parameter value, specified as a character vector. For more information, see “Unit Specification in Simulink Models”.

Example: `'degC'`

Data Types: `char`

value — Value to use in target block parameters

`[]` (default) | valid value

Value to use in target block parameters, specified as any of these valid values:

- Numeric value
- Boolean value
- Instance of enumerated type
- Structure

- Scalar or array

You can use MATLAB expressions to specify the value.

Example Expression	Description
<code>15.23</code>	Specifies a scalar value
<code>[3 4; 9 8]</code>	Specifies a matrix
<code>3+2i</code>	Specifies a complex value
<code>struct('A',20,'B',5)</code>	Specifies a structure with two fields, A and B, with double-precision values 20 and 5. Organize block parameters into structures (see “Organize Related Block Parameter Definitions in Structures”) or initialize the signal elements in a bus (see “Specify Initial Conditions for Bus Signals”).

To use a `Simulink.Parameter` object to store a value of a particular numeric data type, specify the ideal value with the `Value` property, and control the type with the `DataType` property.

If you set the `Value` property by using a typed expression such as `single(32.5)`, the `DataType` property changes to reflect the new type. A best practice is using an expression that is not typed. You can avoid accumulating numerical error through repeated quantizations or data type saturation, especially for fixed-point data types.

Example: `3.15`

Example: `single([3.15 1.23])`

Example: `1.2 + 3.2i`

Example: `true`

Example: `myEnumType.myEnumValue`

Example: `struct('field1',15,'field2',7.32)`

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

Complex Number Support: Yes

Examples

Use Parameter Object to Set Value of Gain Parameter

- 1 At the command prompt, create a `Simulink.Parameter` object.

```
myParam = Simulink.Parameter;
```

- 2 Assign a numeric value to the `Value` property.

```
myParam.Value = 15.23;
```

- 3 Specify other characteristics for the block parameter by adjusting the object properties. For example, to specify the minimum and maximum values the parameter can take, use the `Min` and `Max` properties.

```
myParam.Min = 10.11;  
myParam.Max = 25.27;
```

- 4 In a block dialog box, specify the value of a parameter as `myParam`. For example, in a `Gain` block dialog box, specify **Gain** as `myParam`.

During simulation, the **Gain** parameter uses the value 15.23.

Change Value Stored by Parameter Object

- 1 At the command prompt, create a `Simulink.Parameter` object that stores the value 2.52.

```
myParam = Simulink.Parameter(2.52);
```

- 2 Change the value by accessing the `Value` property of the object. This technique preserves the values of the other properties of the object.

```
myParam.Value = 1.13;
```

Create Parameter Object with Specific Numeric Data Type

To reduce model maintenance, you can leave the `DataType` property at its default value, `auto`. The parameter object acquires a data type from the block parameter that uses the object.

To reduce the risk of the data type changing when you make changes to signal data types and other data types in your model, you can explicitly specify a data type for the parameter object. For example, when you generate code that exports parameter data to your custom code, explicitly specify a data type for the object.

- 1 At the command prompt, create a `Simulink.Parameter` object that stores the value 18.25.

```
myParam = Simulink.Parameter(18.25);
```

The expression `18.25` returns the number 18.25 with the double-precision, floating-point data type `double`. The `Value` property stores the number 18.25 with double precision.

- 2 Use the `DataType` property to specify the single-precision data type `single`.

```
myParam.DataType = 'single';
```

When you simulate or generate code, the parameter object casts the value of the `Value` property, 18.25, to the data type specified by the `DataType` property, `single`.

See Also

`AUTOSAR.Parameter` | `Simulink.CoderInfo` | `Simulink.LookupTable` | `Simulink.Signal`

Topics

“Data Objects”

“Set Block Parameter Values”

“Block Parameter Representation in the Generated Code” (Simulink Coder)

“Determine Where to Store Variables and Objects for Simulink Models”

“Data Types Supported by Simulink”

“Define Data Classes”

Introduced before R2006a

Simulink.RunTimeBlock

Allow Level-2 MATLAB S-function and other MATLAB programs to get information about block while simulation is running

Description

This class allows a Level-2 MATLAB S-function or other MATLAB program to obtain information about a block. Simulink software creates an instance of this class or a derived class for each block in a model. Simulink software passes the object to the callback methods of Level-2 MATLAB S-functions when it updates or simulates a model, allowing the callback methods to get block-related information from and provide such information to Simulink software. See “Write Level-2 MATLAB S-Functions” in Writing S-Functions for more information. You can also use instances of this class in MATLAB programs to obtain information about blocks during a simulation. See “Access Block Data During Simulation” for more information.

Note `Simulink.RunTimeBlock` objects do not support MATLAB sparse matrices. For example, the following line of code attempts to assign a sparse identity matrix to the runtime object's output port data. This line of code in a Level-2 MATLAB S-function produces an error:

```
block.Outport(1).Data = speye(10);
```

Parent Class

None

Derived Classes

`Simulink.MSFcnRuntimeBlock`

Property Summary

Name	Description
“BlockHandle” on page 5-372	Block's handle.
“CurrentTime” on page 5-372	Current simulation time.
“NumDworks” on page 5-372	Number of discrete work vectors used by the block.
“NumOutputPorts” on page 5-373	Number of block output ports.
“NumContStates” on page 5-373	Number of block's continuous states.
“NumDworkDiscStates” on page 5-373	Number of block's discrete states
“NumDialogPrms” on page 5-373	Number of parameters that can be entered on S-function block's dialog box.
“NumInputPorts” on page 5-374	Number of block's input ports.
“NumRuntimePrms” on page 5-374	Number of run-time parameters used by block.
“SampleTimes” on page 5-374	Sample times at which block produces outputs.

Method Summary

Name	Description
“ContStates” on page 5-375	Get a block's continuous states.
“DataTypeIsFixedPoint” on page 5-375	Determine whether a data type is fixed point.
“DatatypeName” on page 5-375	Get name of a data type supported by this block.
“DatatypeSize” on page 5-376	Get size of a data type supported by this block.
“Derivatives” on page 5-376	Get a block's continuous state derivatives.
“DialogPrm” on page 5-377	Get a parameter entered on an S-function block's dialog box.
“Dwork” on page 5-377	Get one of a block's DWork vectors.

Name	Description
“FixedPointNumericType” on page 5-378	Determine the properties of a fixed-point data type.
“InputPort” on page 5-378	Get one of a block's input ports.
“OutputPort” on page 5-379	Get one of a block's output ports.
“RuntimePrm” on page 5-379	Get one of the run-time parameters used by a block.

Properties

BlockHandle

Block's handle.

RO

CurrentTime

Current simulation time.

RO

NumDworks

Number of data work vectors.

RW

`ssGetNumDWork`

NumOutputPorts

Number of output ports.

RW

`ssGetNumOutputPorts`

NumContStates

Number of continuous states.

RW

`ssGetNumContStates`

NumDworkDiscStates

Number of discrete states. In a MATLAB S-function, you need to use `DWorks` to set up discrete states.

RW

`ssGetNumDiscStates`

NumDialogPrms

Number of parameters declared on the block's dialog. In the case of the S-function, it returns the number of parameters listed as a comma-separated list in the **S-function parameters** dialog field.

RW

`ssGetNumSFcnParams`

NumInputPorts

Number of input ports.

RW

`ssGetNumInputPorts`

NumRuntimePrms

Number of run-time parameters used by this block. See “Run-Time Parameters” for more information.

RW

`ssGetNumSFcnParams`

SampleTimes

Block's sample times.

RW for MATLAB S-functions, RO for all other blocks.

Methods

ContStates

Get a block's continuous states.

```
states = ContStates();
```

Get vector of continuous states.

```
ssGetContStates
```

DataTypesFixedPoint

Determine whether a data type is fixed point.

```
bVal = DataTypeIsFixedPoint(dtID);
```

dtID

Integer value specifying the ID of a data type.

Returns `true` if the specified data type is a fixed-point data type.

DatatypeName

Get the name of a data type.

```
name = DatatypeName(dtID);
```

dtID

Integer value specifying ID of a data type.

Returns the name of the data type specified by dtID.

“DatatypeSize” on page 5-376

DatatypeSize

Get the size of a data type.

```
size = DatatypeSize(dtID);
```

dtID

Integer value specifying the ID of a data type.

Returns the size of the data type specified by dtID.

“DatatypeName” on page 5-375

Derivatives

Get derivatives of a block's continuous states.

```
derivs = Derivatives();
```

Get vector of state derivatives.

```
ssGetdX
```

DialogPrm

Get an S-function's dialog parameters.

```
param = DialogPrm(pIdx);
```

pIdx

Integer value specifying the index of the parameter to be returned.

Get the specified dialog parameter. In the case of the S-function, each `DialogPrm` corresponds to one of the elements in the comma-separated list of parameters in the **S-function parameters** dialog field.

`ssGetSFcnParam`, “RuntimePrm” on page 5-379

Dwork

Get one of a block's DWork vectors.

```
dworkObj = Dwork(dwIdx);
```

dwIdx

Integer value specifying the index of a work vector.

Get information about the DWork vector specified by `dwIdx` where `dwIdx` is the index number of the work vector. This method returns an object of type `Simulink.BlockCompDworkData`.

`ssGetDWork`

FixedPointNumericType

Get the properties of a fixed-point data type.

```
eno = FixedPointNumericType(dtID);
```

dtID

Integer value specifying the ID of a fixed-point data type.

Returns an object of `embedded.Numeric` class that contains the attributes of the specified fixed-point data type.

Note `embedded.Numeric` is also the class of the `numericType` objects created by Fixed-Point Designer software. For information on the properties defined by `embedded.Numeric` class, see “`numericType` Object Properties” (Fixed-Point Designer).

InputPort

Get an input port of a block.

```
port = InputPort(pIdx);
```

pIdx

Integer value specifying the index of an input port.

Get the input port specified by `pIdx`, where `pIdx` is the index number of the input port. For example,

```
port = rto.InputPort(1)
```

returns the first input port of the block represented by the run-time object `rto`.

This method returns an object of type `Simulink.BlockPreCompInputPortData` or `Simulink.BlockCompInputPortData`, depending on whether the model that contains

the port is uncompiled or compiled. You can use this object to get and set the input port's uncompiled or compiled properties, respectively.

`ssGetInputPortSignalPtrs`, `Simulink.BlockPreCompInputPortData`,
`Simulink.BlockCompInputPortData`, “OutputPort” on page 5-379

OutputPort

Get an output port of a block.

```
port = OutputPort(pIdx);
```

`pIdx`

Integer value specifying the index of an output port.

Get the output port specified by `pIdx`, where `pIdx` is the index number of the output port. For example,

```
port = rto.OutputPort(1)
```

returns the first output port of the block represented by the run-time object `rto`.

This method returns an object of type `Simulink.BlockPreCompOutputPortData` or `Simulink.BlockCompOutputPortData`, depending on whether the model that contains the port is uncompiled or compiled, respectively. You can use this object to get and set the output port's uncompiled or compiled properties, respectively.

`ssGetInputPortSignalPtrs`, `Simulink.BlockPreCompOutputPortData`,
`Simulink.BlockCompOutputPortData`

RuntimePrm

Get an S-function's run-time parameters.

```
param = RuntimePrm(pIdx);
```

pIdx

Integer value specifying the index of a run-time parameter.

Get the run-time parameter whose index is pIdx. This run-time parameter is a Simulink.BlockData on page 5-177 object of type Simulink.BlockRunTimePrmData.

```
ssGetRunTimeParamInfo
```

Introduced before R2006a

Simulink.SampleTime class

Package: Simulink

Object containing sample time information

Description

The `SampleTime` class represents the sample time information associated with an individual sample time.

Use the methods `Simulink.Block.getSampleTimes` and `Simulink.BlockDiagram.getSampleTimes` to retrieve the values of the `SampleTime` properties for a block and for a block diagram, respectively.

Properties

Value

A two-element array of doubles that contains the period and offset of the sample time

Description

A $1 \times n$ character array that describes the sample time type

ColorRGBValue

A 1×3 array of doubles that contains the red, green and blue (RGB) values of the sample time color

Annotation

A $1 \times n$ character array that represents the annotation of a specific sample time (for example, 'D1')

OwnerBlock

For asynchronous and variable sample times, `OwnerBlock` is a character vector containing the full path to the block that controls the sample time. For all other types of sample times, it is an empty character vector.

ComponentSampleTimes

If the sample time is an async union or if the sample time is hybrid and the component sample times are available, then the array `ComponentSampleTimes` contains `Simulink.SampleTime` objects.

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#) in the [MATLAB Programming Fundamentals](#) documentation.

Examples

Retrieve the sample time information for the 'vdp' model.

```
ts = Simulink.BlockDiagram.getSampleTimes('vdp')
```

Simulink returns:

```
ts =  
  
    1x2 Simulink.SampleTime  
    Package: Simulink  
  
    Properties:  
        Value  
        Description  
        ColorRGBValue  
        Annotation  
        OwnerBlock  
        ComponentSampleTimes  
  
    Methods
```

To examine the values of the properties:

```
ts(1), ts(2)

ans =

Simulink.SampleTime
Package: Simulink

Properties:
    Value: [0 0]
    Description: 'Continuous'
    ColorRGBValue: [0 0 0]
    Annotation: 'Cont'
    OwnerBlock: []
    ComponentSampleTimes: {}

Methods

ans =

Simulink.SampleTime
Package: Simulink

Properties:
    Value: [Inf 0]
    Description: 'Constant'
    ColorRGBValue: [1 0.2600 0.8200]
    Annotation: 'Inf'
    OwnerBlock: []
    ComponentSampleTimes: {}

Methods
```

See Also

[Simulink.Block.getSampleTimes](#) | [Simulink.BlockDiagram.getSampleTimes](#)

Simulink.sdi.DatasetRef class

Package: Simulink.sdi

Access data in Simulation Data Inspector repository

Description

The `Simulink.sdi.DatasetRef` class provides access to data in the Simulation Data Inspector repository without loading the entire set of data into memory. The class is compatible with the `Simulink.SimulationData.DatasetRef` class.

Construction

`dsr_array = Simulink.sdi.DatasetRef` constructs an array containing a `Simulink.sdi.DatasetRef` object for each run in the Simulation Data Inspector.

`dsr_array = Simulink.sdi.DatasetRef(domain)` creates an array containing a `Simulink.sdi.DatasetRef` object for each run, with the contents of each run limited to the selected domain.

`dsr = Simulink.sdi.DatasetRef(runID)` creates a `Simulink.sdi.DatasetRef` object of the run corresponding to the run identifier, `runID`.

`dsr = Simulink.sdi.DatasetRef(runID, domain)` creates a `Simulink.sdi.DatasetRef` object of the run corresponding to `runID` with the contents specified by `domain`.

`dsr = Simulink.sdi.DatasetRef(runID, domain, repositoryPath)` creates a `Simulink.sdi.DatasetRef` object of the run corresponding to `runID` including the contents specified by `domain` from a repository path specified by `repositoryPath`.

Input Arguments

domain — Specify contents of `Simulink.sdi.DatasetRef` objects
'signals' | 'outports' | []

Limits the contents included in the `Simulink.sdi.DatasetRef` object.

- `'signals'` includes only logged signals.
- `'outports'` includes only logged outports.
- `[]` includes all run data.

runID — Run identifier

integer

Specifies the run containing the data for the `Simulink.sdi.DatasetRef` object.

repositoryPath — Path containing the run

string | character vector

Specifies the location of the run containing the data for the `Simulink.sdi.DatasetRef` object.

Properties

Name — Run name

character vector

The name of the run that corresponds with the `Simulink.sdi.DatasetRef` object.

Example: `'Run 1'`

Run — Simulink.sdi.Run object

`Simulink.sdi.Run` object

`Simulink.sdi.Run` object associated with the `Simulink.sdi.DatasetRef` object.

numElements — Number of top-level elements in run

`Simulink.sdi.Run` object

Number of top-level elements in the `Simulink.sdi.Run` object associated with the `Simulink.sdi.DatasetRef` object.

Methods

<code>compare</code>	Compare runs with <code>DatasetRef</code> objects
<code>getAsDatastore</code>	Retrieve element as <code>sdidatastore</code> object
<code>getElement</code>	Retrieve <code>DatasetRef</code> element by index
<code>getElementNames</code>	Get character vectors of element names
<code>getSignal</code>	Return <code>Signal</code> object
<code>plot</code>	Open the Simulation Data Inspector to view and compare data

Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Compare Runs with the `Simulink.sdi.DatasetRef` Object

This example shows how to work with the `Simulink.sdi.DatasetRef` object by comparing two runs of the `ex_sl-demo-absbrake` system with different desired slip ratios.

```
% Simulate model ex_sl-demo-absbrake to create a run of logged signals
load_system('ex_sl-demo-absbrake')
sim('ex_sl-demo-absbrake')

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
run = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = run.getDatasetRef;
```

```
% Get the names of the elements in the object
names = run_DSRef.getElementNames

names =

    2x1 cell array

    {'yout'}
    {'slp' }

% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);

% View signals in outputs
outputs = yout.Values

% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

outputs =

    struct with fields:

        Ww: [1x1 timeseries]
        Vs: [1x1 timeseries]
        Sd: [1x1 timeseries]

% Create another run for a different Desired relative slip
set_param('ex_sldemo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sldemo_absbrake')
DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));
```

```
% Open the Simulation Data Inspector to view signals  
run_DSRef.plot
```

See Also

`Simulink.SimulationData.DatasetRef` | `Simulink.sdi.Run` |
`Simulink.sdi.WorkerRun` | `Simulink.sdi.WorkerRun.getDatasetRef`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

compare

Class: Simulink.sdi.DatasetRef

Package: Simulink.sdi

Compare runs with DatasetRef objects

Syntax

```
[matches, mismatches, results] = dsrObj.compare(other)
```

Description

`[matches, mismatches, results] = dsrObj.compare(other)` returns the number of matches, number of mismatches, and comparison results for a comparison of data in a `Simulink.sdi.DatasetRef` object. The comparison results are returned as a `Simulink.sdi.DiffRunResult` object.

Input Arguments

other — Comparison data

MAT-file | variable

Comparison data, which can come from another `Simulink.sdi.DatasetRef` object, a Dataset in the workspace, or a MAT-file.

Example: 'data.mat'

Example: var

Output Arguments

matches — Number of matching signals

integer

Number of signals that matched within tolerance in the comparison.

mismatches — Number of mismatched signals

integer

Number of signals that did not match within tolerance in the comparison.

results — Simulink.sdi.DiffRunResult object with comparison results

Simulink.sdi.DiffRunResult object

Results of the comparison, returned in a Simulink.sdi.DiffRunResult object.

Examples

Compare Runs with the Simulink.sdi.DatasetRef Object

This example shows how to work with the Simulink.sdi.DatasetRef object by comparing two runs of the ex_sldemo_absbrake system with different desired slip ratios.

```
% Simulate model ex_sldemo_absbrake to create a run of logged signals
load_system('ex_sldemo_absbrake')
sim('ex_sldemo_absbrake')

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
run = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = run.getDatasetRef;

% Get the names of the elements in the object
names = run_DSRef.getElementNames

names =

    2x1 cell array
```

```
{'yout'}
{'slp' }

% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);

% View signals in outputs
outputs = yout.Values

% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

outputs =

    struct with fields:
        Ww: [1x1 timeseries]
        Vs: [1x1 timeseries]
        Sd: [1x1 timeseries]

% Create another run for a different Desired relative slip
set_param('ex_sldemo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sldemo_absbrake')
DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));

% Open the Simulation Data Inspector to view signals
run_DSRef.plot
```

Alternatives

Using the Simulation Data Inspector API, you could create runs for the data you want to compare and use `Simulink.sdi.compareRuns` for the comparison. You can also view runs created from simulation, import data to runs, and compare runs with the Simulation Data Inspector UI.

See Also

`Simulink.sdi.DatasetRef` | `Simulink.sdi.DiffRunResult` | `Simulink.sdi.Run`
| `Simulink.sdi.compareRuns` | `Simulink.sdi.compareSignals` |
`Simulink.sdi.view`

Topics

“Inspect and Compare Data Programmatically”

“How the Simulation Data Inspector Compares Data”

Introduced in R2017b

getAsDatastore

Class: Simulink.sdi.DatasetRef

Package: Simulink.sdi

Retrieve element as sdatastore object

Syntax

```
[elementDatastore, name, index] = SDIDatasetRef.getAsDatastore(arg)
```

Description

[elementDatastore, name, index] = SDIDatasetRef.getAsDatastore(arg) returns the requested element as a matlab.io.datastore.sdatastoreobject, along with the element name and index.

Input Arguments

arg — Element selection criterion

integer | character vector

Search criterion used to retrieve the element from the Simulink.sdi.DatasetRefobject. For name-based searches, specify arg as a character vector. For index-based searches, arg is an integer, representing the index of the desired element.

Example: 'MySignal'

Example: 3

Output Arguments

elementDatastore — Element as sdatastore

sdatastore object

Element as `matlab.io.datastore.sd datastore` object.

name — Element name

character vector

The name of the element.

index — Element index in `DatasetRef` object

integer

The index of the element in the `Simulink.sdi.DatasetRef` object.

Examples

Create an `sd datastore` Object for a Signal

This example shows how to create a `sd datastore` object for a signal in a `Simulink.sdi.DatasetRef` object.

```
% Simulate model sldemo_fuelsys to create a run of logged signals
sim('sldemo_fuelsys')
```

```
% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

```
% Get the run object
run = Simulink.sdi.getRun(runID);
```

```
% Make a Simulink.sdi.DatasetRef object
run_DSRef = run.getDatasetRef;
```

```
% Get the names of the elements in the object
names = run_DSRef.getElementNames
```

```
names =
```

```
10x1 cell array
```

```
    {'EGO Fault Switch:1'          }
    {'air_fuel_ratio'              }
```

```
{'Engine Speed Fault Switch:1' }
{'speed'                        }
{'MAP Fault Switch:1'         }
{'map'                          }
{'ego'                          }
{'Throttle Angle Fault Switch:1'}
{'throttle'                    }
{'fuel'                         }
```

`% Get sddatastore object for fuel signal`
`fuel_ds = run_DSRef.getAsDatastore(10);`

Alternatives

You can construct a `sddatastore` object for a specified signal using `matlab.io.datastore.sddatastore`.

See Also

`Simulink.sdi.DatasetRef` | `Simulink.sdi.DatasetRef.getElement` |
`matlab.io.datastore.SimulationDatastore` |
`matlab.io.datastore.sddatastore`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

getElement

Class: Simulink.sdi.DatasetRef

Package: Simulink.sdi

Retrieve DatasetRef element by index

Syntax

```
[element, name, index] = SDIdatasetRef.getElement(index)
```

Description

[element, name, index] = SDIdatasetRef.getElement(index) returns the element within the Run in the Simulink.sdi.DatasetRef object at the specified index.

Input Arguments

index — Index of element

integer

Location of the element in the Simulink.sdi.DatasetRef object.

Output Arguments

element — Run element in the DatasetRef object

signal

Element from the run in the Simulink.sdi.DatasetRef object.

name — Element name

character vector

Name of the element retrieved from the `Simulink.sdi.DatasetRef` object.

index — Location of element

integer

Location of the element within the `Simulink.sdi.DatasetRef` object.

Examples

Compare Runs with the `Simulink.sdi.DatasetRef` Object

This example shows how to work with the `Simulink.sdi.DatasetRef` object by comparing two runs of the `ex_sldemo_absbrake` system with different desired slip ratios.

```
% Simulate model ex_sldemo_absbrake to create a run of logged signals
load_system('ex_sldemo_absbrake')
sim('ex_sldemo_absbrake')
```

```
% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

```
% Get the run object
run = Simulink.sdi.getRun(runID);
```

```
% Make a Simulink.sdi.DatasetRef object
run_DSRef = run.getDatasetRef;
```

```
% Get the names of the elements in the object
names = run_DSRef.getElementNames
```

```
names =
```

```
2x1 cell array
```

```
    {'yout'}
    {'slp' }
```

```
% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);
```

```
% View signals in outputs
outputs = yout.Values

% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

outputs =

    struct with fields:
        Ww: [1x1 timeseries]
        Vs: [1x1 timeseries]
        Sd: [1x1 timeseries]

% Create another run for a different Desired relative slip
set_param('ex_sldemo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sldemo_absbrake')
DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));

% Open the Simulation Data Inspector to view signals
run_DSRef.plot
```

See Also

[Simulink.sdi.DatasetRef](#) | [Simulink.sdi.DatasetRef.getElementNames](#) | [Simulink.sdi.DatasetRef.getSignal](#)

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

getElementNames

Class: Simulink.sdi.DatasetRef

Package: Simulink.sdi

Get character vectors of element names

Syntax

```
names = dsrObj.getElementNames
```

Description

`names = dsrObj.getElementNames` returns a cell array of character vectors containing the names of the elements in `dsrObj`.

Output Arguments

names — Element names

cell array

Names of the top level elements in the `Simulink.sdi.DatasetRef` object in a cell array.

Examples

Compare Runs with the Simulink.sdi.DatasetRef Object

This example shows how to work with the `Simulink.sdi.DatasetRef` object by comparing two runs of the `ex_sl-demo-absbrake` system with different desired slip ratios.

```
% Simulate model ex_sl-demo-absbrake to create a run of logged signals  
load_system('ex_sl-demo-absbrake')
```

```
sim('ex_sldemo_absbrake')

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
run = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = run.getDatasetRef;

% Get the names of the elements in the object
names = run_DSRef.getElementNames

names =

    2x1 cell array

    {'yout'}
    {'slp' }

% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);

% View signals in outputs
outputs = yout.Values

% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

outputs =

    struct with fields:

        Ww: [1x1 timeseries]
        Vs: [1x1 timeseries]
        Sd: [1x1 timeseries]
```

```
% Create another run for a different Desired relative slip
set_param('ex_sldemo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sldemo_absbrake')
DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));

% Open the Simulation Data Inspector to view signals
run_DSRef.plot
```

See Also

[Simulink.SimulationData.DatasetRef](#) | [Simulink.sdi.DatasetRef](#) |
[Simulink.sdi.DatasetRef.getElement](#) | [Simulink.sdi.DatasetRef.getSignal](#)

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

getSignal

Class: Simulink.sdi.DatasetRef

Package: Simulink.sdi

Return Signal object

Syntax

```
sigObj = SDIDatasetRef.getSignal(searchArg)
```

Description

`sigObj = SDIDatasetRef.getSignal(searchArg)` returns the `Simulink.sdi.Signal` object corresponding to the search argument, `searchArg`.

Input Arguments

searchArg — Search parameter

character vector | integer

The search parameters to select the `Simulink.sdi.Signal` object. The `searchArg` can be a character vector or string targeting a signal name or an integer for an index-based search.

Example: 'throttle'

Example: 2

Output Arguments

sigObj — Simulink.sdi.Signal object

`Simulink.sdi.Signal` object

The `Simulink.sdi.Signal` object corresponding to the search query.

Examples

Compare Runs with the Simulink.sdi.DatasetRef Object

This example shows how to work with the `Simulink.sdi.DatasetRef` object by comparing two runs of the `ex_sl_demo_absbrake` system with different desired slip ratios.

```
% Simulate model ex_sl_demo_absbrake to create a run of logged signals
load_system('ex_sl_demo_absbrake')
sim('ex_sl_demo_absbrake')

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
run = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = run.getDatasetRef;

% Get the names of the elements in the object
names = run_DSRef.getElementNames

names =

    2x1 cell array

    {'yout'}
    {'slp' }
```

```
% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);

% View signals in outputs
outputs = yout.Values

% Get slp signal
slp = run_DSRef.getSignal('slp');
```

```
% Plot signal
slp.Checked = 'true';

outputs =

    struct with fields:

        Ww: [1x1 timeseries]
        Vs: [1x1 timeseries]
        Sd: [1x1 timeseries]

% Create another run for a different Desired relative slip
set_param('ex_sl-demo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sl-demo_absbrake')
DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));

% Open the Simulation Data Inspector to view signals
run_DSRef.plot
```

Alternatives

If the signal is a top-level element in the `Simulink.sdi.DatasetRef` object, you can use the `Simulink.sdi.DatasetRef.getElement` method to get the `Simulink.sdi.Signal` object by index.

See Also

`Simulink.sdi.DatasetRef` | `Simulink.sdi.DatasetRef.getElement` | `Simulink.sdi.getSignal`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

plot

Class: Simulink.sdi.DatasetRef

Package: Simulink.sdi

Open the Simulation Data Inspector to view and compare data

Syntax

```
SDIDatasetRef.plot
```

Description

`SDIDatasetRef.plot` opens the Simulation Data Inspector, where you can view and compare runs and signals.

Examples

Compare Runs with the Simulink.sdi.DatasetRef Object

This example shows how to work with the `Simulink.sdi.DatasetRef` object by comparing two runs of the `ex_sl_demo_absbrake` system with different desired slip ratios.

```
% Simulate model ex_sl_demo_absbrake to create a run of logged signals
load_system('ex_sl_demo_absbrake')
sim('ex_sl_demo_absbrake')

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
run = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
```

```
run_DSRef = run.getDatasetRef;

% Get the names of the elements in the object
names = run_DSRef.getElementNames

names =

    2x1 cell array

    {'yout'}
    {'slp' }

% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);

% View signals in outputs
outputs = yout.Values

% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

outputs =

    struct with fields:

        Ww: [1x1 timeseries]
        Vs: [1x1 timeseries]
        Sd: [1x1 timeseries]

% Create another run for a different Desired relative slip
set_param('ex_sldemo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sldemo_absbrake')
DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));
```

```
% Open the Simulation Data Inspector to view signals  
run_DSRef.plot
```

Alternatives

You can use the `Simulink.sdi.view` function to open the Simulation Data Inspector. For information on using the UI to open the Simulation Data Inspector, see “Populate the Simulation Data Inspector with Your Data”.

See Also

`Simulink.sdi.DatasetRef` | `Simulink.sdi.Signal.plotOnSubPlot` |
`Simulink.sdi.setSubPlotLayout` | `Simulink.sdi.view`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

Simulink.sdi.DiffRunResult class

Package: Simulink.sdi

Access run comparison metadata

Description

The `Simulink.sdi.DiffRunResult` class provides access to the run comparison metadata. You can use the `getSignalByIndex` method to access the data and comparison results for each signal in the run comparison.

Construction

`DiffRunResultObj = Simulink.sdi.compareRuns(runID1, runID2)` returns a `Simulink.sdi.DiffRunResult` object to provide access to the comparison results from comparing the runs corresponding to `runID1` and `runID2`.

Input Arguments

runID1 — Baseline run identifier
integer

Numeric run identifier for the **Baseline** run in the comparison.

runID2 — Compare to run identifier
integer

Numeric identifier for the **Compare to** run in the comparison.

Properties

RunID1 — Baseline signal run ID
integer

Run identifier for the **Baseline** signal of the comparison.

RunID2 — Compare to signal run ID

integer

Run identifier for the **Compare to** signal of the comparison.

MatlabVersion — Version used

character vector

Version of MATLAB used.

DateCreated — Object creation date

datetime

Date and time the Simulink.sdi.DiffRunResult object was created.

Data Types: datetime

Count — Number of signals compared

integer

Number of signals aligned between the two runs in the comparison. For more information on how signals are aligned for comparisons, see “How the Simulation Data Inspector Compares Data”.

Methods

getResultByIndex

Return signal comparison result

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Analyze Simulation Data with Signal Tolerances

You can change tolerance values on a signal-by-signal basis to evaluate the effect of a model parameter change. This example uses the `slexAircraftExample` model and the Simulation Data Inspector to evaluate the effect of changing the time constant for the low-pass filter following the control input.

```
% Load example model
load_system('slexAircraftExample')

% Mark the alpha, rad signal for streaming
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'alpha')

% Simulate system
sim('slexAircraftExample')

% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample', 'modelworkspace');
modelWorkspace.assignin('Ts', 0.2)

% Simulate again
sim('slexAircraftExample')

% Get run data
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);

% Compare runs
diffRun1 = Simulink.sdi.compareRuns(runID1, runID2);

% Get signal result
sigResult1 = diffRun1.getResultByIndex(1);

% Check whether signals matched
sigResult1.match

ans =

    logical
```

```
0

% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = run1.getSignalIDByIndex(1);
sig1 = Simulink.sdi.getSignal(sigID1);

% Change absolute tolerance to 0.2
sig1.absTol = 0.2;

% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1, runID2);
sigResult2 = diffRun2.getResultByIndex(1);

% Check the result
sigResult2.match

ans =

    logical

     1
```

Alternatives

You can view and inspect comparison results using the Simulation Data Inspector UI. For more information, see “Compare Simulation Data”.

See Also

[Simulink.sdi.DatasetRef.compare](#) | [Simulink.sdi.DiffSignalResult](#) | [Simulink.sdi.compareRuns](#) | [Simulink.sdi.compareSignals](#)

Topics

“Inspect and Compare Data Programmatically”
“How the Simulation Data Inspector Compares Data”

Introduced in R2012b

getResultByIndex

Class: Simulink.sdi.DiffRunResult

Package: Simulink.sdi

Return signal comparison result

Syntax

```
diffSigObj = diffRunObj.getResultByIndex(index)
```

Description

`diffSigObj = diffRunObj.getResultByIndex(index)` returns the `Simulink.sdi.DiffSignalResult` object `diffSigObj` corresponding to the `index` in the `Simulink.sdi.DiffRunResult` object, `diffRunObj`.

Input Arguments

index — Index of signal in run
integer

Index of the signal in the `Simulink.sdi.DiffRunResult` object.

Output Arguments

diffSigObj — `Simulink.sdi.DiffSignalResult` object corresponding to the index
`Simulink.sdi.DiffSignalResult` object

`Simulink.sdi.DiffSignalResult` object for the signal at the specified index.

Examples

Analyze Simulation Data with Signal Tolerances

You can change tolerance values on a signal-by-signal basis to evaluate the effect of a model parameter change. This example uses the `slexAircraftExample` model and the Simulation Data Inspector to evaluate the effect of changing the time constant for the low-pass filter following the control input.

```
% Load example model
load_system('slexAircraftExample')

% Mark the alpha, rad signal for streaming
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'alpha')

% Simulate system
sim('slexAircraftExample')

% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample', 'modelworkspace');
modelWorkspace.assignin('Ts', 0.2)

% Simulate again
sim('slexAircraftExample')

% Get run data
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);

% Compare runs
diffRun1 = Simulink.sdi.compareRuns(runID1, runID2);

% Get signal result
sigResult1 = diffRun1.getResultByIndex(1);

% Check whether signals matched
sigResult1.match

ans =

    logical
```

```
0

% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = run1.getSignalIDByIndex(1);
sig1 = Simulink.sdi.getSignal(sigID1);

% Change absolute tolerance to 0.2
sig1.absTol = 0.2;

% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1, runID2);
sigResult2 = diffRun2.getResultByIndex(1);

% Check the result
sigResult2.match

ans =

    logical

     1
```

Alternatives

You can inspect comparison results using the Simulation Data Inspector UI. For more information, see “Compare Simulation Data”.

See Also

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

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2012b

Simulink.sdi.DiffSignalResult class

Package: Simulink.sdi

Access signal comparison results

Description

The `Simulink.sdi.DiffSignalResult` object provides access to the data and metadata created by a signal comparison. A `Simulink.sdi.DiffSignalResult` object gives access the difference signal, tolerance data, and the synchronized signal data.

Construction

`DiffSignalObj = Simulink.sdi.compareSignals(signalID1, signalID2)` creates a `Simulink.sdi.DiffSignalResult` object to provide access to the results of the comparison of the signals corresponding to `sigID1` and `sigID2`.

`DiffSignalObj = DiffRunObj.getResultByIndex(index)` returns a `Simulink.sdi.DiffSignalResult` object for the signal comparison corresponding to the `index` within a `Simulink.sdi.DiffRunResult` object.

Input Arguments

signalID1 — Signal identifier for Baseline signal

integer

Numeric signal identifier for the **Baseline** signal in comparison.

signalID2 — Signal identifier for Compare to signal

integer

Numeric signal identifier for the **Compare to** signal in comparison

index — Index of signal in run

integer

Index of the signal within the run.

Properties

Diff — Difference signal

timeseries

Difference signal resulting from the comparison as a `timeseries` object.

Match — Logical indicator of signal match

logical

Logical indicator of whether comparison signals match within the tolerances.

- 0 indicates that the difference between the signals is not within tolerance.
- 1 indicates that the difference between the signals is within tolerance.

UnitsMatch — Logical indicator of unit match

logical

Logical indicator of whether comparison signals' units match. Comparisons of signals with units that do not match are always marked out of tolerance, and no difference signal is computed.

- 0 indicates that the signals' units do not match.
- 1 indicates that the signals' units match.

MaxDifference — Maximum difference

double

Maximum difference between the two comparison signals.

SignalID1 — Baseline signal identifier

integer

Unique signal identifier for the **Baseline** comparison signal.

SignalID2 — Compare to signal identifier

integer

Unique signal identifier for the **Compare to** comparison signal.

Sync1 — Synchronized Baseline signal

timeseries

Synchronized **Baseline** signal. For more information about synchronization, see “How the Simulation Data Inspector Compares Data”.

Sync2 — Synchronized Compare to signal

timeseries

Synchronized **Compare to** signal. For more information about synchronization, see “How the Simulation Data Inspector Compares Data”.

Tol1 — Tolerance signal

timeseries

Tolerance data for every data point of the comparison. For more information on how the tolerance signal is computed, see “How the Simulation Data Inspector Compares Data”.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Compare Signals Within a Simulation Run

This example uses the `slexAircraftExample` model to demonstrate the comparison of a control system's input and output signals. The example marks the signals for streaming then gets the run object for a simulation run. Signal IDs from the run object specify the signals to be compared.

```
% Load model slexAircraftExample, and mark signals for streaming
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'on')
```

```
% Simulate model slxAircraftExample
sim('slxAircraftExample')

% Get run IDs for most recent run
allIDs = Simulink.sdi.getAllRunIDs;
runID = allIDs(end);

%Get Run object
run = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = run.getSignalIDByIndex(1);
signalID2 = run.getSignalIDByIndex(2);

if (run.isValidSignalID(signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (run.isValidSignalID(signalID1) && run.isValidSignalID(signalID2))
    % Compare signals
    diff = Simulink.sdi.compareSignals(signalID1, signalID2);

    % Check whether signals match within tolerance
    match = diff.match
end

match =

    logical

     0
```

Alternatives

You can view and inspect comparison results using the Simulation Data Inspector UI. For more information, see “Compare Simulation Data”.

See Also

`Simulink.sdi.DiffRunResult` |
`Simulink.sdi.DiffRunResult.getResultByIndex` | `Simulink.sdi.compareRuns`
| `Simulink.sdi.compareSignals`

Topics

“Inspect and Compare Data Programmatically”
“How the Simulation Data Inspector Compares Data”

Introduced in R2012b

Simulink.sdi.Run class

Package: Simulink.sdi

Access run signals and metadata

Description

The `Simulink.sdi.Run` object manages a run's metadata and the signals that comprise the run. You can use several methods to retrieve `Simulink.sdi.Signal` objects to access the signal data and metadata.

Construction

`runObj = Simulink.sdi.Run.create` creates an empty `Simulink.sdi.Run` object.

`runObj = Simulink.sdi.getRun(runID)` creates a `Simulink.sdi.Run` object, `runObj`, for the run corresponding to `runID`.

You can also use the `Simulink.sdi.createRun` and `Simulink.sdi.createRunOrAddToStreamedRun` functions to create `Run` objects.

Input Arguments

runID — Run identifier

integer

Unique number identifying the run.

Properties

id — Run identifier

integer

Unique numerical identification for the run.

Name — Run name

[] (default) | character vector

Name of the run. By default, name is empty.

Example: 'Run 1'

Description — Run description

[] (default) | character vector

Description of the run. By default, Description is empty.

Example: 'Initial simulation'

Tag — Information tag

[] (default) | character vector

Tag for additional run information. By default, Tag is empty. You can use the Tag parameter to categorize your simulation data or attach extra information to simulation runs.

Example: 'Gain = 2'

DateCreated — Run creation time

datetime object

Date and time the run was created.

Data Types: datetime

SignalCount — Number of signals in run

integer

Number of signals contained in the run.

Model — Model that created the run

character vector

Name of the model that created the run.

SimMode — Simulation mode

character vector

Simulation mode used to create the run, for runs created by simulation.

StartTime — Run start time

integer

First time point shared by all signals in the run.

StopTime — Run stop time

integer

Last time point shared by all signals in the run.

SLVersion — Simulink version used to create run

character vector

Version of Simulink used for the simulation that created the run.

ModelVersion — Model version used to create run

character vector

Version of the model simulated to create the run, taken from the **Model Properties**.

UserID — System account

character vector

System account used for the simulation that created the run. UserID only has a value for runs produced with Simulink simulations.

MachineName — Name of machine used for simulation

character vector

Name of the machine used for the simulation that created the run. MachineName only has a value for runs produced with Simulink simulations.

TaskName — Task name

[] (default) | character vector

Name of the simulation task that created the run for runs created with Parallel Computing Toolbox workers.

SolverType — Type of solver used to create run

'Variable-Step' | 'Fixed-Step'

The type of solver used for the simulation that created the run. SolverType only has a value for runs produced with Simulink simulations.

SolverName — Name of solver used to create run

character vector

Name of the solver used for the simulation that created the run. `SolverName` only has a value for runs produced with Simulink simulations.

Example: `ode45`

ModelInitializationTime — Time to initialize model

double

Amount of time to initialize the model for the simulation that created the run. `ModelInitializationTime` only has a value for runs produced with Simulink simulations.

ModelExecutionTime — Time to execute model

double

Execution time of the model simulation that created the run. `ModelExecutionTime` only has a value for runs produced with Simulink simulations.

ModelTerminationTime — Time to terminate simulation

double

Time to terminate the simulation that created the run. `ModelTerminationTime` only has a value for runs produced with Simulink simulations.

ModelTotalElapsedTime — Total simulation time

double

Total time to run model simulation that created the run. `ModelTotalElapsedTime` only has a value for runs produced with Simulink simulations.

Methods

<code>add</code>	Add signals to run
<code>create</code>	Create a <code>Simulink.sdi.Run</code> object
<code>export</code>	Export run to <code>Simulink.SimulationData.Dataset</code> object
<code>getDatasetRef</code>	Create a <code>Simulink.sdi.DatasetRef</code> object for a run
<code>getSignalByIndex</code>	Get <code>Simulink.sdi.Signal</code> object by index
<code>getSignalIDByIndex</code>	Return signal ID for signal at index
<code>isValidSignalID</code>	Determine whether signal ID is valid within a run

Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Plot Signals from a Simulation Run

This example demonstrates how to access the `Simulink.sdi.Run` object for a run created by logging signals to the Simulation Data Inspector. From the `Simulink.sdi.Run` object you can get `Simulink.sdi.Signal` objects that you can use to view data.

```
% Simulate model sldemo_absbrake to create a run
sim('sldemo_fuelsys')

% Get runID for the run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get run object for the run
run = Simulink.sdi.getRun(runID);

% Check signal count of the run
run.signalCount
```

```
ans =  
  
    int32  
  
    10  
  
% Get signal objects for the signals in the run  
signal1 = run.getSignalByIndex(4);  
signal2 = run.getSignalByIndex(9);  
signal3 = run.getSignalByIndex(10);  
  
% Create sub-plot layout to display signals  
Simulink.sdi.setSubPlotLayout(3, 1)  
  
% Plot signals  
signal1.checked = true;  
signal2.plotOnSubPlot(2, 1, true);  
signal3.plotOnSubPlot(3, 1, true);  
  
%View plots in the Simulation Data Inspector  
Simulink.sdi.view
```

Create a Run and View the Data

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.

Create Data for the Run

This example creates `timeseries` objects for a sine and a cosine. To visualize your data, the Simulation Data Inspector requires at least a time vector that corresponds with your data.

```
% Generate timeseries data  
time = linspace(0, 20, 100);  
  
sine_vals = sin(2*pi/5*time);  
sine_ts = timeseries(sine_vals, time);  
sine_ts.Name = 'Sine, T=5';  
  
cos_vals = cos(2*pi/8*time);  
cos_ts = timeseries(cos_vals, time);  
cos_ts.Name = 'Cosine, T=8';
```

Create a Simulation Data Inspector Run and Add Your Data

To give the Simulation Data Inspector access to your data, use the `create` method and create a run. This example modifies some of the run's properties to help identify the data. You can easily view run and signal properties with the Simulation Data Inspector.

```
% Create a run
run = Simulink.sdi.Run.create;
run.Name = 'Sinusoids';
run.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
run.add('vars', sine_ts, cos_ts);
```

Plot Your Data Using the `Simulink.sdi.Signal` Object

The `getSignalByIndex` method returns a `Simulink.sdi.Signal` object that can be used to plot the signal in the Simulation Data Inspector. You can also programmatically control aspects of the plot's appearance, such as the color and style of the line representing the signal. This example customizes the subplot layout and signal characteristics.

```
% Get signal, modify its properties, and change Checked property to true
sine_sig = run.getSignalByIndex(1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';
sine_sig.Checked = true;

% Add another subplot for the cosine signal
Simulink.sdi.setSubPlotLayout(2, 1);

% Plot the cosine signal and customize its appearance
cos_sig = run.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.plotOnSubPlot(2, 1, true);

% View the signal in the Simulation Data Inspector
Simulink.sdi.view
```

Close the Simulation Data Inspector and Save Your Data

```
Simulink.sdi.close('sinusoids.mat')
```

Access Data from a Parallel Simulation

This example executes parallel simulations of the model `slexAircraftExample` with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector API.

Generate Data with Parallel Simulations

Create a vector of filter parameter values to use a different value in each simulation. Then, run a parallel simulation, selecting a different filter value for each worker.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport(1)

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];

parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Change the filter time constant and simulate
    modelWorkspace = get_param('slexAircraftExample', 'modelworkspace');
    modelWorkspace.assignin('Ts', Ts_val)
    sim('slexAircraftExample')

    % Create a worker run for each simulation
    workerRun(index) = Simulink.sdi.WorkerRun.getLatest
end
```

Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

```
ds(7) = Simulink.SimulationData.Dataset;
```

```
for a = 1:7
    ds(a) = workerRun(a).getDataset;
end
ds(1)
```

```
ans =
```

```
Simulink.SimulationData.Dataset '' with 2 elements
```

		Name	BlockPath
1	[1x1 Signal]	alpha, rad	...rcraftExample/Aircraft Dynamics Model
2	[1x1 Signal]	Stick	slexAircraftExample/Pilot

- Use braces { } to access, modify, or add elements using index.

Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the `getDatasetRef` method to reference the data associated with the `WorkerRun`.

```
for b = 1:7
    datasetRef(b) = workerRun(b).getDatasetRef;
end
datasetRef(1)
```

```
ans =
```

```
DatasetRef with properties:
```

```
    Name: 'Run 6: slexAircraftExample'
      Run: [1x1 Simulink.sdi.Run]
numElements: 2
```

Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local Run objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```
for c = 1:7
    Runs(c) = workerRun(c).getLocalRun;
    Ts_val_str = num2str(Ts_vals(c));
    desc = strcat('Ts = ', Ts_val_str);
    Runs(c).Description = desc;
    Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);
end
```

Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

```
Simulink.sdi.cleanupWorkerResources
```

Alternatives

You can view runs and their properties in the Simulation Data Inspector UI. You can also import data to create runs in the Simulation Data Inspector GUI. For more information, see “Populate the Simulation Data Inspector with Your Data”.

See Also

```
Simulink.sdi.Signal | Simulink.sdi.WorkerRun |
Simulink.sdi.WorkerRun.getLocalRun | Simulink.sdi.addToRun |
Simulink.sdi.createRun | Simulink.sdi.getRun |
Simulink.sdi.getRunIDByIndex | Simulink.sdi.setRunNamingRule
```

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2012b

add

Class: Simulink.sdi.Run

Package: Simulink.sdi

Add signals to run

Syntax

```
run.add(sig)
run.add(source, filename)
run.add(source, ts1, ts2)
```

Description

`run.add(sig)` adds the data `sig` to the `Simulink.sdi.Run` object `run` from the base workspace.

`run.add(source, filename)` adds the data in the file `filename` to the `Simulink.sdi.Run` object, `run`.

`run.add(source, ts1, ts2)` allows you to add multiple signals to the run from the base workspace.

Input Arguments

sig — Data to add

Signals to add to the run. Data types that you can add to a run include:

- `timeseries`
- `Simulink.SimulationData.Dataset`
- `Simulink.SimulationOutput`
- `timetable`

- Data logged with `Structure` with `time` format
- Simscape variables

source — Data source selector

'file' | 'vars'

Source of the data to add to the run, specified as a character vector.

- 'file' indicates the data comes from a file.
- 'vars' indicates the data comes from one or more variables in the workspace.

filename — File containing data

character vector

File with data to add to the run.

Example: 'data.mat'

ts1, ts2 — Workspace variables to add to run

timeseries

Data to add to the run in one or more `timeseries` objects.

Examples

Create a Run and View the Data

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.

Create Data for the Run

This example creates `timeseries` objects for a sine and a cosine. To visualize your data, the Simulation Data Inspector requires at least a time vector that corresponds with your data.

```
% Generate timeseries data
time = linspace(0, 20, 100);

sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals, time);
```

```
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals, time);
cos_ts.Name = 'Cosine, T=8';
```

Create a Simulation Data Inspector Run and Add Your Data

To give the Simulation Data Inspector access to your data, use the `create` method and create a run. This example modifies some of the run's properties to help identify the data. You can easily view run and signal properties with the Simulation Data Inspector.

```
% Create a run
run = Simulink.sdi.Run.create;
run.Name = 'Sinusoids';
run.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
run.add('vars', sine_ts, cos_ts);
```

Plot Your Data Using the `Simulink.sdi.Signal` Object

The `getSignalByIndex` method returns a `Simulink.sdi.Signal` object that can be used to plot the signal in the Simulation Data Inspector. You can also programmatically control aspects of the plot's appearance, such as the color and style of the line representing the signal. This example customizes the subplot layout and signal characteristics.

```
% Get signal, modify its properties, and change Checked property to true
sine_sig = run.getSignalByIndex(1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';
sine_sig.Checked = true;

% Add another subplot for the cosine signal
Simulink.sdi.setSubPlotLayout(2, 1);

% Plot the cosine signal and customize its appearance
cos_sig = run.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.plotOnSubPlot(2, 1, true);

% View the signal in the Simulation Data Inspector
Simulink.sdi.view
```

Close the Simulation Data Inspector and Save Your Data

```
Simulink.sdi.close('sinusoids.mat')
```

See Also

[Simulink.sdi.Run](#) | [Simulink.sdi.Run.create](#) | [Simulink.sdi.addToRun](#) | [Simulink.sdi.createRunOrAddToStreamedRun](#)

Topics

“Populate the Simulation Data Inspector with Your Data”

Introduced in R2017b

Simulink.sdi.Run.create

Class: Simulink.sdi.Run

Package: Simulink.sdi

Create a Simulink.sdi.Run object

Syntax

```
runObj = Simulink.sdi.Run.create
```

Description

`runObj = Simulink.sdi.Run.create` creates the empty run object, `runObj`. You can add signals to the Run object with the `Simulink.sdi.Run.add` method or the `Simulink.sdi.addToRun` function. For more information on the `Simulink.sdi.Run` object and its properties, see `Simulink.sdi.Run`.

Output Arguments

runObj — Simulink.sdi.Run object
Simulink.sdi.Run object

Empty Simulink.sdi.Run object.

Examples

Create a Run and View the Data

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.

Create Data for the Run

This example creates `timeseries` objects for a sine and a cosine. To visualize your data, the Simulation Data Inspector requires at least a time vector that corresponds with your data.

```
% Generate timeseries data
time = linspace(0, 20, 100);

sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals, time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals, time);
cos_ts.Name = 'Cosine, T=8';
```

Create a Simulation Data Inspector Run and Add Your Data

To give the Simulation Data Inspector access to your data, use the `create` method and create a run. This example modifies some of the run's properties to help identify the data. You can easily view run and signal properties with the Simulation Data Inspector.

```
% Create a run
run = Simulink.sdi.Run.create;
run.Name = 'Sinusoids';
run.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
run.add('vars', sine_ts, cos_ts);
```

Plot Your Data Using the `Simulink.sdi.Signal` Object

The `getSignalByIndex` method returns a `Simulink.sdi.Signal` object that can be used to plot the signal in the Simulation Data Inspector. You can also programmatically control aspects of the plot's appearance, such as the color and style of the line representing the signal. This example customizes the subplot layout and signal characteristics.

```
% Get signal, modify its properties, and change Checked property to true
sine_sig = run.getSignalByIndex(1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';
sine_sig.Checked = true;
```

```
% Add another subplot for the cosine signal
Simulink.sdi.setSubPlotLayout(2, 1);

% Plot the cosine signal and customize its appearance
cos_sig = run.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.plotOnSubPlot(2, 1, true);

% View the signal in the Simulation Data Inspector
Simulink.sdi.view
```

Close the Simulation Data Inspector and Save Your Data

```
Simulink.sdi.close('sinusoids.mat')
```

See Also

[Simulink.sdi.Run](#) | [Simulink.sdi.Run.add](#) | [Simulink.sdi.createRun](#) | [Simulink.sdi.createRunOrAddToStreamedRun](#)

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

export

Class: Simulink.sdi.Run

Package: Simulink.sdi

Export run to Simulink.SimulationData.Dataset object

Syntax

```
ds = runObj.export
```

Description

`ds = runObj.export` exports the Simulink.sdi.Run object `runObj` to the Simulink.SimulationData.Dataset, `ds`.

Output Arguments

ds — Simulink.SimulationData.Dataset object containing run data

Simulink.SimulationData.Dataset object

Simulink.SimulationData.Dataset object containing the run data.

Examples

Export Run Data

This example shows how to export data from a simulation run to a Dataset.

```
% Load model vdp
load_system('vdp')

% Get handles for signal lines in model
SignalHandles = get_param('vdp', 'Lines');
```

```
% Mark signals for streaming
Simulink.sdi.markSignalForStreaming(SignalHandles(5).Handle, 'on')
Simulink.sdi.markSignalForStreaming(SignalHandles(6).Handle, 'on')

% Simulate model vdp and get run object
sim('vdp')

% Get run object for simulation run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
run = Simulink.sdi.getRun(runID);

% Export run
simDataset = run.export;
```

Alternatives

You can export run data programmatically using the `Simulink.sdi.exportRun` function, or you can use the Simulation Data Inspector UI. For more information, see “Save and Share Simulation Data Inspector Data and Views”.

See Also

`Simulink.sdi.Run` | `Simulink.sdi.exportRun` | `Simulink.sdi.getRun` | `Simulink.sdi.report`

Topics

“Inspect and Compare Data Programmatically”

“Save and Share Simulation Data Inspector Data and Views”

Introduced in R2017b

getDatasetRef

Class: Simulink.sdi.Run

Package: Simulink.sdi

Create a Simulink.sdi.DatasetRef object for a run

Syntax

```
DatasetRef = runObj.getDatasetRef
```

Description

`DatasetRef = runObj.getDatasetRef` creates a `Simulink.sdi.DatasetRef` object with the data in the `Simulink.sdi.Run` object, `runObj`.

Output Arguments

DatasetRef — Simulink.sdi.DatasetRef object

Simulink.sdi.DatasetRef object

Simulink.sdi.DatasetRef object that provides access to the run data.

Examples

Compare Runs with the Simulink.sdi.DatasetRef Object

This example shows how to work with the `Simulink.sdi.DatasetRef` object by comparing two runs of the `ex_sl_demo_absbrake` system with different desired slip ratios.

```
% Simulate model ex_sl_demo_absbrake to create a run of logged signals
load_system('ex_sl_demo_absbrake')
sim('ex_sl_demo_absbrake')
```

```
% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
run = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = run.getDatasetRef;

% Get the names of the elements in the object
names = run_DSRef.getElementNames

names =

    2x1 cell array

    {'yout'}
    {'slp' }

% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);

% View signals in outputs
outputs = yout.Values

% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

outputs =

    struct with fields:
        Ww: [1x1 timeseries]
        Vs: [1x1 timeseries]
        Sd: [1x1 timeseries]
```

```
% Create another run for a different Desired relative slip
set_param('ex_sldemo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sldemo_absbrake')
DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));

% Open the Simulation Data Inspector to view signals
run_DSRef.plot
```

Alternatives

You can also create a `Simulink.sdi.DatasetRef` object using the `Simulink.sdi.DatasetRef` constructor.

See Also

`Simulink.sdi.DatasetRef` | `Simulink.sdi.Run`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

getSignalByIndex

Class: Simulink.sdi.Run

Package: Simulink.sdi

Get Simulink.sdi.Signal object by index

Syntax

```
signal = runObj.getSignalByIndex(index)
```

Description

`signal = runObj.getSignalByIndex(index)` returns a signal object for the signal at the specified index within the Simulink.sdi.Run object, runObj.

Input Arguments

index — Signal index

integer

Index of the signal within the run.

Output Arguments

signal — Simulink.sdi.Signal object

Simulink.sdi.Signal object

Simulink.sdi.Signal object for the signal at the specified index in the run.

Examples

Plot Signals from a Simulation Run

This example demonstrates how to access the `Simulink.sdi.Run` object for a run created by logging signals to the Simulation Data Inspector. From the `Simulink.sdi.Run` object you can get `Simulink.sdi.Signal` objects that you can use to view data.

```
% Simulate model sldemo_absbrake to create a run
sim('sldemo_fuelsys')

% Get runID for the run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get run object for the run
run = Simulink.sdi.getRun(runID);

% Check signal count of the run
run.signalCount

ans =

    int32

     10

% Get signal objects for the signals in the run
signal1 = run.getSignalByIndex(4);
signal2 = run.getSignalByIndex(9);
signal3 = run.getSignalByIndex(10);

% Create sub-plot layout to display signals
Simulink.sdi.setSubPlotLayout(3, 1)

% Plot signals
signal1.checked = true;
signal2.plotOnSubPlot(2, 1, true);
signal3.plotOnSubPlot(3, 1, true);
```

```
%View plots in the Simulation Data Inspector  
Simulink.sdi.view
```

Alternatives

You can access signal properties, view signals, and export data to the workspace using the Simulation Data Inspector UI. For more information, see “Organize Your Simulation Data Inspector Workspace”.

See Also

```
Simulink.sdi.Run | Simulink.sdi.Run.getSignalIDByIndex |  
Simulink.sdi.Run.isValidSignalID | Simulink.sdi.Signal |  
Simulink.sdi.getSignal
```

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2012b

getSignalIDByIndex

Class: Simulink.sdi.Run

Package: Simulink.sdi

Return signal ID for signal at index

Syntax

```
signalID = runObj.getSignalIDByIndex(index)
```

Description

`signalID = runObj.getSignalIDByIndex(index)` returns the signal ID for the signal at the specified index in the `Simulink.sdi.Run` object. You can use the signal ID to create a `Simulink.sdi.Signal` object or to perform a signal comparison with `Simulink.sdi.compareSignals`.

Input Arguments

index — Signal index

integer

Index of the signal within the run.

Output Arguments

signalID — Signal identifier

integer

Unique numeric signal identifier.

Examples

Compare Signals Within a Simulation Run

This example uses the `slexAircraftExample` model to demonstrate the comparison of a control system's input and output signals. The example marks the signals for streaming then gets the run object for a simulation run. Signal IDs from the run object specify the signals to be compared.

```
% Load model slexAircraftExample, and mark signals for streaming
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'on')

% Simulate model slexAircraftExample
sim('slexAircraftExample')

% Get run IDs for most recent run
allIDs = Simulink.sdi.getAllRunIDs;
runID = allIDs(end);

%Get Run object
run = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = run.getSignalIDByIndex(1);
signalID2 = run.getSignalIDByIndex(2);

if (run.isValidSignalID(signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (run.isValidSignalID(signalID1) && run.isValidSignalID(signalID2))
    % Compare signals
    diff = Simulink.sdi.compareSignals(signalID1, signalID2);

    % Check whether signals match within tolerance
    match = diff.match;
end

match =
```

logical

0

Alternatives

You can access signal properties, view signals, and export data to the workspace using the Simulation Data Inspector UI. For more information, see “Organize Your Simulation Data Inspector Workspace”.

See Also

[Simulink.sdi.Run](#) | [Simulink.sdi.Run.getSignalByIndex](#) |
[Simulink.sdi.Run.isValidSignalID](#) | [Simulink.sdi.Signal](#) |
[Simulink.sdi.compareSignals](#) | [Simulink.sdi.getSignal](#)

Topics

[“Inspect and Compare Data Programmatically”](#)

Introduced in R2012b

isValidSignalID

Class: Simulink.sdi.Run

Package: Simulink.sdi

Determine whether signal ID is valid within a run

Syntax

```
isValid = runObj.isValidSignalID(signalID)
```

Description

`isValid = runObj.isValidSignalID(signalID)` returns a logical indication of whether the `signalID` corresponds to a signal in the `Simulink.sdi.Run` object `runObj`.

Input Arguments

signalID — Signal identifier

integer

Unique numeric signal identifier.

Output Arguments

isValid — Logical indicator

logical

Logical indicator of signal ID validity.

- `true` when the `signalID` corresponds to a signal in the run object.
- `false` when the `signalID` does not correspond to a signal in the run object.

Examples

Compare Signals Within a Simulation Run

This example uses the `slexAircraftExample` model to demonstrate the comparison of a control system's input and output signals. The example marks the signals for streaming then gets the run object for a simulation run. Signal IDs from the run object specify the signals to be compared.

```
% Load model slexAircraftExample, and mark signals for streaming
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'on')

% Simulate model slexAircraftExample
sim('slexAircraftExample')

% Get run IDs for most recent run
allIDs = Simulink.sdi.getAllRunIDs;
runID = allIDs(end);

%Get Run object
run = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = run.getSignalIDByIndex(1);
signalID2 = run.getSignalIDByIndex(2);

if (run.isValidSignalID(signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (run.isValidSignalID(signalID1) && run.isValidSignalID(signalID2))
    % Compare signals
    diff = Simulink.sdi.compareSignals(signalID1, signalID2);

    % Check whether signals match within tolerance
    match = diff.match;
end

match =
```

logical

0

See Also

[Simulink.sdi.Run](#) | [Simulink.sdi.Run.getSignalIDByIndex](#) |
[Simulink.sdi.Signal](#) | [Simulink.sdi.compareSignals](#)

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2012b

Simulink.sdi.Signal class

Package: Simulink.sdi

Access signal data and metadata

Description

A `Simulink.sdi.Signal` object provides access to signal data and metadata. The metadata includes properties for visualizing and comparing signals.

Construction

`signal = Simulink.sdi.getSignal(signalID)` returns a `Simulink.sdi.Signal` object corresponding to the `signalID`.

`signal = runObj.getSignal(signalID)` returns a `Simulink.sdi.Signal` object corresponding to the `signalID`.

`signal = runObj.getSignalByIndex(index)` returns a `Simulink.sdi.Signal` object corresponding to the `signalID` at the index specified by `index` in the Run object `runObj`.

Input Arguments

signalID — Signal identifier

integer

Numeric signal identifier for the signal.

index — Signal index within run

integer

The index of the signal within its run.

Example: 1

Properties

Signal Properties (read only)

ID — Signal identifier

integer

Unique number identifying the signal.

RunID — Run identifier

integer

Run identifier for the run that contains the signal.

Name — Signal name

character vector

Name of the signal.

Units — Signal measurement units

character vector

Signal units of measure.

SampleTime — Signal sample time

character vector

The signal's sample time. A value of 'Continuous' indicates a variable-step simulation.

Model — Name of model that produced signal

character vector

Name of the model that produced the signal.

BlockPath — Path of block that defines signal

character vector

The path to the block that produced the signal.

Example: 'sldemo_fuelsys/Engine Gas Dynamics'

FullBlockPath — Path of the block that defines the signal

character vector

Path to the block that generates the signal including the full model hierarchy. For signals within reference models, `FullBlockPath` is a cell array containing the full path. For other signals, `FullBlockPath` is identical to `BlockPath`.

PortIndex — Block port index

integer

Index of the block port that defines the signal.

Dimensions — Dimensions of matrix containing signal

integer array

Dimensions of the matrix that contains the signal.

Channel — Index of signal within matrix

integer array

Indices of the signal for signals that are part of a vector or matrix.

Values — Signal values

timeseries

Time and data values for the signal. For buses, `Values` is a struct.

RootSource — High-level logging structure containing signal imported from workspace

character vector

Name of the high-level logging structure containing the signal for signals imported from the MATLAB workspace.

TimeSource — Source of signal time data imported from workspace

character vector

Name of the variable containing the signal's time data for signals imported from the MATLAB workspace.

DataSource — Source of data imported from workspace

character vector

Name of the array containing the signal data for signals imported from the MATLAB workspace.

Visualization Properties**Checked — Plotting indicator**

false (default) | true

Logical value indicating whether the signal is plotted on any subplot. Setting `Checked` to `false` clears the signal from all subplots. Setting `Checked` to `true` plots the signal on the active subplot.

Data Types: `logical`**LineColor — Signal line color**

1×3 vector

Color of signal in plots, specified as a 1×3 RGB vector.

Example: `[0 114 189]`Data Types: `double`**LineDashed — Signal line style**

'-' | '--' | ':' | '-.'

Signal line style.

- `'-'` specifies a solid line style.
- `'--'` specifies a dashed line style.
- `':'` specifies a dotted line style.
- `'-.'` specifies a dash-dot line style.

InterpMethod — Interpolation method

'linear' (default) | 'zoh'

Interpolation method used in data visualization and synchronization. `'zoh'` specifies zero-order hold, and `'linear'` specifies linear interpolation. For more information about the interpolation options, see “How the Simulation Data Inspector Compares Data”.

Comparison Properties**AbsTol — Absolute tolerance**

0 (default) | double

Positive-valued absolute tolerance of the signal used for signal and run comparisons. For more information about tolerances, see “How the Simulation Data Inspector Compares Data”.

Example: `0.1`

Data Types: `double`

RelTol — Relative tolerance

0 (default) | `double`

Positive-valued relative tolerance of the signal used for signal and run comparisons. For example, `0.1` specifies a 10% tolerance. For more information about tolerances, see “How the Simulation Data Inspector Compares Data”.

Example: `'0.1'`

Data Types: `double`

SyncMethod — Synchronization method

'union' (default) | 'intersection'

Method used to synchronize signal time data for comparison. For more information about the synchronization options, see “How the Simulation Data Inspector Compares Data”.

Methods

<code>export</code>	Export signal object to MATLAB timeseries
<code>getAsTall</code>	Return tall timetable with time and data values
<code>plotOnSubPlot</code>	Plot signal on specified sub-plot

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Define Comparison and Visualization Properties for a Signal

Use `Simulink.sdi.createRun` and `Simulink.sdi.getSignal` to create a signal object. Then, modify the signal's comparison and visualization properties.

```
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
            'SaveFormat','StructureWithTime');

% Use Simulink.sdi.createRun to create a run and return
% the list of signal IDs for signals contained in the run
[~,~,signalIDs] = Simulink.sdi.createRun('My Run','base',{'simOut'});

% Get the signal object corresponding to the first signal ID
signalObj = Simulink.sdi.getSignal(signalIDs(1));

% Define comparison and visualization properties for this signal
signalObj.syncMethod = 'intersection';
signalObj.lineColor = [1,0.4,0.6];
signalObj.lineDashed = '-';
signalObj.checked = true;

% View the signal and its parameters in the Simulation Data Inspector
Simulink.sdi.view;
```

Compare Signals Within a Simulation Run

This example uses the `slexAircraftExample` model to demonstrate the comparison of a control system's input and output signals. The example marks the signals for streaming then gets the run object for a simulation run. Signal IDs from the run object specify the signals to be compared.

```
% Load model slexAircraftExample, and mark signals for streaming
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'on')

% Simulate model slexAircraftExample
sim('slexAircraftExample')

% Get run IDs for most recent run
allIDs = Simulink.sdi.getAllRunIDs;
runID = allIDs(end);

%Get Run object
```

```
run = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = run.getSignalIDByIndex(1);
signalID2 = run.getSignalIDByIndex(2);

if (run.isValidSignalID(signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (run.isValidSignalID(signalID1) && run.isValidSignalID(signalID2))
    % Compare signals
    diff = Simulink.sdi.compareSignals(signalID1, signalID2);

    % Check whether signals match within tolerance
    match = diff.match
end

match =

    logical

     0
```

Plot Signals from a Simulation Run

This example demonstrates how to access the `Simulink.sdi.Run` object for a run created by logging signals to the Simulation Data Inspector. From the `Simulink.sdi.Run` object you can get `Simulink.sdi.Signal` objects that you can use to view data.

```
% Simulate model sldemo_absbrake to create a run
sim('sldemo_fuelsys')

% Get runID for the run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get run object for the run
run = Simulink.sdi.getRun(runID);
```

```

% Check signal count of the run
run.signalCount

ans =

    int32

    10

% Get signal objects for the signals in the run
signal1 = run.getSignalByIndex(4);
signal2 = run.getSignalByIndex(9);
signal3 = run.getSignalByIndex(10);

% Create sub-plot layout to display signals
Simulink.sdi.setSubPlotLayout(3, 1)

% Plot signals
signal1.checked = true;
signal2.plotOnSubPlot(2, 1, true);
signal3.plotOnSubPlot(3, 1, true);

%View plots in the Simulation Data Inspector
Simulink.sdi.view

```

Analyze Simulation Data with Signal Tolerances

You can change tolerance values on a signal-by-signal basis to evaluate the effect of a model parameter change. This example uses the `slexAircraftExample` model and the Simulation Data Inspector to evaluate the effect of changing the time constant for the low-pass filter following the control input.

```

% Load example model
load_system('slexAircraftExample')

% Mark the alpha, rad signal for streaming
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, '

% Simulate system
sim('slexAircraftExample')

% Change input filter time constant

```

```
modelWorkspace = get_param('slexAircraftExample', 'modelworkspace');
modelWorkspace.assignin('Ts', 0.2)

% Simulate again
sim('slexAircraftExample')

% Get run data
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);

% Compare runs
diffRun1 = Simulink.sdi.compareRuns(runID1, runID2);

% Get signal result
sigResult1 = diffRun1.getResultByIndex(1);

% Check whether signals matched
sigResult1.match

ans =

    logical

     0

% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = run1.getSignalIDByIndex(1);
sig1 = Simulink.sdi.getSignal(sigID1);

% Change absolute tolerance to 0.2
sig1.absTol = 0.2;

% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1, runID2);
sigResult2 = diffRun2.getResultByIndex(1);

% Check the result
sigResult2.match

ans =
```

```
logical
```

```
1
```

Alternatives

Use the Simulation Data Inspector UI to view and modify signals and signal properties.

See Also

```
Simulink.sdi.DatasetRef.getSignal | Simulink.sdi.Run |  
Simulink.sdi.Run.getSignalByIndex |  
Simulink.sdi.Run.getSignalIDByIndex | Simulink.sdi.createRun |  
Simulink.sdi.getSignal
```

Topics

“Inspect and Compare Data Programmatically”

“How the Simulation Data Inspector Compares Data”

Introduced in R2012b

export

Class: Simulink.sdi.Signal

Package: Simulink.sdi

Export signal object to MATLAB timeseries

Syntax

```
ts = sigObj.export  
ts = sigObj.export(startTime, endTime)
```

Description

`ts = sigObj.export` exports the Simulink.sdi.Signal object `sigObj` to the timeseries `ts` in the MATLAB workspace.

`ts = sigObj.export(startTime, endTime)` exports the portion of the Simulink.sdi.Signal object defined by `startTime` and `endTime` to the timeseries `ts` in the MATLAB workspace.

Input Arguments

startTime — Export start time

integer

Start time for the signal portion to export.

Example: 0

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

endTime — Export end time

integer

End time for the signal portion to export.

Example: 10

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

Output Arguments

ts — Exported timeseries

timeseries

Timeseries exported from Simulink.sdi.Signal object.

Examples

Export Signal Data to Timeseries

This example shows how to generate a signal and export the signal data to a timeseries.

```
% Simulate model sldemo_fuelsys to create a run of logged signals
sim('sldemo_fuelsys');

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
run = Simulink.sdi.getRun(runID);

% Get a signal
signal = run.getSignalByIndex(2);

% Export signal data to timeseries
ts = signal.export;
```

Alternatives

You can export data to MATLAB or a MAT-file from the Simulation Data Inspector UI. For more information, see “Save and Share Simulation Data Inspector Data and Views”.

See Also

`Simulink.sdi.Signal` | `Simulink.sdi.exportRun`

Topics

“Inspect and Compare Data Programmatically”

“Save and Share Simulation Data Inspector Data and Views”

Introduced in R2017b

getAsTall

Class: Simulink.sdi.Signal

Package: Simulink.sdi

Return tall timetable with time and data values

Syntax

```
tt = sigObj.getAsTall
```

Description

`tt = sigObj.getAsTall` returns a tall timetable of the time and data values in the `Simulink.sdi.Signal` object `sigObj`. For more information on working with tall arrays, see “Tall Arrays” (MATLAB).

Output Arguments

tt — Tall timetable

tall timetable

Tall timetable containing the data from the `Simulink.sdi.Signal` object.

Examples

Get Tall Timetable of Signal Data

This example shows how to generate a tall timetable from signal data in a `Simulink.sdi.Signal` object.

```
% Simulate the model sldemo_fuelsys to create a run of logged signals  
sim('sldemo_fuelsys');
```

```
% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
run = Simulink.sdi.getRun(runID);

% Get a signal
signal = run.getSignalByIndex(2);

% Get tall timetable of signal
tt = signal.getAsTall
```

```
tt =
```

```
Mx1 tall timetable
```

Time	Data
0 sec	0.068493
0.00056199 sec	0.092452
0.0033719 sec	0.21101
0.01 sec	0.48273
0.02 sec	0.88522
0.03 sec	1.2763
0.04 sec	1.6563
0.05 sec	2.0255
:	:
:	:

See Also

[Simulink.sdi.Signal](#) | [Simulink.sdi.Signal.export](#) | [Simulink.sdi.exportRun](#) | [tall](#)

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

plotOnSubPlot

Class: Simulink.sdi.Signal

Package: Simulink.sdi

Plot signal on specified sub-plot

Syntax

```
signalObj.plotOnSubplot(r, c, checked)
```

Description

`signalObj.plotOnSubplot(r, c, checked)` plots or clears the signal corresponding to the `Simulink.sdi.Signal` object, `sigObj`, on the sub-plot specified by `r` and `c`.

Input Arguments

r — Row index

scalar

Row index for the sub-plot.

Example: 1

c — Column index

scalar

Column index for the sub-plot.

Example: 2

checked — Signal checked parameter state

true | false

Desired state for signal on sub-plot.

- `true` plots the signal on the sub-plot.
- `false` clears the signal from the sub-plot.

Data Types: `logical`

Examples

Plot Signals from a Simulation Run

This example demonstrates how to access the `Simulink.sdi.Run` object for a run created by logging signals to the Simulation Data Inspector. From the `Simulink.sdi.Run` object you can get `Simulink.sdi.Signal` objects that you can use to view data.

```
% Simulate model sldemo_absbrake to create a run
sim('sldemo_fuelsys')

% Get runID for the run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get run object for the run
run = Simulink.sdi.getRun(runID);

% Check signal count of the run
run.signalCount

ans =

    int32

    10

% Get signal objects for the signals in the run
signal1 = run.getSignalByIndex(4);
signal2 = run.getSignalByIndex(9);
signal3 = run.getSignalByIndex(10);

% Create sub-plot layout to display signals
Simulink.sdi.setSubPlotLayout(3, 1)
```

```
% Plot signals
signal1.checked = true;
signal2.plotOnSubPlot(2, 1, true);
signal3.plotOnSubPlot(3, 1, true);

%View plots in the Simulation Data Inspector
Simulink.sdi.view
```

Alternatives

You can use the Simulation Data Inspector GUI to modify your plot layout and where you plot signals. For more information, see “Inspect Simulation Data”.

See Also

`Simulink.sdi.Signal` | `Simulink.sdi.setSubPlotLayout`

Topics

“Inspect and Compare Data Programmatically”

“Create Plots Using the Simulation Data Inspector”

Introduced in R2017b

Simulink.sdi.WorkerRun class

Package: Simulink.sdi

Access simulation data from parallel workers

Description

The `Simulink.sdi.WorkerRun` class provides access to run data generated on Parallel Computing Toolbox parallel workers. Create a `Simulink.sdi.WorkerRun` object on the worker, and then use the object to access data in your local MATLAB session.

Construction

`workerRun = Simulink.sdi.WorkerRun(runID)` creates a `Simulink.sdi.WorkerRun` object with the run identifier specified by `runID`.

`workerRun = Simulink.sdi.WorkerRun.getLatest` creates a `Simulink.sdi.WorkerRun` object of the most recent run.

Input Arguments

runID — Run identifier

integer

Run identifier

Methods

<code>getDataset</code>	Create Dataset of worker run data
<code>getDatasetRef</code>	Create DatasetRef for worker run
<code>getLatest</code>	Create worker run for latest run
<code>getLocalRun</code>	Create local run from worker run

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Access Data from a Parallel Simulation

This example executes parallel simulations of the model `slexAircraftExample` with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector API.

Generate Data with Parallel Simulations

Create a vector of filter parameter values to use a different value in each simulation. Then, run a parallel simulation, selecting a different filter value for each worker.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport(1)

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];

parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Change the filter time constant and simulate
    modelWorkspace = get_param('slexAircraftExample', 'modelworkspace');
    modelWorkspace.assignin('Ts', Ts_val)
    sim('slexAircraftExample')

    % Create a worker run for each simulation
```

```
        workerRun(index) = Simulink.sdi.WorkerRun.getLatest
    end
```

Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

```
ds(7) = Simulink.SimulationData.Dataset;
```

```
for a = 1:7
    ds(a) = workerRun(a).getDataset;
end
ds(1)
```

```
ans =
```

```
Simulink.SimulationData.Dataset '' with 2 elements
```

		Name	BlockPath
1	[1x1 Signal]	alpha, rad	...rcraftExample/Aircraft Dynamics Model
2	[1x1 Signal]	Stick	slexAircraftExample/Pilot

- Use braces { } to access, modify, or add elements using index.

Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the `getDatasetRef` method to reference the data associated with the `WorkerRun`.

```
for b = 1:7
    datasetRef(b) = workerRun(b).getDatasetRef;
end
datasetRef(1)
```

```
ans =
```

```
DatasetRef with properties:
```

```
    Name: 'Run 6: slexAircraftExample'
      Run: [1x1 Simulink.sdi.Run]
```

```
numElements: 2
```

Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local Run objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```
for c = 1:7
    Runs(c) = workerRun(c).getLocalRun;
    Ts_val_str = num2str(Ts_vals(c));
    desc = strcat('Ts = ', Ts_val_str);
    Runs(c).Description = desc;
    Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);
end
```

Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

```
Simulink.sdi.cleanupWorkerResources
```

Alternatives

You can also access, view, and analyze simulation data from Parallel Computing Toolbox workers using the Simulation Data Inspector UI.

See Also

```
Simulink.sdi.Run
```

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

getDataset

Class: Simulink.sdi.WorkerRun

Package: Simulink.sdi

Create Dataset of worker run data

Syntax

```
dataset = workerRun.getDataset  
dataset = workerRun.getDataset(domain)
```

Description

`dataset = workerRun.getDataset` returns a `Simulink.SimulationData.Dataset` object of the data corresponding to the `Simulink.sdi.WorkerRun` object.

`dataset = workerRun.getDataset(domain)` returns a `Simulink.SimulationData.Dataset` object of the data corresponding to the `Simulink.sdi.WorkerRun` object limited to the scope specified by `domain`.

Input Arguments

domain — Scope specifier

'signals' | 'outports'

Scope limiting argument that selects the data to return in the `Simulink.SimulationData.Dataset` object.

- 'signals' limits the data returned in the `Dataset` to signals in the `WorkerRun`.
- 'outports' limits the data returned in the `Dataset` to output data in the `WorkerRun`.

Output Arguments

dataset — **Simulink.SimulationData.Dataset** object

Simulink.SimulationData.Dataset object

Simulink.SimulationData.Dataset object containing the data from the Simulink.sdi.WorkerRun object.

Examples

Access Data from a Parallel Simulation

This example executes parallel simulations of the model `slexAircraftExample` with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector API.

Generate Data with Parallel Simulations

Create a vector of filter parameter values to use a different value in each simulation. Then, run a parallel simulation, selecting a different filter value for each worker.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport(1)

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];

parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Change the filter time constant and simulate
    modelWorkspace = get_param('slexAircraftExample', 'modelworkspace');
```

```
modelWorkspace.assignin('Ts', Ts_val)
sim('slexAircraftExample')

% Create a worker run for each simulation
workerRun(index) = Simulink.sdi.WorkerRun.getLatest
end
```

Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

```
ds(7) = Simulink.SimulationData.Dataset;
```

```
for a = 1:7
    ds(a) = workerRun(a).getDataset;
end
ds(1)
```

```
ans =
```

```
Simulink.SimulationData.Dataset '' with 2 elements
```

		Name	BlockPath
1	[1x1 Signal]	alpha, rad	...rcraftExample/Aircraft Dynamics Model
2	[1x1 Signal]	Stick	slexAircraftExample/Pilot

- Use braces { } to access, modify, or add elements using index.

Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the `getDatasetRef` method to reference the data associated with the `WorkerRun`.

```
for b = 1:7
    datasetRef(b) = workerRun(b).getDatasetRef;
end
datasetRef(1)
```

```
ans =
```

DatasetRef with properties:

```
Name: 'Run 6: slexAircraftExample'  
Run: [1x1 Simulink.sdi.Run]  
numElements: 2
```

Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local Run objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```
for c = 1:7  
    Runs(c) = workerRun(c).getLocalRun;  
    Ts_val_str = num2str(Ts_vals(c));  
    desc = strcat('Ts = ', Ts_val_str);  
    Runs(c).Description = desc;  
    Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);  
end
```

Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

```
Simulink.sdi.cleanupWorkerResources
```

See Also

[Simulink.SimulationData.Dataset](#) | [Simulink.sdi.WorkerRun.getDatasetRef](#)
| [Simulink.sdi.WorkerRun.getLatest](#) |
[Simulink.sdi.WorkerRun.getLocalRun](#)

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

getDatasetRef

Class: Simulink.sdi.WorkerRun

Package: Simulink.sdi

Create DatasetRef for worker run

Syntax

```
datasetRef = workerRun.getDatasetRef  
datasetRef = workerRun.getDatasetRef(domain)
```

Description

`datasetRef = workerRun.getDatasetRef` returns a `Simulink.SimulationData.DatasetRef` object referencing the data in the `Simulink.sdi.WorkerRun` object, `workerRun`.

`datasetRef = workerRun.getDatasetRef(domain)` returns a `Simulink.SimulationData.DatasetRef` object referencing the data in the `Simulink.sdi.WorkerRun` object, `workerRun`, limited to the scope specified by `domain`.

Input Arguments

domain — Scope limiting input

'signals' | 'outports'

Scope limiting argument that selects the data to reference in the `Simulink.SimulationData.DatasetRef` object.

- 'signals' limits the data referenced in the `DatasetRef` to signals in the `WorkerRun`.
- 'outports' limits the data referenced in the `DatasetRef` to output data in the `WorkerRun`.

Output Arguments

datasetRef — Simulink.SimulationData.DatasetRef object

Simulink.sdi.DatasetRef object

Simulink.sdi.DatasetRef object referencing the data in the Parallel Computing Toolbox worker run.

Examples

Access Data from a Parallel Simulation

This example executes parallel simulations of the model `slexAircraftExample` with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector API.

Generate Data with Parallel Simulations

Create a vector of filter parameter values to use a different value in each simulation. Then, run a parallel simulation, selecting a different filter value for each worker.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport(1)

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];

parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Change the filter time constant and simulate
    modelWorkspace = get_param('slexAircraftExample', 'modelworkspace');
```

```

modelWorkspace.assignin('Ts', Ts_val)
sim('slexAircraftExample')

% Create a worker run for each simulation
workerRun(index) = Simulink.sdi.WorkerRun.getLatest
end

```

Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

```
ds(7) = Simulink.SimulationData.Dataset;
```

```

for a = 1:7
    ds(a) = workerRun(a).getDataset;
end
ds(1)

```

```
ans =
```

```
Simulink.SimulationData.Dataset '' with 2 elements
```

		Name	BlockPath
1	[1x1 Signal]	alpha, rad	...rcraftExample/Aircraft Dynamics Model
2	[1x1 Signal]	Stick	slexAircraftExample/Pilot

- Use braces { } to access, modify, or add elements using index.

Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the `getDatasetRef` method to reference the data associated with the `WorkerRun`.

```

for b = 1:7
    datasetRef(b) = workerRun(b).getDatasetRef;
end
datasetRef(1)

```

```
ans =
```

DatasetRef with properties:

```
Name: 'Run 6: slexAircraftExample'
Run: [1x1 Simulink.sdi.Run]
numElements: 2
```

Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local Run objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```
for c = 1:7
    Runs(c) = workerRun(c).getLocalRun;
    Ts_val_str = num2str(Ts_vals(c));
    desc = strcat('Ts = ', Ts_val_str);
    Runs(c).Description = desc;
    Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);
end
```

Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

```
Simulink.sdi.cleanupWorkerResources
```

See Also

[Simulink.SimulationData.DatasetRef](#) | [Simulink.sdi.WorkerRun.getDataset](#)
| [Simulink.sdi.WorkerRun.getLatest](#) |
[Simulink.sdi.WorkerRun.getLocalRun](#)

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

Simulink.sdi.WorkerRun.getLatest

Class: Simulink.sdi.WorkerRun

Package: Simulink.sdi

Create worker run for latest run

Syntax

```
runObj = Simulink.sdi.WorkerRun.getLatest
```

Description

`runObj = Simulink.sdi.WorkerRun.getLatest` creates a `Simulink.sdi.WorkerRun` object for the latest run on a Parallel Computing Toolbox worker.

Output Arguments

runObj — Local `Simulink.sdi.Run` object

`Simulink.sdi.WorkerRun` object

`Simulink.sdi.WorkerRun` object to access the data from the latest Parallel Computing Toolbox worker run.

Examples

Access Data from a Parallel Simulation

This example executes parallel simulations of the model `slexAircraftExample` with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector API.

Generate Data with Parallel Simulations

Create a vector of filter parameter values to use a different value in each simulation. Then, run a parallel simulation, selecting a different filter value for each worker.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport(1)

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];

parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Change the filter time constant and simulate
    modelWorkspace = get_param('slexAircraftExample', 'modelworkspace');
    modelWorkspace.assignin('Ts', Ts_val)
    sim('slexAircraftExample')

    % Create a worker run for each simulation
    workerRun(index) = Simulink.sdi.WorkerRun.getLatest
end
```

Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

```
ds(7) = Simulink.SimulationData.Dataset;

for a = 1:7
    ds(a) = workerRun(a).getDataset;
end
ds(1)
```

```
ans =  
  
Simulink.SimulationData.Dataset '' with 2 elements  
  
      Name      BlockPath  
-----  
1  [1x1 Signal]  alpha, rad  ...rcraftExample/Aircraft Dynamics Model  
2  [1x1 Signal]  Stick      slexAircraftExample/Pilot  
  
- Use braces { } to access, modify, or add elements using index.
```

Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the `getDatasetRef` method to reference the data associated with the `WorkerRun`.

```
for b = 1:7  
    datasetRef(b) = workerRun(b).getDatasetRef;  
end  
datasetRef(1)
```

```
ans =  
  
DatasetRef with properties:  
  
    Name: 'Run 6: slexAircraftExample'  
    Run: [1x1 Simulink.sdi.Run]  
numElements: 2
```

Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local `Run` objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```
for c = 1:7  
    Runs(c) = workerRun(c).getLocalRun;  
    Ts_val_str = num2str(Ts_vals(c));  
    desc = strcat('Ts = ', Ts_val_str);  
    Runs(c).Description = desc;
```

```
Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);  
end
```

Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

```
Simulink.sdi.cleanupWorkerResources
```

See Also

[Simulink.sdi.WorkerRun](#) | [Simulink.sdi.WorkerRun.getDataset](#) |
[Simulink.sdi.WorkerRun.getDatasetRef](#) |
[Simulink.sdi.WorkerRun.getLocalRun](#)

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

getLocalRun

Class: Simulink.sdi.WorkerRun

Package: Simulink.sdi

Create local run from worker run

Syntax

```
runObj = workerRun.getLocalRun
```

Description

`runObj = workerRun.getLocalRun` creates the local `Simulink.sdi.Run` object `runObj` for the `Simulink.sdi.WorkerRun` object `workerRun`. Use `getLocalRun` in the client MATLAB to access the `Simulink.sdi.WorkerRun` data.

Output Arguments

runObj — Local `Simulink.sdi.Run` object

`Simulink.sdi.Run` object

Local `Simulink.sdi.Run` object.

Examples

Access Data from a Parallel Simulation

This example executes parallel simulations of the model `slexAircraftExample` with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector API.

Generate Data with Parallel Simulations

Create a vector of filter parameter values to use a different value in each simulation. Then, run a parallel simulation, selecting a different filter value for each worker.

```
% Make sure the Simulation Data Inspector is empty, and PCT support is
% enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport(1)

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];

parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % Load system and select signals to log
    load_system('slexAircraftExample')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
    Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

    % Change the filter time constant and simulate
    modelWorkspace = get_param('slexAircraftExample', 'modelworkspace');
    modelWorkspace.assignin('Ts', Ts_val)
    sim('slexAircraftExample')

    % Create a worker run for each simulation
    workerRun(index) = Simulink.sdi.WorkerRun.getLatest
end
```

Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

```
ds(7) = Simulink.SimulationData.Dataset;

for a = 1:7
    ds(a) = workerRun(a).getDataset;
end
ds(1)
```

```
ans =  
  
Simulink.SimulationData.Dataset '' with 2 elements  
  
      Name      BlockPath  
-----  
1 [1x1 Signal]  alpha, rad  ...rcraftExample/Aircraft Dynamics Model  
2 [1x1 Signal]  Stick      slexAircraftExample/Pilot  
  
- Use braces { } to access, modify, or add elements using index.
```

Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the `getDatasetRef` method to reference the data associated with the `WorkerRun`.

```
for b = 1:7  
    datasetRef(b) = workerRun(b).getDatasetRef;  
end  
datasetRef(1)
```

```
ans =  
  
DatasetRef with properties:  
  
    Name: 'Run 6: slexAircraftExample'  
    Run: [1x1 Simulink.sdi.Run]  
numElements: 2
```

Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local `Run` objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```
for c = 1:7  
    Runs(c) = workerRun(c).getLocalRun;  
    Ts_val_str = num2str(Ts_vals(c));  
    desc = strcat('Ts = ', Ts_val_str);  
    Runs(c).Description = desc;
```

```
    Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);  
end
```

Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

```
Simulink.sdi.cleanupWorkerResources
```

See Also

`Simulink.sdi.Run` | `Simulink.sdi.WorkerRun`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b

Simulink.Signal

Specify attributes of signal

Description

This class enables you to create workspace objects that you can use to assign or validate the attributes of a signal or discrete state, such as its data type, numeric type, dimensions, and so on.

You can use a signal object to:

- Assign values to signal attributes that are left unassigned (have a value of `-1` or `auto`) by the signal source.
- Validate signal attributes whose values are explicitly assigned by the signal source. Such attributes have values other than `-1` or `auto`. Successful validation guarantees that the signal has the attributes that you intended it to have.

You can create a `Simulink.Signal` object in the MATLAB workspace or in a model workspace.

Use signal objects to assign or validate signal or discrete state attributes by giving the signal or discrete state the same name as the workspace variable that references the `Simulink.Signal` object.

For more information about using signal objects, see “Use Simulink.Signal Objects to Specify and Control Signal Attributes” and “Data Objects”.

Creation

Create a `Simulink.Signal` object:

- By using the Model Data Editor. See “For Signals”.
- By using the Model Explorer. See “Create Data Objects from Built-In Data Class Package Simulink”.

- Directly from a signal properties dialog box or the Property Inspector in a model. See “Create Signal Object from Signal Properties Dialog Box”.
- By using the `Simulink.Signal` function, described below.

Syntax

```
signalObj = Simulink.Signal
```

Description

`signalObj = Simulink.Signal` returns a `Simulink.Signal` object with default property values.

Properties

For information about properties in the property dialog box of a `Simulink.Signal` object, see “Property Dialog Box”.

CoderInfo — Specifications for generating code for signal

`Simulink.CoderInfo` object

Information used by Simulink Coder for generating code for this signal. The value of this property is an object of `Simulink.CoderInfo` class.

For more information, see “Storage Classes for Signals and States” (Simulink Coder) and “Simulink Package Custom Storage Classes” (Embedded Coder).

Complexity — Numeric complexity of signal

'auto' (default) | 'real' | 'complex'

Character vector specifying the numeric type of this signal. Valid values are 'auto' (determined by Simulink), 'real', or 'complex'.

Data Types: char

Data Type — Data type of signal

'auto' (default) | character vector

Character vector specifying the data type of this signal.

The default value, 'auto', specifies that Simulink should determine the data type. You can specify a built-in data type (for example, 'uint8' or 'single') or a custom data type. To specify a custom data type, use a MATLAB expression that specifies the type, (for example, the name of a `Simulink.NumericType` object that you create in the base workspace).

To specify a bus object as the data type for the signal object, use the 'Bus: <object_name>' syntax. See “Bus Support” for details about what you need to do if you specify a bus object as the data type.

Example: 'auto'

Example: 'int8'

Example: 'fixdt(1,16,5)'

Example: 'myAliasTypeObject'

Example: 'Enum: myEnumType'

Example: 'Bus: myBusObject'

Data Types: char

Description — Custom description of signal

' ' (empty character vector) (default) | character vector

Description of this signal. This field is intended for use in documenting this signal.

This property is used by the Simulink Report Generator and for code generation.

If you have an Embedded Coder license, you can add the signal description as a comment for the variable declaration in generated code:

- Specify a storage class for the signal object other than `Auto`.
- On the **Code Generation > Comments** pane of the model Configuration Parameters dialog box, select the model configuration parameter **Simulink data object descriptions**. For more information, see “Simulink data object descriptions” (Simulink Coder).

Example: 'This signal represents the rotation speed of the engine.'

Data Types: char

Dimensions — Dimensions of signal

-1 (default) | row vector | character vector

Scalar or vector specifying the dimensions of this signal.

Valid values are -1 (the default) specifying any dimensions, N specifying a vector signal of size N, or [M N] specifying an M×N matrix signal.

To use symbolic dimensions, specify a character vector.

Example: [1 3]

Example: '[1 myDimParam]'

Data Types: double | char

Max — Maximum value of signal

[] (empty) (default) | real double scalar

Maximum value that this signal can have.

The default value is [] (unspecified). Specify a finite, real, double, scalar value.

Note If you specify a bus object as the data type for a signal, do not set the maximum value for bus data on the signal property dialog box. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`.

Simulink uses this value in the following ways:

- When updating the diagram or starting a simulation, Simulink generates an error if the initial value of the signal is greater than the maximum value or if the maximum value is outside the range of the data type of the signal.
- When you enable the **Simulation range checking** diagnostic, Simulink alerts you during simulation if the signal value is greater than the maximum value (see “Simulation range checking”).

Example: 5.32

Data Types: double

Min — Minimum value of signal

[] (empty) (default) | real double scalar

Minimum value that this signal can have.

The default value is [] (unspecified). Specify a finite, real, double, scalar value.

Note If you specify a bus object as the data type for a signal, do not set the minimum value for bus data on the signal property dialog box. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see `Simulink.BusElement`.

Simulink uses this value in the following ways:

- When updating the diagram or starting a simulation, Simulink generates an error if the signal's initial value is less than the minimum value or if the minimum value is outside the range for the data type of the signal.
- When you enable the **Simulation range checking** diagnostic, Simulink alerts you during simulation if the signal value is less than the minimum value (see “Simulation range checking”).

Example: `-0.92`

Data Types: `double`

InitialValue — Initial value of signal or state

' ' (empty character vector) (default) | character vector

Signal or state value before a simulation takes its first time step.

You can specify any MATLAB expression, including the name of a workspace variable, that evaluates to a numeric scalar value or array.

At the command prompt or in a script, even if you use a number, specify the initial value as a character vector.

```
mySigObject.InitialValue='5.3';
```

```
mySigObject.InitialValue = 'myNumericVariable';
```

To specify an initial value for a signal that uses a numeric data type other than double, cast the initial value to the signal data type. For example, you can specify `'single(73.3)'` to use 73.3 as the initial value for a signal of data type `single`.

If you use a bus object as the data type for the signal object, set `InitialValue` to a character vector containing either 0 or a MATLAB structure that matches the bus object. See “Bus Support” for details.

If the initial value evaluates to a MATLAB structure, then in the **Configuration Parameters** dialog box, set “Underspecified initialization detection” to `Simplified`.

If necessary, Simulink converts the initial value to ensure type, complexity, and dimension consistency with the corresponding block parameter value. If you specify an invalid value or expression, an error message appears when you update the model. Also, Simulink performs range checking of the initial value. The software alerts you when the initial value of the signal lies outside a range that corresponds to its specified minimum and maximum values and data type.

Classic initialization mode: In this mode, initial value settings for signal objects that represent the following signals and states override the corresponding block parameter initial values if undefined (specified as `[]`):

- Output signals of conditionally executed subsystems and Merge blocks
- Block states

Simplified initialization mode: In this mode, initial values of signal objects associated with the following blocks are ignored. The initial values of the corresponding blocks are used instead.

- Output blocks of conditionally executed subsystems
- Merge blocks

Example: `'15.23'`

Example: `'myInitParam'`

Data Types: `char`

sampleTime — Sample time of signal

-1 (default) | `double` scalar or array

Rate at which this signal should be updated.

See “Specify Sample Time” for details.

Example: `0.001`

Example: `2`

Data Types: `double`

Unit — Physical unit of signal value

`' '` (default) | valid unit

Physical unit used for expressing this signal value (for example, inches).

For more information, see “Unit Specification in Simulink Models”.

Example: `'degC'`

Data Types: `char`

Examples

Simulink.Signal Examples

For examples that show how to use `Simulink.Signal` objects, see “Use `Simulink.Signal` Objects to Specify and Control Signal Attributes” and “Data Objects”.

See Also

`AUTOSAR.Signal` | `Simulink.CoderInfo` | `Simulink.Parameter`

Topics

“Determine Where to Store Variables and Objects for Simulink Models”

“Control Signal Data Types”

“Control Signals and States in Code by Applying Storage Classes” (Simulink Coder)

“Define Data Classes”

“Signal Basics”

“Data Objects”

“Data Types Supported by Simulink”

“MPT Data Object Properties” (Embedded Coder)

Introduced before R2006a

Simulink.SimulationData.BlockPath

Fully specified Simulink block path

Description

Simulink creates block path objects when creating dataset objects for signal logging and data store logging. `Simulink.SimulationData.Signal` and `Simulink.SimulationData.DataStoreMemory` objects include block path objects.

You can create a block path that you can use with the `Simulink.SimulationData.Dataset.getElement` method to access a specific dataset element. If you want to create a dataset in MATLAB to use as a baseline to compare against a signal logging or data store logging dataset, then you need to create the block paths as part of that dataset.

The `Simulink.SimulationData.BlockPath` class is very similar to the `Simulink.BlockPath` class.

You do not have to have Simulink installed to use the `Simulink.SimulationData.BlockPath` class. However, you must have Simulink installed to use the `Simulink.BlockPath` class. If you have Simulink installed, consider using `Simulink.BlockPath` instead of `Simulink.SimulationData.BlockPath`, because the `Simulink.BlockPath` class includes a method for checking the validity of block path objects without you having to update the model diagram.

Property Summary

Name	Description
SubPath on page 5-184	Individual component within the block specified by the block path

Method Summary

Name	Description
BlockPath on page 5-184	Create a block path.
convertToCell on page 5-187	Convert a block path to a cell array of character vectors.
getBlock on page 5-188	Get a single block path in the model reference hierarchy.
getLength on page 5-189	Get the length of the block path.

Properties

SubPath

Represents an individual component within the block specified by the block path.

For example, if the block path refers to a Stateflow chart, you can use SubPath to indicate the chart signals. For example:

```
Block Path:
    'sf_car/shift_logic'

SubPath:
    'gear_state.first'
```

character vector

RW

Methods

BlockPath

Create block path

```
blockpath_object = Simulink.SimulationData.BlockPath()  
blockpath_object = Simulink.SimulationData.BlockPath(blockpath)  
blockpath_object = Simulink.SimulationData.BlockPath(paths)  
blockpath_object = Simulink.SimulationData.BlockPath(paths, subpath)
```

blockpath

The block path object that you want to copy.

paths

A character vector or cell array of character vectors that Simulink uses to build the block path.

Specify each character vector in order, from the top model to the specific block for which you are creating a block path.

Each character vector must be a path to a block within the Simulink model. The block must be:

- A block in a single model
- A Model block (except for the last character vector, which may be a block other than a Model block)
- A block that is in a model that is referenced by a Model block that is specified in the previous character vector

subpath

A character vector that represents an individual component within a block.

blockpath_object

The block path that you create.

`blockpath_object = Simulink.SimulationData.BlockPath()` creates an empty block path.

`blockpath_object = Simulink.SimulationData.BlockPath(blockpath)` creates a copy of the block path of the block path object that you specify with the `source_blockpath` argument.

`blockpath = Simulink.SimulationData.BlockPath(paths)` creates a block path from the character vector or cell array of character vectors that you specify with the `paths` argument. Each character vector represents a path at a level of model hierarchy.

`blockpath = Simulink.SimulationData.BlockPath(paths, subpath)` creates a block path from the character vector or cell array of character vectors that you specify with the `paths` argument and creates a path for the individual component (for example, a signal) of the block.

Create a block path object called `bp1`, using a cell array of character vectors representing elements of the block path.

```
bp1 = Simulink.SimulationData.BlockPath(...
{'sldemo_mdref_depgraph/thermostat', ...
'sldemo_mdref_heater/Fahrenheit to Celsius', ...
'sldemo_mdref_F2C/Gain1'})
```

The resulting block path reflects the model reference hierarchy for the block path.

```
bp1 =

    Simulink.BlockPath
    Package: Simulink

    Block Path:
        'sldemo_mdref_depgraph/thermostat'
        'sldemo_mdref_heater/Fahrenheit to Celsius'
        'sldemo_mdref_F2C/Gain1'
```

convertToCell

Convert block path to cell array of character vectors

```
cellarray = Simulink.SimulationData.BlockPath.convertToCell()
```

```
cellarray
```

The cell array of character vectors representing the elements of the block path.

`cellarray = Simulink.SimulationData.BlockPath.convertToCell()` converts a block path to a cell array of character vectors.

```
bp1 = Simulink.SimulationData.BlockPath(...
    {'sldemo_mdldref_depgraph/thermostat', ...
    'sldemo_mdldref_heater/Fahrenheit to Celsius', ...
    'sldemo_mdldref_F2C/Gain1'})
cellarray_for_bp1 = bp1.convertToCell()
```

The result is a cell array representing the elements of the block path.

```
cellarray_for_bp1 =
    'sldemo_mdldref_depgraph/thermostat'
    'sldemo_mdldref_heater/Fahrenheit to Celsius'
    'sldemo_mdldref_F2C/Gain1'
```

getBlock

Get single block path in model reference hierarchy

```
block = Simulink.SimulationData.BlockPath.getBlock(index)
```

`index`

The index of the block for which you want to get the block path. The index reflects the level in the model reference hierarchy. An index of 1 represents a block in the top-level model, an index of 2 represents a block in a model referenced by the block of index 1, and an index of n represents a block that the block with index $n-1$ references.

`block`

The block representing the level in the model reference hierarchy specified by the `index` argument.

`blockpath = Simulink.SimulationData.BlockPath.getBlock(index)` returns the block path of the block specified by the `index` argument.

Get the block for the second level in the model reference hierarchy.

```
bp1 = Simulink.SimulationData.BlockPath(...
{'sldemo_mdref_depgraph/thermostat', ...
'sldemo_mdref_heater/Fahrenheit to Celsius', ...
'sldemo_mdref_F2C/Gain1'})
blockpath = bp1.getBlock(2)
```

The result is the thermostat block, which is at the second level in the block path hierarchy.

```
blockpath =
sldemo_mdref_heater/Fahrenheit to Celsius
```

getLength

Get length of block path

```
length = Simulink.SimulationData.BlockPath.getLength()
```

```
length
```

The length of the block path. The length is the number of levels in the model reference hierarchy.

`length = Simulink.SimulationData.BlockPath.getLength()` returns a numeric value that corresponds to the number of levels in the model reference hierarchy for the block path.

Get the length of block path `bp1`.

```
bp1 = Simulink.SimulationData.BlockPath(...
{'sldemo_mdref_depgraph/thermostat', ...
```

```
'sldemo_mdref_heater/Fahrenheit to Celsius', ...  
'sldemo_mdref_F2C/Gain1'})  
length_bp1 = bp1.getLength()
```

The result reflects that the block path has three elements.

```
length_bp1 =  
  
    3
```

See Also

[Simulink.BlockPath](#) | [Simulink.SimulationData.Dataset](#)

Introduced in R2012b

Simulink.SimulationData.Dataset class

Package: Simulink.SimulationData

Create Simulink.SimulationData.Dataset object

Description

Simulink creates `Simulink.SimulationData.Dataset` objects to store data elements when:

- Performing signal logging, which use the `Dataset` format
- Logging states or outputs, if you use the default format of `Dataset`.
- Logging a data store

Using the `Dataset` format for state and output logging offers several advantages compared to `Array`, `Structure`, or `Structure` with `time`. For details, see “Format for State Information Saved Without `SimState`”.

To generate a `Simulink.SimulationData.Dataset` object from the root-level `Inport` blocks in a model, you can use the `createInputDataset` function. Signals in the generated dataset have the properties of the `Inport` blocks and the corresponding ground values at model start and stop times. You can create `timeseries` and `timetable` objects for the time and values for signals for which you want to load data for simulation. The other signals use ground values.

You can use curly braces (`{}`) to streamline indexing syntax to access, set, and add elements in a dataset, instead of using `get`, `getElement`, `setElement`, or `addElement` methods. To get or set an element using curly braces, the index must be a scalar that is not greater than the number of elements in the dataset variable. To add an element, the index must be a scalar that is greater than the total number of elements in the dataset by one. The `get`, `getElement`, `setElement`, or `addElement` methods support specifying an element by name or block path, as well as by index.

For individual non-bus signal data, you can specify these types of data for `Dataset` elements:

- `timeseries`
- `timetable`
- `matlab.io.datastore.SimulationDatastore`
- double vectors or structure of double data
- `timeseries`
- a `Simulink.SimulationData.Signal`, `Simulink.SimulationData.State`, or `Simulink.SimulationData.DataStoreMemory` object

For bus signals, use a structure with a data element for each leaf signal, using one of these formats:

- A MATLAB `timeseries` object
- A MATLAB `timetable` object
- A `matlab.io.datastore.SimulationDatastore` object
- An empty matrix
- An array that meets one of these requirements:
 - An array with time in the first column and the remaining columns each corresponding to an input port. See “Create Data Arrays for Root-Level Inputs”.
 - An $n \times 1$ array for a root inport that drives a function-call subsystem.
- Another structure, with data elements for each signal that are consistent with these requirements for a structure for bus data

Variable-size signals are not supported for `Dataset` data values.

Construction

`convertedDataset = Simulink.SimulationData.Dataset (loggedDataToConvert)` converts the `loggedDataToConvert` to a `Simulink.SimulationData.Dataset` object. You can then use the `Simulink.SimulationData.Dataset.concat` method to combine elements of two `Dataset` objects.

`constructedDataset = Simulink.SimulationData.Dataset (variableName, 'DatasetName', 'dsname')` constructs a

Simulink.SimulationData.Dataset object, adds variable `variableName`, and names the data set `dsname`.

Input Arguments

loggedDataToConvert — Data element to convert

character vector

Data element to convert to a data set, specified as a character vector. You can convert elements such as:

- Array
- Structure

Note Structure inputs cannot be arrays or matrices.

- Structure with time
- MATLAB time series
- Structure of MATLAB time-series elements
- ModelDataLogs

variableName — Variable to add to data set

character vector

Variable to add to data set, specified as a character vector.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'DatasetName', 'dsname'

DatasetName — Data set name

character vector

Data set name, specified as a character vector.

Output Arguments

convertedDataset — Converted data set

Simulink.SimulationData.Dataset object

Converted data set, returned as a Simulink.SimulationData.Dataset object.

constructedDataset — Constructed data set

Simulink.SimulationData.Dataset object

Constructed data set, returned as a Simulink.SimulationData.Dataset object.

Properties

Name — Name of the data set

same as the logging variable (default) | character vector

Name of the data set, specified as a character vector or logging variable (for example, `logout` for signal logging). Specify a name when you want to distinguish easily one data set from another. For example, you could reset the name when comparing multiple simulations. This property is read/write.

```
ds = Simulink.SimulationData.Dataset
ds.Name = 'Dataset1'
```

Total Elements — Total number of elements

double

Total number of elements in data set, specified as a double. This property is read only. To get this value, use the `Simulink.SimulationData.Dataset.numElements` method.

Methods

<code>addElement</code>	Add element to end of data set
<code>concat</code>	Concatenate dataset to another dataset
<code>get</code>	Get element or collection of elements from dataset
<code>getElementNames</code>	Return names of all elements in dataset
<code>find</code>	Get element or collection of elements from dataset
<code>numElements</code>	Get number of elements in data set
<code>plot</code>	Plot dataset elements in Signal Preview window or Simulation Data Inspector
<code>setElement</code>	Change element stored at specified index

Tip To get the names of `Dataset` variables in the MAT-file, using the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function processes faster than using the `who` or `whos` functions.

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Concatenate Dataset `ds1` to Dataset `ds`

Convert data from two To Workspace blocks, convert to `Dataset` format, and concatenate them. `myvdp` is the `vdp` model with two To Workspace blocks with variables named `simout` and `simout1`. These blocks log data in time-series format.

```
mdl = 'myvdp';
open_system(mdl);
sim(mdl)
ds = Simulink.SimulationData.Dataset(simout);
```

```
ds1 = Simulink.SimulationData.Dataset(simout1);  
dsfinal = concat(ds,ds1)
```

Access, Change, and Add Dataset Elements

Use curly brace indexing syntax to work with a `logouts` signal logging dataset that has three elements.

Get the second element of the `logouts` dataset.

```
logouts{2}
```

Change the name of the third element.

```
logouts{3}.Name = 'thirdSignal'
```

Add a fourth element.

```
time = 0.1*(0:100)';  
element4 = Simulink.SimulationData.Signal;  
element4.Name = 'C';  
element4.Values = timeseries(3*sin(time),time);  
logouts{4} = element4
```

- “Convert Logged Data to Dataset Format”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Load Big Data for Simulations”

See Also

```
Simulink.ModelDataLogs | Simulink.SimulationData.BlockPath |  
Simulink.SimulationData.DataStoreMemory |  
Simulink.SimulationData.Dataset.addElement |  
Simulink.SimulationData.Dataset.concat |  
Simulink.SimulationData.Dataset.get |  
Simulink.SimulationData.Dataset.getElementNames |  
Simulink.SimulationData.Dataset.numElements |
```

```
Simulink.SimulationData.Dataset.plot |  
Simulink.SimulationData.Dataset.setElement |  
Simulink.SimulationData.DatasetRef |  
Simulink.SimulationData.DatasetRef.getDatasetVariableNames |  
Simulink.SimulationData.Signal | createInputDataset | loadIntoMemory
```

Topics

“Convert Logged Data to Dataset Format”

“Export Signal Data Using Signal Logging”

“Log Data Stores”

“Migrate Scripts That Use Legacy ModelDataLogs API”

“Load Big Data for Simulations”

Introduced in R2011a

Simulink.SimulationData.DatasetRef class

Package: Simulink.SimulationData

Create `Simulink.SimulationData.DatasetRef` object

Description

To use a reference for accessing a `Simulink.SimulationData.Dataset` object stored in a MAT-file, create a `Simulink.SimulationData.DatasetRef` object. You can use this reference to avoid running out of memory, by retrieving data signal by signal, for data that you log to persistent storage. You can stream a `DatasetRef` object into a root-level input port or you can use it to create a `SimulationDatastore` object to use for streaming. For details, see “Load Big Data for Simulations”.

For parallel simulations, for which you specify an array of `Simulink.SimulationInput` objects, if you are logging to file, Simulink:

- Creates `Simulink.SimulationData.DatasetRef` objects to access output data in the MAT-file and includes those objects in the `SimulationOutput` object data
- Enables the `CaptureErrors` argument for simulation

Construction

`datasetRefObj = Simulink.SimulationData.DatasetRef(location, identifier)` creates a reference to the contents of a `Simulink.SimulationData.Dataset` variable stored in a MAT-file.

Input Arguments

location — MAT-file containing `Simulink.SimulationData.Dataset` object to reference

character vector

MAT-file containing `Simulink.SimulationData.Dataset` object to reference, specified as a character vector. The character vector is a path to the MAT-file. Do not use a file name from one locale in a different locale.

identifier — Name of variable in MAT-file

character vector

Name of a `Simulink.SimulationData.Dataset` variable in MAT-file, specified as a character vector. When you log to persistent storage, Simulink uses the variable names specified for each kind of logging.

Suppose that you use the default variable name for signal logging (`logouts`) and default MAT-file name for persistent storage logging (`mat.out`). After you simulate the model, then to create a reference to the `Dataset` object for signal logging, at the MATLAB command line, enter:

```
sigLogRef = Simulink.SimulationData.DatasetRef('out.mat','logouts');
```

Output Arguments

datasetRefObj — Reference to Dataset object

`Simulink.SimulationData.DatasetRef` object

Reference to logging dataset, returned as a `Simulink.SimulationData.DatasetRef` object.

Properties

Location — MAT-file containing `Simulink.SimulationData.Dataset` object to reference

character vector

MAT-file containing `Simulink.SimulationData.Dataset` object to reference, specified as a character vector. The character vector is a path to the MAT-file. Include the `.mat` extension in the file name. Do not use a file name from one locale in a different locale.

Identifier — Name of variable in MAT-file

character vector

Name of a `Simulink.SimulationData.Dataset` variable in MAT-file, specified as a character vector. When you log to persistent storage, Simulink uses the variable names specified for each kind of logging (for example, 'logout' for signal logging data).

Methods

Use the `numElements`, `getElement`, and `getElementNames` methods for a `Simulink.SimulationData.DatasetRef` object the same way that you use those methods for a `Simulink.SimulationData.Dataset` object.

Method	Purpose
<code>numElements</code>	Get number of elements from dataset
<code>getElementNames</code>	Return names of all elements in dataset
<p><code>get</code></p> <p>The <code>get</code> method is an alias for the <code>getElement</code> method.</p> <hr/> <p>Note You can use curly braces to streamline indexing syntax to access elements in a dataset reference, instead of using <code>get</code> or <code>getElement</code> methods. To get an element using curly braces, the index must be a scalar that is not greater than the number of elements in the variable. The <code>get</code> and <code>getElement</code> methods support specifying an element by name or block path, as well as by index.</p>	Get element from dataset
<code>getAsDatastore</code>	Get <code>matlab.io.datastore.SimulationDatastore</code> representation of element from a <code>DatasetRef</code> object
<code>plot</code>	Plot elements from dataset in Signal Preview window

Tip To get the names of `Dataset` variables in the MAT-file, using the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function processes faster than using the `who` or `whos` functions.

Copy Semantics

You can copy `DatasetRef` object properties by value. However, copying the `DatasetRef` object produces a handle object. Copying the `DatasetRef` object does not copy the data in the MAT-file that the object references. For details about copy operations, see [Copying Objects \(MATLAB\)](#) in the MATLAB documentation.

Examples

Create References to Persistent Storage Dataset Objects

This example shows how to construct and use `Simulink.SimulationData.DatasetRef` objects to access data for a model that logs to persistent storage. This simple example shows the basic steps for logging to persistent storage. This example does not represent a realistic situation for logging to persistent storage, because it shows a short simulation with small memory requirements.

Open the `vdp` model.

In the **Configuration Parameters > Data Import/Export** pane, select these parameters:

- **States**
- **Log Dataset data to file**

Set the **Format** parameter to `Dataset`.

Leave the other parameter settings as they are and click **Apply**.

In the model, click a signal and from the action bar, select **Enable Data Logging**.

Simulate the model.

Get a list of `Dataset` variable names in the `out.mat` file.

```
varNames = Simulink.SimulationData.DatasetRef.getDatasetVariableNames('out.mat')  
  
varNames =  
  
    1x2 cell array  
  
    'logout'    'xout'
```

Create a reference to the logged states data that is stored in `out.mat`. The variable for the logged states data is `xout`.

```
statesLogRef = Simulink.SimulationData.DatasetRef('out.mat','xout')  
  
statesLogRef =  
  
    Simulink.SimulationData.DatasetRef  
    Characteristics:  
        Location: out.mat (/my_files/out.mat)  
        Identifier: xout  
  
    Resolved Dataset: 'xout' with 2 elements  
  
        Name  BlockPath  
    _____  _____  
    1  ' '      vdp/x1  
    2  ' '      vdp/x2
```

Create a reference to the signal logging data that is stored in `out.mat`. The variable for the signal logging data is `logout`.

```
sigLogRef = Simulink.SimulationData.DatasetRef('out.mat','logout')  
  
sigLogRef =  
  
    Simulink.SimulationData.DatasetRef  
    Characteristics:  
        Location: out.mat (/my_files/out.mat)  
        Identifier: logout  
  
    Resolved Dataset: 'logout' with 1 element  
  
        Name  BlockPath  
    _____  _____  
    1  x1      vdp/x1
```

Use the `numElements` to access the number of elements in the logged states dataset.

```
statesLogRef.numElements

ans =

     2
```

Use the `DatasetRef` to access the first element of the signal logging dataset.

```
sigLogRef{1}

ans =

    Simulink.SimulationData.Signal
    Package: Simulink.SimulationData

    Properties:
        Name: 'x1'
    PropagatedName: ''
        BlockPath: [1x1 Simulink.SimulationData.BlockPath]
        PortType: 'outport'
        PortIndex: 1
        Values: [1x1 timeseries]
```

Methods, Superclasses

Delete the persistent storage MAT-file and try to use one of the `DatasetRef` objects.

```
delete('out.mat');
statesLogRef.get(1)

File does not exist.
```

The `statesLogRef` still exists, but it is a reference to a `Dataset` object that is in a file that no longer exists.

- “Log Data to Persistent Storage”
- “Load Big Data for Simulations”
- “Convert Logged Data to Dataset Format”

See Also

`Simulink.SimulationData.Dataset` |
`Simulink.SimulationData.DatasetRef.getDatasetVariableNames` |
`matlab.io.datastore.SimulationDatastore`

Topics

“Log Data to Persistent Storage”
“Load Big Data for Simulations”
“Convert Logged Data to Dataset Format”

Introduced in R2016a

Simulink.SimulationData.DataStoreMemory

Container for data store logging information

Description

Simulink uses `Simulink.SimulationData.DataStoreMemory` objects to store logging information from Data Store Memory blocks during simulation. The objects contain information about the blocks that write to the data store.

Property Summary

Name	Description
BlockPath on page 5-519	Location of Data Store Memory block for the logged data store
DSMWriterBlockPaths on page 5-520	Location of Data Store Write blocks that write to the data store
DSMWriters on page 5-520	Data Store Write blocks for each signal value
Name on page 5-520	Name of the data store dataset
Scope on page 5-521	Scope of the data store: 'local' or 'global'
Values on page 5-521	Time and data that were logged

Properties

BlockPath

Location of Data Store Memory block for the logged data store.

character vector

RW

DSMWriterBlockPaths

Location of blocks that write to the data store. Each element of the array contains the full block path of one writer block.

Vector of `Simulink.SimulationData.BlockPath` objects

RO

DSMWriters

The number of writes in the data store.

The `nth` element of `DSMWriters` contains the index of the element in `DSMWriterBlockPaths` that contains the block path of the writer that performed the `nth` write to `Values`.

Integer vector

RO

Name

Name of the data store dataset

character vector

RO

Scope

Scope of the data store: 'local' or 'global'

character vector

RW

Values

Time and data that were logged

MATLAB timeseries

RW

See Also

[Data Store Memory](#) | [Data Store Write](#) | [Simulink.SimulationData.BlockPath](#) | [Simulink.SimulationData.Dataset](#)

Topics

“Log Data Stores”

Simulink.SimulationData.LoggingInfo

Signal logging override settings

Description

This object specifies a set of signal logging override settings.

Use a `Simulink.SimulationData.LoggingInfo` object to specify the signal logging override settings for a signal. You can use this object for the `LoggingInfo` property of a `Simulink.SimulationData.SignalLoggingInfo` object.

Property Summary

Name	Description
<code>DataLogging</code> on page 5-523	Signal logging mode.
<code>NameMode</code> on page 5-523	Source of signal logging name.
<code>LoggingName</code> on page 5-523	Custom signal logging name.
<code>DecimateData</code> on page 5-524	Use subset of sample points.
<code>Decimation</code> on page 5-524	Decimation value (n): Simulink logs every nth data point.
<code>LimitDataPoints</code> on page 5-524	Limit number of data points to log.
<code>MaxPoints</code> on page 5-524	Maximum number of data points to log (N). The set of logged data points is the last N data points generated by the simulation.

Method Summary

Name	Description
<code>LoggingInfo</code> on page 5-525	Create a set of signal logging override settings for a signal.

Properties

DataLogging

Signal logging mode.

Indicates whether logging is enabled for this signal.

logical value — {true} | false

RW

NameMode

Source of signal logging name.

Indicates whether the signal logging name is a custom name ('true') or whether the signal logging name is the same as the signal name ('false').

logical value — true | {false}

RW

LoggingName

Custom signal logging name

The custom signal logging name to use for this signal, if the `NameMode` property is `true`.

character vector

RW

DecimateData

Log a subset of sample points, selecting data points at a specified interval. The first sample point is always logged.

logical value — true | {false}

RW

Decimation

Decimation value (*n*). If the `DecimateData` property is true, then Simulink logs every *n*th data point.

positive integer

RW

LimitDataPoints

Limit the number of data points to log.

logical value — true | {false}

RW

MaxPoints

Maximum number of data points to log (*N*). If the `LimitDataPoints` property is true, then the set of logged data points includes the last *N* data points generated by the simulation.

positive integer

RW

Methods

LoggingInfo

Create a `Simulink.SimulationData.LoggingInfo` object.

```
logging_info_object = Simulink.SimulationData.LoggingInfo()  
logging_info_object = Simulink.SimulationData.LoggingInfo(object)
```

`object`

A signal logging override settings object whose property values the constructor uses for the new `Simulink.SimulationData.LoggingInfo` object. The signal logging override object that you specify must be one of the following types of objects:

- `Simulink.SimulationData.LoggingInfo` object
- `Simulink.LoggingInfo` object

`logging_info_object`

A `Simulink.SimulationData.LoggingInfo` object.

`logging_info_object = Simulink.SimulationData.LoggingInfo()` creates a `Simulink.SimulationData.LoggingInfo` object that has default property values.

`logging_info_object = Simulink.SimulationData.LoggingInfo(object)` creates a `Simulink.SimulationData.LoggingInfo` object that copies the property values from the signal logging override object that you specify with the `object` argument.

The following example creates a `Simulink.SimulationData.LoggingInfo` object with default settings, changes the `DecimateData` and `Decimation` properties, and uses the object for the `LoggingInfo` property of a `Simulink.SimulationData.SignalLoggingInfo` object `mi`.

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'ex_mdref_cou
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'ex_bus_loggin
log_info = Simulink.SimulationData.LoggingInfo();
log_info.DecimateData = true;
log_info.Decimation = 2;
mi = Simulink.SimulationData.SignalLoggingInfo('ex_bus_logging');
mi.LoggingInfo = log_info
```

```
Simulink.SimulationData.SignalLoggingInfo
  Package: Simulink.SimulationData
```

```
  BlockPath:
    'ex_bus_logging'
```

```
  OutputPortIndex: 1
```

```
  LoggingInfo:
    DataLogging: 1
    NameMode: 0
    LoggingName: ''
    DecimateData: 1
    Decimation: 2
    LimitDataPoints: 0
    MaxPoints: 5000
```

See Also

[Simulink.ModelDataLogs](#) | [Simulink.SimulationData.BlockPath](#) |
[Simulink.SimulationData.DataStoreMemory](#) |
[Simulink.SimulationData.ModelLoggingInfo](#) |
[Simulink.SimulationData.Signal](#) |
[Simulink.SimulationData.SignalLoggingInfo](#)

Topics

“Override Signal Logging Settings from MATLAB”
“Migrate Scripts That Use Legacy ModelDataLogs API”

“Export Signal Data Using Signal Logging”
“Log Data Stores”

Introduced in R2012b

Simulink.SimulationData.ModelLoggingInfo

Signal logging override settings for a model

Description

This class is a collection of `Simulink.SimulationData.SignalLoggingInfo` objects that specify all signal logging override settings for a model.

Use methods and properties of this class to:

- Turn off logging for a signal or a Model block.
- Change logging settings for any signals that are already marked for logging within a model.

You can control whether a top-level model and referenced models use override signal logging settings or use the signal logging settings specified by the model. Use the `LoggingMode` and `LogAsSpecifiedByModels` properties to control which logging settings to apply.

Logging Mode for Models	Property Settings
For top-level model and all referenced models, use logging settings specified in the model.	Set <code>LoggingMode</code> to <code>LogAllAsSpecifiedInModel</code> .
For top-level model and all referenced models, use override signal logging settings.	Set <code>LoggingMode</code> to <code>OverrideSignals</code> .
For top-level model and referenced models, use a mix of override signal logging settings and the signal logging settings specified in the model.	Set <code>LoggingMode</code> to <code>OverrideSignals</code> . In the <code>LogAsSpecifiedByModels</code> cell array, include the models that you do not want to use the override signal logging settings.

For more information and examples, see “Override Signal Logging Settings from MATLAB”.

Property Summary

Name	Description
LoggingMode on page 5-529	Signal logging override status
LogAsSpecifiedByModels on page 5-530	Source of signal logging settings for the top-level model or a top-level Model block
Signals on page 5-531	All signals that have signal override settings

Method Summary

Name	Description
findSignal on page 5-536	Find signals within the Signals vector, using block path and output port index.
verifySignalAndModelPaths on page 5-539	Verify signal and model paths for the model signal logging override object.
getLogAsSpecifiedInModel on page 5-537	Determine whether the model logs signals as specified in the model or uses override settings.
setLogAsSpecifiedInModel on page 5-538	Set the logging mode for the top-level model or a top-level Model block.
createFromModel on page 5-531	Create and populate a model signal logging override object with all logged signals in the model reference hierarchy.
ModelLoggingInfo on page 5-534	Set signals to log or override logging settings.

Properties

LoggingMode

Signal logging override status. Values are:

- `OverrideSignals` — (Default) Uses the logging settings for signals, as specified in the `Signals` property. For models where `getLogAsSpecifiedInModel` is:

- `true` — Logs all signals, as specified in the model.
- `false` — Logs only the signals specified in the `Signals` property.
- `LogAllAsSpecifiedInModel` — Logs signals in the top-level model and all referenced models, as specified in the model. Simulink honors the signal logging indicators (blue antennae) and ignores the `Signals` property.

To change the logging mode for the top-level model or for a given reference model, use the `setLogAsSpecifiedInModel` method.

character array

RW

LogAsSpecifiedByModels

When `LoggingMode` is set to `'OverrideSignals'`, this cell array specifies whether the top-level model or a top-level Model block logs all signals based on the signal logging settings defined in the model.

- For the top-level model and top-level Model blocks that the cell array includes, Simulink ignores the `Signals` property overrides.
- For a model or Model block that the cell array does *not* include, Simulink uses the `Signals` property to determine which signals to log.

When `LoggingMode` is set to `'LogAllAsSpecifiedInModel'`, Simulink ignores the `LogAsSpecifiedByModels` property.

Use the `getLogAsSpecifiedInModel` method to determine whether the top-level model or top-level Model block logs signals as specified in the model (default logging), and use `setLogAsSpecifiedInModel` to turn default logging on and off.

cell array — For the top-level model, specify the model name. For Model blocks, specify the block path.

RW

Signals

Vector of `Simulink.SimulationData.SignalLoggingInfo` objects for all signals with signal logging override settings.

vector of `Simulink.SimulationData.SignalLoggingInfo` objects

RW

Methods

`createFromModel`

Create a `Simulink.SimulationData.ModelLoggingInfo` object for a top-level model, with override settings for each logged signal in the model.

```
model_logging_info_object = ...
Simulink.SimulationData.ModelLoggingInfo.createFromModel(...
model,options)
```

`model`

Name of the top-level model for which to create a
`Simulink.SimulationData.ModelLoggingInfo` object.

`options`

You can use any combination of the following option name and value pairs to control the kinds of systems from which to include logged signals.

- `FollowLinks`
 - `on` — (Default) Include logged signals from inside of libraries.
 - `off` — Skip all libraries.
- `LookUnderMasks`

- `all` — (Default) Include logged signals from all masked subsystems.
- `none` — Skip all masked subsystems.
- `graphical` — Include logged signals from masked subsystems that do not have a workspace or dialog box.
- `functional` — Include logged signals from masked subsystems that do not have a dialog box.
- `Variants`
 - `ActiveVariants` — (Default) Include logged signals from only active subsystem and model reference variants.
 - `AllVariants` — Include logged signals from all subsystem and model reference variants.
- `RefModels`
 - `on` — (Default) Include logged signals from referenced models.
 - `off` — Skip all referenced models.

If you select more than one option, then the created `Simulink.SimulationData.ModelLoggingInfo` object includes signals that fit the combinations (the “AND”) of the specified options. For example, if you set `FollowLinks` to `on` and set `RefModels` to `off`, then the model signal logging override object does not include signals from library links that exist inside of referenced models.

```
model_logging_override_object
```

```
    Simulink.SimulationData.ModelLoggingInfo object for the top-level model.
```

```
model_logging_info_object =
```

```
    Simulink.SimulationData.ModelLoggingInfo.createFromModel(model) creates a Simulink.SimulationData.ModelLoggingInfo object for the model that includes logged signals for the following kinds of systems:
```

- Libraries
- Masked subsystems
- Referenced models

- Active variants

`model_logging_override_object = Simulink.SimulationData.ModelLoggingInfo.createFromModel(model, options)` creates a `Simulink.SimulationData.ModelLoggingInfo` object for the model. The included logged signals reflect the options settings for the following kinds of systems:

- Libraries
- Masked subsystems
- Referenced models
- Variants

The following example creates a model logging override object for the `sldemo_mdref_bus` model and automatically adds each logged signal in the model to that object:

```
mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel(...
'sldemo_mdref_bus')

mi =

    ModelLoggingInfo with properties:

                Model: 'sldemo_mdref_bus'
        LoggingMode: 'OverrideSignals'
  LogAsSpecifiedByModels: {}
                Signals: [1x3 Simulink.SimulationData.SignalLoggingInfo]
```

To apply the model override object settings, use:

```
set_param(sldemo_mdref_bus, 'DataLoggingOverride', mi);
```

The following example explicitly specifies the kinds of systems from which to include signals, rather than use the default settings for each kind of system. This example specifies to include signals from all model reference variants (instead of using the default of including only active variant).

The `sldemo_mdref_variants` model has two variants:

`sldemo_mrv_nonlinear_controller` and `sldemo_controller`. In this example, in each variant, you configure a signal for signal logging, and then create a `Simulink.SimulationData.ModelLoggingInfo` object. The resulting object includes, in the `Signals` property, two signals (one from each variant).

```
sldemo_mrv_nonlinear_controller;
sldemo_mrv_nonlinear_controller;
ph = get_param('sldemo_mrv_nonlinear_controller/Add', 'PortHandles');
set_param(ph.Outport(1), 'DataLogging', 'on');
ph1 = get_param('sldemo_mrv_nonlinear_controller/Add', 'PortHandles');
set_param(ph1.Outport(1), 'DataLogging', 'on');
mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel(...
    'sldemo_mdhref_variants', 'Variants', 'AllVariants')

V_NONLINEAR_CONTROLLER =

Simulink.Variant
    Condition: 'CTRL==1'

V_NONLINEAR_CONTROLLER =

Simulink.Variant
    Condition: 'CTRL==2'

CTRL =

    1

mi =

    Simulink.SimulationData.ModelLoggingInfo
    Package: Simulink.SimulationData

    Properties:
        Model: 'sldemo_mdhref_variants'
        LoggingMode: 'OverrideSignals'
        LogAsSpecifiedByModels: {}
        Signals: [1x2 Simulink.SimulationData.SignalLoggingInfo]

    Methods
```

ModelLoggingInfo

Specify signals to log or override logging settings.

```
model_logging_override_object = ....
Simulink.SimulationData.ModelLoggingInfo(model)
```

model

Name of the top-level model for which to create a
Simulink.SimulationData.ModelLoggingInfo object

`model_logging_override_object`

Simulink.SimulationData.ModelLoggingInfo object created for the specified top-level model.

`model_logging_override_object=`

Simulink.SimulationData.ModelLoggingInfo(model) creates a Simulink.SimulationData.ModelLoggingInfo object for the specified top-level model.

If you use the Simulink.SimulationData.ModelLoggingInfo constructor, specify a Simulink.SimulationData.SignalLoggingInfo object for each logged signal for which you want to override logging settings.

To check that you have specified valid signal logging override settings for a model, use the `verifySignalAndModelPaths` method with the Simulink.SimulationData.ModelLoggingInfo object for the model.

The following example shows how to log all signals as specified in the top-level model and all referenced models.

```
mi = Simulink.SimulationData.ModelLoggingInfo('sldemo_mdref_bus');
mi.LoggingMode = 'LogAllAsSpecifiedInModel'
```

```
mi =
```

```
ModelLoggingInfo with properties:
```

```
    Model: 'sldemo_mdref_bus'
  LoggingMode: 'LogAllAsSpecifiedInModel'
LogAsSpecifiedByModels: {}
    Signals: []
```

To apply the model override object settings, use:

```
set_param(sldemo_mdref_bus, 'DataLoggingOverride', mi);
```

The following example shows how to log only signals in the top-level model:

```
mi = ...
Simulink.SimulationData.ModelLoggingInfo('sldemo_mdref_bus');
mi.LoggingMode = 'OverrideSignals';
mi = mi.setLogAsSpecifiedInModel('sldemo_mdref_bus', true);
set_param('sldemo_mdref_bus', 'DataLoggingOverride', mi);
```

findSignal

Find signals within the `Signals` vector, using a block path and optionally an output port index.

```
signal_indices = ...
    model_logging_override_object.findSignal(block_path)
signal_indices = ...
    model_logging_override_object.findSignal(...
    block_path, port_index)
```

`block_path`

Source block to search. The `block_path` must be one of the following:

- Character vector
- Cell array of character vectors
- `Simulink.BlockPath` object

`port_index`

Index of the output port to search. Specify a scalar greater than, or equal to, 1.

`signal_indices`

Vector of numeric indices into the signals vector of the `Simulink.SimulationData.ModelLoggingInfo` object.

```
signal_indices = model_logging_override_object.findSignal(
block_path) finds the indices of the signals for the block path that you specify.
```

To find a *single* instance of a signal within a referenced model, use a `Simulink.BlockPath` object or a cell array with a *full path*.

To find *all* instances of a signal within a referenced model, use a character vector with the *relative* path of the signal within the referenced model.

To find a logged chart signal within a Stateflow chart, use a `Simulink.BlockPath` object and set the `SubPath` property to the name of the Stateflow chart signal.

`signal_indices = model_logging_override_object.findSignal(block_path, port_index)` finds the indices of the output signal for the port that you specify, for the block path that you specify.

Do not use the `port_index` argument for Stateflow chart signals.

To find a signal that is *not* in a Stateflow chart and that does *not* appear in multiple instances of a referenced model:

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'ex_bus_logging')))
% Open the referenced model
ex_mdref_counter_bus
mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel(...
    'ex_bus_logging');
% Click the COUNTERBUSCreator block that is the source of
% the logged COUNTERBUS signal
signal_index = mi.findSignal(gcb)

signal_index =

     1
```

To find a signal in a specific instance of a referenced model that is not in a Stateflow chart, use the following approach:

```
signal_index = mi.findSignal({'ex_bus_logging/CounterA', ...
    'ex_mdref_counter_bus/Bus Creator'})

signal_index =

     4
```

For an example that uses the `findSignal` method with a Stateflow chart, see “Override Logging Properties with the Command-Line API” (Stateflow) in the Stateflow documentation.

getLogAsSpecifiedInModel

Determine whether the model logs as specified in the model or uses override settings.

```
logging_mode = ...
getLogAsSpecifiedInModel(model_logging_override_object, path)
```

`model_logging_override_object`

A `Simulink.SimulationData.ModelLoggingInfo` object.

`path`

The path is a character vector that specifies one of the following:

- Name of the top-level model
- Block path of a Model block in the top-level model

`logging_mode`

The `logging_mode` is:

- `true`, if the model specified by `path` is logged as specified in the model.
- `false`, if the model specified by `path` is logged using the override settings specified in the `Signals` property.

`logging_mode =`

`model_logging_override_object.getLogAsSpecifiedInModel(path)` returns:

- `true`, if the model specified by `path` is logged as specified in the model.
- `false`, if the model specified by `path` is logged using the override settings specified in the `Signals` property.

In the following example, the `Simulink.SimulationData.ModelLoggingInfo` object `mi` uses the override settings specified in its `Signals` property.

```
mi = Simulink.SimulationData.ModelLoggingInfo('sldemo_mdlref_bus');  
logging_mode = getLogAsSpecifiedInModel(mi, 'sldemo_mdlref_bus')
```

```
logging_mode =
```

```
0
```

setLogAsSpecifiedInModel

Set logging mode for top-level model or top-level Model block

```
setLogAsSpecifiedInModel(override_object, path)
```

override_object

Simulink.SimulationData.ModelLoggingInfo object.

path

Character vector that specifies one of the following:

- Name of the top-level model
- Block path of a Model block in the top-level model

value

Logging mode:

- true, if the model specified by path is logged as specified in the model
- false, if the model specified by path is logged using the override settings specified in the Signals property.

setLogAsSpecifiedInModel(override_object, path, value) sets the LoggingMode property for a top-level model or a Model block in the top-level model.

The following example shows how to log only signals in the top-level model, using the logging settings specified in that model:

```
sldemo_mdrefref_bus;
mi = Simulink.SimulationData.ModelLoggingInfo('sldemo_mdrefref_bus');
mi.LoggingMode = 'OverrideSignals';
mi = setLogAsSpecifiedInModel(mi, 'sldemo_mdrefref_bus', true);
set_param('sldemo_mdrefref_bus', 'DataLoggingOverride', mi);
```

verifySignalAndModelPaths

Verify paths in Simulink.SimulationData.ModelLoggingInfo object.

```
verified_object = verifySignalAndModelPaths...
(model_logging_override_object, action)
```

`model_logging_override_object`

The `Simulink.SimulationData.ModelLoggingInfo` object to verify. This argument is required.

`action`

The action that the function performs if verification fails. This argument is optional. Specify one of the following values:

- `error` — (default) Throw an error when verification fails
- `warnAndRemove` — Issue a warning when verification fails and update the `Simulink.SimulationData.ModelLoggingInfo` object.
- `remove` — Silently update the `Simulink.SimulationData.ModelLoggingInfo` object.

`verified_object`

If the method detects no invalid paths, it returns the validated object. For example:

```
verified_object =  
  
Simulink.SimulationData.ModelLoggingInfo  
Package: Simulink.SimulationData  
  
Properties:  
    Model: 'logging_top'  
    LoggingMode: 'OverrideSignals'  
    LogAsSpecifiedByModels: {}  
    Signals: [1x1 Simulink.SimulationData.SignalLoggingInfo]
```

If the method detects an invalid path, it performs the action specified by the `action` argument. By default, it issues an error message.

```
verified_object = verifySignalAndModelPaths(  
model_logging_override_object, action)
```

For a `Simulink.SimulationData.ModelLoggingInfo` object, verify that:

- All character vectors in the `LogAsSpecifiedByModels` property are either the name of the top-level model or the block path of a Model block in the top-level model.
- The block paths for signals in the `Signals` property refer to valid blocks within the hierarchy of the top-level model.

- The `OutputPortIndex` property for all signals in the `Signals` property are valid for the given block.
- All signals in the `Signals` property refer to *logged* signals.

The `action` argument specifies what action the method performs. By default, the method returns an error if it detects an invalid path.

If you use the `Simulink.SimulationData.ModelLoggingInfo` constructor and specify a `Simulink.SimulationData.SignalLoggingInfo` object for each signal, then consider using the `verifySignalAndModelPaths` method to verify that your object definitions are valid.

The following example shows how to validate the signal and block paths in a `Simulink.SimulationData.ModelLoggingInfo` object. Because the `action` argument is `warnAndRemove`, if the validation fails, the `verifySignalAndModelPaths` method issues a warning and updates the `Simulink.SimulationData.ModelLoggingInfo` object.

```
mi = Simulink.SimulationData.ModelLoggingInfo('sldemo mdlref_bus');  
verified_object = verifySignalAndModelPaths...  
    (mi, 'warnAndRemove')
```

See Also

`Simulink.BlockPath` | `Simulink.ModelDataLogs` |
`Simulink.SimulationData.DataStoreMemory` |
`Simulink.SimulationData.LoggingInfo` | `Simulink.SimulationData.Signal` |
`Simulink.SimulationData.SignalLoggingInfo`

Topics

“Override Signal Logging Settings from MATLAB”
“Migrate Scripts That Use Legacy `ModelDataLogs` API”
“Export Signal Data Using Signal Logging”
“Log Data Stores”

Introduced in R2012b

Simulink.SimulationData.SignalLoggingInfo

Signal logging override settings for signal

Description

This object contains the signal override signal logging settings for a single logged signal.

Property Summary

Name	Description
BlockPath on page 5-542	Simulink.BlockPath of source block of a signal to log.
OutputPortIndex on page 5-543	Index of an output port to log.
LoggingInfo on page 5-543	Simulink.SimulationData.LoggingInfo object containing all logging override settings for a signal.

Method Summary

Name	Description
SignalLoggingInfo on page 5-544	Create a signal logging override object for a signal.

Properties

BlockPath

Simulink.BlockPath of source block of signal to log. The block path represents the full model reference hierarchy.

To specify a specific instance of a signal, use an absolute path, reflecting the model reference hierarchy, starting at the top model. For example:

```
sig_log_info = Simulink.SimulationData.SignalLoggingInfo(...  
{'sldemo_mdlref_bus/CounterA', ...  
'sldemo_mdlref_counter_bus/Bus Creator'})
```

Simulink.BlockPath

RW

OutputPortIndex

Index of the output port to log. The index is a 1-based numeric value.

nonzero integer

RW

LoggingInfo

Simulink.SimulationData.LoggingInfo object containing logging override settings for a signal. The logging settings specify whether signal logging is overridden for this signal. The logging settings also can specify a logging name, a decimation factor, and a maximum number of data points.

Simulink.SimulationData.LoggingInfo object

RW

Methods

SignalLoggingInfo

Construct a `Simulink.SimulationData.SignalLoggingInfo` object.

```
signal_logging_info_object = ...  
    Simulink.SimulationData.SignalLoggingInfo()  
signal_loggingInfo_object = ...  
    Simulink.SimulationData.SignalLoggingInfo(path)  
signalLoggingInfo_object = ...  
    Simulink.SimulationData.SignalLoggingInfo(path,index)
```

`path`

The block path of the source block for which the signal logging override settings apply. If you use this argument without also using the `port` argument, then Simulink sets the output port index to 1.

`index`

Output port index to which the signal logging override settings apply.

`signal_logging_object`

`Simulink.SimulationData.SignalLoggingInfo` object that represents the override settings of a signal.

```
signal_logging_override_object =  
Simulink.SimulationData.SignalLoggingInfo() creates a  
Simulink.SimulationData.LoggingInfo object that contains default logging  
settings for a signal.
```

```
signal_logging_override_object =  
Simulink.SimulationData.SignalLoggingInfo(path) creates a  
Simulink.SimulationData.LoggingInfo object, using the specified block path, and  
sets the output port index to 1.
```

```
signal_logging_override_object =  
Simulink.SimulationData.SignalLoggingInfo(path, port) creates a
```

Simulink.SimulationData.LoggingInfo object that contains default logging settings for the specified block path and output port index.

The following example creates a Simulink.SimulationData.SignalLoggingInfo object for the first output port of the Bus Creator block in the sldemo_mdref_bus model.

```
sldemo_mdref_bus;
mi = Simulink.SimulationData.ModelLoggingInfo(...
'sldemo_mdref_bus');
mi.LoggingMode = 'OverrideSignals';
mi.Signals = ...
    Simulink.SimulationData.SignalLoggingInfo(...
        {'sldemo_mdref_bus/CounterA', ...
'sldemo_mdref_counter_bus/Bus Creator'}, 1)
```

The output is:

```
mi =

    Data.ModelLoggingInfo with properties:

        Model: 'sldemo_mdref_bus'
    LoggingMode: 'OverrideSignals'
    LogAsSpecifiedByModels: {}
        Signals: [1x1 Simulink.SimulationData.SignalLoggingInfo]

    Methods
```

See Also

Simulink.ModelDataLogs | Simulink.SimulationData.BlockPath |
 Simulink.SimulationData.DataStoreMemory |
 Simulink.SimulationData.LoggingInfo |
 Simulink.SimulationData.ModelLoggingInfo |
 Simulink.SimulationData.Signal

Topics

“Override Signal Logging Settings from MATLAB”
 “Migrate Scripts That Use Legacy ModelDataLogs API”
 “Export Signal Data Using Signal Logging”
 “Log Data Stores”

Introduced in R2012b

Simulink.SimulationData.Signal

Container for signal logging information

Description

Simulink uses `Simulink.SimulationData.Signal` objects to store signal logging information during simulation. The objects contain information about the source block for the signal, including the port type and index.

Property Summary

Name	Description
BlockPath on page 5-547	Block path for the source block for the signal
Name on page 5-548	Name of signal element to use for name-based access
PropagatedName on page 5-548	Propagated signal name, if any
PortIndex on page 5-548	Numeric index of port that was logged
PortType on page 5-549	Type of port that was logged: for signal logging, the port type is 'outport'
Values on page 5-549	Time and data that were logged

Properties

BlockPath

Block path for the source block for the signal

```
Simulink.SimulationData.BlockPath
```

RW

Name

Name of signal element to use for name-based access

character vector

RW

PropagatedName

Propagated name of signal element

Signal logging and root Output block logging data for a signal captures the propagated signal name if the logging format is `Dataset` and:

- For signal logging, you:
 - Mark the signal for signal logging and in the Signal Properties dialog box select **Show Propagated Signals**.
 - Enable **Configuration Parameters > Data Import/Export > Signal logging**.
- For root Output block logging, you select **Configuration Parameters > Data Import/Export > Output**.

The propagated signal name does not include angle brackets (<>).

character vector

RO

PortIndex

Numeric index of port that was logged

scalar real integer

RW

PortType

Type of port that was logged: for signal logging, the port type is 'output'

character vector

RW

Values

Time and data that were logged.

For an example of how to use the `Values` property and plot logged signal data, in the `sldemo_mdhref_bus` example, see “Logging Signal Data.”

- MATLAB `timeseries` object
- Structure of MATLAB `timeseries` objects (for bus signals)
- Array of structures of MATLAB `timeseries` objects (for array of buses signals)
- Array of MATLAB `timeseries` objects (for nonbus signals in a For Each subsystem)

RW

See Also

`Simulink.BlockPath` | `Simulink.SimulationData.Dataset` | `timeseries`

Topics

“View and Access Signal Logging Data”

“Create MATLAB Timeseries Data for Root-Level Inputs”

“Load Bus Data to Root-Level Input Ports”

“Log Signals in For Each Subsystems”

Simulink.SimulationData.State class

Package: Simulink.SimulationData

State logging element

Description

Simulink uses `Simulink.SimulationData.State` objects to store state logging information during simulation. The objects contain state information about which block the state data is coming from and the type of state.

Properties

Name — Name of state element to use for name-based access

character vector

Name of state element to use for name-based access, specified as a character vector. If you do not specify a name, 'CSTATE' or 'DSTATE' is used, depending on whether it is a continuous or discrete state.

BlockPath — Block path for state source block

a `Simulink.SimulationData.BlockPath` object

Block path for state source block, specified as a `Simulink.SimulationData.BlockPath` object

Label — Type of state

'CSTATE' | 'DSTATE'

Type of state, returned as 'CSTATE' or 'DSTATE'. Read-only property.

- 'CSTATE' – Continuous state
- 'DSTATE' – Discrete state

Values — State element information

single MATLAB `timeseries` object | a structure of MATLAB `timeseries` objects

State element information, specified as a single MATLAB `timeseries` object or as a structure of MATLAB `timeseries` objects.

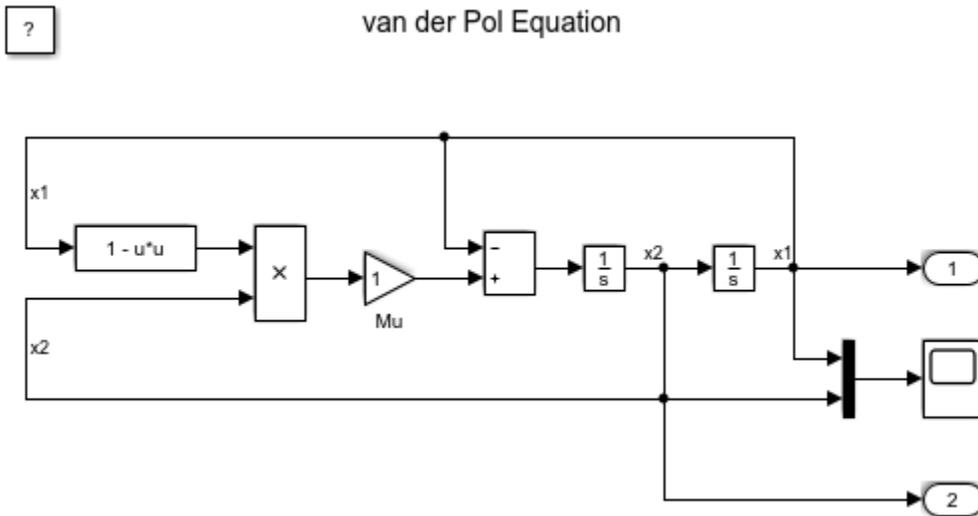
Examples

Final State Information in Structure with Dataset Format

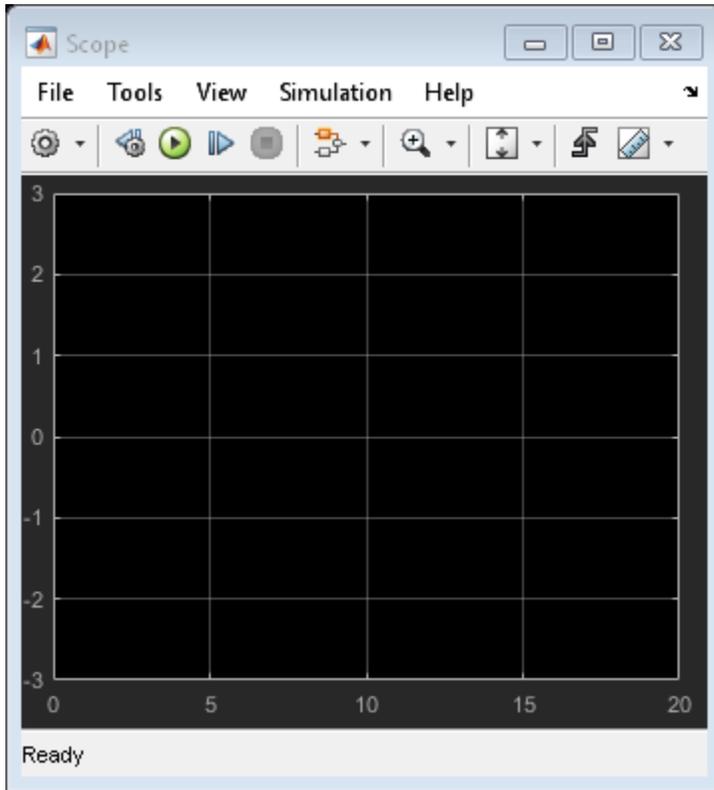
Saved final state information in `Dataset` format and access the state data after simulation.

Open the `vdp` model and specify to log final states in `Dataset` format. Use the default logged state variable, `xFinal`.

```
open_system('vdp');
set_param(gcs, 'SaveFinalState', 'on', 'SaveFormat', 'Dataset');
```

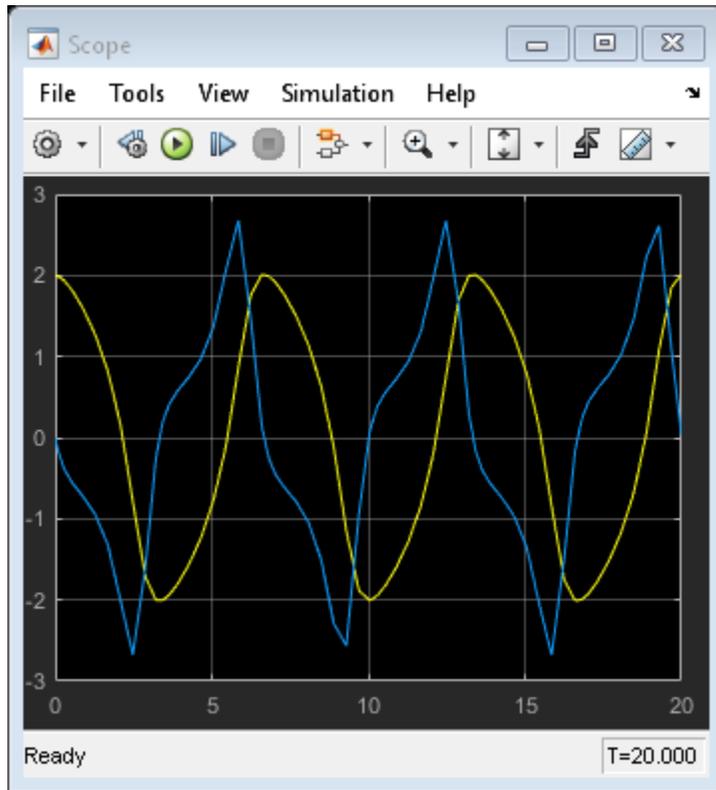


Copyright 2004-2013 The MathWorks, Inc.



Simulate the model.

```
sim('vdp');
```



View the state logging information in `xFinal`.

```
xFinal
```

```
xFinal =
```

```
Simulink.SimulationData.Dataset 'xFinal' with 2 elements
```

	Name	BlockPath
1	[1x1 State]	vdp/x1
2	[1x1 State]	vdp/x2

- Use braces `{ }` to access, modify, or add elements using index.

Examine the first element of the state dataset.

```
xFinal.get(1)
```

```
ans =
```

```
Simulink.SimulationData.State  
Package: Simulink.SimulationData  
  
Properties:  
    Name: ''  
    BlockPath: [1x1 Simulink.SimulationData.BlockPath]  
    Label: CSTATE  
    Values: [1x1 timeseries]
```

See Also

Simulink.SimulationData.Dataset

Topics

“State Information”

Introduced in R2015a

Simulink.SimulationData.Unit class

Package: Simulink.SimulationData

Store units for simulation data

Description

Simulink creates `Simulink.SimulationData.Unit` objects to store unit information for signals when:

- Performing signal logging, which uses the `Dataset` format
- Logging root Output blocks, if in **Configurations Parameters** you select the **Output** parameter and set **Format** to `Dataset`
- Logging to a To Workspace block or To File block, if you set the **Save format** block parameter to the default of `Timeseries`

Construction

`unitsObj = Simulink.SimulationData.Unit(unitName)` creates a `Simulink.SimulationData.Unit` object with the unit that you specify.

Input Arguments

unitName — Name of logging data units

character vector

Name of logging data units, specified as a character vector.

Output Arguments

unitsObj — Logging data units

`Simulink.SimulationData.Unit` object

Logging data units, returned as a `Simulink.SimulationData.Unit` object.

Properties

Name — Name of the units

character vector

Name of the units, specified as a character vector.

Methods

Method	Purpose
<code>Simulink.SimulationData.Unit.setName</code>	Specify name of logging data unit

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Create and Use Inches Logging Units

Create a `Simulink.SimulationData.Unit` object representing inches.

```
inchesUnit = Simulink.SimulationData.Unit('inches');
```

Create a MATLAB `timeseries` object and set its `Units` field to the `Simulink.SimulationData.Unit` object.

```
ts = timeseries(1:10);
ts.DataInfo.Units = inchesUnit;
ts.DataInfo.Units
```

```
ans =
```

```
Units with properties:
```

Name: 'inches'

- “Log Signal Data That Uses Units”
- “Load Signal Data That Uses Units”
- “Convert Logged Data to Dataset Format”
- “Prepare Model Inputs and Outputs”

See Also

`Simulink.SimulationData.Dataset`

Topics

- “Log Signal Data That Uses Units”
- “Load Signal Data That Uses Units”
- “Convert Logged Data to Dataset Format”
- “Prepare Model Inputs and Outputs”

Introduced in R2016a

Simulink.SimulationInput class

Package: Simulink

Creates `SimulationInput` objects to make changes to a model for multiple or individual simulations

Description

The `Simulink.SimulationInput` object allows you to make changes to a model and run simulations with those changes. These changes are temporarily applied to the model. Through `Simulink.SimulationInput` object, you can change:

- Initial state
- Model parameters
- Block parameters
- External inputs
- Variables

Through the `Simulink.SimulationInput` object, you can also specify MATLAB functions to run at the start and the end of each simulation by using `in.PreSimFcn` and `in.PostSimFcn`, respectively. `Simulink.SimulationInput` does not support the ability to allow model references to have their own data dictionary.

Construction

`in = Simulink.SimulationInput('modelName')` creates a `SimulationInput` object for a model.

Input Arguments

modelName — Name of the model
character vector

Create a `Simulink.SimulationInput` object by passing the name of the model as an argument.

Example: `in = Simulink.SimulationInput('cstr')`

Properties

modelName — Name of the model

character vector

Name of the model for which the `SimulationInput` object is created.

InitialState — Initial state

`Simulink.SimState.ModelSimState` object

Initial state of the model for a simulation specified as a `Simulink.SimState.ModelSimState` object

ExternalInput — External Input

numerical array, dataset object, timeseries, character array

External inputs added to the model for a simulation.

BlockParameters — Block parameters

array of `Simulink.Simulation.BlockParameter`

Block parameters of the model that are modified.

Variables — Variables

array of `Simulink.Simulation.Variable`

Variables of the model that are modified.

ModelParameters — Model parameters

array of `Simulink.Simulation.ModelParameter`

Model parameters of the model that are modified .

PreSimFcn — Function

MATLAB function

MATLAB function to run before the start of the simulation.

PostSimFcn — Function

MATLAB function

MATLAB function to run after the simulations.

UserString — User string

character array

Brief description of the simulation specified as a character array.

Methods

Method	Purpose
<code>setModelParameter</code>	Set model parameters to be used for a specific simulation through <code>SimulationInput</code> object.
<code>setBlockParameter</code>	Set block parameters to be used for a specific simulation through <code>SimulationInput</code> object.
<code>setInitialState</code>	Set initial state to be used for a specific simulation through <code>SimulationInput</code> object.
<code>ExternalInput</code>	Set external inputs for a simulation through <code>SimulationInput</code> object.
<code>setVariable</code>	Set variables for a simulation through <code>SimulationInput</code> object.
<code>PreSimFcn</code>	Specify a MATLAB function to run before start of each simulation through <code>SimulationInput</code> object.
<code>PostSimFcn</code>	Specify a MATLAB function to run after each simulation is complete through <code>SimulationInput</code> object.
<code>applyToModel</code>	Apply changes to the model specified through a <code>SimulationInput</code> object.
<code>validate</code>	Validate the contents of the <code>SimulationInput</code> object.

Examples

Create a Simulink.SimulationInput Object

This example shows you how to create a `SimulationInput` object.

Create a single `SimulationInput` object for a model.

```
model = 'sldemo_househeat';  
in = Simulink.SimulationInput(model);
```

Create an Array of Simulink.SimulationInput Objects

This example shows you how to create an array of `SimulationInput` objects.

Create an array of `SimulationInput` objects by using the `for` loop.

```
model = 'vdp';  
for i = 10:-1:1  
    in(i) = Simulink.SimulationInput(model);  
end
```

Set Block Parameters using an Array of Simulink.SimulationInput Objects

This example modifies the block parameters of a model through the `SimulationInput` object.

Open the model

```
mdl = 'sldemo_househeat';  
open_system(mdl);
```

Create a `SimulationInput` object for this model.

```
in = Simulink.SimulationInput(mdl);
```

Modify block parameter.

```
in = in.setBlockParameter('sldemo_househeat/Set Point', 'Value', '300');
```

Simulate the model.

```
out = sim(in)
```

See Also

ExternalInput | PostSimFcn | PreSimFcn | Simulation Manager |
Simulink.SimulationInput | applyToModel | parsim | setBlockParameter |
setInitialState | setModelParameter | setVariable | validate

Topics

“Run Multiple Simulations”

“Parallel Simulations Workflow”

Introduced in R2017a

applyToModel

Apply changes to the model specified through a `SimulationInput` object, `in`

Syntax

```
in.applyToModel
```

Description

`in.applyToModel` applies the changes specified through the `SimulationInput` object to the model. You can use it to debug a model or to interactively analyze a simulation.

Examples

Apply Changes Made Through the `Simulink.SimulationInput` Object to the Model

This example shows how to modify a model through a `SimulationInput` object and save those modifications.

Open the model and create a `SimulationInput` object.

```
open_system('sldemo_househeat');  
in = Simulink.SimulationInput('sldemo_househeat');
```

Modify block parameter, model parameters and a variable through `SimulationInput` object.

```
in = in.setBlockParameter('sldemo_househeat/Set Point', 'Value', '75');  
in = in.setVariable('cost', 50, 'Workspace', 'sldemo_househeat');  
in = in.setModelParameter('StartTime', '1', 'StopTime', '5');
```

Apply the modifications made in the above step to the model.

`in.applyToModel`

See Also

`ExternalInput` | `PostSimFcn` | `PreSimFcn` | `Simulation Manager` |
`Simulink.SimulationInput` | `parsim` | `setBlockParameter` | `setInitialState`
| `setModelParameter` | `setVariable` | `validate`

Topics

“Run Multiple Simulations”

“Parallel Simulations Workflow”

Introduced in R2017a

ExternalInput

Set external inputs for a simulation through `SimulationInput` object, `in`

Syntax

```
in.ExternalInput = [t, u1, ..uN]
in.ExternalInput = ds
in.ExternalInput = ts
```

Description

`in.ExternalInput = [t, u1, ..uN]` allows you to directly specify numerical arrays as inputs to a model if a model has root inports.

`in.ExternalInput = ds` allows you to directly specify dataset objects as external inputs to a model if a model has root inports..

`in.ExternalInput = ts` allows you to directly specify timeseries object as external input if a model has a single root inport.

Examples

Set Numerical Arrays as External Inputs Through a `Simulink.SimulationInput` Object

This example shows how to set numerical arrays as external inputs.

Open the model

```
open_system('sldemo_mdhref_counter');
```

Create a `SimulationInput` object for this model.

```
in = Simulink.SimulationInput('sldemo_mdhref_counter');
```

Prepare external inputs.

```
t = (0:0.01:10)';  
u1 = 5*ones(size(t));  
u2 = 10*sin(t);  
u3 = -5*ones(size(t));
```

Set external inputs to the model.

```
in.ExternalInput = [t, u1, u2, u3];
```

Simulate the model.

```
out = sim(in);
```

Input Arguments

[t, u1, .. uN] — Numerical array

Numerical array

Numerical array to be used as an external input.

ds — Dataset object

`Simulink.SimulationData.Dataset` object

Dataset object to be used as an external input

ts — Time series

Time object handle

Time series to be used as an external input

See Also

`PostSimFcn` | `PreSimFcn` | `Simulation Manager` | `Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setInitialState` | `setModelParameter` | `setVariable` | `validate`

Topics

“Run Multiple Simulations”

“Parallel Simulations Workflow”

Introduced in R2017a

PostSimFcn

Specify a MATLAB function to run after each simulation is complete through `SimulationInput` object, in

Syntax

```
in.PostSimFcn = @(y) myfunction(arg1, arg2...)
```

Description

`in.PostSimFcn = @(y) myfunction(arg1, arg2...)` runs after each simulation is complete. The `Simulink.SimulationOutput` object is passed as the argument `y` to this function. `myfunction` is any MATLAB function and can be used to do the post processing on the output. To return post processed data, you must return it as values in a struct. These values are then packed into the `Simulink.SimulationOutput` output to replace the usual logged data or add new data to the `Simulink.SimulationOutput` object.

Examples

Specify a MATLAB function for Postprocessing of the Output

This example specifies a MATLAB Function through `SimulationInput` object to run after each simulation is complete.

Create a `PostSimFcn` to get the mean of the output.

```
function newout = postsim(out);  
newout.mean = mean(out.yout);  
end
```

Create a `SimulationInput` object for a model.

```
in = Simulink.SimulationInput('vdp');  
in.PostSimFcn = @(x) postsim(x);  
in = setModelParameter(in, 'SaveOutput', 'on');
```

Simulate the model.

```
out = sim(in)
```

View your result

```
out.mean
```

It is best practice to avoid using 'ErrorMessage' and 'SimulationMetadata' as field names in the function.

Input Arguments

y — Copy of `Simulink.SimulationOutput` object for postprocessing

`Simulink.SimulationOutput` object

This is a `Simulink.SimulationOutput` object which is an input to myfunction.

See Also

[ExternalInput](#) | [PreSimFcn](#) | [Simulation Manager](#) |
[Simulink.SimulationInput](#) | [applyToModel](#) | [parsim](#) | [setBlockParameter](#) |
[setInitialState](#) | [setModelParameter](#) | [setVariable](#) | [validate](#)

Topics

“Run Multiple Simulations”

“Parallel Simulations Workflow”

Introduced in R2017a

PreSimFcn

Specify a MATLAB function to run before start of each simulation through `SimulationInput` object, in

Syntax

```
in.PreSimFcn = @(x) myfunction(arg1, arg2...)
```

Description

`in.PreSimFcn = @(x) myfunction(arg1, arg2...)` runs before each simulation starts. The `Simulink.SimulationInput` object is passed as an argument `x` to this function. `myfunction` is any MATLAB function and can be used to modify the `Simulink.SimulationInput` object. If you use `myfunction` to modify the `Simulink.SimulationInput` object, you must return `Simulink.SimulationInput` object as the only output argument.

Examples

Specify a MATLAB Function to Run Before Each Simulation

This example shows how to specify a MATLAB function through `SimulationInput` object to run at before start of each simulation.

Create a `PreSimFcn` function.

```
function presim(in)
signalbuilder('sf_car/User Inputs', 'ActiveGroup', in.Variables.Value)
```

Open the model.

```
model = 'sf_car';
open_system(model);
```

Create an array of `SimulationInput` objects for this model. Use `in.PreSimFcn` to run `presim` before simulation.

```
n = 4;
for idx = n:-1:1
    in(idx) = Simulink.SimulationInput(model);
    in(idx) = in(idx).setVariable('SigIndex', idx);
    in(idx).PreSimFcn = @(x) presim(x);
end
```

Simulate the model.

```
out = sim(in)
```

Input Arguments

x — A `Simulink.SimulationInput` object as input to the myfunction

`Simulink.SimulationInput` object

This is an input to myfunction in which you can modify the `Simulink.SimulationInput` object.

See Also

[ExternalInput](#) | [PostSimFcn](#) | [Simulation Manager](#) | [Simulink.SimulationInput](#) | [applyToModel](#) | [parsim](#) | [setBlockParameter](#) | [setInitialState](#) | [setModelParameter](#) | [setVariable](#) | [validate](#)

Topics

“Run Multiple Simulations”

“Parallel Simulations Workflow”

Introduced in R2017a

setBlockParameter

Set block parameters to be used for a specific simulation through `SimulationInput` object, `in`

Syntax

```
in = in.setBlockParameter(BlockPath, ParameterName, Value, ...  
ParameterNameN, ValueN)
```

Description

`in = in.setBlockParameter(BlockPath, ParameterName, Value, ...
ParameterNameN, ValueN)` sets the parameter on the block specified at `BlockPath` with the properties `ParameterName` and `Value`. You can set multiple block parameters in a model using the same `SimulationInput` object. For more information on block parameter, see “Block-Specific Parameters” on page 6-128.

Examples

Modify a Block Parameter for a Simulation

This example modifies the block parameters of a model through the `SimulationInput` object.

Open the model

```
mdl = 'sldemo_househeat';  
open_system(mdl);
```

Create a `SimulationInput` object for this model.

```
in = Simulink.SimulationInput(mdl);
```

Modify block parameter.

```
in = in.setBlockParameter('sldemo_househeat/Set Point','Value','300');
```

Simulate the model.

```
out = sim(in)
```

Input Arguments

BlockPath — Path of the block

character vector

BlockPath is the path of the block for which the parameter is changed

Example: 'sldemo_househeat/Set Point'

ParameterName — Block parameter name

character vector

Specify optional comma-separated pairs of `ParameterName`, `Value` arguments. `ParameterName` is the parameter name and `Value` is the corresponding value. `ParameterName` must appear inside single quotes (' '). Block parameter values are typically specified as character vectors. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. `ParameterNameN,ValueN` pairs follow the same syntax as `set_param`.

Example: 'Value', '350'

See Also

[ExternalInput](#) | [PostSimFcn](#) | [PreSimFcn](#) | [Simulation Manager](#) | [Simulink.SimulationInput](#) | [applyToModel](#) | [parsim](#) | [setInitialState](#) | [setModelParameter](#) | [setVariable](#) | [validate](#)

Topics

“Run Multiple Simulations”
“Parallel Simulations Workflow”

Introduced in R2017a

setInitialState

Set initial state to be used for a specific simulation through `SimulationInput` object, in

Syntax

```
in.setInitialState = xInitial
```

Description

`in.setInitialState = xInitial` sets the initial state of a model to `xInitial`, a `Simulink.SimState.ModelSimState` object.

Input Arguments

xInitial — `Simulink.SimState.ModelSimState` object

`Simulink.SimState.ModelSimState`

You can change the `Initial State` of a model by assigning it to a `Simulink.SimState.ModelSimState` object.

See Also

`ExternalInput` | `PostSimFcn` | `PreSimFcn` | `Simulation Manager` | `Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` | `setModelParameter` | `setVariable` | `validate`

Topics

“Run Multiple Simulations”

“Parallel Simulations Workflow”

Introduced in R2017a

setModelParameter

Set model parameters to be used for a specific simulation through `SimulationInput` object, `in`

Syntax

```
in = in.setModelParameter(ParameterName,Value,...ParameterNameN,  
ValueN)
```

Description

`in = in.setModelParameter(ParameterName,Value,...ParameterNameN,ValueN)` sets a model parameter `Name` with a `Value`. You can add multiple model parameters to the model using the same `SimulationInput` object. For more information on model parameters, see “Model Parameters” on page 6-2.

Examples

Modify a Model Parameter for a Simulation

This example modifies the model parameters of through the `SimulationInput` object

Open the model.

```
mdl = 'sldemo_househeat';  
open_system(mdl);
```

Create a `SimulationInput` object for this model.

```
in = Simulink.SimulationInput(mdl);
```

Modify model parameters.

```
in = in.setModelParameter('StartTime','1','StopTime','4');
```

Simulate the model.

```
out = sim(in)
```

Input Arguments

ParameterName — Block parameter name

character vector

Specify optional comma-separated pairs of `ParameterName, Value` arguments. `ParameterName` is the parameter name and `Value` is the corresponding value. `ParameterName` must appear inside single quotes (' '). Model parameter values are typically specified as character vectors. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`. `ParameterNameN, ValueN` pairs follow the same syntax as `set_param`.

Example: `'StartTime', '1'`

See Also

[ExternalInput](#) | [PostSimFcn](#) | [PreSimFcn](#) | [Simulation Manager](#) | [Simulink.SimulationInput](#) | [applyToModel](#) | [parsim](#) | [setBlockParameter](#) | [setInitialState](#) | [setVariable](#) | [validate](#)

Topics

“Run Multiple Simulations”

“Parallel Simulations Workflow”

Introduced in R2017a

setVariable

Set variables for a simulation through `SimulationInput` object, `in`

Syntax

```
in = in.setVariable(Name,Value)
in = in.setVariable(Name,Value,'Workspace', 'ModelName')
```

Description

`in = in.setVariable(Name,Value)` assigns a `Value` to variable `Name`. You can add multiple variables to the model using the same `SimulationInput` object.

`in = in.setVariable(Name,Value,'Workspace', 'ModelName')` assigns the `Value` to variable `Name`. Variables that are defined through the `SimulationInput` object are placed in the global workspace scope by default. The term global workspace is specific to the `Simulink.SimulationInput` object and its methods. Variables in the global workspace scope take precedence if a variable with the same name exists in the base workspace or the data dictionary. The variables in the model workspace take precedence over the global workspace scope. To change the value of a model workspace variable, set the scope by specifying the model name when you add the variable to the `SimulationInput` object.

For information on using nonscalar variables, structure variables and parameter objects, see “Sweep Nonscalars, Structures, and Parameter Objects”.

Examples

Modify a Variable for a Simulation

This example modifies the model parameters of through the `SimulationInput` object.

Open the model.

```
mdl = 'sldemo_househeat';  
open_system(mdl);
```

Create a `SimulationInput` object for this model

```
in = Simulink.SimulationInput(mdl);
```

Set the variable value to 50.

```
in = in.setVariable('cost',50);
```

By default, this variable is placed in the global workspace scope.

Simulate the model.

```
out = sim(in)
```

Modify a Variable for a Simulation in the Model Workspace

This example modifies the model parameters of through the `SimulationInput` object.

Set path and open the model.

```
mdl = 'sldemo_househeat';  
open_system(mdl);
```

Create a `SimulationInput` object for this model

```
in = Simulink.SimulationInput(mdl);
```

Set the variable to 50 and set the scope to model workspace.

```
in = in.setVariable('cost',50,'Workspace','sldemo_househeat');
```

Simulate the model.

```
out = sim(in)
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of *Name*, *Value* arguments. *Name* is the argument name and *Value* is the literal value of the variable. *Name* must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as *Name1*, *Value1*, . . . , *NameN*, *ValueN*.

Example: 'cost', '65'

Workspace — Workspace for the variable

character vector

Set the scope of the defined variable by specifying the model name

Example: 'Workspace', 'sldemo_househeat'

See Also

[ExternalInput](#) | [PostSimFcn](#) | [PreSimFcn](#) | [Simulation Manager](#) | [Simulink.SimulationInput](#) | [applyToModel](#) | [parsim](#) | [setBlockParameter](#) | [setInitialState](#) | [setModelParameter](#) | [validate](#)

Topics

“Run Multiple Simulations”

“Parallel Simulations Workflow”

Introduced in R2017a

validate

Validate the contents of the `SimulationInput` object, `in`

Syntax

```
in.validate
```

Description

`in.validate` validates the changes made to the model through the `SimulationInput` object. This method validates all the changes made to the model through the `SimulationInput` object.

Examples

Validate Changes Made Through the SimulationInput Object

This example modifies and validates the variable of the model through the `SimulationInput` object.

Open the model.

```
mdl = 'sldemo_househeat';  
open_system(mdl);
```

Create a `SimulationInput` object for this model

```
in = Simulink.SimulationInput(mdl);
```

Modify a model parameter

```
in = in.setModelParameter('InvalidParamName', '5');
```

Validate this change

`in.validate`

See Also

`ExternalInput` | `PostSimFcn` | `PreSimFcn` | `Simulation Manager` |
`Simulink.SimulationInput` | `applyToModel` | `parsim` | `setBlockParameter` |
`setInitialState` | `setModelParameter` | `setVariable`

Topics

“Run Multiple Simulations”

“Parallel Simulations Workflow”

Introduced in R2017a

Simulink.SimulationMetadata class

Package: Simulink

Access metadata of simulation runs

Description

The `SimulationMetadata` class contains information about a simulation run including:

- Model information
- Timing information
- Execution and diagnostic information
- Custom character vector to tag the simulation
- Custom data to describe the simulation

`SimulationMetadata` packages this information with the `SimulationOutput` object. To use `SimulationMetadata`, use one of these approaches:

- In **Configuration Parameters > Data Import/Export**, under **Save options**, select **Single simulation output**.
- Use `set_param` to set `ReturnWorkspaceOutputs` to on.

```
set_param(model_name, 'ReturnWorkspaceOutputs', 'on');
```

To retrieve the `SimulationMetadata` object, use the `getSimulationMetadata` method on a `SimulationOutput` object.

Properties

ModelInfo — Information about the model and simulation operating environment
structure

The `ModelInfo` structure has these fields.

Field Name	Type	Description
ModelName	char	Name of the model
ModelVersion	char	Version of the model
ModelFilePath	char	Absolute location of the .mdl/.slx file
UserID	char	System user ID of the machine used for the simulation
MachineName	char	Hostname of the machine used for the simulation
Platform	char	Operating system of the machine used for the simulation
ModelStructuralChecksum	4-by-1 uint32	Structural checksum of the model calculated after an update diagram
SimulationMode	char	Simulation mode
StartTime	double	Simulation start time
StopTime	double	Time at which the simulation was terminated
SolverInfo	structure	Solver information: <ul style="list-style-type: none"> • Fixed-step solvers – Solver type, name, and fixed step size • Variable solvers – Solver type, name, and max step size (initial setting)
SimulinkVersion	structure	Version of Simulink
LoggingInfo	structure	Metadata about logging to persistent storage: <ul style="list-style-type: none"> • LoggingToFile field — Indicates whether logging to persistent storage is enabled ('on' or 'off') • LoggingFileName field — Specifies the resolved file name for the persistent storage MAT-file (if LoggingToFile is 'on').

ExecutionInfo — Structure to store information about a simulation run
structure

Structure to store information about a simulation run, including the reason a simulation stopped and any diagnostics reported during the simulation. The structure has these fields.

Field Name	Type	Description
StopEvent	Nontranslated character vector	<p>Reason the simulation stopped, represented by one of these.</p> <ul style="list-style-type: none"> • ReachedStopTime – Simulation stopped upon reaching stop time and no errors were reported during execution. StopEvent has value ReachedStopTime, even if errors are reported in the stop callbacks, which are executed after the simulation ends. • ModelStop – Simulation stopped by a block or by solver before reaching stop time • StopCommand – Simulation stopped manually by clicking the Stop button or programmatically using the set_param command • DiagnosticError – Simulation stopped because an error was reported during simulation • KeyboardControlC – Simulation stopped using keystroke Ctrl+C. • PauseCommand – Simulation paused manually by clicking the Pause button or programmatically using the set_param command • ConditionalPause – Simulation paused using a conditional breakpoint • PauseTime – Simulation paused at or after specified pause time • StepForward – Simulation paused after clicking step forward

Field Name	Type	Description
		<ul style="list-style-type: none"> • StepBackward – Simulation paused after clicking step backward • TimeOut – Simulation stopped because execution time exceeded timeout specified by TimeOut
StopEventSource	Simulink.SimulationData.BlockPath	Source of stop event, if it is a valid Simulink object
StopEventDescription	Translated character vector	Super set of information stored in StopEvent and StopEventSource
ErrorDiagnostic	struct	<p>Error reported during simulation, represented by these fields.</p> <ul style="list-style-type: none"> • Diagnostic – MSLDiagnostic object that includes object paths, ID, message, cause, and stack • SimulationPhase – Represented by one of these: Initialization, Execution, or Termination • SimulationTime – Simulation time represented as a double, if reported during Execution; else, represented as [] <p>By passing the name–value pair 'ReturnErrorsInOutput', 'on' to the sim command, errors generated during simulation are reported in ExecutionInfo.ErrorDiagnostic. The sim command does not capture generated errors.</p>

Field Name	Type	Description
WarningDiagnostics	Array of struct	<p>Array of all warnings reported during the simulation. Each array item is represented by these fields.</p> <ul style="list-style-type: none"> Diagnostic – MSLDiagnostic object that includes object paths, ID, message, cause, and stack SimulationPhase – Represented by one of these: Initialization, Execution, or Termination SimulationTime – Simulation time represented as a double, if reported during Execution; else, represented as []

TimingInfo — Structure to store profiling information about the simulation structure

Structure to store profiling information about the simulation, including the time stamps for the start and end of the simulation. The structure has these fields.

Field Name	Type	Description
WallClockTimestampStart	character vector	Wall clock time when the simulation started, in YYYY-MM-DD HH:MI:SS format with microsecond resolution
WallClockTimestampStop	character vector	Wall clock time when the simulation stopped, in YYYY-MM-DD HH:MI:SS format with microsecond resolution
InitializationElapsedWallTime	double	Time spent before execution, in seconds
ExecutionElapsedWallTime	double	Time spent during execution, in seconds
TerminationElapsedWallTime	double	Time spent after execution in, seconds
TotalElapsedWallTime	double	Total time spent in initialization, execution, and termination, in seconds

The `ExecutionElapsedWallTime` includes the time that Simulink spent to roll back or step back in a simulation. The `ExecutionElapsedWallTime` does not include the time spent between steps. For example, if you use `Stepper` to step through a simulation, the `ExecutionElapsedWallTime` time does not include the time when the simulation is in a paused state. For more information about using `Stepper`, see “How Simulation Stepper Helps With Model Analysis”.

UserString — Custom character vector to describe the simulation

character vector

Use `Simulink.SimulationOutput.setUserString` to directly store a character vector in the `SimulationMetadata` object that is contained in the `SimulationOutput` object.

UserData — Custom data to store in SimulationMetadata object that is contained in the SimulationOutput object

character vector

Use `Simulink.SimulationOutput.setUserData` to store custom data in the `SimulationMetadata` object that is contained in the `SimulationOutput` object.

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Get a SimulationMetadata Object for vdp Simulation

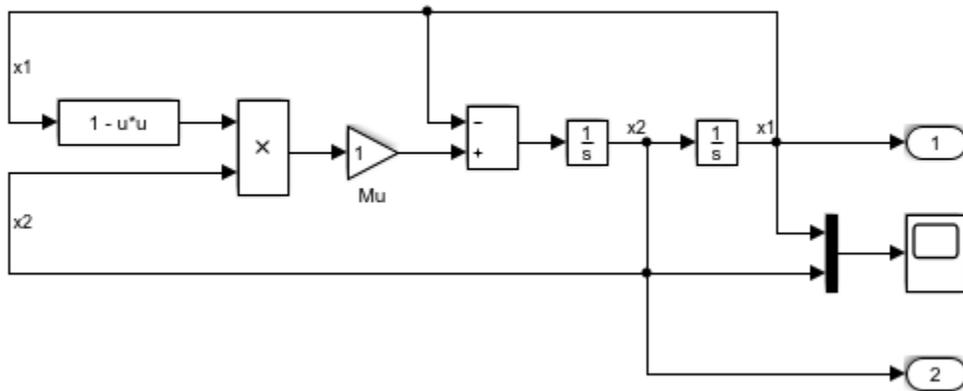
Simulate the `vdp` model. Retrieve metadata from a `SimulationMetadata` object of the simulation.

Simulate the `vdp` model. Save the results of the `Simulink.SimulationOutput` object in `simout`.

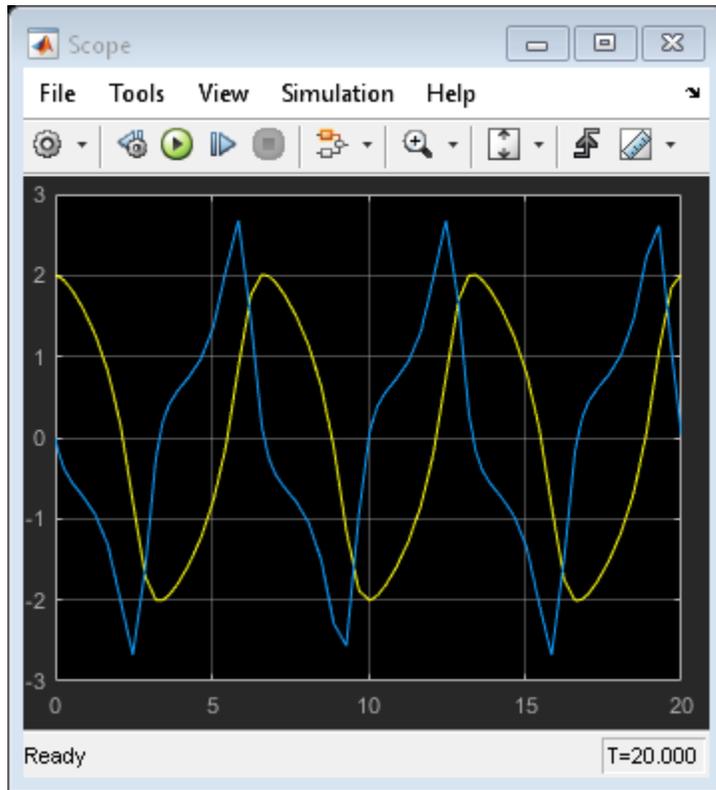
```
open_system('vdp');  
simout = sim(bdroot, 'ReturnWorkspaceOutputs', 'on');
```

?

van der Pol Equation



Copyright 2004-2013 The MathWorks, Inc.



Retrieve metadata information about this simulation using `mData`. This is the `SimulationMetadata` object that `simout` contains.

```
mData=simout.getSimulationMetadata()
```

```
mData =
```

```
SimulationMetadata with properties:
```

```
    ModelInfo: [1x1 struct]
    TimingInfo: [1x1 struct]
    ExecutionInfo: [1x1 struct]
    UserString: ''
    UserData: []
```

Store custom data or string in simout.

```
simout=simout.setUserData(struct('param1','value1','param2','value2','param3','value3'))
simout=simout.setUserString('Store first simulation results');
```

Retrieve the custom data you stored from mData.

```
mData=simout.getSimulationMetadata()
disp(mData.UserData)
```

```
mData =
```

```
SimulationMetadata with properties:

    ModelInfo: [1x1 struct]
    TimingInfo: [1x1 struct]
    ExecutionInfo: [1x1 struct]
    UserString: 'Store first simulation results'
    UserData: [1x1 struct]

    param1: 'value1'
    param2: 'value2'
    param3: 'value3'
```

Retrieve the custom string you stored from mData.

```
mData=simout.getSimulationMetadata()
disp(mData.UserString)
```

```
mData =
```

```
SimulationMetadata with properties:

    ModelInfo: [1x1 struct]
    TimingInfo: [1x1 struct]
    ExecutionInfo: [1x1 struct]
    UserString: 'Store first simulation results'
    UserData: [1x1 struct]
```

Store first simulation results

See Also

`Simulink.SimulationOutput.getSimulationMetadata` |

`Simulink.SimulationOutput.setUserData` |

`Simulink.SimulationOutput.setUserString`

Introduced in R2015a

Simulink.SimulationOutput class

Package: Simulink

Access object values of simulation results

Description

The `SimulationOutput` class contains all simulation outputs, including workspace variables.

You can use dot notation to access the data for simulation outputs. For example, to return data for the `xout` variable for a `simOut` `SimulationOutput` object, use a `simOut.tout` command.

Alternatively, you can use `Simulink.SimulationOutput.who` and either `Simulink.SimulationOutput.get` or `Simulink.SimulationOutput.find` methods to access the output variable names and their respective values.

Properties

SimulationMetadata — Metadata for simulation runs

`Simulink.SimulationMetadata` object

Metadata for simulation runs, returned as a `Simulink.SimulationMetadata` object. Fields other than the `UserData` and `UserString` fields are read only.

ErrorMessage — Simulation logging error messages

char vector

Simulation logging error message, returned as a char vector. (read only)

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#) in the [MATLAB Programming Fundamentals](#) documentation.

Examples

View Simulation Output and Metadata

Simulate a model and place the results of the `Simulink.SimulationOutput` object in `simOut` and view the simulation metadata.

Simulate the `vdp` model.

```
simOut = sim('vdp', 'SimulationMode', 'normal', 'AbsTol', '1e-5', ...
            'SaveState', 'on', 'StateSaveName', 'xoutNew', ...
            'SaveOutput', 'on', 'OutputSaveName', 'youtNew')
```

```
Simulink.SimulationOutput:
```

```
    xoutNew: [65x2 double]
    youtNew: [65x2 double]
```

```
    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
    ErrorMessage: [0x0 char]
```

Get the values of the variable `youtNew`.

```
simOut.youtNew
```

Simulink returns and displays the values.

Get the timing information for the simulation.

```
myMetadata = simOut.SimulationMetadata
```

```
myMetadata =
```

```
SimulationMetadata with properties:

    ModelInfo: [1x1 struct]
    TimingInfo: [1x1 struct]
    ExecutionInfo: [1x1 struct]
    UserString: ''
    UserData: []

myMetadata.TimingInfo

ans =

    struct with fields:

        WallClockTimestampStart: '2016-12-30 08:47:51.739935'
        WallClockTimestampStop: '2016-12-30 08:47:58.185579'
        InitializationElapsedWallTime: 5.9166
        ExecutionElapsedWallTime: 0.1910
        TerminationElapsedWallTime: 0.3380
        TotalElapsedWallTime: 6.4456
```

See Also

[Simulink.SimulationData.BlockPath](#) | [Simulink.SimulationData.Dataset](#) | [loadIntoMemory](#)

Topics

[“Migrate Scripts That Use Legacy ModelDataLogs API”](#)
[“Export Simulation Data”](#)

Simulink.SubsysDataLogs

Container for subsystem signal data logs

Description

Note Before R2016a, the `Simulink.SubsysDataLogs` class was used in conjunction with the `ModelDataLogs` logging data format. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format.

However, you can use data that was logged in a previous release using `ModelDataLogs` format.

In releases before R2016a, Simulink created instances of this class to contain logs for signals in a subsystem models were logged in `ModelDataLogs` format. Objects of this class have a variable number of properties. The first property, named `Name`, is the name of the subsystem whose log data this object contains. The remaining properties are signal log or signal log container objects containing the data logged for the subsystem specified by this object's `Name` property.

For example, suppose you have this logged data from a model run in a release earlier than R2016a:

```
Simulink.SubsysDataLogs (Gain):
  Name           elements  Simulink Class
  a              1        Timeseries
  m              2        TsArray
```

You can use either fully qualified log names or the `unpack` command to access the signal logs contained by a `SubsysDataLogs` object. For example, to access the amplitudes logged for signal `a` in the preceding example, you could enter the following at the MATLAB command line:

```
data = logout.Gain.a.Data;
```

or

```
>> logout.unpack('all');  
data = a.Data;
```

See Also

“Load Signal Data for Simulation”, `Simulink.ModelDataLogs`,
`Simulink.Timeseries`, `Simulink.TsArray`, `Simulink.SimulationData.Dataset`,
`who`, `whos`, `unpack`

Introduced before R2006a

Simulink.SuppressedDiagnostic class

Package: Simulink

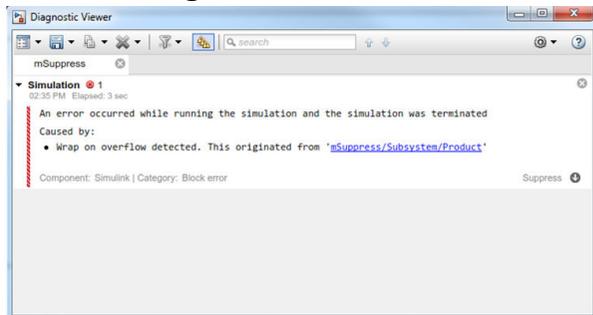
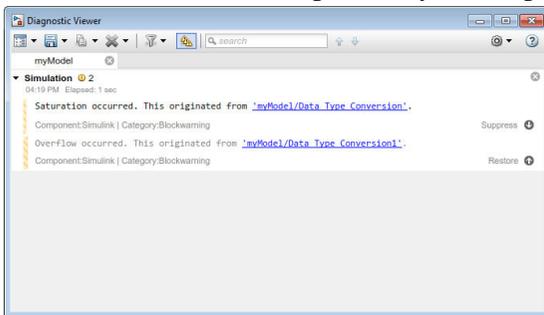
Suppress diagnostic messages from a specified block

Description

A `Simulink.SuppressedDiagnostic` object contains information related to diagnostic warnings or errors that are suppressed from being thrown during simulation.

Construction

The Diagnostic Viewer in Simulink includes an option to suppress certain diagnostics. This feature enables you to suppress warnings or errors for specific objects in your model. Click the **Suppress** button next to the warning in the Diagnostic Viewer to suppress the warning from the specified source. This action creates a `Simulink.SuppressedDiagnostic` object. You can access this object at the MATLAB command line using the `Simulink.getSuppressedDiagnostics` function. You can restore the diagnostic by clicking **Restore this warning**.



`DiagnosticObject = Simulink.SuppressedDiagnostic(source, message_id)` creates a suppressed diagnostic object. The object suppresses all instances of diagnostics represented by `message_id` thrown by the blocks specified by `source`.

Input Arguments

source — System, block, or model object throwing diagnostic

model | subsystem | block path | block handle

The source of the diagnostic, specified as a model, subsystem, block path, block handle, cell array of block paths, or cell array of block handles.

To get the block path, use the `gcb` function.

To get the block handle, use the `getSimulinkBlockHandle` function.

Data Types: char | cell

message_id — message identifier of diagnostic

message identifier | cell array of message identifiers

The message identifier of the diagnostic, specified as a character vector or a cell array of character vectors. You can find the message identifier of diagnostics thrown during simulation by accessing the `ExecutionInfo` property of the `Simulink.SimulationMetadata` object associated with a simulation. You can also use the `lastwarn` function.

Data Types: char | cell

Properties

Comments — Comments associated with the suppression object

character vector

Comments associated with the suppression object, specified as a character vector. This property is optional.

Data Types: char

ID — Message identifier of the diagnostic that was suppressed

character vector

The message identifier of the diagnostic that was suppressed, specified as a character vector.

Data Types: char

LastModified — Date the suppression object was last modified

character vector

Date the suppression object was last modified, specified as a character vector. This property is read-only.

Data Types: `char`**LastModifiedBy** — Name of the user who was last to add or edit the suppression object

character vector

Name of the user who last to add or edit the suppression object, specified as a character vector. This property is optional.

Data Types: `char`**Source** — block path of the source of the diagnostic

character vector

The block path of the model object that has a suppressed diagnostic, specified as a character vector.

Data Types: `char`

Methods

`restore` Remove specified diagnostic suppressions

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a Simulink.SuppresseDiagnostic Object

Using the model from “Suppress Diagnostic Messages Programmatically”, create and then restore a diagnostic suppression.

Create a Simulink.SuppresseDiagnostic object, suppression to suppress the parameter precision loss warning thrown by the Constant block, one.

```
suppression = Simulink.SuppresseDiagnostic('Suppressor_CLI_Demo/one', ...  
'SimulinkFixedPoint:util:fxpParameterPrecisionLoss');
```

The parameter precision loss warning is no longer thrown in future simulations of this model.

Add accountability information to the object by editing the LastModifiedBy and Comments properties of the object.

```
suppression.LastModifiedBy = 'John Doe';  
suppression.Comments = 'Reviewed: Joe Schmoe'
```

```
suppression =
```

```
SuppresseDiagnostic with properties:
```

```
    Source: 'Suppressor_CLI_Demo/one'  
    Id: 'SimulinkFixedPoint:util:fxpParameterPrecisionLoss'  
LastModifiedBy: 'John Doe'  
    Comments: 'Reviewed: Joe Schmoe'  
    LastModified: '2016-Jun-01 17:25:21'
```

You can restore the diagnostic using the restore method.

```
restore(suppression);
```

- “Suppress Diagnostic Messages Programmatically”

See Also

Simulink.SuppresseDiagnostic | Simulink.SuppresseDiagnostic.restore
| Simulink.getSuppresseDiagnostics | Simulink.restoreDiagnostic |
Simulink.suppressDiagnostic

Topics

“Suppress Diagnostic Messages Programmatically”

Class Attributes (MATLAB)

Property Attributes (MATLAB)

Introduced in R2016b

restore

Class: Simulink.SuppressedDiagnostic

Package: Simulink

Remove specified diagnostic suppressions

Syntax

```
restore(SuppressedDiagnostic)
```

Description

`restore(SuppressedDiagnostic)` removes the specified suppressed diagnostic object.

Input Arguments

SuppressedDiagnostic — Suppressed diagnostic object to restore

`Simulink.SuppressedDiagnostic` object

`Simulink.SuppressedDiagnostic` object

Examples

Restore a Suppressed Diagnostic

Using the model from “Suppress Diagnostic Messages Programmatically”, create and then restore a diagnostic suppression.

Create a `Simulink.SuppressedDiagnostic` object, suppression to suppress the parameter precision loss warning from the Constant block, one.

```
suppression = Simulink.SuppressedDiagnostic('Suppressor_CLI_Demo/one', ...  
    'SimulinkFixedPoint:util:fxpParameterPrecisionLoss');
```

You can restore the diagnostic using the `restore` method.

```
restore(suppression);
```

Restore All Suppressed Diagnostics

Using the model from “Suppress Diagnostic Messages Programmatically”, restore all diagnostic suppressions associated with a model.

Use the `Simulink.suppressDiagnostic` function to suppress the parameter precision loss and parameter underflow warnings from the Constant block, `one`.

```
diags = {'SimulinkFixedPoint:util:fxpParameterPrecisionLoss', 'SimulinkFixedPoint:util:fxpParameterUnderflowWarning'};
Simulink.suppressDiagnostic('Suppressor_CLI_Demo/one', diags);
```

Use the `Simulink.getSuppressedDiagnostics` function to get all suppressions associated with the model, returned as an array of `Simulink.SuppressedDiagnostic` objects.

```
suppressed_diagnostics = Simulink.getSuppressedDiagnostics('Suppressor_CLI_Demo')
```

```
suppressed_diagnostics =
```

```
1×2 SuppressedDiagnostic array with properties:
```

```
Source
Id
LastModifiedBy
Comments
LastModified
```

Restore all diagnostics using the `restore` method and iterating through the `suppressed_diagnostics` array.

```
for iter = 1:numel(suppressed_diagnostics)
    restore(suppressed_diagnostics(iter));
end
```

- “Suppress Diagnostic Messages Programmatically”

See Also

`Simulink.SuppressedDiagnostic` | `Simulink.getSuppressedDiagnostics` |
`Simulink.restoreDiagnostic` | `Simulink.suppressDiagnostic`

Topics

“Suppress Diagnostic Messages Programmatically”

Introduced in R2016b

Simulink.TimeInfo

Provide information about time data in `Simulink.Timeseries` object

Description

Simulink software creates instances of these objects to describe the time data that it includes in `Simulink.Timeseries` objects.

Note The `Simulink.Timeseries` class is supported for backwards compatibility. The `ModelDataLogs` format created `Simulink.Timeseries` objects for signal logging data. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy `ModelDataLogs` API”.

Properties

Name	Access	Description
Units	RW	The units, e.g., 'seconds', in which the time series data are expressed in the associated <code>Simulink.Timeseries</code> object.

Name	Access	Description
Start	RW	If the associated signal is not in a conditionally executed subsystem, this field contains the simulation time of the first signal value recorded in the associated <code>Simulink.Timeseries</code> object. If the signal is in a conditionally executed subsystem, this field contains an array of times when the system became active.
end	RW	If the associated signal is not in a conditionally executed subsystem, this field contains the simulation time of the last signal value recorded in the associated <code>Simulink.Timeseries</code> object. If the signal is in a conditionally executed subsystem, this field contains an array of times when the system became inactive.
Increment	RW	The interval between simulation times at which signal data is logged in the associated <code>Simulink.Timeseries</code> object. If the signal is aperiodic (continuous signal with variable-step solver), this property has a value of <code>NaN</code> . A signal is periodic if it has a discrete sample time (not continuous or constant) or is continuous with a fixed-step solver.
Length	W	The number of signal samples recorded in the associated <code>Simulink.Timeseries</code> object, i.e., the length of the arrays referenced by the object's <code>Time</code> and <code>Data</code> properties.

See Also

`Simulink.Timeseries` , `Simulink.SimulationData.Dataset`

Introduced before R2006a

Simulink.Timeseries

Store data for any signal except mux or bus signal

Description

Note The `Simulink.Timeseries` class is supported for backwards compatibility. The `ModelDataLogs` format created `Simulink.Timeseries` objects for signal logging data. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy `ModelDataLogs` API”.

Simulink software creates instances of this class to store signal data that it logs for any signal except a mux or bus signal, which are stored in a `Simulink.TsArray`. See “Export Signal Data Using Signal Logging” for more information.

Properties

Name	Access	Description
Name	RW	Name of this signal log.
BlockPath	RW	Path of the block that output the signal logged in this signal log.

Name	Access	Description
PortIndex	RW	Index of the output port that emitted the signal logged in this signal log.
SignalName	RW	Name of the signal logged in this signal log.
ParentName	RW	Name of the parent of the signal recorded in this log, if the signal is an element of a mux or a virtual bus; otherwise, the same as SignalName.
TimeInfo	RW	An object of <code>Simulink.TimeInfo</code> class that describes the time data in this log.
Time	RW	An array containing the simulation times at which signal data was logged.
Data	RW	An array containing the signal data.

See Also

“Export Signal Data Using Signal Logging”, `Simulink.TimeInfo`, `Simulink.SimulationData.Dataset`, `Simulink.ModelDataLogs`, `Simulink.SubsysDataLogs`, `Simulink.TsArray`, `who`, `whos`, `unpack`

Introduced before R2006a

Simulink.TsArray

Store data for mux or bus signal

Description

Note Before R2016a, the `Simulink.TsArray` class was used in conjunction with the `ModelDataLogs` logging data format. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format.

However, you can use data that was logged in a previous release using `ModelDataLogs` format.

In releases earlier than R2016a, Simulink software created instances of this class to contain the data that it logs for a mux or bus signal. Other types of signals were stored in a `Simulink.Timeseries`.

Objects of the `Simulink.TsArray` class have a variable number of properties. The first property, called `Name`, specifies the log name of the logged signal. The remaining properties reference logs for the elements of the logged signal: `Simulink.Timeseries` objects for elementary signals and `Simulink.TsArray` objects for mux or bus signals. The name of each property is the log name of the corresponding signal.

For example, suppose you have this logged data from a model run in a release earlier than R2016a that was configured to log in `ModelDataLogs` format.

```
logout.b2
```

```
Simulink.TsArray (untitled/Bus Creator1):
  Name          elements  Simulink Class
  x1            1         Timeseries
  b1            2         TsArray
```

The `Simulink.ModelDataLogs` object, named `logout`, contains a `Simulink.TsArray` object, named `b2`, that contains the logs for the elements of `b2` (that

is, the elementary signal `x1` and the bus signal `b1`). Entering the fully qualified name of the `Simulink.TsArray` object, (`logouts.b2`) at the MATLAB command line reveals the structure of the signal log for this model.

You can use either fully qualified log names or the `unpack` command to access the signal logs contained by a `Simulink.TsArray` object. For example, to access the amplitudes logged for signal `x1` in the preceding example, you can enter the following at the MATLAB command line:

```
data = logouts.b2.x1.Data;
```

or

```
logouts.unpack('all');  
data = x1.Data;
```

See Also

`Simulink.ModelDataLogs`, `Simulink.SubsysDataLogs`, `Simulink.Timeseries`, `Simulink.SimulationData.Dataset`, `who`, `whos`, `unpack`

Introduced before R2006a

Simulink.Variant class

Package: Simulink

Specify conditions that control variant selection

Description

An object of the `Simulink.Variant` class represents a conditional expression called a variant control. The object allows you to specify a Boolean expression that activates a specific variant choice when it evaluates to `true`.

A variant control comprises one or more variant control variables, specified using MATLAB variables or `Simulink.Parameter` objects.

You specify variant controls for each variant choice represented in a Variant Subsystem or Model Variant block. For a given Variant Subsystem or Model Variant block, only one variant control can evaluate to `true` at a time. When a variant control evaluates to `true`, Simulink activates the variant choice that corresponds to that variant control.

Construction

`variantControl = Simulink.Variant(conditionExpression)` creates a variant control.

Properties

conditionExpression — Variant condition expression

'(default)' | character vector

Variant condition expression, specified as a character vector containing one or more of these operands and operators.

Operands

- Variable names that resolve to MATLAB variables or `Simulink.Parameter` objects with integer or enumerated data type and scalar literal values
- Variable names that resolve to `Simulink.Variant` objects
- Scalar literal values that represent integer or enumerated values

Operators

- Parentheses for grouping
- Arithmetic, relational, logical, or bit-wise operators

The variant condition expression evaluates to a Boolean value. This property has read and write access.

Example: `'(Fuel==2 || Emission==1) && Ratio==2'`

Examples

Create Variant Controls Using MATLAB Variables

Use MATLAB variables when you want to simulate the model but are not considering code generation.

Create MATLAB variables with scalar literal values.

```
Fuel = 3;  
Emission = 1;  
Ratio = 3;
```

Develop conditional expressions using the variables.

```
Variant1 = Simulink.Variant('Fuel==1 && Emission==2');  
Variant2 = Simulink.Variant('(Fuel==2 || Emission==1) && Ratio==2');  
Variant3 = Simulink.Variant('Fuel==3 || Ratio==4');
```

Create Variant Controls Using `simulink.Parameter` Objects

If you want to generate preprocessor conditionals for code generation, use `Simulink.Parameter` objects instead of MATLAB variables.

Create variant `Simulink.Parameter` objects with scalar literal values.

```
Fuel = Simulink.Parameter(3);  
Emission = Simulink.Parameter(1);  
Ratio = Simulink.Parameter(3);
```

Specify the custom storage class for these objects as `ImportedDefine` so that the values are specified by an external header file.

Other valid values for the custom storage class are `Define` and `CompilerFlag`.

```
Fuel.CoderInfo.StorageClass = 'Custom';  
Fuel.CoderInfo.CustomStorageClass = 'ImportedDefine';  
  
Emission.CoderInfo.StorageClass = 'Custom';  
Emission.CoderInfo.CustomStorageClass = 'ImportedDefine';  
  
Ratio.CoderInfo.StorageClass = 'Custom';  
Ratio.CoderInfo.CustomStorageClass = 'ImportedDefine';
```

Develop conditional expressions using the variables and create variant controls.

```
Variant1 = Simulink.Variant('Fuel==1 && Emission==2');  
Variant2 = Simulink.Variant('(Fuel==2 || Emission==1) && Ratio==2');  
Variant3 = Simulink.Variant('Fuel==3 || Ratio==4');
```

See Also

“Operators and Operands in Variant Condition Expressions”

Topics

“Define, Configure, and Activate Variants”

“Convert Variant Control Variables into Simulink.Parameter Objects”

“Approaches for Specifying Variant Controls”

Simulink.VariantConfigurationData class

Package: Simulink

Class representing a variant configurations data object

Description

The variant configuration data object, stores a collection of variant configurations, constraints, and the name of the default active configuration. The `Simulink.VariantConfigurationData` class has properties that enable you to add, modify, or remove variant configurations, constraints, and control variables. Use an instance of `Simulink.VariantConfigurationData` class to do the following:

- Define and edit variant configurations.
- Add control variables to variant configurations.
- Add copy of variant configuration.
- Delete existing variant configurations, constraints, and sub model configurations.
- Set a specific configuration as default active.
- Validate model using default or a specific variant configuration.
- Query or create variant configurations data object for a given model.

Properties

VariantConfigurations

Set of variant configurations. The names of the configurations must be unique and valid MATLAB variable names.

Constraints

Set of constraints that must always be satisfied by the model for all variant configurations. The name of the constraints must be unique and valid MATLAB variable names.

DefaultConfigurationName

Name of the variant configuration to be used by default for validation.

Methods

<code>addConfiguration</code>	Add a new variant configuration to the variant configuration data object
<code>addConstraint</code>	Add a constraint to the variant configuration data object
<code>addControlVariables</code>	Add control variables to an existing variant configuration
<code>addCopyOfConfiguration</code>	Add a copy of an existing variant configuration to the variant configuration data object
<code>addSubModelConfigurations</code>	Add to a variant configuration the names of the configurations to be used for submodels
<code>existsFor</code>	Check if variant configuration data object exists for a model
<code>getConfiguration</code>	Returns the variant configuration with a given name from a variant configuration data object
<code>getDefaultConfiguration</code>	Returns default variant configuration, if any, for a variant configuration data object
<code>getFor</code>	Get existing variant configuration data object for a model
<code>getOrCreateFor</code>	Get existing or create a new variant configuration data object for a model
<code>removeConfiguration</code>	Remove a variant configuration with a given name from the variant configuration data object
<code>removeConstraint</code>	Remove a constraint from the variant configuration data object
<code>removeControlVariable</code>	Remove a control variable from a variant configuration
<code>removeSubModelConfiguration</code>	Remove from a variant configuration, the configuration to be used for a sub model.
<code>setDefaultConfigurationName</code>	Set name of the default variant configuration for a variant configuration data object
<code>validateModel</code>	Validate all variant blocks in the model and submodels in the hierarchy during simulation
<code>VariantConfigurationData</code>	Object constructor with optional arguments for variant configurations, constraints, and default configuration name

Examples

```
load_system(model);  
% Create variant config and associate it with model  
variantConfig = Simulink.VariantConfigurationData;  
set_param(model, 'VariantConfigurationObject', 'variantConfig');
```

See Also

Topics

“Variant Manager Overview”

addConfiguration

Class: Simulink.VariantConfigurationData

Package: Simulink

Add a new variant configuration to the variant configuration data object

Syntax

```
vcdataObj.addConfiguration(name)
vcdataObj.addConfiguration(name,description)
vcdataObj.addConfiguration(name,description,controlVars)
vcdataObj.addConfiguration(name,description,controlVars,
subModelConfigurations)
```

Description

`vcdataObj.addConfiguration(name)` adds a new variant configuration with a given name to the variant configuration data object.

`vcdataObj.addConfiguration(name,description)` adds a new variant configuration with a given name and optional description to the variant configuration data object.

`vcdataObj.addConfiguration(name,description,controlVars)` adds a new variant configuration with a given name, optional description, and control variables to the variant configuration data object.

`vcdataObj.addConfiguration(name,description,controlVars,subModelConfigurations)` adds a new variant configuration with a given name, optional description, control variables, and submodel configurations to the variant configuration data object.

Input Arguments

name

Name of variant configuration being added.

description

Description text for the variant configuration being added.

controlVars

Control variables for the variant configuration being added. This argument must be a vector of structures with required fields: `Name` and `Value`. The values assigned to the `Name` field must be unique and valid MATLAB variable names. The `Value` field can contain either character vectors or `Simulink.Parameter` objects. The values of control variables are checked during validation of the variant configuration.

subModelConfigurations

Vector of structures containing fields: `ModelName`, `ConfigurationName`. The names of submodels must be unique and valid MATLAB variable names and configuration names must be valid MATLAB variables.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add a variant configuration LinInterExp
vcdataObj.addConfiguration('LinInterExp')
```

See Also

```
Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addControlVariables |
Simulink.VariantConfigurationData.addSubModelConfigurations
```

addConstraint

Class: Simulink.VariantConfigurationData

Package: Simulink

Add a constraint to the variant configuration data object

Syntax

```
vcdataObj.addConstraint(nameOfConstraint)
vcdataObj.addConstraint(nameOfConstraint, condition)
vcdataObj.addConstraint(nameOfConstraint, condition, description)
```

Description

`vcdataObj.addConstraint(nameOfConstraint)` adds a new constraint with a given name to `vcdataObj`.

`vcdataObj.addConstraint(nameOfConstraint, condition)` adds a new constraint with a given name and condition expression to `vcdataObj`.

`vcdataObj.addConstraint(nameOfConstraint, condition, description)` adds a new constraint with a given name, condition expression, and description to `vcdataObj`.

Input Arguments

nameOfConstraint

Name of constraint being added. Must be unique and valid MATLAB variable name.

condition

Boolean expression that must evaluate to true. When the expression evaluates to true, it means the constraint is satisfied.

description

Text that describes the constraint.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add a constraint named LinNotExtern
vcdataObj.addConstraint('LinNotExtern','((Ctrl~=1)...
    || (PlantLocation ~=1))','Description of the constraint')
```

See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addConfiguration |
Simulink.VariantConfigurationData.removeConfiguration |
Simulink.VariantConfigurationData.removeConstraint

addControlVariables

Class: Simulink.VariantConfigurationData

Package: Simulink

Add control variables to an existing variant configuration

Syntax

```
vcdataObj.addControlVariables(nameOfConfiguration, controlVars)
```

Description

`vcdataObj.addControlVariables(nameOfConfiguration, controlVars)`, adds control variables to a variant configuration.

Input Arguments

nameOfConfiguration

Specifies the name of an existing configuration.

controlVars

Control variables being added. This argument must be a vector of structures with required fields: `Name` and `Value`. The values assigned to the `Name` field must be unique and valid MATLAB variable names. The `Value` field can contain either character vectors or `Simulink.Parameter` objects. The values of control variables are checked during validation of the variant configuration.

Examples

```
% Define the variant configuration data object  
vcdataObj = Simulink.VariantConfigurationData;
```

```
% Add a variant configuration named LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
'Linear Internal Experimental Plant Controller');

% Add control variables SmartSensor1Mod and PlanLocation
vcdataObj.addControlVariables('LinInterExp',...
    cell2struct({'SmartSensor1Mod', '2';...
                'PlantLocation', '1'},...
                {'Name', 'Value'}, 2))
```

See Also

[Simulink.VariantConfigurationData](#) |
[Simulink.VariantConfigurationData.addSubModelConfigurations](#) |
[Simulink.VariantConfigurationData.removeControlVariable](#) |
[Simulink.VariantConfigurationData.removeSubModelConfiguration](#)

addCopyOfConfiguration

Class: Simulink.VariantConfigurationData

Package: Simulink

Add a copy of an existing variant configuration to the variant configuration data object

Syntax

```
vcdataObj.addCopyOfConfiguration(nameOfExistingConfiguration)
vcdataObj.addCopyOfConfiguration(nameOfExistingConfiguration,
nameOfTobeAddedConfiguration)
```

Description

`vcdataObj.addCopyOfConfiguration(nameOfExistingConfiguration)`, adds a new configuration with a default name (default name is based on existing configuration name being copied) as a copy of the existing configuration to the variant configuration data object.

`vcdataObj.addCopyOfConfiguration(nameOfExistingConfiguration, nameOfTobeAddedConfiguration)`, adds a new configuration with a specified name, as a copy of the existing configuration, to the variant configuration data object.

Input Arguments

nameOfExistingConfiguration

Name of existing configuration.

Default:

nameOfTobeAddedConfiguration

Name of new configuration to be added as a copy of the configuration.

Default:**Examples**

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the variant configuration LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
    'Linear Internal Experimental Plant Controller');

% Add a copy of variant configuration LinInterExp
% and name the copy as LinExtExp
vcdataObj.addCopyOfConfiguration('LinInterExp','LinExtExp')
```

See Also

```
Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addConfiguration |
Simulink.VariantConfigurationData.removeConfiguration |
Simulink.VariantConfigurationData.setDefaultConfiguration
```

addSubModelConfigurations

Class: Simulink.VariantConfigurationData

Package: Simulink

Add to a variant configuration the names of the configurations to be used for submodels

Syntax

```
vcdataObj.addSubModelConfigurations (nameOfConfiguration,  
subModelConfigurations)
```

Description

`vcdataObj.addSubModelConfigurations (nameOfConfiguration, subModelConfigurations)`, specifies names of the configurations to be used for submodels.

Input Arguments

nameOfConfiguration

Name for the configuration of submodels that are model references.

subModelConfigurations

Vector of structures containing fields: `ModelName`, `ConfigurationName`. The names of submodels must be unique and valid MATLAB variable names and configuration names must be valid MATLAB variables.

Examples

```
% Add the path to the model file  
addpath(fullfile(docroot, 'toolbox', 'simulink', 'examples'));
```

```
% Load the model
load_system('slexVariantManagementExample');

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the variant configuration LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
    'Linear Internal Experimental Plant Controller');

% Add a new submodel configuration to LinInterExp
vcdataObj.addSubModelConfigurations('LinInterExp',...
    [struct('ModelName', 'slexVariantManagementExternalPlantMdlRef',...
        'ConfigurationName', 'LowFid')])
```

See Also

[Simulink.VariantConfigurationData](#) |
[Simulink.VariantConfigurationData.addControlVariables](#) |
[Simulink.VariantConfigurationData.removeControlVariable](#) |
[Simulink.VariantConfigurationData.removeSubModelConfiguration](#)

existsFor

Class: Simulink.VariantConfigurationData

Package: Simulink

Check if variant configuration data object exists for a model

Syntax

```
Simulink.VariantConfigurationData.existsFor(modelNameOrHandle)
```

Description

`Simulink.VariantConfigurationData.existsFor(modelNameOrHandle)` returns true if the variant configuration data object exists for the model.

Input Arguments

modelNameOrHandle

Name or handle to the model.

Examples

```
% Add the path to the model file
addpath(fullfile(docroot,'toolbox','simulink','examples'));

% Load the model
load_system('slexVariantManagementExample');

% Checks whether a variant configuration
% data object exists for model
[exists] = Simulink.VariantConfigurationData.existsFor...
('slexVariantManagementExample')
```

See Also

`Simulink.VariantConfigurationData` |
`Simulink.VariantConfigurationData.getFor`

getConfiguration

Class: Simulink.VariantConfigurationData

Package: Simulink

Returns the variant configuration with a given name from a variant configuration data object

Syntax

```
vcdataObj.getConfiguration(nameOfConfiguration)
```

Description

`vcdataObj.getConfiguration(nameOfConfiguration)` returns a specific variant configuration that is associated with the variant configuration data object.

Input Arguments

nameOfConfiguration

Name of the variant configuration to be returned.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the variant configuration LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
    'Linear Internal Experimental Plant Controller');

% Add a control variable SmartSensor1Mod
vcdataObj.addControlVariables('LinInterExp',...
    [struct('Name','SmartSensor1Mod','Value','2')]);
```

```
% Obtain information on the variant configuration..  
% LinInterExp from the variant configuration data object  
vc = vcdDataObj.getConfiguration('LinInterExp')
```

See Also

[Simulink.VariantConfigurationData](#) |
[Simulink.VariantConfigurationData.addConfiguration](#) |
[Simulink.VariantConfigurationData.getDefaultConfiguration](#) |
[Simulink.VariantConfigurationData.removeConfiguration](#)

getDefaultConfiguration

Class: Simulink.VariantConfigurationData

Package: Simulink

Returns default variant configuration, if any, for a variant configuration data object

Syntax

```
vcdataObj.getDefaultConfiguration
```

Description

`vcdataObj.getDefaultConfiguration` returns the default variant configuration. If no default variant configuration is defined, then `[]` is returned.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the variant configuration named LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
    'Linear Internal Experimental Plant Controller');

% Add the variant configuration LinInterStd
vcdataObj.addConfiguration('LinInterStd',...
    'Linear Internal Standard Plant Controller');

% Set LinExtExp as the default variant configuration
vcdataObj.setDefaultConfigurationName('LinExtExp');

% Obtain the default variant configuration
defvc = vcdataObj.getDefaultConfiguration
```

See Also

`Simulink.VariantConfigurationData` |

`Simulink.VariantConfigurationData.getConfiguration` |

`Simulink.VariantConfigurationData.setDefaultConfigurationName`

getFor

Class: Simulink.VariantConfigurationData

Package: Simulink

Get existing variant configuration data object for a model

Syntax

```
Simulink.VariantConfigurationData.getFor(modelNameOrHandle)
```

Description

`Simulink.VariantConfigurationData.getFor(modelNameOrHandle)`, returns the variant configuration object for the model. If no default variant configuration is defined, then `[]` is returned.

Input Arguments

modelNameOrHandle

Model name or handle.

Examples

```
% Add the path to the model file
addpath(fullfile(docroot,'toolbox','simulink','examples'));

% Load the model
load_system('slexVariantManagementExample');

% Obtain variant configuration data object for the model
% slexVariantManagementExample
vcdataObj = Simulink.VariantConfigurationData.getFor...
('slexVariantManagementExample')
```

See Also

`Simulink.VariantConfigurationData` |
`Simulink.VariantConfigurationData.existsFor` |
`Simulink.VariantConfigurationData.getOrCreateFor`

getOrCreateFor

Class: Simulink.VariantConfigurationData

Package: Simulink

Get existing or create a new variant configuration data object for a model

Syntax

```
Simulink.VariantConfigurationData.getOrCreateFor(modelNameOrHandle)
```

Description

`Simulink.VariantConfigurationData.getOrCreateFor(modelNameOrHandle)`, returns the object if the variant configuration data objects exists otherwise, creates an empty object.

Input Arguments

modelNameOrHandle

Model name or handle to the model.

Examples

```
% Add the path to the model file
addpath(fullfile(docroot, 'toolbox', 'simulink', 'examples'));

% Load the model
load_system('slexVariantManagementExample');

% Obtain existing or create an empty variant configuration
% data object for the slexVariantManagementExample model
vcdataObj = Simulink.VariantConfigurationData.getOrCreateFor...
('slexVariantManagementExample')
```

See Also

`Simulink.VariantConfigurationData` |
`Simulink.VariantConfigurationData.existsFor` |
`Simulink.VariantConfigurationData.getFor`

removeConfiguration

Class: Simulink.VariantConfigurationData

Package: Simulink

Remove a variant configuration with a given name from the variant configuration data object

Syntax

```
vcdataObj.removeConfiguration(nameOfConfiguration)
```

Description

`vcdataObj.removeConfiguration(nameOfConfiguration)` removes the configuration from the variant configuration data object.

Input Arguments

nameOfConfiguration

Name of the configuration to be removed.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the LinInterExp variant configuration
% to the variant configuration data object
vcdataObj.addConfiguration('LinInterExp',...
    'Linear Internal Experimental Plant Controller');

% Remove the LinInterExp configuration
```

```
% from the variant configuration data object  
vcdataObj.removeConfiguration('LinInterExp')
```

See Also

```
Simulink.VariantConfigurationData |  
Simulink.VariantConfigurationData.addConfiguration |  
Simulink.VariantConfigurationData.getConfiguration
```

removeConstraint

Class: Simulink.VariantConfigurationData

Package: Simulink

Remove a constraint from the variant configuration data object

Syntax

```
vcdataObj.removeConstraint(nameOfConstraint)
```

Description

`vcdataObj.removeConstraint(nameOfConstraint)`, removes the constraint from the variant configuration data object.

Input Arguments

nameOfConstraint

Name of the constraint to be removed.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add a constraint named LinNotExtern
vcdataObj.addConstraint('LinNotExtern','((Ctrl~=1)...
    || (PlantLocation ~=1))',..
    'Description of the constraint');

% Remove the constraint LinNotExtern
% from the variant configuration
vcdataObj.removeConstraint('LinNotExtern')
```

See Also

`Simulink.VariantConfigurationData` |

`Simulink.VariantConfigurationData.addConstraint`

removeControlVariable

Class: Simulink.VariantConfigurationData

Package: Simulink

Remove a control variable from a variant configuration

Syntax

```
vcdataObj.removeControlVariable(nameOfConfiguration,  
nameOfControlVariable)
```

Description

`vcdataObj.removeControlVariable(nameOfConfiguration, nameOfControlVariable)` removes a control variable from a variant configuration.

Input Arguments

nameOfConfiguration

Name of the variant configuration.

nameOfControlVariable

Name of the control variable to be deleted.

Examples

```
% Define the variant configuration data object  
vcdataObj = Simulink.VariantConfigurationData;  
  
% Add a variant configuration named LinInterExp  
vcdataObj.addConfiguration('LinInterExp',...  
    'Linear Internal Experimental Plant Controller');
```

```
% Add control variables SmartSensor1Mod and PlanLocation
vcdataObj.addControlVariables('LinInterExp',...
    [struct('Name','SmartSensor1Mod','Value','2')]);

% Remove the control variable SmartSensor1Mod
% from the configuration LinInterExp
vcdataObj.removeControlVariable('LinInterExp',...
    'SmartSensor1Mod')
```

See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addControlVariables

removeSubModelConfiguration

Class: Simulink.VariantConfigurationData

Package: Simulink

Remove from a variant configuration, the configuration to be used for a sub model.

Syntax

```
vcdataObj.removeSubModelConfiguration(nameOfConfiguration,  
nameOfSubModel)
```

Description

`vcdataObj.removeSubModelConfiguration(nameOfConfiguration, nameOfSubModel)`, removes the configuration specified for a submodel.

Input Arguments

nameOfConfiguration

Name of the submodel configuration to be removed.

nameOfSubModel

Name of the submodel from which the configuration must be removed.

Examples

```
% Load the model  
load_system('slexVariantManagementExample');  
  
% Define the variant configuration data object  
vcdataObj = Simulink.VariantConfigurationData;
```

```
% Add the variant configuration named LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
    'Linear Internal Experimental Plant Controller',controlvars);

% Add a new submodel configuration to LinInterExp
vcdataObj.addSubModelConfigurations('LinInterExp',...
    [struct('ModelName','slexVariantManagementExternalPlantMdlRef',...
        'ConfigurationName','LowFid')]);

% Remove the submodel configuration LinInterExp
% from the submodel slexVariantManagementExternalPlantMdlRef
vcdataObj.removeSubModelConfiguration('LinInterExp',...
    'slexVariantManagementExternalPlantMdlRef')
```

See Also

[Simulink.VariantConfigurationData](#) |
[Simulink.VariantConfigurationData.addSubModelConfigurations](#)

setDefaultConfigurationName

Class: Simulink.VariantConfigurationData

Package: Simulink

Set name of the default variant configuration for a variant configuration data object

Syntax

```
vcdataObj.setDefaultConfigurationName(nameOfConfiguration)
```

Description

`vcdataObj.setDefaultConfigurationName(nameOfConfiguration)` sets the default configuration name. A variant configuration must exist with the same name. If an empty value is passed, then the default configuration name is cleared.

Input Arguments

nameOfConfiguration

Name of the configuration to be set as the default.

Examples

```
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the LinInterExp variant configuration
vcdataObj.addConfiguration('LinInterExp',...
    'Linear Internal Experimental Plant Controller');

% Set the configuration LinInterExp as default
vcdataObj.setDefaultConfigurationName('LinInterExp');
```

```
% Obtain the default variant configuration  
dconfig = vcdatabObj.getDefaultConfiguration
```

See Also

[Simulink.VariantConfigurationData](#) |
[Simulink.VariantConfigurationData.getDefaultConfiguration](#)

validateModel

Class: Simulink.VariantConfigurationData

Package: Simulink

Validate all variant blocks in the model and submodels in the hierarchy during simulation

Syntax

```
Simulink.VariantConfigurationData.validateModel(modelName)  
Simulink.VariantConfigurationData.validateModel(modelName,  
configName)
```

Description

`Simulink.VariantConfigurationData.validateModel(modelName)`, validates the model and referenced models during simulation.

`Simulink.VariantConfigurationData.validateModel(modelName, configName)`, validates the model and referenced models during simulation optionally using a variant configuration.

Input Arguments

modelName

Name of the model

configName

Name of the configuration to be validated

Examples

```
% Add the path to the model file
addpath(fullfile(docroot, 'toolbox', 'simulink', 'examples'));

% Load the model
load_system('slexVariantManagementExample');

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add a variant configuration LinInterExp
vcdataObj.addConfiguration('LinInterExp');

% Add control variables to LinInterExp
vcdataObj.addControlVariables('LinInterExp',...
    cell2struct({'Ctrl', '1';...
               'PlantLocation', '2';...
               'SimType', '2'},...
               {'Name', 'Value'}, 2));

% Associate this object with the model
set_param('slexVariantManagementExample',...
    'VariantConfigurationObject', 'vcdataObj');

% Validate the model slexVariantManagementExample using
% the configuration LinInterExp
[valid, errors] = Simulink.VariantConfigurationData.validateModel...
    ('slexVariantManagementExample', 'LinInterExp')
```

See Also

```
Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.existsFor |
Simulink.VariantConfigurationData.getFor |
Simulink.VariantConfigurationData.getOrCreateFor
```

VariantConfigurationData

Class: Simulink.VariantConfigurationData

Package: Simulink

Object constructor with optional arguments for variant configurations, constraints, and default configuration name

Syntax

```
vardataObj = Simulink.VariantConfigurationData(  
variantConfigurations)
```

Description

`vardataObj = Simulink.VariantConfigurationData(
variantConfigurations)`, constructor that creates an empty variant configuration data object. Optionally, can also accept constraints and a default configuration name as inputs.

Input Arguments

variantConfigurations

Configurations that are part of the variant configuration data object.

constraints

Constraints to be satisfied by the model.

defaultConfigurationName

Name of the default configuration

Examples

```
% Create an empty variant configuration data object  
vcdataObj = Simulink.VariantConfigurationData
```

See Also

```
Simulink.VariantConfigurationData |  
Simulink.VariantConfigurationData.addConfiguration |  
Simulink.VariantConfigurationData.addConstraint |  
Simulink.VariantConfigurationData.addControlVariables |  
Simulink.VariantConfigurationData.addSubModelConfigurations
```

Simulink.VariantManager class

Package: Simulink

Class representing a set of Variant Manager functionality

Description

The variant manager class provides a set of methods to access Variant Manager functionality from MATLAB command-line. Use an instance of `Simulink.VariantManager` class to:

- Convert the Subsystems or Model block to a Variant Subsystem.
- Find variables used in Variant control expressions.
- Generate a reduced model for specified variant configurations.
- Display or control behavior of a variant condition legend.

Method

<code>variantLegend</code>	Display or control behavior of variant condition legend
<code>reduceModel</code>	Generate reduced model for specified variant configurations
<code>convertToVariant</code>	Convert Subsystem or Model block to Variant Subsystem block

See Also

Topics

“Variant Manager Overview”

`Simulink.VariantManager.variantLegend`

`Simulink.VariantManager.convertToVariant`

variantLegend

Class: Simulink.VariantManager

Package: Simulink

Display or control behavior of variant condition legend

Syntax

```
Simulink.VariantManager.variantLegend(modelName, action)
```

Description

`Simulink.VariantManager.variantLegend(modelName, action)` displays or performs a specified action on the variant condition legend.

Input Arguments

modelName — Model for which the variant legend is displayed

character vector

Model for which the variant legend is displayed, specified as a character vector.

action — Task to be performed on the variant legend

'open' | 'print' | 'showCodeConditions' | 'close'

Task to be performed on the variant condition legend for the model. You can specify the task as:

- 'open' — Displays the variant condition legend for a model. The model must be open. If the legend is opened for the first time, the model is updated.
- 'print' — Prints the data in the variant condition legend. The legend must be open. There is no preview before printing the legend.

- 'showCodeConditions' — Displays code generation conditions column in the variant condition legend. The legend must be open. showCodeConditions is used as a name-value pair and accepts 'on' or 'off' as its values.
- 'close' — Closes the variant condition legend belonging to the specified model.

Examples

```
model = 'sldemo_variant_subsystems';
open_system(model);
% Open the variant condition legend
Simulink.VariantManager.variantLegend(model, 'open');
% Display the code generation conditions
Simulink.VariantManager.variantLegend(model, 'showCodeConditions', 'on');
```

See Also

Variant Subsystem

Topics

“Create a Simple Variant Model”

“What Are Variants and When to Use Them”

Introduced in R2017b

reduceModel

Class: Simulink.VariantManager

Package: Simulink

Generate reduced model for specified variant configurations

Syntax

```
Simulink.VariantManager.reduceModel(Model)
Simulink.VariantManager.reduceModel(Model, Name, Value)
```

Description

`Simulink.VariantManager.reduceModel(Model)` creates a reduced model for the specified configuration. The referenced models are also reduced. Library blocks used by the model are not reduced. By default, the name of the reduced model and any reduced child referenced model name is the original model name suffixed with `_r`.

`Simulink.VariantManager.reduceModel(Model, Name, Value)` specifies the reduction parameters in the `Name` and `Value` arguments form.

Input Arguments

Model — Model to be reduced

character vector

Required field. Model to be reduced, specified as a character vector.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' ') and is case-insensitive whereas, the value string is case-sensitive. You can

specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Configurations — Variant configuration name

`{}` | character vector | 'cell array'

Specifies the configuration that is must be retained after reduction. By default, the active variant configuration is used to reduce the model.

OutputFolder — Folder to store the reduced model

character vector

Specifies the folder to place the reduced models and related artifacts. By default, the reduced models are generated in `./reducedModel` subfolder in the original model folder.

PreserveSignalAttributes — Preserve the signal attributes in the reduced model

`{true}` | false

When the value is `true`, the Variant Reducer preserves the compiled signal attributes between the original and reduced models by adding signal specification blocks at appropriate block ports in the reduced model. Compiled signal attributes include signal data types, signal dimensions, compiled sample times, etc.

Verbose — Displays step details

`true` | `{false}`

When the value is `true`, the Variant Reducer displays details of the steps performed during model reduction.

ModelSuffix — Reduced model name suffix

`{_r}`

Specifies the suffix to append to the reduced models and the related artifacts.

Examples

```
% Reduce model based on its variant control variable values in the base workspace.  
  
Simulink.VariantManager.reduceModel('sldemo_variant_subsystems');  
  
% Reduce model with Name-Value pair
```

```
Simulink.VariantManager.reduceModel('iv_model', Configurations, ...
    {'LinInterStd', 'NonLinExterHighFid'}, OutputFolder, 'outdir');

% Reduce a model by specifying a variant configuration in the form of struct of variant control variable.
Simulink.VariantManager.reduceModel('iv_model', ...
    struct('V', 1, 'W', [1 2]));
```

See Also

Variant Subsystem

Topics

“Create a Simple Variant Model”

“What Are Variants and When to Use Them”

Simulink.VariantManager

Simulink.VariantManager.variantLegend

Introduced in R2016a

convertToVariant

Class: Simulink.VariantManager

Package: Simulink

Convert Subsystem or Model block to Variant Subsystem block

Syntax

```
variant_subsystem = convertToVariant(block)
```

Description

`variant_subsystem = convertToVariant(block)` converts a Subsystem or Model block to a Variant Subsystem block that includes the subsystem or referenced model as a variant. The model uses variants based on the variant controls. A Variant Subsystem can contain subsystems, referenced models, or both.

Note Future releases will no longer support using a Model block to contain model variants. You can use the `convertToVariant` method to convert model variants so that Model blocks are contained in a Variant Subsystem. Use of a Variant Subsystem block provides these advantages:

- Allows you to mix Model and Subsystem blocks as variant systems
- Supports flexible I/O, so that all variants do not need to have the same number of input and output ports

For an example of a model that uses a Variant Subsystem block as a container for variant models, see “Model Reference Variants”.

Limitations

You cannot convert a Subsystem block that meets either of these conditions:

- The subsystem is conditionally executed (in other words, has control ports).
- The Subsystem block is in a Simscape model that has Editing Mode set to **Restricted**.

Input Arguments

block — Subsystem or Model block to convert

block path | block handle

The path or block handle of the Subsystem or Model block to convert to a Variant Subsystem block. Specify a block path as a character vector and a block handle as a scalar.

Output Arguments

variantSubsystem — Handle of Variant Subsystem block

scalar returned by Simulink

If you specify an output argument, the method returns the block handle of the Variant Subsystem block created by the conversion.

Examples

```
open_system('autoMRVar');  
Simulink.VariantManager.convertToVariant('autoMRVar/Engine');
```

Alternative Functionality

Simulink Editor

In the Simulink Editor, right-click the Model block and select **Subsystems & Model Reference > Convert to > Variant Subsystem**.

See Also

Variant Subsystem

Topics

“Create a Simple Variant Model”

“What Are Variants and When to Use Them”

Introduced in R2017b

Simulink.WorkspaceVar class

Package: Simulink

Contains information about workspace variables and blocks that use them

Note `Simulink.WorkspaceVar` will be removed in a future release. Use `Simulink.VariableUsage` instead.

Description

A `Simulink.WorkspaceVar` object describes attributes of a workspace variable and lists the blocks that use the variable. The `Simulink.findVars` function returns one or more `Simulink.WorkspaceVar` objects that embody the results of searching for variables.

Tip

Only a `Simulink.WorkspaceVar` constructor can set any field value in a `Simulink.WorkspaceVar` object. The fields are otherwise read-only.

Fields

Name

The name of the variable described by the `Simulink.WorkspaceVar` object

Workspace

The name of the workspace in which the variable resides, for example:

Workspace value	Meaning
'base workspace'	The MATLAB base workspace

Workspace value	Meaning
'MyModel '	The model workspace for the model MyModel.
'MyModel/Mask1 '	The mask workspace for the masked block Mask1 in the model MyModel.

WorkspaceType

The type of workspace in which the variable resides. The possible values are:

- 'base'
- 'model'
- 'mask'

UsedByBlocks

A cell array of character vectors. Each character vector names a block that uses the variable. `Simulink.findVars` populates this field.

Construction

`var = Simulink.WorkspaceVar (VarName, WkspName)`, where both arguments are character vectors, returns a `Simulink.WorkspaceVar` object with Name `VarName` and Workspace `WkspName`. The inputs need not exist in the model. Simulink will provide a `WorkspaceType` automatically.

`vars = Simulink.WorkspaceVar (VarNames, WkspName)`, where `VarNames` is a cell array of character vectors, returns a vector of `Simulink.WorkspaceVar` objects, each with a specified name and `Workspace = WkspName`.

Methods

`[VarsOut] = VarsIn1.setdiff (VarsIn2)` — Calls `setdiff` to return the difference between `VarsIn1` and `VarsIn2`. The arguments and return value are vectors of `Simulink.WorkspaceVar` objects.

`[VarsOut] = VarsIn1.intersect (VarsIn2)` — Calls `intersect` to return the intersection between `VarsIn1` and `VarsIn2`. The arguments and return value are vectors of `Simulink.WorkspaceVar` objects.

Examples

Create a `Simulink.WorkspaceVar` object for the variable 'k' in the base workspace.

```
var = Simulink.WorkspaceVar('k', 'base workspace');
```

Return a vector of `Simulink.WorkspaceVar` objects, one object for each variable returned by `who`.

```
[vars] = Simulink.WorkspaceVar (who, WkspName)
```

Return a vector of `Simulink.WorkspaceVar` objects, one object for each variable returned by `whos`.

```
[vars] = Simulink.WorkspaceVar (who, WkspName)
```

Create a vector of `Simulink.WorkspaceVar` objects that describes all the variables in a model workspace

```
hws = get_param('mymodel', 'ModelWorkspace');  
vars=Simulink.WorkspaceVar(hws.whos, 'MyModel')
```

Create a vector of `Simulink.WorkspaceVar` objects that describes all the variables in a mask workspace

```
maskVars = get_param('mymodel/maskblock', 'MaskWSVariables');  
vars = Simulink.WorkspaceVar(maskVars, 'mymodel/maskblock');
```

See Also

- `Simulink.findVars`
- `setdiff`
- `intersect`

Simulink.VariableUsage class

Package: Simulink

Store information about the relationship between variables and blocks in models

Description

A `Simulink.VariableUsage` object describes where a variable is used in models. Use this information to:

- Prepare to permanently store the variables in files and workspaces. For more information about storing variables for a model, see “Determine Where to Store Variables and Objects for Simulink Models”.
- Reduce the number of variables that you need to store by eliminating unused variables.
- Prepare to partition variables and establish variable ownership when you work in a team.

To analyze variable usage in models, use `Simulink.VariableUsage` objects together with the `Simulink.findVars` function. The function returns and accepts `Simulink.VariableUsage` objects as arguments. For more information, see `Simulink.findVars`.

A `Simulink.VariableUsage` object can also describe the usage of an enumerated data type.

Only a `Simulink.VariableUsage` constructor or the `Simulink.findVars` function can set property values in a `Simulink.VariableUsage` object. The properties are otherwise read only.

Construction

The `Simulink.findVars` function returns `Simulink.VariableUsage` objects.

`vars = Simulink.VariableUsage(VarNames, SourceName)` creates an array of `Simulink.VariableUsage` objects to describe the variables `VarNames`. The constructor

sets the `Name` property of each object to one of the variable names specified by `VarNames`, and sets the `Source` property of all the objects to the source specified by `SourceName`.

You can specify `VarNames` with variables that are not used in any loaded models.

Input Arguments

VarNames — Names of target variables

character vector | cell array of character vectors

Names of target variables, specified as a character vector or a cell array of character vectors. The constructor creates a `Simulink.VariableUsage` object for each variable name.

Example: 'k'

Example: {'k', 'asdf', 'fuelFlow'}

Data Types: char | cell

SourceName — Name of variable source

character vector

Name of the source that defines the target variables, specified as a character vector. For example, you can specify the MATLAB base workspace or a data dictionary as a source. The constructor also determines and sets the `SourceType` property of each of the returned `Simulink.VariableUsage` objects.

Example: 'base workspace'

Example: 'myModel'

Example: 'myDictionary.sldd'

Data Types: char

Properties

Name — Name of variable or enumerated type

character vector

The name of the variable or enumerated data type the object describes, returned as a character vector.

Source — Name of defining workspace

character vector

The name of the workspace or data dictionary that defines the described variable, returned as a character vector. The table shows some examples.

Source value	Meaning
'base workspace'	MATLAB base workspace
'MyModel'	Model workspace for the model MyModel
'MyModel/Mask1'	Mask workspace for the masked block Mask1 in the model MyModel
'sldemo_fuelsys_dd_controller.sldd'	The data dictionary named 'sldemo_fuelsys_dd_controller.sldd'

The table shows some examples if you created the `Simulink.VariableUsage` object by using the `Simulink.findVars` function to find enumerated data types.

Source value	Meaning
'BasicColors.m'	The enumerated type is defined in the MATLAB file 'BasicColors.m'.
' '	The enumerated type is defined dynamically and has no source.
'sldemo_fuelsys_dd_controller.sldd'	The enumerated type is defined in the data dictionary named 'sldemo_fuelsys_dd_controller.sldd'.

SourceType — Type of defining workspace

character vector

The type of the workspace that defines the variable, returned as a character vector. The possible values are:

- 'base workspace'
- 'model workspace'
- 'mask workspace'
- 'data dictionary'

If you created the `Simulink.VariableUsage` object by using the `Simulink.findVars` function to find enumerated data types, the possible values are:

- 'MATLAB file'
- 'dynamic class'
- 'data dictionary'

Users — Model blocks that use the variable or models that use the enumerated type
cell array of character vectors

Model blocks that use the variable or models that use the enumerated type, returned as a cell array of character vectors. Each character vector names a block or model that uses the variable or enumerated type. The `Simulink.findVars` function populates this field.

Methods

`intersect` Intersection of two arrays of `Simulink.VariableUsage` objects

`setdiff` Return difference between two arrays of `Simulink.VariableUsage` objects

Examples

Return a `Simulink.VariableUsage` object for the variable 'k' in the base workspace.

```
var = Simulink.VariableUsage('k', 'base workspace');
```

Return an array of `Simulink.VariableUsage` objects containing one object for each variable returned by the `whos` command.

```
vars = Simulink.VariableUsage(whos, 'base workspace')
```

Return an array of `Simulink.VariableUsage` objects that describes all the variables in a model workspace.

```
hws = get_param('mymodel', 'ModelWorkspace');
vars = Simulink.VariableUsage(hws.whos, 'MyModel')
```

Return an array of `Simulink.VariableUsage` objects that describes all the variables in a mask workspace.

```
maskVars = get_param('mymodel/maskblock', 'MaskWSVariables');  
vars = Simulink.VariableUsage(maskVars, 'mymodel/maskblock');
```

See Also

[Simulink.data.existsInGlobal](#) | [Simulink.findVars](#)

Topics

[“Model Exploration”](#)

[“Variables”](#)

intersect

Class: Simulink.VariableUsage

Package: Simulink

Intersection of two arrays of Simulink.VariableUsage objects

Syntax

```
VarsOut = intersect(VarsIn1,VarsIn2)
```

Description

`VarsOut = intersect(VarsIn1,VarsIn2)` returns an array that identifies the variables described in `VarsIn1` and in `VarsIn2`, which are arrays of Simulink.VariableUsage objects. If a variable is described by a Simulink.VariableUsage object in `VarsIn1` and in `VarsIn2`, the function returns a Simulink.VariableUsage object that stores the variable usage information from both objects in the `Users` property.

`intersect` compares the `Name`, `Source`, and `SourceType` properties of the Simulink.VariableUsage objects in `VarsIn1` with the same properties of the objects in `VarsIn2`. If `VarsIn1` and `VarsIn2` each contain Simulink.VariableUsage objects that have the same values for these three properties, they both describe the same variable.

To create Simulink.VariableUsage objects that describe the usage of variables in a model, use the Simulink.findVars function.

Input Arguments

VarsIn1 — First array of variables for comparison

array of Simulink.VariableUsage objects

First array of variables for comparison, specified as an array of `Simulink.VariableUsage` objects.

VarsIn2 — Second array of variables for comparison

array of `Simulink.VariableUsage` objects

Second array of variables for comparison, specified as an array of `Simulink.VariableUsage` objects.

Output Arguments

VarsOut — Variables described in both input arrays

array of `Simulink.VariableUsage` objects

Variables that are described in both input arrays, returned as an array of `Simulink.VariableUsage` objects. The function returns an object for each variable that is described in `VarsIn1` and in `VarsIn2`.

Examples

Compare Variables Used by Models

Given two models, discover the variables needed by both models.

```
model1Vars = Simulink.findVars('model1');  
model2Vars = Simulink.findVars('model2');  
commonVars = intersect(model1Vars,model2Vars);
```

See Also

`Simulink.VariableUsage` | `Simulink.findVars` | `setdiff`

Topics

“Model Exploration”

“Variables”

setdiff

Class: Simulink.VariableUsage

Package: Simulink

Return difference between two arrays of Simulink.VariableUsage objects

Syntax

```
VarsOut = setdiff(VarsIn1,VarsIn2)
```

Description

`VarsOut = setdiff(VarsIn1,VarsIn2)` returns an array that identifies the variables described in `VarsIn1` but not in `VarsIn2`, which are arrays of Simulink.VariableUsage objects. If a variable is described by a Simulink.VariableUsage object in `VarsIn1` but not in `VarsIn2`, the function returns a copy of the object.

`setdiff` compares the `Name`, `Source`, and `SourceType` properties of the Simulink.VariableUsage objects in `VarsIn1` with the same properties of the objects in `VarsIn2`. If `VarsIn1` and `VarsIn2` each contain a Simulink.VariableUsage object with the same values for these three properties, the objects describe the same variable, and `setdiff` does not return an object to describe it.

To create Simulink.VariableUsage objects that describe the usage of variables in a model, use the Simulink.findVars function.

Input Arguments

VarsIn1 — First array of variables for comparison

array of Simulink.VariableUsage objects

First array of variables for comparison, specified as an array of Simulink.VariableUsage objects.

VarsIn2 — Second array of variables for comparison

array of `Simulink.VariableUsage` objects

Second array of variables for comparison, specified as an array of `Simulink.VariableUsage` objects.

Output Arguments

varsOut — Variables described in first array but not second array

array of `Simulink.VariableUsage` objects

Variables that are described in the first input array but not in the second input array, returned as an array of `Simulink.VariableUsage` objects. The function returns an object for each variable that is described in `VarsIn1` but not in `VarsIn2`.

Examples

Determine Variable Usage Difference Between Models

Given two models, discover the variables that are needed by the first model but not the second model.

```
model1Vars = Simulink.findVars('model1');  
model2Vars = Simulink.findVars('model2');  
commonVars = setdiff(model1Vars,model2Vars);
```

Find Variables Not Used by Model

Locate all variables in the base workspace that are not used by a loaded model that has been recently compiled.

```
models = find_system('type','block_diagram','LibraryType','None');  
base_vars = Simulink.VariableUsage(who,'base workspace');
```

```
used_vars = Simulink.findVars(models,'WorkspaceType','base');  
unusedVars = setdiff(base_vars,used_vars);
```

See Also

[Simulink.VariableUsage](#) | [Simulink.findVars](#) | [intersect](#)

Topics

[“Model Exploration”](#)

[“Variables”](#)

Simulink.data.Dictionary class

Package: Simulink.data

Configure data dictionary

Description

An object of the `Simulink.data.Dictionary` class represents a data dictionary. The object allows you to perform operations on the data dictionary such as save or discard changes, import data from the base workspace, and add other data dictionaries as references.

Construction

The functions `Simulink.data.dictionary.create` and `Simulink.data.dictionary.open` create a `Simulink.data.Dictionary` object.

Properties

DataSources — Referenced data dictionaries

cell array of character vectors

Referenced data dictionaries by file name, returned as a cell array of character vectors. This property only lists directly referenced dictionaries whose parent is the `Simulink.data.Dictionary` object. This property is read only.

HasUnsavedChanges — Indicator of unsaved changes

0 | 1

Indicator of unsaved changes to the data dictionary, returned as 0 or 1. The value is 1 if changes have been made since last data dictionary save and 0 if not. This property is read only.

NumberOfEntries — Total number of entries in data dictionary

integer

Total number of entries in data dictionary, including those in referenced dictionaries, returned as an integer. This property is read only.

Methods

<code>addDataSource</code>	Add reference data dictionary to parent data dictionary
<code>close</code>	Close connection between data dictionary and <code>Simulink.data.Dictionary</code> object
<code>discardChanges</code>	Discard changes to data dictionary
<code>filepath</code>	Full path and file name of data dictionary
<code>getSection</code>	Return <code>Simulink.data.dictionary.Section</code> object to represent data dictionary section
<code>hide</code>	Remove data dictionary from Model Explorer
<code>importEnumTypes</code>	Import enumerated type definitions to data dictionary
<code>importFromBaseWorkspace</code>	Import base workspace variables to data dictionary
<code>listEntry</code>	List data dictionary entries
<code>removeDataSource</code>	Remove reference data dictionary from parent data dictionary
<code>saveChanges</code>	Save changes to data dictionary
<code>show</code>	Show data dictionary in Model Explorer

Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Create New Data Dictionary and Data Dictionary Object

Create a data dictionary file `myNewDictionary.sldd` and a `Simulink.data.Dictionary` object representing the new data dictionary. Assign the object to variable `dd1`.

```
dd1 = Simulink.data.dictionary.create('myNewDictionary.sldd')
```

```
dd1 =
```

```
data dictionary with properties:
```

```
    DataSources: {0x1 cell}  
HasUnsavedChanges: 0  
    NumberOfEntries: 0
```

Open Existing Data Dictionary

Create a `Simulink.data.Dictionary` object representing the existing data dictionary `myDictionary_ex_API.sldd`. Assign the object to variable `dd2`.

```
dd2 = Simulink.data.dictionary.open('myDictionary_ex_API.sldd')
```

```
dd2 =
```

```
Dictionary with properties:
```

```
    DataSources: {'myRefDictionary_ex_API.sldd'}  
HasUnsavedChanges: 0  
    NumberOfEntries: 4
```

- “Store Data in Dictionary Programmatically”

See Also

`Simulink.data.dictionary.Entry` | `Simulink.data.dictionary.Section` |
`Simulink.data.dictionary.create` | `Simulink.data.dictionary.open`

Topics

“Store Data in Dictionary Programmatically”

“What Is a Data Dictionary?”

Introduced in R2015a

addDataSource

Class: Simulink.data.Dictionary

Package: Simulink.data

Add reference data dictionary to parent data dictionary

Syntax

```
addDataSource(dictionaryObj, refDictionaryFile)
```

Description

`addDataSource(dictionaryObj, refDictionaryFile)` adds a data dictionary, `refDictionaryFile`, as a reference dictionary to a parent dictionary `dictionaryObj`, a `Simulink.data.Dictionary` object.

The parent dictionary contains all the entries that are defined in the referenced dictionary until the referenced dictionary is removed from the parent dictionary. The `DataSource` property of an entry indicates the dictionary that defines the entry.

Input Arguments

dictionaryObj — Parent data dictionary

`Simulink.data.Dictionary` object

Parent data dictionary, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the target dictionary with a `Simulink.data.Dictionary` object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

refDictionaryFile — File name of data dictionary to reference

character vector

File name of data dictionary to reference, specified as a character vector that includes the `.sldd` extension. The data dictionary file must be on your MATLAB path.

Example: `'mySubDictionary_ex_API.sldd'`

Data Types: `char`

Examples

Add a Reference Data Dictionary to a Parent Data Dictionary

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Add the data dictionary `mySubDictionary_ex_API.sldd` as a reference dictionary to `myDictionary_ex_API.sldd`.

```
addDataSource(myDictionaryObj, 'mySubDictionary_ex_API.sldd');
```

Confirm the addition by viewing the `DataSources` property of variable `myDictionaryObj`. The property returns the name of the newly referenced dictionary.

```
myDictionaryObj.DataSources
```

```
ans =
```

```
    'myRefDictionary_ex_API.sldd'  
    'mySubDictionary_ex_API.sldd'
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use the Model Explorer window to manage reference dictionaries. See “Partition Dictionary Data Using Referenced Dictionaries” for more information.

See Also

`Simulink.data.Dictionary` | `removeDataSource`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

close

Class: Simulink.data.Dictionary

Package: Simulink.data

Close connection between data dictionary and Simulink.data.Dictionary object

Syntax

```
close(dictionaryObj)
```

Description

`close(dictionaryObj)` closes the connection between the Simulink.data.Dictionary object `dictionaryObj` and the data dictionary it represents. `dictionaryObj` remains as a Simulink.data.Dictionary object but no longer represents any data dictionary.

Input Arguments

dictionaryObj — Target Simulink.data.Dictionary object

handle to Simulink.data.Dictionary object

Target Simulink.data.Dictionary object, specified as a handle to the object.

Tips

- Use the `close` function in a custom MATLAB function to disassociate a Simulink.data.Dictionary object from a data dictionary. Custom MATLAB functions can create and store variables and objects in function workspaces but cannot delete those variables and objects.

- The `close` function does not affect the content or the state of the represented data dictionary. The function does not discard unsaved changes to the represented dictionary or entries. You can save or discard them later.

See Also

`Simulink.data.Dictionary`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

discardChanges

Class: Simulink.data.Dictionary

Package: Simulink.data

Discard changes to data dictionary

Syntax

```
discardChanges(dictionaryObj)
```

Description

`discardChanges(dictionaryObj)` discards all changes made to the specified data dictionary since the last time changes to the dictionary were saved using the `saveChanges` function. `discardChanges` also discards changes made to referenced data dictionaries. The changes to the target dictionary and its referenced dictionaries are permanently lost.

Input Arguments

dictionaryObj — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the target dictionary with a `Simulink.data.Dictionary` object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

Examples

Discard Changes to Data Dictionary

Create a `Simulink.data.Dictionary` object representing the data dictionary `myDictionary_ex_API.sldd` and assign the object to variable `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd')  
myDictionaryObj =
```

```
Dictionary with properties:
```

```
    DataSources: {'myRefDictionary_ex_API.sldd'}  
    HasUnsavedChanges: 0  
    NumberOfEntries: 4
```

Make a change to `myDictionary_ex_API.sldd` by adding an entry named `myNewEntry` with value 237. View the `HasUnsavedChanges` property of `myDictionaryObj` to confirm a change was made.

```
addEntry(dDataSectObj, 'myNewEntry', 237);  
myDictionaryObj
```

```
myDictionaryObj =
```

```
Dictionary with properties:
```

```
    DataSources: {'myRefDictionary_ex_API.sldd'}  
    HasUnsavedChanges: 1  
    NumberOfEntries: 5
```

Discard all changes to `myDictionary_ex_API.sldd`. The `HasUnsavedChanges` property of `myDictionaryObj` indicates changes were discarded.

```
discardChanges(myDictionaryObj)  
myDictionaryObj
```

```
myDictionaryObj =
```

```
Dictionary with properties:
```

```
    DataSources: {'myRefDictionary_ex_API.sldd'}
```

```
HasUnsavedChanges: 0  
NumberOfEntries: 4
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use the Model Explorer window to discard changes to data dictionaries. See “View and Revert Changes to Dictionary Entries” for more information.

See Also

`Simulink.data.Dictionary | saveChanges`

Topics

“Store Data in Dictionary Programmatically”
“What Is a Data Dictionary?”

Introduced in R2015a

filepath

Class: Simulink.data.Dictionary

Package: Simulink.data

Full path and file name of data dictionary

Syntax

```
dictionaryFilePath = filepath(dictionaryObj)
```

Description

`dictionaryFilePath = filepath(dictionaryObj)` returns the full path and file name of the data dictionary `dictionaryObj`, a `Simulink.data.Dictionary` object.

Input Arguments

dictionaryObj — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the target dictionary with a `Simulink.data.Dictionary` object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

Examples

Return Path of Data Dictionary File

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Return the full path of `myDictionary_ex_API.sldd` and assign it to variable `myDictionaryFilePath`.

```
myDictionaryFilePath = filepath(myDictionaryObj)
```

```
myDictionaryFilePath =
```

```
C:\Users\jsmith\myDictionary_ex_API.sldd
```

- “Store Data in Dictionary Programmatically”

See Also

`Simulink.data.Dictionary`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

getSection

Class: Simulink.data.Dictionary

Package: Simulink.data

Return Simulink.data.dictionary.Section object to represent data dictionary section

Syntax

```
sectionObj = getSection(dictionaryObj,sectionName)
```

Description

`sectionObj = getSection(dictionaryObj,sectionName)` returns a Simulink.data.dictionary.Section object representing one section, `sectionName`, of a data dictionary `dictionaryObj`, a Simulink.data.Dictionary object.

Input Arguments

dictionaryObj — Data dictionary containing target section

Simulink.data.Dictionary object

Data dictionary containing target section, specified as a Simulink.data.Dictionary object. Before you use this function, represent the dictionary with a Simulink.data.Dictionary object by using, for example, the Simulink.data.dictionary.create or Simulink.data.dictionary.open function.

sectionName — Name of target data dictionary section

character vector

Name of target data dictionary section, specified as a character vector.

Example: 'Design Data'

Example: 'Configurations'

Data Types: char

Examples

Create New Data Dictionary Section Object

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
dDataSectObj = getSection(myDictionaryObj, 'Design Data')
```

```
dDataSectObj =
```

```
    Section with properties:
```

```
        Name: 'Design Data'
```

- “Store Data in Dictionary Programmatically”

See Also

`Simulink.data.Dictionary` | `Simulink.data.dictionary.Section`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

hide

Class: Simulink.data.Dictionary

Package: Simulink.data

Remove data dictionary from Model Explorer

Syntax

```
hide(dictionaryObj)
```

Description

`hide(dictionaryObj)` removes the data dictionary `dictionaryObj` from the **Model Hierarchy** pane of Model Explorer. The target dictionary no longer appears as a node in the model hierarchy tree. Use this function when you are finished working with a data dictionary and want to reduce clutter in the Model Explorer.

Input Arguments

dictionaryObj — Target data dictionary

Simulink.data.Dictionary object

Target data dictionary, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the target dictionary with a `Simulink.data.Dictionary` object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

Examples

Hide Data Dictionary from Model Explorer

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Open Model Explorer and display the new data dictionary as the selected tree node in the **Model Hierarchy** pane.

```
show(myDictionaryObj)
```

With Model Explorer open, at the MATLAB command prompt, call the `hide` function to observe the removal of `myDictionary_ex_API.sldd` from the model hierarchy tree.

```
hide(myDictionaryObj)
```

- “Store Data in Dictionary Programmatically”

Tips

- To add a data dictionary as a node in the model hierarchy tree in Model Explorer, use the `show` function or use the interface to open and view the dictionary in Model Explorer.
- The `hide` function does not affect the content of the target dictionary.

Alternatives

You can remove a data dictionary from the **Model Hierarchy** pane of Model Explorer by right-clicking the dictionary tree node and selecting **Close**.

See Also

`Simulink.data.Dictionary` | `show`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

importEnumTypes

Class: Simulink.data.Dictionary

Package: Simulink.data

Import enumerated type definitions to data dictionary

Syntax

```
importedTypes = importEnumTypes(dictionaryObj, targetTypes)
[importedTypes, importFailures] = importEnumTypes(dictionaryObj,
targetTypes)
```

Description

`importedTypes = importEnumTypes(dictionaryObj, targetTypes)` imports to the data dictionary `dictionaryObj` the definitions of one or more enumerated types `targetTypes`. `importEnumTypes` does not import MATLAB variables created using enumerated types but instead, in support of those variables, imports the definitions of the types. The target data dictionary stores the definition of a successfully imported type as an entry. This syntax returns a list of the names of successfully imported types. `importEnumTypes` saves changes made to the target dictionary, so before you use `importEnumTypes`, confirm that unsaved changes are acceptable.

`[importedTypes, importFailures] = importEnumTypes(dictionaryObj, targetTypes)` additionally returns a list of any target types that were not successfully imported. You can inspect the list to determine the reason for each failure.

Input Arguments

dictionaryObj — Target data dictionary

Simulink.data.Dictionary object

Target data dictionary, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the target dictionary with a `Simulink.data.Dictionary`

object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

targetTypes — Enumerated type definitions to import

cell array of character vectors

Enumerated type definitions to import, specified as a cell array of character vectors. If any target types are defined using `classdef` blocks in MATLAB files or P-files, the files must be available on your MATLAB path so that `importEnumTypes` can disable them.

Example: `{ 'myEnumType' }`

Example: `{ 'myFirstEnumType', 'mySecondEnumType', 'myThirdEnumType' }`

Data Types: `cell`

Output Arguments

importedTypes — Target types successfully imported

array of structures

Target enumerated type definitions successfully imported, returned as an array of structures. Each structure in the array represents one imported type. The `className` field of each structure identifies a type by name and the `renamedFiles` field identifies any renamed MATLAB files or P-files.

importFailures — Target types not imported

array of structures

Enumerated type definitions targeted but not imported, returned as an array of structures. Each structure in the array represents one type not imported. The `className` field of each structure identifies a type by name and the `reason` field explains the failure.

Examples

Import Enumerated Data to Data Dictionary

Create a data dictionary `myNewDictionary.sldd` in your current working folder and a `Simulink.data.Dictionary` object representing the new data dictionary. Assign the object to the variable `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.create('myNewDictionary.sldd');
```

Run the script in the MATLAB file `myDataEnum_ex_API.m`. The file defines an enumerated type named `InstrumentTypes` using the `Simulink.defineIntEnumType` function and creates three variables based on the new type. Then, import the new variables from the base workspace to `myDictionary_ex_API.sldd`.

```
myDataEnum_ex_API
importFromBaseWorkspace(myDictionaryObj, 'varList', ...
{'firstEnumVariable', 'secondEnumVariable', 'thirdEnumVariable'});
```

Clear the imported variables from the base workspace. Before you can import an enumerated data type definition to the target data dictionary, you must clear the base workspace of any variables created using the target type.

```
clear firstEnumVariable
clear secondEnumVariable
clear thirdEnumVariable
```

Import the data type definition to `myDictionary_ex_API.sldd`.

```
importEnumTypes(myDictionaryObj, {'InstrumentTypes'})
```

```
ans =
```

```
    className: 'InstrumentTypes'
    renamedFiles: {}
```

- “Enumerations in Data Dictionary”
- “Store Data in Dictionary Programmatically”

Tips

- Before you can import an enumerated data type definition to a data dictionary, you must clear the base workspace of any variables created using the target type.

- You can define an enumerated type using a `classdef` block in a MATLAB file or a P-file. `importEnumTypes` imports type definitions directly from these files if you specify the names of the types to import using the input argument `targetTypes` and if the files defining the types are on your MATLAB path.
- To avoid conflicting definitions for imported types, `importEnumTypes` renders MATLAB files or P-files ineffective by appending `.save` to their names. The `.save` extensions cause variables to rely on the definitions in the target data dictionary and not on the definitions in the files. You can remove the `.save` extensions to restore the files to their original state.
- You can use `importEnumTypes` to import enumerated types defined using the `Simulink.defineIntEnumType` function. Because such types are not defined using MATLAB files or P-files, `importEnumTypes` does not rename any files.
- Use the function `Simulink.findVars` to generate a list of the enumerated types that are used by a model. Then, use the list with `importEnumTypes` to import the definitions of the types to a data dictionary. See “Enumerations in Data Dictionary” for more information.

See Also

`Simulink.data.Dictionary` | `importFromBaseWorkspace`

Topics

“Enumerations in Data Dictionary”

“Store Data in Dictionary Programmatically”

Introduced in R2015a

importFromBaseWorkspace

Class: Simulink.data.Dictionary

Package: Simulink.data

Import base workspace variables to data dictionary

Syntax

```
importedVars = importFromBaseWorkspace(dictionaryObj)
importedVars = importFromBaseWorkspace(dictionaryObj, Name, Value)
[importedVars, existingVars] = importFromBaseWorkspace(____)
```

Description

`importedVars = importFromBaseWorkspace(dictionaryObj)` imports all variables from the MATLAB base workspace to the data dictionary `dictionaryObj` without overwriting existing entries in the dictionary. If any base workspace variables are already in the dictionary, the function present a warning and a list.

This syntax returns a list of names of the successfully imported variables. A variable is considered successfully imported only if `importFromBaseWorkspace` assigns the value of the variable to the corresponding entry in the target data dictionary.

`importedVars = importFromBaseWorkspace(dictionaryObj, Name, Value)` imports base workspace variables to a data dictionary, with additional options specified by one or more `Name, Value` pair arguments.

`[importedVars, existingVars] = importFromBaseWorkspace(____)` additionally returns a list of variables that were not overwritten. Use this syntax if `existingVarsAction` is set to 'none', the default value, which prevents existing dictionary entries from being overwritten.

Input Arguments

dictionaryObj — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the target dictionary with a `Simulink.data.Dictionary` object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

clearWorkspaceVars — Flag to clear base workspace of imported variables

`false` (default) | `true`

Flag to clear the base workspace of any successfully imported variables, specified as the comma-separated pair consisting of 'clearWorkspaceVars' and `true` or `false`.

Example: 'clearWorkspaceVars',`true`

Data Types: `logical`

existingVarsAction — Action to take for existing dictionary variables

'none' (default) | 'error' | 'overwrite'

Action to take for existing dictionary variables, specified as the comma-separated pair consisting of 'existingVarsAction' and 'none', 'error', or 'overwrite'.

If you specify 'none', `importFromBaseWorkspace` attempts to import target variables but does not import or make any changes to variables that are already in the data dictionary.

If you specify 'error', `importFromBaseWorkspace` returns an error, without importing any variables, if any target variables are already in the data dictionary.

If you specify `'overwrite'`, `importFromBaseWorkspace` imports all target variables and overwrites any variables that are already in the data dictionary.

Example: `'existingVarsAction','error'`

Data Types: `char`

varList — Variables to import

cell array of character vectors

Names of specific base workspace variables to import, specified as the comma-separated pair consisting of `'varList'` and a cell array of character vectors. If you want to import only one variable, specify the name inside a cell array. If you do not specify `'varList'`, `importFromBaseWorkspace` imports all variables from the MATLAB base workspace.

Example: `'varList',{'a','myVariable','fuelFlow'}`

Example: `'varList',{'fuelFlow'}`

Data Types: `cell`

Output Arguments

importedVars — Successfully imported variables

cell array of character vectors

Names of successfully imported variables, returned as a cell array of character vectors. A variable is considered successfully imported only if `importFromBaseWorkspace` assigns the value of the variable to the corresponding entry in the target data dictionary.

existingVars — Variables that were not imported

cell array of character vectors

Names of target variables that were not imported due to their existence in the target data dictionary, returned as a cell array of character vectors. `existingVars` has content only if `'existingVarsAction'` is set to `'none'` which is also the default. In that case `importFromBaseWorkspace` imports only variables that are not already in the target data dictionary.

Examples

Import All Base Workspace Variables to Data Dictionary

In the MATLAB base workspace, create variables to import.

```
a = 'Char Variable';  
myVariable = true;  
fuelFlow = 324;
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import all base workspace variables to the data dictionary and return a list of successfully imported variables. If any base workspace variables are already in `myDictionary_ex_API.sldd`, `importFromBaseWorkspace` presents a warning and a list of the affected variables.

```
importFromBaseWorkspace(myDictionaryObj);
```

```
Warning: The following variables were not imported because  
they already exist in the dictionary:  
    fuelFlow
```

Specify Variables to Import to Data Dictionary from Base Workspace

In the MATLAB base workspace, create variables to import.

```
b = 'Char Variable';  
mySecondVariable = true;  
airFlow = 324;
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import only the new base workspace variables to the data dictionary.

```
importFromBaseWorkspace(myDictionaryObj, 'varList', ...  
{ 'b', 'mySecondVariable', 'airFlow' });
```

Import Variables from Base Workspace and Overwrite Conflicts

In the MATLAB base workspace, create a variable to import.

```
fuelFlow = 324;
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`. `myDictionary_ex_API.sldd` already contains an entry called `fuelFlow`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import the variable `fuelFlow` and overwrite the corresponding entry in `myDictionary_ex_API.sldd`.

```
importFromBaseWorkspace(myDictionaryObj, 'varList', {'fuelFlow'}, ...  
'existingVarsAction', 'overwrite');
```

`importFromBaseWorkspace` assigns the value of the base workspace variable `fuelFlow` to the value of the corresponding entry in `myDictionary_ex_API.sldd`.

Return Variables Not Imported to Data Dictionary from Base Workspace

Return a list of variables that are not imported from the MATLAB base workspace because they are already in the target data dictionary.

In the MATLAB base workspace, create variables to import.

```
fuelFlow = 324;  
myNewVariable = 'This is a character vector.'
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`. `myDictionary_ex_API.sldd` already contains an entry called `fuelFlow`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import the variables `fuelFlow` and `myNewVariable` to the data dictionary. Specify names for the output arguments of `importFromBaseWorkspace` to return the names of successfully and unsuccessfully imported variables.

```
[importedVars,existingVars] = importFromBaseWorkspace(myDictionaryObj,...  
'varList',{'fuelFlow','myNewVariable'})
```

```
importedVars =  
  
    'myNewVariable'
```

```
existingVars =  
  
    'fuelFlow'
```

`importFromBaseWorkspace` does not import the variable `fuelflow` because it is already in the target data dictionary.

- “Store Data in Dictionary Programmatically”

Tips

- `importFromBaseWorkspace` can import MATLAB variables created from enumerated data types but cannot import the definitions of the enumerated types. Use the `importEnumTypes` function to import enumerated data type definitions to a data dictionary. If you import variables of enumerated data types to a data dictionary but do not import the enumerated type definitions, the dictionary is less portable and might not function properly if used by someone else.

Alternatives

- When you use the Simulink Editor to link a model to a data dictionary, you can choose to import model variables from the base workspace. See “Migrate Single Model to Use Dictionary” for more information.
- You can also use the Model Explorer window to drag-and-drop variables from the base workspace into a data dictionary.

See Also

`Simulink.data.Dictionary` | `importEnumTypes`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

listEntry

Class: Simulink.data.Dictionary

Package: Simulink.data

List data dictionary entries

Syntax

```
listEntry(dictionaryObj)  
listEntry(dictionaryObj, Name, Value)
```

Description

`listEntry(dictionaryObj)` displays in the MATLAB Command Window a table of information about all the entries in the data dictionary `dictionaryObj`, a `Simulink.data.Dictionary` object. The displayed information includes the name of each entry, the name of the section containing each entry, the status of each entry, the date and time each entry was last modified, the last user name to modify each entry, and the class of the value each entry contains. By default, the function sorts the list of entries alphabetically by entry name.

`listEntry(dictionaryObj, Name, Value)` displays the entries in a data dictionary with additional options specified by one or more `Name, Value` pair arguments.

To return the value of a data dictionary entry at the command prompt, use the `getValue` method of a `Simulink.data.dictionary.Entry` object. See “Store Data in Dictionary Programmatically”.

Input Arguments

dictionaryObj — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the target dictionary with a `Simulink.data.Dictionary`

object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Ascending — Sort order of list

`true` (default) | `false`

Sort order of the list of data dictionary entries, specified as the comma-separated pair consisting of 'Ascending' and `true` or `false`. If you specify `false`, `listEntry` sorts the list in descending order.

Example: 'Ascending', `false`

Data Types: `logical`

Class — Criteria to filter list by class

character vector

Criteria to filter the list of data dictionary entries by class, specified as the comma-separated pair consisting of 'Class' and a character vector identifying a valid class. The function lists only entries whose values are of the specified class.

Example: 'Class', 'Simulink.Parameter'

Data Types: `char`

LastModifiedBy — Criteria to filter list by user name of last modifier

character vector

Criteria to filter the list of data dictionary entries by the user name of the last user to modify each entry, specified as the comma-separated pair consisting of 'LastModifiedBy' and a character vector identifying the specified user name. The function lists only entries that were last modified by the specified user name.

Example: 'LastModifiedBy', 'jsmith'

Data Types: `char`

Limit — Maximum number of entries to list

integer

Maximum number of entries to list, specified as the comma-separated pair consisting of 'Limit' and an integer. The function lists up to the specified number of entries starting from the top of the sorted and filtered list.

Example: 'Limit', 9

Data Types: double

Name — Criteria to filter list by entry name

character vector

Criteria to filter the list of data dictionary entries by entry name, specified as the comma-separated pair consisting of 'Name' and a character vector defining the filter criteria. You can use an asterisk character, *, as a wildcard to represent any number of characters. The function lists only entries whose names match the filter criteria.

Example: 'Name', 'fuelFlow'

Example: 'Name', 'fuel*'

Data Types: char

Section — Criteria to filter list by data dictionary section

character vector

Criteria to filter the list of data dictionary entries by section, specified as the comma-separated pair consisting of 'Section' and a character vector identifying the target section. The function lists only entries that are contained in the target section.

Example: 'Section', 'Design Data'

SortBy — Flag to sort list by specific property

'Name' (default) | 'Section' | 'LastModified' | 'LastModifiedBy'

Flag to sort the list of data dictionary entries by a specific property, specified as the comma-separated pair consisting of 'SortBy' and a character vector identifying a property in the list of entries. Valid properties include 'Name', 'Section', 'LastModified', and 'LastModifiedBy'.

Example: 'SortBy', 'LastModifiedBy'

Examples

List All Entries in Data Dictionary

Represent the data dictionary `sldemo_fuelsys_dd_controller.sldd` with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
```

List all the entries in the data dictionary.

```
listEntry(myDictionaryObj)
```

Sort List of Data Dictionary Entries in Descending Order

Represent the data dictionary `sldemo_fuelsys_dd_controller.sldd` with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
```

List all the entries in the data dictionary and sort the list in descending order by entry name.

```
listEntry(myDictionaryObj, 'Ascending', false)
```

Filter List of Data Dictionary Entries by Name

Represent the data dictionary `sldemo_fuelsys_dd_controller.sldd` with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
```

List only the entries in the data dictionary whose names begin with `max`.

```
listEntry(myDictionaryObj, 'Name', 'max*')
```

Sort List of Data Dictionary Entries by Time of Modification

Represent the data dictionary `sldemo_fuelsys_dd_controller.sldd` with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
```

List all the entries in the dictionary and sort the list by the date and time each entry was last modified.

```
listEntry(myDictionaryObj, 'SortBy', 'LastModified')
```

- “Store Data in Dictionary Programmatically”

See Also

`Simulink.data.Dictionary` | `Simulink.data.dictionary.Entry` | `evalin`

Topics

“Store Data in Dictionary Programmatically”

“What Is a Data Dictionary?”

Introduced in R2015a

removeDataSource

Class: Simulink.data.Dictionary

Package: Simulink.data

Remove reference data dictionary from parent data dictionary

Syntax

```
removeDataSource(dictionaryObj, refDictionaryFile)
```

Description

`removeDataSource(dictionaryObj, refDictionaryFile)` removes a referenced data dictionary, `refDictionaryFile`, from a parent dictionary `dictionaryObj`, a `Simulink.data.Dictionary` object.

The parent dictionary no longer contains the entries that are defined in the referenced dictionary.

Input Arguments

dictionaryObj — Parent data dictionary

`Simulink.data.Dictionary` object

Parent data dictionary, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the target dictionary with a `Simulink.data.Dictionary` object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

refDictionaryFile — File name of referenced data dictionary

character vector

File name of referenced data dictionary, specified as a character vector that includes the `.sldd` extension. The data dictionary file must be on your MATLAB path.

Example: 'myRefDictionary_ex_API.sldd'

Data Types: char

Examples

Remove Referenced Data Dictionary from Parent Data Dictionary

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`. The `DataSources` property of `myDictionaryObj` indicates `myDictionary_ex_API.sldd` references `myRefDictionary_ex_API.sldd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd')
```

```
myDictionaryObj =
```

```
Dictionary with properties:
```

```
    DataSources: {'myRefDictionary_ex_API.sldd'}
  HasUnsavedChanges: 0
    NumberOfEntries: 4
```

Remove `myRefDictionary_ex_API.sldd` from `myDictionary_ex_API.sldd`.

```
removeDataSource(myDictionaryObj, 'myRefDictionary_ex_API.sldd');
```

View the properties of the `Simulink.data.Dictionary` object `myDictionaryObj`, which represents the parent data dictionary. The `DataSources` property confirms the removal of `myRefDictionary_ex_API.sldd`.

```
myDictionaryObj
```

```
myDictionaryObj =
```

```
Dictionary with properties:
```

```
    DataSources: {0x1 cell}
```

```
HasUnsavedChanges: 1  
NumberOfEntries: 3
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer to manage reference dictionaries. See “Partition Dictionary Data Using Referenced Dictionaries” for more information.

See Also

`Simulink.data.Dictionary | addDataSource`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

saveChanges

Class: Simulink.data.Dictionary

Package: Simulink.data

Save changes to data dictionary

Syntax

```
saveChanges(dictionaryObj)
```

Description

`saveChanges(dictionaryObj)` saves all changes made to a data dictionary `dictionaryObj`, a `Simulink.data.Dictionary` object. `saveChanges` also saves changes made to referenced data dictionaries. The previous states of the target dictionary and its referenced dictionaries are permanently lost.

Input Arguments

dictionaryObj — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the target dictionary with a `Simulink.data.Dictionary` object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

Examples

Save Changes to Data Dictionary

Create a new data dictionary `myNewDictionary.sldd` and represent the Design Data section with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.create('myNewDictionary.sldd')
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

```
myDictionaryObj =

    data dictionary with properties:

        DataSources: {0x1 cell}
        HasUnsavedChanges: 0
        NumberOfEntries: 0
```

Change `myNewDictionary.sldd` by adding an entry named `myNewEntry` with value 237. View the `HasUnsavedChanges` property of `myDictionaryObj` to confirm a change was made.

```
addEntry(dDataSectObj, 'myNewEntry', 237);
myDictionaryObj
```

```
myDictionaryObj =

    Dictionary with properties:

        DataSources: {0x1 cell}
        HasUnsavedChanges: 1
        NumberOfEntries: 1
```

Save all changes to `myNewDictionary.sldd`. The `HasUnsavedChanges` property of `myDictionaryObj` indicates changes were saved.

```
saveChanges(myDictionaryObj)
myDictionaryObj
```

```
myDictionaryObj =

    Dictionary with properties:

        DataSources: {0x1 cell}
```

```
HasUnsavedChanges: 0  
NumberOfEntries: 1
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer to save changes to a data dictionary by right-clicking on the dictionary tree node in the **Model Hierarchy** pane and selecting **Save Changes**.

See Also

`Simulink.data.Dictionary` | `discardChanges`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

show

Class: Simulink.data.Dictionary

Package: Simulink.data

Show data dictionary in Model Explorer

Syntax

```
show(dictionaryObj)
show(dictionaryObj, openModelExplorer)
```

Description

`show(dictionaryObj)` opens Model Explorer and displays the data dictionary `dictionaryObj` as the selected tree node in the **Model Hierarchy** pane.

`show(dictionaryObj, openModelExplorer)` enables you to add the target dictionary to the **Model Hierarchy** pane without opening Model Explorer.

Input Arguments

dictionaryObj — Target data dictionary

`Simulink.data.Dictionary` object

Target data dictionary, specified as a `Simulink.data.Dictionary` object. Before you use this function, represent the target dictionary with a `Simulink.data.Dictionary` object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

openModelExplorer — Flag to open Model Explorer

`true` (default) | `false`

Flag to open Model Explorer, specified as `true` or `false`.

Data Types: `logical`

Examples

Show Data Dictionary in Model Explorer

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Open Model Explorer and display `myDictionary_ex_API` as the selected node of the model hierarchy tree in the **Model Hierarchy** pane.

```
show(myDictionaryObj)
```

Add Data Dictionary to Model Hierarchy Tree

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Add `myDictionary_ex_API.sldd` to the model hierarchy tree without opening Model Explorer.

```
show(myDictionaryObj, false)
```

You can confirm the addition of `myDictionary_ex_API` to the model hierarchy tree by manually opening Model Explorer.

- “Store Data in Dictionary Programmatically”

Tips

- Use the `hide` function to remove a data dictionary from the tree in the **Model Hierarchy** pane of Model Explorer. The dictionary does not appear in the hierarchy again until you use the `show` function or you open and view the dictionary in the Model Explorer using the interface.

See Also

`Simulink.data.Dictionary` | `hide`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

Simulink.data.dictionary.Entry class

Package: Simulink.data.dictionary

Configure data dictionary entry

Description

An object of the `Simulink.data.dictionary.Entry` class represents one entry of a data dictionary. The object allows you to perform operations such as assign the entry a value or change the name of the entry.

Before you can create a new `Simulink.data.dictionary.Entry` object, you must create a `Simulink.data.dictionary.Section` object representing the data dictionary section that contains the target entry. However, once created, the `Simulink.data.dictionary.Entry` object exists independently of the `Simulink.data.dictionary.Section` object. Use the function `getSection` to create a `Simulink.data.dictionary.Section` object.

Construction

The functions `addEntry`, `getEntry`, and `find` create `Simulink.data.dictionary.Entry` objects.

Properties

DataSource — File name of containing data dictionary

character vector

File name of containing data dictionary, specified as a character vector. Changes you make to this property affect the represented data dictionary entry.

Example: `'myDictionary.sldd'`

Data Types: `char`

LastModified — Date and time of last modification

character vector

Date and time of last modification to entry, returned in Coordinated Universal Time (UTC) as a character vector. This property is read only.

LastModifiedBy — Name of last user to modify entry

character vector

Name of last user to modify entry, returned as a character vector. This property is read only.

Name — Name of entry

character vector

Name of entry, specified as a character vector. Changes you make to this property affect the represented data dictionary entry.

Data Types: char

Status — State of entry

'New' | 'Modified' | 'Unchanged' | 'Deleted'

State of entry, returned as 'New', 'Modified', 'Unchanged', or 'Deleted'. The state is valid since the last data dictionary save. If the state is 'Deleted', the represented entry was deleted from its data dictionary. This property is read only.

Methods

deleteEntry	Delete data dictionary entry
discardChanges	Discard changes to data dictionary entry
find	Search in array of data dictionary entries
getValue	Return value of data dictionary entry
setValue	Set value of data dictionary entry
showChanges	Display changes made to data dictionary entry

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Add Entry to Data Dictionary and Modify its Value

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Add an entry `myEntry` with value 27 to the Design Data section of `myDictionary_ex_API.sldd`. Assign the returned `Simulink.data.dictionary.Entry` object to variable `e`.

```
e = addEntry(dDataSectObj, 'myEntry', 27)
```

```
e =
```

```
Entry with properties:
```

```
    Name: 'myEntry'  
    Value: 27  
    DataSource: 'myDictionary_ex_API.sldd'  
    LastModified: '2014-Aug-26 18:42:08.439709'  
    LastModifiedBy: 'jsmith'  
    Status: 'New'
```

Change the value of `myEntry` from 27 to the character vector 'My New Value'.

```
setValue(e, 'My New Value')
```

```
e
```

```
e =
```

```
Entry with properties:
```

```

        Name: 'myEntry'
        Value: 'My New Value'
        DataSource: 'myDictionary_ex_API.sldd'
        LastModified: '2014-Aug-26 18:45:58.336598'
        LastModifiedBy: 'jsmith'
        Status: 'New'

```

Return Value of Data Dictionary Entry

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
fuelFlowObj = getEntry(dDataSectObj, 'fuelFlow');

```

Return the value of the entry `fuelFlow` and assign the value to the variable `fuelFlowValue`.

```

fuelFlowValue = getValue(fuelFlowObj)

fuelFlowValue =

```

237

Move Entry Within Data Dictionary Hierarchy

Represent the `Design Data` section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`. `myDictionary_ex_API.sldd` references the data dictionary `myRefDictionary_ex_API.sldd`.

```

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj, 'Design Data');

```

Create a `Simulink.data.dictionary.Entry` object representing the entry `fuelFlow`, which resides in `myDictionary_ex_API.sldd`. Assign the object to variable `e`.

```

e = getEntry(dDataSectObj, 'fuelFlow')

```

```
e =  
  
Entry with properties:  
  
    Name: 'fuelFlow'  
    Value: 237  
    DataSource: 'myDictionary_ex_API.sldd'  
    LastModified: '2014-Sep-05 13:12:06.099278'  
    LastModifiedBy: 'jsmith'  
    Status: 'Unchanged'
```

Migrate the entry `fuelFlow` to the reference data dictionary `myRefDictionary_ex_API.sldd` by modifying the `DataSource` property of `e`.

```
e.DataSource = 'myRefDictionary_ex_API.sldd'  
  
e =  
  
Entry with properties:  
  
    Name: 'fuelFlow'  
    Value: 237  
    DataSource: 'myRefDictionary_ex_API.sldd'  
    LastModified: '2014-Sep-05 13:12:06.099278'  
    LastModifiedBy: 'jsmith'  
    Status: 'Modified'
```

Because `myDictionary_ex_API.sldd` references `myRefDictionary_ex_API.sldd`, both dictionaries belong to the same dictionary hierarchy, allowing you to migrate the entry `fuelFlow` between them.

- “Store Data in Dictionary Programmatically”

See Also

`Simulink.data.Dictionary` | `Simulink.data.dictionary.Section` | `getEntry`

Topics

“Store Data in Dictionary Programmatically”
“What Is a Data Dictionary?”

Introduced in R2015a

deleteEntry

Class: Simulink.data.dictionary.Entry

Package: Simulink.data.dictionary

Delete data dictionary entry

Syntax

```
deleteEntry(entryObj)
```

Description

`deleteEntry(entryObj)` deletes the data dictionary entry represented by `entryObj`, a `Simulink.data.dictionary.Entry` object. The represented entry no longer exists in the data dictionary that defined it.

The function sets the `Status` properties of any `Simulink.data.dictionary.Entry` objects representing the deleted entry to `'Deleted'`. You can access only the `Status` properties of the objects.

Input Arguments

entryObj — Target data dictionary entry

`Simulink.data.dictionary.Entry` object

Target data dictionary entry, specified as a `Simulink.data.dictionary.Entry` object. Before you use this function, represent the target entry with a `Simulink.data.dictionary.Entry` object by using, for example, the `getEntry` function.

Examples

Delete Data Dictionary Entry

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');  
fuelFlowObj = getEntry(dDataSectObj, 'fuelFlow');
```

Delete the entry `fuelFlow` from the data dictionary `myDictionary_ex_API.sldd`. `myDictionary_ex_API.sldd` no longer contains the `fuelFlow` entry.

```
deleteEntry(fuelFlowObj)
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use the Model Explorer window to view the contents of a data dictionary and delete entries.

See Also

`Simulink.data.dictionary.Entry` | `addEntry`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

discardChanges

Class: Simulink.data.dictionary.Entry

Package: Simulink.data.dictionary

Discard changes to data dictionary entry

Syntax

```
discardChanges(entryObj)
```

Description

`discardChanges(entryObj)` discards all changes made to the data dictionary entry `entryObj`, a `Simulink.data.dictionary.Entry` object, since the last time the containing data dictionary was saved using the `saveChanges` function. The changes to the entry are permanently lost.

Input Arguments

entryObj — Target data dictionary entry

`Simulink.data.dictionary.Entry` object

Target data dictionary entry, specified as a `Simulink.data.dictionary.Entry` object. Before you use this function, represent the target entry with a `Simulink.data.dictionary.Entry` object by using, for example, the `getEntry` function.

Examples

Discard Changes to Data Dictionary Entry

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
fuelFlowObj = getEntry(dDataSectObj, 'fuelFlow');
```

Change the entry `fuelFlow` by assigning it the new value 493. Confirm a change was made by viewing the `Status` property of `fuelFlowObj`.

```
setValue(fuelFlowObj, 493);
fuelFlowObj
```

```
fuelFlowObj =
```

```
Entry with properties:
```

```
    Name: 'fuelFlow'
    Value: 493
    DataSource: 'myDictionary_ex_API.sldd'
    LastModified: '2014-Sep-05 13:14:30.661978'
    LastModifiedBy: 'jsmith'
    Status: 'Modified'
```

Discard all changes to the entry `fuelFlow`. The `Status` property of `fuelFlowObj` shows that changes were discarded.

```
discardChanges(fuelFlowObj)
fuelFlowObj
```

```
fuelFlowObj =
```

```
Entry with properties:
```

```
    Name: 'fuelFlow'
    Value: 237
    DataSource: 'myDictionary_ex_API.sldd'
    LastModified: '2014-Sep-05 13:12:06.099278'
```

```
LastModifiedBy: 'jsmith'  
Status: 'Unchanged'
```

- “Store Data in Dictionary Programmatically”

Tips

- You can use the `discardChanges` function or the `saveChanges` function with an entire data dictionary, discarding or saving changes to all entries in the dictionary at once. However, only the `discardChanges` function can additionally operate on individual entries. You cannot use the `saveChanges` function to save changes to individual entries.

Alternatives

You can use Model Explorer and the Comparison Tool to discard changes to data dictionary entries. See “View and Revert Changes to Dictionary Entries” for more information.

See Also

`Simulink.data.dictionary.Entry` | `saveChanges`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

find

Class: Simulink.data.dictionary.Entry

Package: Simulink.data.dictionary

Search in array of data dictionary entries

Syntax

```
foundEntries = find(targetEntries, PName1, PValue1, ..., PNameN, PValueN)
foundEntries = find(targetEntries, PName1, PValue1, ..., PNameN, PValueN,
options)
```

Description

`foundEntries = find(targetEntries, PName1, PValue1, ..., PNameN, PValueN)` searches the array of data dictionary entries `targetEntries` using search criteria `PName1, PValue1, ..., PNameN, PValueN`, and returns an array of entries matching the criteria. This syntax matches the search criteria with the properties of the target entries, which are `Simulink.data.dictionary.Entry` objects, but not with the properties of their values. See `Simulink.data.dictionary.Entry` for a list of data dictionary entry properties.

`foundEntries = find(targetEntries, PName1, PValue1, ..., PNameN, PValueN, options)` searches for data dictionary entries using additional search options. For example, you can match the search criteria with the values of the target entries.

Input Arguments

targetEntries — Data dictionary entries to search

array of `Simulink.data.dictionary.Entry` objects

Data dictionary entries to search, specified as an array

`Simulink.data.dictionary.Entry` objects. Before you use this function, represent

the target entries with `Simulink.data.dictionary.Entry` objects by using, for example, the `getEntry` function.

Example: `[myEntryObj1,myEntryObj2,myEntryObj3]`

PName1, PValue1, . . . , PNameN, PValueN — Search criteria

name-value pairs representing properties

Search criteria, specified as one or more name-value pairs representing names and values of properties of the target data dictionary entries. For a list of the properties of a data dictionary entry, see `Simulink.data.dictionary.Entry`. If you specify more than one name-value pair, the returned entries meet all of the criteria.

If you include the `'-value'` option to search in the values of the target entries, the search criteria apply to the values of the entries rather than to the entries themselves.

Example: `'LastModifiedBy','jsmith'`

Example: `'DataSource','myRefDictionary_ex_API.slidd'`

options — Additional search options

supported option codes

Additional search options, specified as one or more of the following supported option codes.

<code>'-value'</code>	This option causes <code>find</code> to search only in the values of the target data dictionary entries. Specify this option before any other search criteria or options arguments.
<code>'-and', '-or', '-xor', or '-not'</code> logical operators	These options modify or combine multiple search criteria or other option codes.
<code>'-property', propertyName</code>	This name-value pair causes <code>find</code> to search for entries or values that have the property <code>propertyName</code> regardless of the value of the property. Specify <code>propertyName</code> as a character vector.
<code>'-class', className</code>	This name-value pair causes <code>find</code> to search for entries or values that are objects of the class <code>className</code> . Specify <code>className</code> as a character vector.

<code>'-isa', className</code>	This name-value pair causes <code>find</code> to search for entries or values that are objects of the class or of any subclass derived from the class <code>className</code> . Specify <code>className</code> as a character vector.
<code>'-regexp'</code>	This option allows you to use regular expressions in your search criteria. This option affects only search criteria that follow <code>'-regexp'</code> .

Example: `'-value'`

Example: `'-value', '-property', 'CoderInfo'`

Example: `'-value', '-class', 'Simulink.Parameter'`

Output Arguments

foundEntries — Data dictionary entries matching search criteria

array of `Simulink.data.dictionary.Entry` objects

Data dictionary entries matching the specified search criteria, returned as an array of `Simulink.data.dictionary.Entry` objects.

Examples

Search Data Dictionary Entry Values for Specific Class

Search in an array of data dictionary entries `myEntryObjs` for entries whose values are objects of the class `Simulink.Parameter`.

```
foundEntries = find(myEntryObjs, '-value', '-class', 'Simulink.Parameter')
```

Search Data Dictionary Entries for Modifying User

Search in an array of data dictionary entries `myEntryObjs` for entries that were last modified by the user `jsmith`.

```
foundEntries = find(myEntryObjs, 'LastModifiedBy', 'jsmith')
```

Search Data Dictionary Entries Using Multiple Criteria

Search in an array of data dictionary entries `myEntryObjs` for entries that were last modified by the user `jsmith` or whose names begin with `fuel`.

```
foundEntries = find(myEntryObjs, 'LastModifiedBy', 'jsmith', '-or', ...  
'-regexp', 'Name', 'fuel*')
```

Search Data Dictionary Entries Using Regular Expressions

Search in an array of data dictionary entries `myEntryObjs` for entries whose names begin with `Press`.

```
foundEntries = find(myEntryObjs, '-regexp', 'Name', 'Press*')
```

Search Data Dictionary Entries for Specific Value

Search in an array of data dictionary entries `myEntryObjs` for entries whose values are 273. If you find more than one entry, store the entries in an array called `foundEntries`.

```
foundEntries = [];  
for i = 1:length(myEntryObjs)  
    if getValue(myEntryObjs(i)) == 273  
        foundEntries = [foundEntries myEntryObjs(i)];  
    end  
end
```

Search Data Dictionary Entry Values for Specific Property

Search in an array of data dictionary entries `myEntryObjs` for entries whose values have a property `DataType`.

```
foundEntries = find(myEntryObjs, '-value', '-property', 'DataType')
```

- “Store Data in Dictionary Programmatically”

See Also

`Simulink.data.dictionary.Entry` | `find`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

getValue

Class: Simulink.data.dictionary.Entry

Package: Simulink.data.dictionary

Return value of data dictionary entry

Syntax

```
entryValue = getValue(entryObj)
```

Description

`entryValue = getValue(entryObj)` returns the value of the data dictionary entry `entryObj`, a `Simulink.data.dictionary.Entry` object.

To programmatically access variables for the purpose of sweeping block parameter values, consider using `Simulink.SimulationInput` objects instead of modifying the variables through the programmatic interface of the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values”.

Input Arguments

entryObj — Target data dictionary entry

`Simulink.data.dictionary.Entry` object

Target data dictionary entry, specified as a `Simulink.data.dictionary.Entry` object. Before you use this function, represent the target entry with a `Simulink.data.dictionary.Entry` object by using, for example, the `getEntry` function.

Examples

Return Value of Data Dictionary Entry

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.slidd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.slidd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');  
fuelFlowObj = getEntry(dDataSectObj, 'fuelFlow');
```

Return the value of the entry `fuelFlow` and assign the value to variable `fuelFlowValue`.

```
fuelFlowValue = getValue(fuelFlowObj)  
  
fuelFlowValue =
```

```
    237
```

- “Store Data in Dictionary Programmatically”

See Also

`Simulink.data.dictionary.Entry` | `setValue`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

setValue

Class: Simulink.data.dictionary.Entry

Package: Simulink.data.dictionary

Set value of data dictionary entry

Syntax

```
setValue(entryObj, newValue)
```

Description

`setValue(entryObj, newValue)` assigns the value `newValue` to the data dictionary entry `entryObj`, a `Simulink.data.dictionary.Entry` object.

To programmatically access variables for the purpose of sweeping block parameter values, consider using `Simulink.SimulationInput` objects instead of modifying the variables through the programmatic interface of the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values”.

Input Arguments

entryObj — Target data dictionary entry

`Simulink.data.dictionary.Entry` object

Target data dictionary entry, specified as a `Simulink.data.dictionary.Entry` object. Before you use this function, represent the target entry with a `Simulink.data.dictionary.Entry` object by using, for example, the `getEntry` function.

newValue — Value to assign to data dictionary entry

MATLAB expression

Value to assign to data dictionary entry, specified as a MATLAB expression. The expression must return a value that is supported by the data dictionary section that contains the entry.

Example: 27.5

Example: myBaseWorkspaceVariable

Example: Simulink.Parameter

Examples

Set Value of Data Dictionary Entry

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
fuelFlowObj = getEntry(dDataSectObj, 'fuelFlow');
```

Set the value of the entry `fuelFlow` to 493. Then, view the `Value` property of `fuelFlowObj` to observe the change.

```
setValue(fuelFlowObj, 493)
fuelFlowObj
```

```
fuelFlowObj =
```

```
Entry with properties:
```

```
    Name: 'fuelFlow'
    Value: 493
    DataSource: 'myDictionary_ex_API.sldd'
    LastModified: '2014-Sep-05 13:37:22.161124'
    LastModifiedBy: 'jsmith'
    Status: 'Modified'
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use the Model Explorer window to view and change the values of data dictionary entries.

See Also

`Simulink.data.dictionary.Entry | getValue`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

showChanges

Class: Simulink.data.dictionary.Entry

Package: Simulink.data.dictionary

Display changes made to data dictionary entry

Syntax

```
showChanges(entryObj)
```

Description

`showChanges(entryObj)` opens the Comparison Tool to show changes made to the data dictionary entry `entryObj`, a `Simulink.data.dictionary.Entry` object. The Comparison Tool displays the properties of `entryObj` as they were when the data dictionary was last saved and as they were when the `showChanges` function was called.

Input Arguments

entryObj — Target data dictionary entry

`Simulink.data.dictionary.Entry` object

Target data dictionary entry, specified as a `Simulink.data.dictionary.Entry` object. Before you use this function, represent the target entry with a `Simulink.data.dictionary.Entry` object by using, for example, the `getEntry` function.

Examples

View Unsaved Changes to Data Dictionary Entry

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.slidd`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.slidd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');  
fuelFlowObj = getEntry(dDataSectObj, 'fuelFlow');
```

Make a change to the entry `fuelFlow` by assigning it the new value 494.

```
setValue(fuelFlowObj, 494);
```

Observe the unsaved change to the entry `fuelFlow`. The Comparison Tool opens and compares side by side the current state of the entry with its most recently saved state.

```
showChanges(fuelFlowObj)
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer and the Comparison Tool to view changes to data dictionary entries. See “View and Revert Changes to Dictionary Entries” for more information.

See Also

`Simulink.data.dictionary.Entry` | `discardChanges`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

Simulink.data.dictionary.EnumTypeDefinition class

Package: Simulink.data.dictionary

Store enumerated type definition in data dictionary

Description

An object of the `Simulink.data.dictionary.EnumTypeDefinition` class defines an enumerated data type in a data dictionary. You store the object in a data dictionary entry so models linked to the dictionary can use the enumerated type definition.

In the MATLAB base workspace, objects of this class retain information about an enumerated type but do not define the type for use by other variables or by models.

Construction

When you use the function `importEnumTypes` to import the definitions of enumerated types to a data dictionary, Simulink creates a `Simulink.data.dictionary.EnumTypeDefinition` object in the dictionary for each imported definition. The dictionary stores each object in an individual entry.

The constructor `Simulink.data.dictionary.EnumTypeDefinition` creates an instance of this class with default property values and a single enumeration member that has underlying integer value 0.

Properties

AddClassNameToEnumNames — Flag to control enumeration identifiers in generated code
`false` (default) | `true`

Flag to prefix enumerations with the class name in generated code, specified as `true` or `false`.

If you specify `true`, when you generate code the identifier of each enumeration member begins with the name of the enumeration class. For example, an enumeration class

LEDcolor with enumeration members GREEN and RED defines the enumeration members in generated code as LEDcolor_GREEN and LEDcolor_RED.

Data Types: logical

DataScope — Flag to control data type definition in generated code

'Auto' (default) | 'Imported' | 'Exported'

Flag to control data type definition in generated code, specified as 'Auto', 'Imported', or 'Exported'. The table describes the behavior of generated code for each value.

Value	Action
Auto (default)	<p>If you do not specify the property Headerfile, export the data type definition to <i>model_types.h</i>, where <i>model</i> is the model name.</p> <p>If you specify Headerfile, import the data type definition from the specified header file.</p>
Exported	<p>Export the data type definition to a separate header file.</p> <p>If you do not specify the property Headerfile, the header file name defaults to <i>type.h</i>, where <i>type</i> is the data type name.</p>
Imported	<p>Import the data type definition from a separate header file.</p> <p>If you do not specify the property Headerfile, the header file name defaults to <i>type.h</i>, where <i>type</i> is the data type name.</p>

DefaultValue — Default enumeration member

' ' (default) | character vector

Default enumeration member, specified as a character vector. Specify DefaultValue as the name of an enumeration member you have already defined.

When you create a `Simulink.data.dictionary.EnumTypeDefinition` object, `DefaultValue` is an empty character vector, `''`, and Simulink uses the first enumeration member as the default member.

Example: `'enumMember1'`

Description — Description of enumerated data type in generated code

`''` (default) | character vector

Description of the enumerated data type, specified as a character vector. Use this property to explain the purpose of the type in generated code.

Example: `'Two possible colors of LED indicator: GREEN and RED.'`

Data Types: `char`

HeaderFile — Name of header file defining enumerated data type in generated code

`''` (default) | character vector

Name of the header file that defines the enumerated data type in generated code, specified as a character vector. Use a `.h` extension to specify the file name.

If you do not specify `HeaderFile`, generated code uses a default header file name that depends on the value of the `DataScope` property .

Example: `'myTypeIncludeFile.h'`

Data Types: `char`

StorageType — Data type of underlying integer values

`''` (default) | character vector

Data type of the integer values underlying the enumeration members, specified as a character vector. Generated code stores the underlying integer values using the data type you specify.

You can specify one of these supported integer types:

- `'int8'`
- `'int16'`
- `'int32'`
- `'uint8'`

- 'uint16'

To store the underlying integer values in generated code using the native integer type of the target hardware, specify `StorageType` as an empty character vector, `''`, which is the default value.

Example: 'int16'

''

Methods

`appendEnumeral` Add enumeration member to enumerated data type definition in data dictionary

`removeEnumeral` Remove enumeration member from enumerated data type definition in data dictionary

Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Programmatically Create Enumerated Type Definition in Data Dictionary

Create an object that can store the definition of an enumerated type. By default, the new type defines a single enumeration member `enum1` with underlying integer value 0.

```
myColors = Simulink.data.dictionary.EnumTypeDefinition
```

```
myColors =
```

```
    Simulink.data.dictionary.EnumTypeDefinition  
    enum1
```

Add some enumeration members to the definition of the type.

```

appendEnumeral(myColors, 'Orange', 1, '')
appendEnumeral(myColors, 'Black', 2, '')
appendEnumeral(myColors, 'Cyan', 3, '')
myColors

myColors =

    Simulink.data.dictionary.EnumTypeDefinition
        enum1
        Orange
        Black
        Cyan

```

Remove the default enumeration member `enum1`. Since `enum1` is the first enumeration member in the list, identify it with index 1.

```

removeEnumeral(myColors, 1)
myColors

myColors =

    Simulink.data.dictionary.EnumTypeDefinition
        Orange
        Black
        Cyan

```

Customize the enumerated type by configuring the properties of the object representing it.

```

myColors.Description = 'These are my favorite colors.';
myColors.DefaultValue = 'Cyan';
myColors.HeaderFile = 'colorsType.h';

```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');

```

Import the object that defines the enumerated type `myColors` to the dictionary.

```

importFromBaseWorkspace(myDictionaryObj, 'varList', {'myColors'});

```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer to add and modify enumerated data types stored in a data dictionary.

See Also

`Simulink.data.Dictionary`

Topics

“Store Data in Dictionary Programmatically”

“Use Enumerated Data in Simulink Models”

Introduced in R2015a

appendEnumeral

Class: Simulink.data.dictionary.EnumTypeDefinition

Package: Simulink.data.dictionary

Add enumeration member to enumerated data type definition in data dictionary

Syntax

```
appendEnumeral (typeObj, memberName, memberValue, memberDesc)
```

Description

`appendEnumeral (typeObj, memberName, memberValue, memberDesc)` adds an enumeration member to the enumerated type definition stored by `typeObj`, a `Simulink.data.dictionary.EnumTypeDefinition` object.

Input Arguments

typeObj — Target enumerated type definition

`Simulink.data.dictionary.EnumTypeDefinition` object

Target enumerated type definition, specified as a `Simulink.data.dictionary.EnumTypeDefinition` object.

memberName — Name of new enumeration member

character vector

Name of the new enumeration member, specified as a character vector.

Example: `'myNewEnumMember'`

Data Types: `char`

memberValue — Integer value underlying new enumeration member

integer

Integer value underlying the new enumeration member, specified as an integer.

The definition of the enumeration class determines the integer data type used in generated code to store the underlying values of enumeration members.

Example: 3

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

memberDesc — Description of new enumeration member

character vector

Description of the new enumeration member, specified as a character vector.

If you do not want to supply a description for the enumeration member, use an empty character vector.

Example: `'Enumeration member number 1.'`

Example: `''`

Data Types: `char`

Examples

Programmatically Create Enumerated Type Definition in Data Dictionary

Create an object that can store the definition of an enumerated type. By default, the new type defines a single enumeration member `enum1` with underlying integer value 0.

```
myColors = Simulink.data.dictionary.EnumTypeDefinition
```

```
myColors =
```

```
    Simulink.data.dictionary.EnumTypeDefinition  
    enum1
```

Add some enumeration members to the definition of the type.

```
appendEnumeral(myColors, 'Orange', 1, '')  
appendEnumeral(myColors, 'Black', 2, '')
```

```
appendEnumeral(myColors, 'Cyan', 3, '')
myColors

myColors =

    Simulink.data.dictionary.EnumTypeDefinition
        enum1
        Orange
        Black
        Cyan
```

Remove the default enumeration member `enum1`. Since `enum1` is the first enumeration member in the list, identify it with index 1.

```
removeEnumeral(myColors, 1)
myColors

myColors =

    Simulink.data.dictionary.EnumTypeDefinition
        Orange
        Black
        Cyan
```

Customize the enumerated type by configuring the properties of the object representing it.

```
myColors.Description = 'These are my favorite colors.';
myColors.DefaultValue = 'Cyan';
myColors.HeaderFile = 'colorsType.h';
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import the object that defines the enumerated type `myColors` to the dictionary.

```
importFromBaseWorkspace(myDictionaryObj, 'varList', {'myColors'});
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer to add enumeration members to the enumerated data type represented by a `Simulink.data.dictionary.EnumTypeDefinition` object.

See Also

`Simulink.data.dictionary.EnumTypeDefinition` |
`Simulink.data.dictionary.EnumTypeDefinition.removeEnumeral`

Topics

“Store Data in Dictionary Programmatically”
“Use Enumerated Data in Simulink Models”

Introduced in R2015a

removeEnumeral

Class: Simulink.data.dictionary.EnumTypeDefinition

Package: Simulink.data.dictionary

Remove enumeration member from enumerated data type definition in data dictionary

Syntax

```
removeEnumeral(typeObj, memberNum)
```

Description

`removeEnumeral(typeObj, memberNum)` removes an enumeration member from the enumerated type definition stored by `typeObj`, a `Simulink.data.dictionary.EnumTypeDefinition` object.

Input Arguments

typeObj — Target enumerated type definition

`Simulink.data.dictionary.EnumTypeDefinition` object

Target enumerated type definition, specified as a `Simulink.data.dictionary.EnumTypeDefinition` object.

memberNum — Index of target enumeration member

integer

Index of target enumeration member, specified as an integer.

The first enumeration member in an enumerated type definition has index 1. For example, suppose an enumerated type `BasicColors` has this definition:

```
myColors =
```

```
    Simulink.data.dictionary.EnumTypeDefinition
```

```
Orange  
Black  
Cyan
```

To remove the enumeration member `Black`, specify `memberNum` as 2. To remove the enumeration member `Cyan`, specify 3.

Do not specify `memberNum` using the integer value underlying an enumeration member. The integer value underlying the member is not equivalent to the index of the member.

Example: 3

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

Examples

Programmatically Create Enumerated Type Definition in Data Dictionary

Create an object that can store the definition of an enumerated type. By default, the new type defines a single enumeration member `enum1` with underlying integer value 0.

```
myColors = Simulink.data.dictionary.EnumTypeDefinition
```

```
myColors =
```

```
    Simulink.data.dictionary.EnumTypeDefinition  
    enum1
```

Add some enumeration members to the definition of the type.

```
appendEnumeral(myColors, 'Orange', 1, '')  
appendEnumeral(myColors, 'Black', 2, '')  
appendEnumeral(myColors, 'Cyan', 3, '')  
myColors
```

```
myColors =
```

```
    Simulink.data.dictionary.EnumTypeDefinition  
    enum1  
    Orange
```

```

    Black
    Cyan

```

Remove the default enumeration member `enum1`. Since `enum1` is the first enumeration member in the list, identify it with index 1.

```

removeEnumeral(myColors,1)
myColors

myColors =

    Simulink.data.dictionary.EnumTypeDefinition
        Orange
        Black
        Cyan

```

Customize the enumerated type by configuring the properties of the object representing it.

```

myColors.Description = 'These are my favorite colors.';
myColors.DefaultValue = 'Cyan';
myColors.HeaderFile = 'colorsType.h';

```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');

```

Import the object that defines the enumerated type `myColors` to the dictionary.

```

importFromBaseWorkspace(myDictionaryObj, 'varList', {'myColors'});

```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer to remove enumeration members from the enumerated data type represented by a `Simulink.data.dictionary.EnumTypeDefinition` object.

See Also

`Simulink.data.dictionary.EnumTypeDefinition` |
`Simulink.data.dictionary.EnumTypeDefinition.appendEnumeral`

Topics

“Store Data in Dictionary Programmatically”
“Use Enumerated Data in Simulink Models”

Introduced in R2015a

Simulink.data.dictionary.Section class

Package: Simulink.data.dictionary

Configure data dictionary section

Description

An object of the `Simulink.data.dictionary.Section` class represents one section of a data dictionary, such as Design Data or Configurations. The object allows you to perform operations on the section such as add or delete entries and import data from files.

Before you can create a `Simulink.data.dictionary.Section` object, you must create a `Simulink.data.Dictionary` object representing the target data dictionary. Once created, the `Simulink.data.dictionary.Section` object exists independently of the `Simulink.data.Dictionary` object.

Construction

The function `getSection` creates a `Simulink.data.dictionary.Section` object.

Properties

Name — Name of data dictionary section

character vector

Name of data dictionary section, returned as a character vector. This property is read only.

Methods

<code>addEntry</code>	Add new entry to data dictionary section
<code>assignin</code>	Assign value to data dictionary entry
<code>deleteEntry</code>	Delete data dictionary entry
<code>evalin</code>	Evaluate MATLAB expression in data dictionary section
<code>exist</code>	Check existence of data dictionary entry
<code>exportToFile</code>	Export data dictionary entries from section to MAT-file or MATLAB file
<code>find</code>	Search in data dictionary section
<code>getEntry</code>	Create <code>Simulink.data.dictionary.Entry</code> object to represent data dictionary entry
<code>importFromFile</code>	Import variables from MAT-file or MATLAB file to data dictionary section

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create New Data Dictionary Section Object

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
dDataSectObj = getSection(myDictionaryObj, 'Design Data')
```

```
dDataSectObj =
```

Section with properties:

Name: 'Design Data'

- “Store Data in Dictionary Programmatically”

See Also

`Simulink.data.Dictionary` | `getSection`

Topics

“Store Data in Dictionary Programmatically”

“What Is a Data Dictionary?”

Introduced in R2015a

addEntry

Class: Simulink.data.dictionary.Section

Package: Simulink.data.dictionary

Add new entry to data dictionary section

Syntax

```
addEntry(sectionObj,entryName,entryValue)
entryObj = addEntry(sectionObj,entryName,entryValue)
```

Description

`addEntry(sectionObj,entryName,entryValue)` adds an entry, with name `entryName` and value `entryValue`, to the data dictionary section `sectionObj`, a `Simulink.data.dictionary.Section` object.

`entryObj = addEntry(sectionObj,entryName,entryValue)` returns a `Simulink.data.dictionary.Entry` object representing the newly added data dictionary entry.

Input Arguments

sectionObj — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a `Simulink.data.dictionary.Section` object. Before you use this function, represent the target section with a `Simulink.data.dictionary.Section` object by using, for example, the `getSection` function.

entryName — Name of new data dictionary entry

character vector

Name of new data dictionary entry, specified as a character vector.

Example: 'myNewEntry'

Data Types: char

entryValue — Value of new data dictionary entry

MATLAB expression

Value of new data dictionary entry, specified as a MATLAB expression that returns any valid data dictionary content.

Example: 27.5

Example: myBaseWorkspaceVariable

Example: Simulink.Parameter

Examples

Add Entry to Design Data Section of Data Dictionary

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Add an entry to the Design Data section of `myDictionary_ex_API.sldd` an entry `myNewEntry` with value 237.

```
addEntry(dDataSectObj, 'myNewEntry', 237)
```

Add New Simulink.Parameter Object to Data Dictionary

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Add an entry to the Design Data section of `myDictionary_ex_API.sldd`. Name the new entry `myNewParam` and assign a `Simulink.Parameter` object to the value.

```
addEntry(dDataSectObj, 'myNewParam', Simulink.Parameter)
```

The expression `Simulink.Parameter` constructs a new `Simulink.Parameter` object, and the `addEntry` function assigns the object to the value of the new data dictionary entry `myNewParam`.

- “Store Data in Dictionary Programmatically”

Tips

- `addEntry` returns an error if the entry name you specify with `entryName` is already the name of an entry in the target data dictionary section or in the same section of any referenced dictionaries.

Alternatives

You can use Model Explorer to add entries to a data dictionary in the same way you can use it to add variables to a model workspace or the base workspace.

See Also

`Simulink.data.dictionary.Entry` | `Simulink.data.dictionary.Section` | `assignin`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

assignin

Class: Simulink.data.dictionary.Section

Package: Simulink.data.dictionary

Assign value to data dictionary entry

Syntax

```
assignin(sectionObj,entryName,entryValue)
```

Description

`assignin(sectionObj,entryName,entryValue)` assigns the value `entryValue` to the data dictionary entry `entryName` in the data dictionary section `sectionObj`, a `Simulink.data.dictionary.Section` object. If an entry with the specified name is not in the target section, `assignin` creates the entry with the specified name and value.

If an entry with the name specified by input argument `entryName` is not defined in the target data dictionary section but is defined in a referenced dictionary, `assignin` does not create a new entry in the target section but operates on the entry in the referenced dictionary.

To programmatically access variables for the purpose of sweeping block parameter values, consider using `Simulink.SimulationInput` objects instead of modifying the variables through the programmatic interface of the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values”.

Input Arguments

sectionObj — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a `Simulink.data.dictionary.Section` object. Before you use this function, represent the target section with a

Simulink.data.dictionary.Section object by using, for example, the `getSection` function.

entryName — Name of target data dictionary entry

character vector

Name of target data dictionary entry, specified as a character vector. If a matching entry does not already exist, the functions creates a new entry using the specified name.

Example: 'myEntry'

Data Types: char

entryValue — Value to assign to data dictionary entry

MATLAB expression

Value to assign to data dictionary entry, specified as a MATLAB expression that returns any valid data dictionary content.

Example: 27.5

Example: myBaseWorkspaceVariable

Example: Simulink.Parameter

Examples

Assign Value to Data Dictionary Entry

Assign a value to a data dictionary entry by operating on a `Simulink.data.dictionary.Section` object.

Represent the Design Data section of the data dictionary `myDictionary_ex_API.slidd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.slidd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Assign the value 237 to an entry `myAssignedEntry` in the data dictionary `myDictionary_ex_API.slidd`. If an entry named `myAssignedEntry` is not in `myDictionary_ex_API.slidd`, create it.

```
assignin(dDataSectObj, 'myAssignedEntry', 237)
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use the Model Explorer window to view and change the values of data dictionary entries.

See Also

`Simulink.data.dictionary.Section | setValue`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

deleteEntry

Class: Simulink.data.dictionary.Section

Package: Simulink.data.dictionary

Delete data dictionary entry

Syntax

```
deleteEntry(sectionObj,entryName)
deleteEntry(sectionObj,entryName,'DataSource',dictionaryName)
```

Description

`deleteEntry(sectionObj,entryName)` deletes a data dictionary entry `entryName` from the data dictionary section `sectionObj`, a `Simulink.data.dictionary.Section` object. If there are multiple entries with the specified name in a hierarchy of reference dictionaries, the function deletes all the entries. If you represent a data dictionary entry with one or more `Simulink.data.dictionary.Entry` objects and later delete the entry using the `deleteEntry` function, the objects remain with their `Status` property set to `'Deleted'`.

`deleteEntry(sectionObj,entryName,'DataSource',dictionaryName)` deletes an entry that is defined in the data dictionary `DictionaryName`. Use this syntax to uniquely identify an entry that is defined more than once in a hierarchy of referenced data dictionaries.

Input Arguments

sectionObj — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a `Simulink.data.dictionary.Section` object. Before you use this function, represent the target section with a

Simulink.data.dictionary.Section object by using, for example, the getSection function.

entryName — Name of target data dictionary entry

character vector

Name of target data dictionary entry, specified as a character vector.

Example: 'myEntry'

Data Types: char

dictionaryName — Name of data dictionary that defines target entry

character vector

File name of data dictionary that defines the target entry, specified as a character vector including the .sldd extension.

Example: 'mySubDictionary_ex_API.sldd'

Data Types: char

Examples

Delete Entry from Data Dictionary Section

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj. The Design Data section of myDictionary_ex_API.sldd already contains an entry named fuelFlow.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Delete the entry fuelFlow from the data dictionary myDictionary_ex_API.sldd. myDictionary_ex_API.sldd no longer contains the fuelFlow entry.

```
deleteEntry(dDataSectObj, 'fuelFlow')
```

Delete Entry from Reference Data Dictionary

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Delete the entry `myRefEntry` from the data dictionary

`myRefDictionary_ex_API.sldd`. `myDictionary_ex_API.sldd` references `myRefDictionary_ex_API.sldd`, and `myRefDictionary_ex_API.sldd` defines an entry `myRefEntry`.

```
deleteEntry(dDataSectObj, 'myRefEntry', 'DataSource', ...  
            'myRefDictionary_ex_API.sldd')
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use the Model Explorer window to delete entries from a data dictionary in the same way you can delete variables from a model workspace or the base workspace.

See Also

`Simulink.data.dictionary.Entry` | `Simulink.data.dictionary.Section` | `addEntry`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

evalin

Class: Simulink.data.dictionary.Section

Package: Simulink.data.dictionary

Evaluate MATLAB expression in data dictionary section

Syntax

```
returnValue = evalin(sectionObj,expression)
```

Description

`returnValue = evalin(sectionObj,expression)` evaluates a MATLAB expression in the data dictionary section `sectionObj` and returns the values returned by `expression`.

To programmatically access variables for the purpose of sweeping block parameter values, consider using `Simulink.SimulationInput` objects instead of modifying the variables through the programmatic interface of the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values”.

Input Arguments

sectionObj — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a `Simulink.data.dictionary.Section` object. Before you use this function, represent the target section with a `Simulink.data.dictionary.Section` object by using, for example, the `getSection` function.

expression — MATLAB expression to evaluate

character vector

MATLAB expression to evaluate, specified as a character vector.

Example: 'a = 5.3'

Example: 'whos'

Example: 'CurrentSpeed.Value = 290.73'

Data Types: char

Examples

List All Entries in Data Dictionary Section

Represent the Design Data section of the data dictionary `myDictionary_ex_API.slidd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.slidd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Execute the `whos` command in the Design Data section of `myDictionary_ex_API.slidd`.

```
evalin(dDataSectObj, 'whos')
```

Name	Size	Bytes	Class	Attributes
fuelFlow	1x1	8	double	
myRefEntry	1x1	1	logical	
parameterGain37	1x1	112	Simulink.Parameter	

- “Store Data in Dictionary Programmatically”

Tips

- `evalin` allows you to treat a data dictionary section as a MATLAB workspace. You can think of entries contained in the section as workspace variables you can manipulate with MATLAB expressions.

See Also

`Simulink.data.dictionary.Section` | `Simulink.data.evalinGlobal`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

exist

Class: Simulink.data.dictionary.Section

Package: Simulink.data.dictionary

Check existence of data dictionary entry

Syntax

```
doesExist = exist(sectionObj,entryName)
```

Description

`doesExist = exist(sectionObj,entryName)` determines if the data dictionary section `sectionObj` contains an entry by the name of `entryName` and returns an indication.

Input Arguments

sectionObj — Target data dictionary section

Simulink.data.dictionary.Section object

Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the `getSection` function.

entryName — Name of target entry

character vector

Name of target entry, specified as a character vector.

Example: 'myEntry'

Data Types: char

Output Arguments

doesExist — Indication of entry existence

0 | 1

Indication of entry existence, returned as 0 if false and 1 if true.

Examples

Determine if Data Dictionary Entry Exists

Determine if an entry exists in a data dictionary by searching for the name of the entry

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Determine if an entry `fuelFlow` exists in the Design Data section of `myDictionary_ex_API.sldd`.

```
exist(dDataSectObj, 'fuelFlow')
```

```
ans =
```

```
1
```

Determine if an entry `myEntry` exists in the Design Data section of `myDictionary_ex_API.sldd`.

```
exist(dDataSectObj, 'myEntry')
```

```
ans =
```

```
0
```

- “Store Data in Dictionary Programmatically”

Tips

- `exist` also determines if a matching entry exists in the same section of any referenced data dictionaries. For example, if `sectionObj` represents the Design Data section of a data dictionary `myDictionary_ex_API.sldd`, `exist` searches the Design Data section of `myDictionary_ex_API.sldd` and the Design Data sections of any dictionaries referenced by `myDictionary_ex_API.sldd`.

Alternatives

You can use Model Explorer to search a data dictionary for an entry.

See Also

`Simulink.data.dictionary.Section` | `Simulink.data.existsInGlobal` | `find`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

exportToFile

Class: Simulink.data.dictionary.Section

Package: Simulink.data.dictionary

Export data dictionary entries from section to MAT-file or MATLAB file

Syntax

```
exportToFile(sectionObj, fileName)
```

Description

`exportToFile(sectionObj, fileName)` exports to a MAT or MATLAB file all the values of the entries contained in the data dictionary section `sectionObj`, a `Simulink.data.dictionary.Section` object. `exportToFile` exports the values of all entries, including those defined in referenced dictionaries.

Input Arguments

sectionObj — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a `Simulink.data.dictionary.Section` object. Before you use this function, represent the target section with a `Simulink.data.dictionary.Section` object by using, for example, the `getSection` function.

fileName — Name of MAT or MATLAB file

character vector

Name of target MAT or MATLAB file, specified as a character vector. `exportToFile` supplies a file extension `.mat` if you do not specify an extension.

Example: `'myNewFile.mat'`

Example: 'myNewFile.m'

Data Types: char

Examples

Export Data Dictionary Entries to MAT or MATLAB Files

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

Represent the Configurations section of `myDictionary_ex_API.sldd` with an object named `configSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');  
configSectObj = getSection(myDictionaryObj, 'Configurations');
```

Export the entries from the Design Data section of `myDictionary_ex_API.sldd` to a MATLAB file in your current working folder.

```
exportToFile(dDataSectObj, 'myDictionaryDesignData.m')
```

Export the entries from the Configurations section of `myDictionary_ex_API.sldd` to a MAT-file in your current working folder.

```
exportToFile(configSectObj, 'myDictionaryConfigurations.mat')
```

```
Exported 1 entries from scope 'Configurations'  
to MAT-file myDictionaryConfigurations.mat.
```

- “Store Data in Dictionary Programmatically”

Limitation

The `exportToFile` method does not export enumerated data types (which are stored as `Simulink.data.dictionary.EnumTypeDefinition` objects). To transfer or copy an enumerated type from one dictionary to another, use the `getEntry` and `addEntry` methods of `Simulink.data.dictionary.Section` objects.

Alternatives

You can use Model Explorer to export data dictionary entries to a file. See “Export Design Data from Dictionary” for more information.

See Also

`Simulink.data.dictionary.Section` | `importFromFile`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

find

Class: Simulink.data.dictionary.Section

Package: Simulink.data.dictionary

Search in data dictionary section

Syntax

```
foundEntries = find(sectionObj, PName1, PValue1, ..., PNameN, PValueN)
foundEntries = find(sectionObj, PName1, PValue1, ..., PNameN, PValueN,
options)
```

Description

`foundEntries = find(sectionObj, PName1, PValue1, ..., PNameN, PValueN)` searches the data dictionary section `sectionObj` using search criteria `PName1, PValue1, ..., PNameN, PValueN`, and returns an array of matching entries that were found in the target section. This syntax matches the search criteria with the properties of the entries in the target section but not with the properties of their values. See `Simulink.data.dictionary.Entry` for a list of data dictionary entry properties.

`foundEntries = find(sectionObj, PName1, PValue1, ..., PNameN, PValueN, options)` searches for data dictionary entries using additional search options. For example, you can match the search criteria with the values of the entries in the target section.

Input Arguments

sectionObj — Data dictionary section to search

Simulink.data.dictionary.Section object

Data dictionary section to search, specified as a `Simulink.data.dictionary.Section` object. Before you use this function, represent the target section with a

Simulink.data.dictionary.Section object by using, for example, the getSection function.

PName1,PValue1,...,PNameN,PValueN — Search criteria

name-value pairs representing properties

Search criteria, specified as one or more name-value pairs representing names and values of properties of the entries in the target data dictionary section. For a list of the properties of a data dictionary entry, see Simulink.data.dictionary.Entry. If you specify more than one name-value pair, the returned entries meet all of the criteria.

If you include the '-value' option to search in the values of the entries, the search criteria apply to the values of the entries rather than to the entries themselves.

Example: 'LastModifiedBy','jsmith'

Example: 'DataSource','myRefDictionary_ex_API.slidd'

options — Additional search options

supported option codes

Additional search options, specified as one or more of the following supported option codes.

'-value'	This option causes find to search only in the values of the entries in the target data dictionary section. Specify this option before any other search criteria or options arguments.
'-and','-or','-xor','-not' logical operators	These options modify or combine multiple search criteria or other option codes.
'-property',propertyName	This name-value pair causes find to search for entries or values that have the property propertyName regardless of the value of the property. Specify propertyName as a character vector.
'-class',className	This name-value pair causes find to search for entries or values that are objects of the class className. Specify className as a character vector.

'-isa', className	This name-value pair causes <code>find</code> to search for entries or values that are objects of the class or of any subclass derived from the class <code>className</code> . Specify <code>className</code> as a character vector.
'-regexp'	This option allows you to use regular expressions in your search criteria. This option affects only search criteria that follow '-regexp'.

Example: '-value'

Example: '-value', '-property', 'CoderInfo'

Example: '-value', '-class', 'Simulink.Parameter'

Output Arguments

foundEntries — Data dictionary entries matching search criteria

array of `Simulink.data.dictionary.Entry` objects

Data dictionary entries matching the specified search criteria, returned as an array of `Simulink.data.dictionary.Entry` objects.

Examples

Return Array of All Entries in Data Dictionary Section

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Return all of the entries stored in the Design Data section of the data dictionary `myDictionary_ex_API.sldd`.

```
allEntries = find(dDataSectObj)
```

Search Data Dictionary Section for Specific Class

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Search in the Design Data section of `myDictionary_ex_API.sldd` for entries whose values are objects of the `Simulink.Parameter` class.

```
foundEntries = find(dDataSectObj, '-value', '-class', 'Simulink.Parameter')
```

Search Data Dictionary Section for Modifying User

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Search in the Design Data section of `myDictionary_ex_API.sldd` for entries that were last modified by the user `jsmith`.

```
foundEntries = find(dDataSectObj, 'LastModifiedBy', 'jsmith')
```

Search Data Dictionary Section Using Multiple Criteria

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Search in the Design Data section of `myDictionary_ex_API.sldd` for entries that were last modified by the user `jsmith` or whose names begin with `fuel`.

```
foundEntries = find(dDataSectObj, 'LastModifiedBy', 'jsmith', '-or', ...  
    '-regexp', 'Name', 'fuel*')
```

Search Data Dictionary Section Using Regular Expressions

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Search in the Design Data section of `myDictionary_ex_API.sldd` for entries whose names begin with `fuel`.

```
foundEntries = find(dDataSectObj, '-regexp', 'Name', 'fuel*')
```

Search Data Dictionary Section for Specific Value

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Return all of the entries stored in the Design Data section of the data dictionary `myDictionary_ex_API.sldd`.

```
allEntries = find(dDataSectObj);
```

Find the entries with value 237. If you find more than one entry, store the entries in an array called `foundEntries`.

```
foundEntries = [];  
for i = 1:length(allEntries)  
    if getValue(allEntries(i)) == 237  
        foundEntries = [foundEntries allEntries(i)];  
    end  
end
```

Search Data Dictionary Section for Specific Property

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Search in the Design Data section of `myDictionary_ex_API.sldd` for entries whose values have a property `DataType`.

```
foundEntries = find(dDataSectObj, '-value', '-property', 'DataType')
```

- “Store Data in Dictionary Programmatically”

Alternatives

You can use Model Explorer to search a data dictionary for entries using arbitrary criteria.

See Also

`Simulink.data.dictionary.Entry` | `Simulink.data.dictionary.Section` | `exist` | `find`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

getEntry

Class: Simulink.data.dictionary.Section

Package: Simulink.data.dictionary

Create Simulink.data.dictionary.Entry object to represent data dictionary entry

Syntax

```
entryObj = getEntry(sectionObj,entryName)
entryObj = getEntry(sectionObj,
entryName,'DataSource',dictionaryName)
```

Description

`entryObj = getEntry(sectionObj,entryName)` returns an array of Simulink.data.dictionary.Entry objects representing data dictionary entries `entryName` found in the data dictionary section `sectionObj`, a Simulinkdata.dictionary.Section object. `getEntry` returns multiple objects if multiple entries have the specified name in a reference hierarchy of data dictionaries.

`entryObj = getEntry(sectionObj,entryName,'DataSource',dictionaryName)` returns an object representing a data dictionary entry that is defined in the data dictionary `dictionaryName`. Use this syntax to uniquely identify an entry that is defined more than once in a hierarchy of referenced data dictionaries.

Input Arguments

sectionObj — Target data dictionary section

Simulink.data.dictionary.Section object

Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a

Simulink.data.dictionary.Section object by using, for example, the getSection function.

entryName — Name of target data dictionary entry

character vector

Name of target data dictionary entry, specified as a character vector.

Example: 'myEntry'

Data Types: char

dictionaryName — Name of data dictionary containing target entry

character vector

File name of data dictionary containing the target entry, specified as a character vector including the .sldd extension.

Example: 'mySubDictionary_ex_API.sldd'

Data Types: char

Output Arguments

entryObj — Target data dictionary entry

Simulink.data.dictionary.Entry object

Target data dictionary entry, returned as one or more Simulink.data.dictionary.Entry objects.

Examples

Set Value of Data Dictionary Entry

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```
fuelFlowObj = getEntry(dDataSectObj, 'fuelFlow');
```

Set the value of the entry `fuelFlow` to 493.

```
setValue(fuelFlowObj, 493)
```

Set Value of Entry in Reference Dictionary

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Represent the data dictionary entry `myRefEntry` with a `Simulink.data.dictionary.Entry` object named `refEntryObj`. `myDictionary_ex_API.sldd` references `myRefDictionary_ex_API.sldd`, and `myRefDictionary_ex_API.sldd` defines an entry `myRefEntry`.

```
refEntryObj = getEntry(dDataSectObj, 'myRefEntry', 'DataSource', ...  
    'myRefDictionary_ex_API.sldd');
```

Set the value of the entry `myRefEntry` to 493.

```
setValue(refEntryObj, 493)
```

- “Store Data in Dictionary Programmatically”

See Also

`Simulink.data.dictionary.Entry` | `Simulink.data.dictionary.Section` | `addEntry` | `getValue`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

importFromFile

Class: Simulink.data.dictionary.Section

Package: Simulink.data.dictionary

Import variables from MAT-file or MATLAB file to data dictionary section

Syntax

```
importedVars = importFromFile(sectionObj, fileName)
importedVars = importFromFile(sectionObj,
    fileName, 'existingVarsAction', existAction)
[importedVars, existingVars] = importFromFile( ___ )
```

Description

`importedVars = importFromFile(sectionObj, fileName)` imports variables defined in the MAT-file or MATLAB file `fileName` to the data dictionary section `sectionObj` without overwriting any variables that are already in the target section. If any variables are already in the target section, the function displays a warning and a list in the MATLAB Command Window. This syntax returns a list of variables that were successfully imported. A variable is considered successfully imported only if `importFromFile` assigns the value of the variable to the corresponding entry in the target data dictionary.

`importedVars = importFromFile(sectionObj, fileName, 'existingVarsAction', existAction)` imports variables that are already in the target section by taking a specified action `existAction`. For example, you can choose to use the variable values in the target file to overwrite the corresponding values in the target section.

`[importedVars, existingVars] = importFromFile(___)` returns a list of variables in the target section that were not overwritten. Use this syntax if `existingVarsAction` is set to 'none', the default value, which prevents existing dictionary entries from being overwritten.

Input Arguments

sectionObj — Target data dictionary section

`Simulink.data.dictionary.Section` object

Target data dictionary section, specified as a `Simulink.data.dictionary.Section` object. Before you use this function, represent the target section with a `Simulink.data.dictionary.Section` object by using, for example, the `getSection` function.

fileName — Name of MAT or MATLAB file

character vector

Name of target MAT or MATLAB file, specified as a character vector. `importFromFile` automatically supplies a file extension `.mat` if you do not specify an extension.

Example: `'myFile.mat'`

Example: `'myFile.m'`

Data Types: `char`

existAction — Action to take for existing dictionary variables

`'none'` (default) | `'overwrite'` | `'error'`

Action to take for existing dictionary variables, specified as `'none'`, `'overwrite'`, or `'error'`.

If you specify `'none'`, `importFromFile` attempts to import target variables but does not import or make any changes to variables that are already in the data dictionary section.

If you specify `'overwrite'`, `importFromFile` imports all target variables and overwrites any variables that are already in the data dictionary section.

If you specify `'error'`, `importFromFile` returns an error, without importing any variables, if any target variables are already in the data dictionary section.

Example: `'overwrite'`

Data Types: `char`

Output Arguments

importedVars — Successfully imported variables

cell array of character vectors

Names of successfully imported variables, returned as a cell array of character vectors. A variable is considered successfully imported only if `importFromFile` assigns its value to the corresponding entry in the target data dictionary.

existingVars — Variables that were not imported

cell array of character vectors

Names of target variables that were not imported due to their existence in the target data dictionary, returned as a cell array of character vectors. `existingVars` has content only if `existAction` is set to 'none', which is also the default. In that case `importFromFile` imports only variables that are not already in the target data dictionary.

Examples

Import to Data Dictionary from File

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');  
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Import all variables contained in the file `myData_ex_API.m` to the data dictionary and return a list of successfully imported variables. If any variables are already in `myDictionary_ex_API.sldd`, `importFromFile` returns a warning and a list of the affected variables.

```
importFromFile(dDataSectObj, 'myData_ex_API.m')
```

```
Warning: The following variables were not imported because  
they already exist in the dictionary:
```

```
    fuelFlow
```

```
ans =

    'myFirstEntry'
    'mySecondEntry'
    'myThirdEntry'
```

Import Variables from File and Overwrite Conflicts

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Import all variables contained in the file `myData_ex_API.m` to the data dictionary, overwrite any variables that are already in the dictionary, and return a list of successfully imported variables.

```
importFromFile(dDataSectObj, 'myData_ex_API.m', 'existingVarsAction', 'overwrite')
```

```
ans =

    'fuelFlow'
    'myFirstEntry'
    'mySecondEntry'
    'myThirdEntry'
```

Return Variables Not Imported to Data Dictionary from File

Return a list of variables that are not imported from a file because they are already in the target data dictionary

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
```

Import all variables contained in the file `myData_ex_API.m` to the data dictionary. Specify names for the output arguments of `importFromFile` to return the names of successfully and unsuccessfully imported variables.

```
[importedVars,existingVars] = importFromFile(dDataSectObj, 'myData_ex_API.m')  
  
importedVars =  
  
    'myFirstEntry'  
    'mySecondEntry'  
    'myThirdEntry'  
  
  
existingVars =  
  
    'fuelFlow'
```

`importFromFile` does not import the variable `fuelFlow` because it is already in the target data dictionary.

- “Store Data in Dictionary Programmatically”

Tips

- `importFromFile` can import MATLAB variables created from enumerated data types but cannot import the definitions of the enumerated types. Use the `importEnumTypes` function to import enumerated data type definitions to a data dictionary. If you import variables of enumerated data types to a data dictionary but do not import the enumerated type definitions, the dictionary is less portable and might not function properly if used by someone else.

Alternatives

You can use the Model Explorer to import variables to a data dictionary from a file. See “Import Data to Dictionary from File” for more information.

See Also

`Simulink.data.dictionary.Section` | `exportToFile` | `importEnumTypes`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a

Simulink.DualScaledParameter

Specify name, value, units, and other properties of Simulink dual-scaled parameter

Description

This class extends the `Simulink.Parameter` class so that you can define an object that stores two scaled values of the same physical value. For example, for temperature measurement, you can store a Fahrenheit scale and a Celsius scale with conversion defined by a computation method that you provide. Given one scaled value, the `Simulink.DualScaledParameter` computes the other scaled value using the computation method.

A dual-scaled parameter has:

- A calibration value. The value that you prefer to use.
- A main value. The real-world value that Simulink uses.
- An internal stored integer value. The value that is used in the embedded code.

You can use `Simulink.DualScaledParameter` objects in your model for both simulation and code generation. The parameter computes the internal value before code generation via the computation method. This offline computation results in leaner generated code.

If you provide the calibration value, the parameter computes the main value using the computation method. This method can be a first-order rational function.

$$y = \frac{ax + b}{cx + d}$$

- x is the calibration value.
- y is the main value.
- a and b are the coefficients of the `CalToMain` compute numerator.
- c and d are the coefficients of the `CalToMain` compute denominator.

If you provide the calibration minimum and maximum values, the parameter computes minimum and maximum values of the main value. Simulink performs range checking of

parameter values. The software alerts you when the parameter object value lies outside a range that corresponds to its specified minimum and maximum values and data type.

You can use the `Simulink.DualScaledParameter` dialog box to define a `Simulink.DualScaledParameter` object. To open the dialog box:

- 1 In the Model Explorer, select the base workspace or a model workspace and select **Add > Add Custom**.
- 2 In the Model Explorer — Select Object dialog box, set **Object class** to `Simulink.DualScaledParameter`.

Property Dialog Box

Main Attributes Tab

The image shows a screenshot of the Simulink Property Dialog Box for a parameter named 'param'. The dialog box has a title bar with the text 'Simulink.DualScaledParameter: param' and a close button. It features two tabs: 'Calibration Attributes' and 'Main Attributes', with the latter being the active tab. The 'Main Attributes' section contains several input fields and a dropdown menu:

- Value:** A text field containing '[]'.
- Data type:** A dropdown menu set to 'auto' with a '>>' button to its right.
- Dimensions:** A text field containing '[0 0]'.
- Complexity:** A text field containing 'real'.
- Minimum:** A text field containing '[]'.
- Maximum:** A text field containing '[]'.
- Unit:** An empty text field.
- Code generation options:** A section containing a dropdown menu for 'Storage class' set to 'Auto'.
- Description:** A large empty text area.

At the bottom of the dialog box, there are four buttons: 'OK', 'Cancel', 'Help', and 'Apply'.

This tab shows the properties inherited from the `Simulink.Parameter` class. For more information, see `Simulink.Parameter`.

Calibration Attributes Tab

The image shows a dialog box titled "Simulink.DualScaledParameter: param" with a close button (X) in the top right corner. The dialog has two tabs: "Calibration Attributes" (selected) and "Main Attributes".

Under the "Calibration Attributes" tab, there are several input fields:

- Calibration value: []
- Calibration minimum: [] Calibration maximum: []
- CalToMain compute numerator: []
- CalToMain compute denominator: []
- Calibration name: ""
- Calibration units: ""

Below these fields is a "Parameter validation" section with a "Is configuration valid:" label and a text input field containing the value "true".

At the bottom of the dialog is a "Diagnostic message:" label and a large, empty text area.

At the bottom right of the dialog are four buttons: "OK", "Cancel", "Help", and "Apply".

Calibration value

Calibration value of the parameter. The value that you prefer to use. The default value is [] (unspecified). Specify a finite, real, double value.

Before specifying **Calibration value**, you must specify **CalToMain compute numerator** and **CalToMain compute denominator** to define the computation method. The parameter uses the computation method and the calibration value to calculate the main value that Simulink uses.

Calibration minimum

Minimum value for the calibration parameter. The default value is [] (unspecified). Specify a finite, real, double scalar value.

Before specifying **Calibration minimum**, you must specify **CalToMain compute numerator** and **CalToMain compute denominator** to define the computation method. The parameter uses the computation method and the calibration minimum value to calculate the minimum or maximum value that Simulink uses. A first order rational function is strictly monotonic, either increasing or decreasing. If it is increasing, setting the calibration minimum sets the main minimum value. If it is decreasing, setting the calibration minimum sets the main maximum.

If the parameter value is less than the minimum value or if the minimum value is outside the range of the parameter data type, Simulink generates a warning. In these cases, when updating the diagram or starting a simulation, Simulink generates an error.

Calibration maximum

Maximum value for the calibration parameter. The default value is [] (unspecified). Specify a finite, real, double scalar value.

Before specifying **Calibration maximum**, you must specify **CalToMain compute numerator** and **CalToMain compute denominator** to define the computation method. The parameter uses the computation method and the calibration maximum value to calculate the corresponding maximum or minimum value that Simulink uses. A first order rational function is strictly monotonic, either increasing or decreasing. If it is increasing, setting the calibration maximum sets the main maximum value. If it is decreasing, setting the calibration maximum sets the main minimum.

If the parameter value is less than the minimum value or if the minimum value is outside the range of the parameter data type, Simulink generates a warning. In these

cases, when updating the diagram or starting a simulation, Simulink generates an error.

CalToMain compute numerator

Specify the numerator coefficients *a* and *b* of the first-order linear equation:

$$y = \frac{ax + b}{cx + d}$$

The default value is [] (unspecified). Specify finite, real, double scalar values for *a* and *b*. For example, [1 1] or, for reciprocal scaling, 1.

Once you have applied CalToMain compute numerator, you cannot change it.

CalToMain compute denominator

Specify the denominator coefficients *c* and *d* of the first-order linear equation:

$$y = \frac{ax + b}{cx + d}$$

The default value is [] (unspecified). Specify finite, real, double scalar values for *c* and *d*. For example, [1 1].

Once you have applied CalToMain compute denominator, you cannot change it.

Calibration name

Specify the name of the calibration parameter. The default value is ''. Specify a character vector value, for example, 'T1'.

Calibration units

Specify the measurement units for this calibration value. This field is intended for use in documenting this parameter. The default value is ''. Specify a character vector value, for example, 'Fahrenheit'.

Is configuration valid

Simulink indicates whether the configuration is valid. The default value is `true`. If Simulink detects an issue with the configuration, it sets this field to `false` and provides information in the **Diagnostic message** field. You cannot set this field.

Diagnostic message

If you specify invalid parameter settings, Simulink displays a message in this field. Use the diagnostic information to help you fix an invalid configuration issue. You cannot set this field.

Properties

Name	Access	Description
CalibrationValue	RW	Calibration value of this parameter. (Calibration value)
CalibrationMin	RW	Calibration minimum value of this parameter. (Calibration minimum)
CalibrationMax	RW	Calibration maximum value of this parameter. (Calibration maximum)
CalToMainCompuNumerator	RW	Numerator coefficients of the computation method. (CalToMain compute numerator) Once you have applied <code>CalToMainCompuNumerator</code> , you cannot change it.
CalToMainCompuDenominator	RW	Denominator coefficients of the computation method. (CalToMain compute denominator) Once you have applied <code>CalToMainCompuDenominator</code> , you cannot change it.
CalibrationName	RW	Name of the calibration parameter. (Calibration name)
CalibrationDocUnits	RW	Measurement units for this calibration parameter's value. (Calibration units)
IsConfigurationValid	RO	Information about validity of configuration. (Is configuration valid)
DiagnosticMessage	RO	If the configuration is invalid, diagnostic information to help you fix the issue. (Diagnostic message)

Example

Create and Update a Dual-Scaled Parameter

Create a `Simulink.DualScaledParameter` object that stores a temperature as both Fahrenheit and Celsius.

Create a `Simulink.DualScaledParameter` object.

```
Temp = Simulink.DualScaledParameter;
```

Set the computation method that converts between Fahrenheit and Celsius.

```
Temp.CalToMainCompuNumerator = [1 -32];  
Temp.CalToMainCompuDenominator = [1.8];
```

Set the value of the temperature that you want to see in Fahrenheit.

```
Temp.CalibrationValue = 212
```

```
Temp =
```

```
DualScaledParameter with properties:
```

```
    CalibrationValue: 212  
    CalibrationMin: []  
    CalibrationMax: []  
    CalToMainCompuNumerator: [1 -32]  
    CalToMainCompuDenominator: 1.8000  
    CalibrationName: ''  
    CalibrationDocUnits: ''  
    IsConfigurationValid: 1  
    DiagnosticMessage: ''  
    Value: 100  
    CoderInfo: [1x1 Simulink.CoderInfo]  
    Description: ''  
    DataType: 'auto'  
    Min: []  
    Max: []  
    Unit: ''  
    Complexity: 'real'  
    Dimensions: [1 1]
```

The `Simulink.DualScaledParameter` calculates `Temp.Value` which is the value that Simulink uses. `Temp.CalibrationValue` is 212 (degrees Fahrenheit), so `Temp.Value` is 100 (degrees Celsius).

Name the value and specify the units.

```
Temp.CalibrationName = 'TempF';  
Temp.CalibrationDocUnits = 'Fahrenheit';
```

Set calibration minimum and maximum values.

```
Temp.CalibrationMin = 0;  
Temp.CalibrationMax = 300;
```

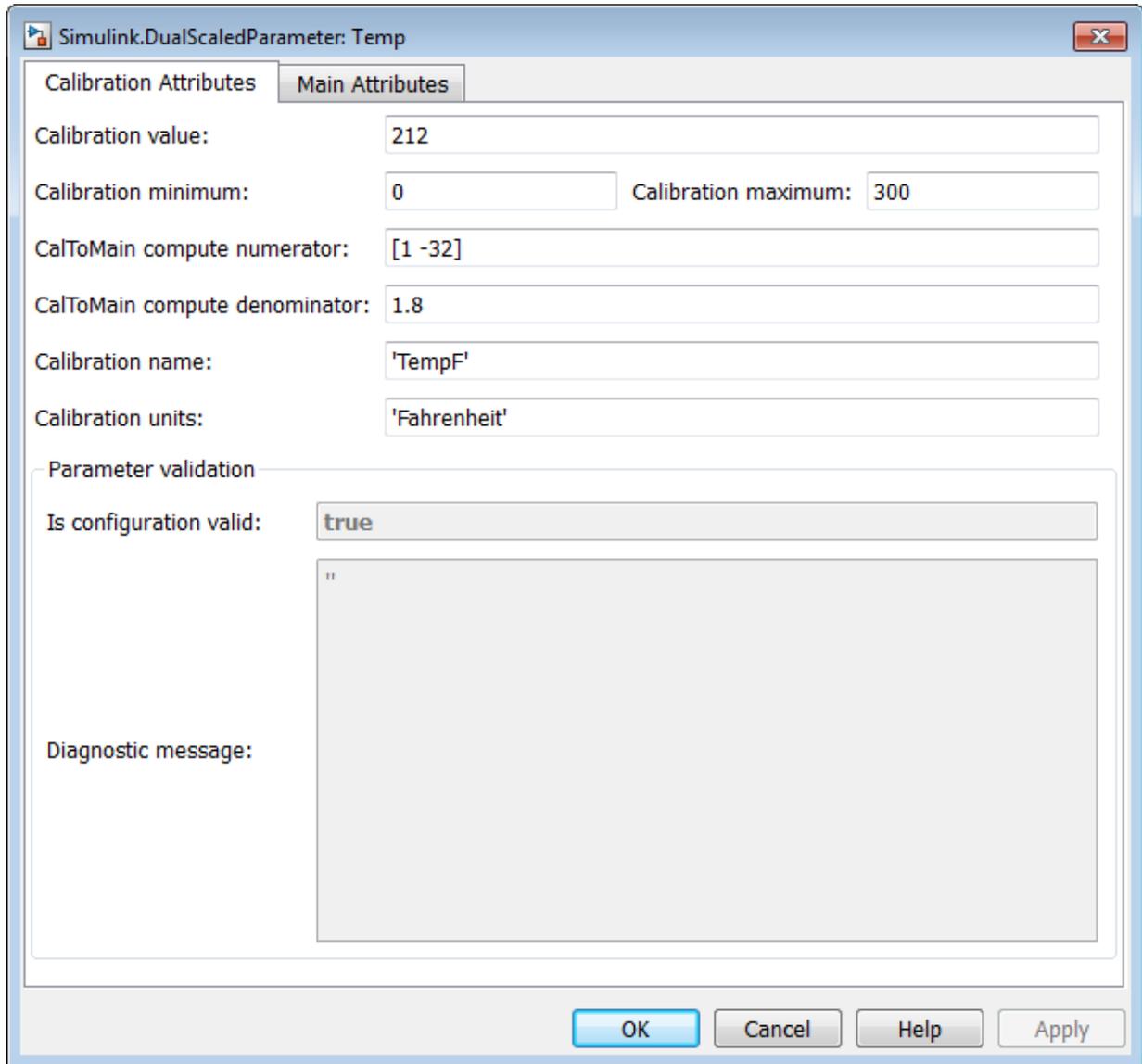
If you specify a calibration value outside this allowable range, Simulink generates a warning.

Specify the units that Simulink uses.

```
Temp.Unit = 'degC';
```

Open the `Simulink.DualScaledParameter` dialog box.

```
open Temp
```



The **Calibration Attributes** tab displays the calibration value and the computation method that you specified.

In the dialog box, click the **Main Attributes** tab.

The image shows a dialog box titled "Simulink.DualScaledParameter: Temp" with two tabs: "Calibration Attributes" and "Main Attributes". The "Main Attributes" tab is selected. The dialog contains the following fields and controls:

- Value:** A text input field containing the number "100".
- Data type:** A dropdown menu set to "auto" with a ">>" button to its right.
- Dimensions:** A text input field containing "[1 1]".
- Complexity:** A dropdown menu set to "real".
- Minimum:** A text input field containing "-17.7778".
- Maximum:** A text input field containing "148.889".
- Unit:** A text input field containing "degC".
- Code generation options:** A section header above a dropdown menu.
- Storage class:** A dropdown menu set to "Auto".
- Description:** A large empty text area.

At the bottom of the dialog are four buttons: "OK", "Cancel", "Help", and "Apply".

This tab displays information about the value used by Simulink.

See Also

`AUTOSAR.DualScaledParameter` | `Simulink.Parameter`

Topics

“Configure AUTOSAR Data for Measurement and Calibration” (Embedded Coder)
“Fixed Point”

Introduced in R2013b

Simulink.Mask class

Package: Simulink

Control masks programmatically

Description

Use an instance of `Simulink.Mask` class to perform the following operations:

- Create, copy, and delete masks.
- Create, edit, and delete mask parameters.
- Determine the block that owns the mask.
- Obtain workspace variables defined for a mask.

Properties

Type

Specifies the mask type of the associated block.

Type: character vector

Default: Empty character vector

Description

Specifies the block description of the associated block.

Type: character vector

Default: Empty character vector

Help

Specifies the help text that is displayed for the mask.

Type: character vector

Default: Empty character vector

Initialization

Specifies the initialization commands for the associated block.

Type: character vector

Default: Empty character vector

SelfModifiable

Indicates that the block can modify itself and its content.

Type: boolean

Values: 'on' | 'off'

Default: 'off'

Display

Specifies MATLAB code for drawing the block icon.

Type: character vector

Default: Empty character vector

IconFrame

Sets the visibility of the block frame. (Visible is on, Invisible is off).

Type: boolean

Values: 'on' | 'off'

Default: 'on'

IconOpaque

Sets the transparency of the icon (Opaque is on, Transparent is off).

Type: boolean

Values: 'on' | 'off'

Default: 'on'

RunInitForIconRedraw

Specifies whether Simulink must run mask initialization before executing the mask icon commands.

Type: enum

Values: 'auto' | 'on' | 'off'

Default: 'auto'

IconRotate

Sets icon to rotate with the block.

Type: enum

Values: 'none' | 'port'

Default: 'none'

PortRotate

Specifies the port rotation policy for the masked block.

Type: enum

Values: 'default' | 'physical'

Default: 'default'

IconUnits

Specifies the units for the drawing commands.

Type: enum

Values: 'pixel' | 'autoscale' | 'normalized'

Default: 'autoscale'

Methods

addParameter	Add a parameter to a mask
copy	Copy a mask from one block to another
create	Create a mask on a Simulink block
delete	Unmask a block and delete the mask from memory
get	Get a block mask as a mask object
addDialogControl	Add dialog control elements to mask dialog box
getDialogControl	Search for a specific dialog control on the mask
getOwner	Determine the block that owns a mask
getParameter	Get a mask parameter using its name
getWorkspaceVariables	Get all the variables defined in the mask workspace for a masked block
numParameters	Determine the number of parameters in a mask
removeDialogControl	Remove dialog control element from mask dialog box
removeParameter	Remove parameter from mask dialog box
removeAllParameters	Remove all existing parameters from a mask
set	Set the properties of an existing mask

See Also

Topics

“Control Masks Programmatically”

“Block Masks”

addParameter

Class: Simulink.Mask

Package: Simulink

Add a parameter to a mask

Syntax

```
p = Simulink.Mask.get(blockName)
p.addParameter(Name, Value)
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.addParameter(Name, Value)` appends a parameter to the mask. If you do not specify name–value pairs as arguments with this command, Simulink creates an unnamed mask parameter with control type set to `edit`.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Type

Type of control that is used to specify the value of this parameter.

Default: edit

TypeOptions

The options that are displayed within a popup control or in a promoted parameter. This field is a cell array.

Default: empty

Name

The name of the mask parameter. This name is assigned to the mask workspace variable created for this parameter.

Default: empty

Prompt

Text that identifies the parameter on the Mask Parameters dialog box.

Default: empty

Value

The default value of the mask parameter in the Mask Parameters dialog box.

Default: Type specific; depends on the `Type` of the parameter

Evaluate

Option to specify whether parameter must be evaluated.

Default: 'on'

Tunable

Option to specify whether parameter is tunable.

Default: 'on'

Enabled

Option to specify whether user can set parameter value.

Default: 'on'

Visible

Option to set whether mask parameter is hidden or visible to the user.

Default: 'on'

Callback

Container for MATLAB code that executes when user makes a change in the Mask Parameters dialog box and clicks **Apply**.

Default: empty

TabName

The name of the tab in the Mask Parameters dialog box where the parameter appears.

Default: empty

Container

Option to specifies a container for the child dialog control. The permitted values are 'panel', 'group', and 'tab'.

Examples

- 1 Get mask as an object using a masked block's path.

```
p = Simulink.Mask.get('myModel/Subsystem');
```

- 2 Add a parameter to the mask without specifying name–value pairs for parameter attributes.

```
p.addParameter;
```

- 3 Add a mask parameter of type popup that cannot be evaluated and appears on a tab named **Properties** in the Mask Parameters dialog box.

```
p.addParameter('Type','popup','TypeOptions',...  
{'Red' 'Blue' 'Green'},'Evaluate','off','TabName','Properties');
```

See Also

“Block Masks” | `Simulink.Mask`

Simulink.Mask.copy

Class: Simulink.Mask

Package: Simulink

Copy a mask from one block to another

Syntax

```
pSource = Simulink.Mask.get(srcBlockName)
pDest = Simulink.Mask.create(destBlockName)
pDest.copy(pSource)
```

Description

`pSource = Simulink.Mask.get(srcBlockName)` gets the mask on the source block specified by `blockName` as a mask object.

`pDest = Simulink.Mask.create(destBlockName)` creates an empty mask on the destination block specified by `destBlockName`.

`pDest.copy(pSource)` overwrites the destination mask with the source mask.

Input Arguments

srcBlockName

The handle to the source block or the path to the source block inside the model.

Note The source block should be masked.

destBlockName

The handle to the destination block or the path to the destination block inside the model.

Note The destination block should have an empty mask. Otherwise, the copied mask will overwrite the non-empty mask.

Examples

- 1 Create an empty mask on the destination block using the block's path.

```
pDest = Simulink.Mask.create('myModel/Subsystem');
```

- 2 Get source mask as an object using the source block's path.

```
pSource = Simulink.Mask.get('myModel/Abs');
```

- 3 Make the destination mask a copy of the source mask.

```
pDest.copy(pSource);
```

See Also

“Block Masks” | `Simulink.Mask`

Simulink.Mask.create

Class: Simulink.Mask

Package: Simulink

Create a mask on a Simulink block

Syntax

```
p = Simulink.Mask.create(blockName)
```

Description

`p = Simulink.Mask.create(blockName)` creates an empty mask on the block specified by `blockName`. If the specified block is already masked, an error message appears.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Examples

- 1 Create a mask using a block's handle.

Note In the model, select the block to be masked.

```
p = Simulink.Mask.create(gcbh);
```

- 2 Create a mask using the block's path.

```
p = Simulink.Mask.create('myModel/Subsystem');
```

See Also

“Block Masks” | `Simulink.Mask`

delete

Class: Simulink.Mask

Package: Simulink

Unmask a block and delete the mask from memory

Syntax

```
p = Simulink.Mask.get(blockName)
p.delete
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.delete` unmasks the block and deletes the mask from memory.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Examples

- 1 Get mask as an object using a masked block's path.

```
p = Simulink.Mask.get('myModel/Subsystem');
```

- 2 Unmask the block using the mask object and delete the mask from memory.

```
p.delete;
```

See Also

“Block Masks” | `Simulink.Mask`

Simulink.Mask.get

Class: Simulink.Mask

Package: Simulink

Get a block mask as a mask object

Syntax

```
p = Simulink.Mask.get(blockName)
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object. If the specified block is not masked, a null value returns.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Examples

- 1 Get mask as an object using a masked block's handle.

Note In the model, select the masked block.

```
p = Simulink.Mask.get(gcbh);
```

- 2 Get mask as an object using a masked block's path.

```
p = Simulink.Mask.get('myModel/Subsystem');
```

See Also

“Block Masks” | `Simulink.Mask`

addDialogControl

Class: Simulink.Mask

Package: Simulink

Add dialog control elements to mask dialog box

Syntax

```
successIndicator = maskObj.addDialogControl(controlType,  
controlIdentifier)  
successIndicator = maskObj.addDialogControl(Name,Value)
```

Description

`successIndicator = maskObj.addDialogControl(controlType, controlIdentifier)` adds dialog control elements like text, hyperlinks, or tabs to mask dialog box. First get the mask object and assign it to the variable `maskObj`

`successIndicator = maskObj.addDialogControl(Name,Value)` specifies the Name and Value arguments for an element on the mask dialog box. You can specify multiple Name-Value pairs.

Input Arguments

controlType — Value type of dialog control element

character vector

Type of dialog control element, specified

- 'panel'
- 'group'
- 'tabcontainer'
- 'tab'

- 'collapsiblepanel'
- 'text'
- 'image'
- 'hyperlink'
- 'pushbutton'

controlIdentifier — Unique identifier for the element
character vector

Specifies the programmatic identifier for the element of mask dialog box. Use a name that is unique and does not have space between words. For more information, see “Variable Names” (MATLAB).

Name-Value Pair Arguments

Specify optional comma-separated pairs of *Name*, *Value* arguments. *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes (' ') and is case-sensitive. You can specify several name and value pair arguments in any order as *Name1*, *Value1*, . . . , *NameN*, *ValueN*.

Type

Type of control that is used to specify the value of this dialog control element. *Type* is a required argument. The permitted values are 'panel', 'group', 'tabcontainer', 'tab', 'collapsiblepanel', 'text', 'image', 'hyperlink', and 'pushbutton'.

Default: empty

Name

The identifier of the dialog control element. *Name* is a required argument. This field is available for all dialog control types.

Default: empty

Prompt

Text that is displayed in the dialog control element on the Mask dialog box. This field is available for all except for panel and image dialog control types.

Default: empty

Tooltip

Tooltip for the dialog control element.

Default: empty

Enabled

Option to specify whether you can set value for the dialog control element. This field is available for all dialog control types.

Default: 'on'

Visible

Option to set whether the dialog control element is hidden or visible to the user. This field is available for all dialog control types.

Default: 'on'

Callback

Container for MATLAB code that executes when you edit the dialog control element and click **Apply**. This field is available only for the hyperlink and pushbutton dialog control types.

Default: empty

Row

Option to set whether the dialog control is placed in the new row or the same row. This field is available for all dialog control types.

Default: empty

Filepath

Contains the path to an image file. This field is available for image, and pushbutton dialog control types.

Default: empty

Container

Option to specifies a container for the child dialog control. The permitted values are the names of 'panel', 'group', and 'tab' dialog controls.

Examples

Add Dialog Control Elements to Mask Dialog Box

Get mask object and add dialog control element to it.

```
% Get mask object on model Engine
maskObj = Simulink.Mask.get('Engine/Gain');

% Add hyperlink to mask dialog box

maskLink = maskObj.addDialogControl('hyperlink','link');
maskLink.Prompt = 'Mathworks Home Page';
maskLink.Callback = 'web(''www.mathworks.com'')'

% Alternative method to add hyperlink

maskLink = maskObj.addDialogControl('hyperlink','link');
maskLink.Prompt = 'www.mathworks.com';

% Add text to mask dialog box

maskText = maskObj.addDialogControl('text','text_tag');
maskText.Prompt = 'Enable range checking';

% Add button to mask dialog box

maskButton = maskObj.addDialogControl('pushbutton','button_tag');
maskButton.Prompt = 'Compute';
```

Add Dialog Control Elements to Mask Dialog Box Tabs

Create tabs on the mask dialog box and add elements to these tabs.

```

% Get mask object on a block named 'GainBlock'

maskObj = Simulink.Mask.get('GainBlock/Gain');

% Create a tab container

maskObj.addDialogControl('tabcontainer','allTabs');
tabs = maskObj.getDialogControl('allTabs');

% Create tabs and name them

maskTab1 = tabs.addDialogControl('tab','First');
maskTab1.Prompt = 'First tab';

maskTab2 = tabs.addDialogControl('tab','Second');
maskTab2.Prompt = 'Second tab';

% Add elements to one of the tabs

firstTab = tabs.getDialogControl('First');
firstTab.addDialogControl('text','textOnFirst');
firstTab.getDialogControl('textOnFirst').Prompt = 'Tab one';

```

Add Dialog Control Element Using Name-Value Pair

Add dialog control element and specify values for it

```

% Get mask object on model Engine

maskObj = Simulink.Mask.get('Engine/Gain');

% Add a dialog box and specify values for it

maskDialog = maskObj.addDialogControl('Type','text',...
'Prompt','hello','Visible','off');

```

See Also

“Block Masks” | Control Masks Programmatically | Simulink.Mask

Introduced in R2014a

getDialogControl

Class: Simulink.Mask

Package: Simulink

Search for a specific dialog control on the mask

Syntax

```
[control, phandle] = handle.getDialogControl(cname)
```

Description

[control, phandle] = handle.getDialogControl(cname) , search for a specific child dialog control recursively on the mask dialog.

Input Arguments

cname

Name of the dialog control being searched on the mask dialog.

Default:

Output Arguments

control

Target dialog control being searched on the mask dialog.

phandle

Parent of the dialog control being searched mask dialog.

Examples

Find a dialog control

Find a text dialog control on the mask dialog box. `maskObj` is the handle to the mask object. The `getDialogControl` method returns the handle to the dialog control (`hdlgctrl`) and handle to the parent dialog control (`phandle`).

```
[hdlgctrl, phandle] = maskObj.getDialogControl('txt_var')
```

See Also

“Block Masks” | `Simulink.Mask`

getOwner

Class: Simulink.Mask

Package: Simulink

Determine the block that owns a mask

Syntax

```
p = Simulink.Mask.get(blockName)
p.getOwner
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.getOwner` returns the interface to the block that owns the mask.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Examples

- 1 Get mask as an object using a masked block's path.

```
p = Simulink.Mask.get('myModel/Subsystem');
```

- 2 Get the interface to the block that owns the mask.

```
p.getOwner;
```

See Also

“Block Masks” | `Simulink.Mask`

getParameter

Class: Simulink.Mask

Package: Simulink

Get a mask parameter using its name

Syntax

```
p = Simulink.Mask.get(blockName)
param = p.getParameter(paramName)
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`param = p.getParameter(paramName)` returns an array of mask parameters.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

paramName

The name of the parameter you want to get.

Examples

- 1 Get mask as an object using a masked block's path.

```
p = Simulink.Mask.get('myModel/Subsystem');
```

- 2 Get a mask parameter by using its name.

```
param = p.getParameter('intercept');
```

See Also

“Block Masks” | `Simulink.Mask`

getWorkspaceVariables

Class: Simulink.Mask

Package: Simulink

Get all the variables defined in the mask workspace for a masked block

Syntax

```
p = Simulink.Mask.get(blockName)
vars = p.getWorkspaceVariables
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`vars = p.getWorkspaceVariables` returns as a structure all the variables defined in the mask workspace for the masked block.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Examples

- 1 Get mask as an object using a masked block's path.

```
p = Simulink.Mask.get('myModel/Subsystem');
```

- 2 Get all the variables defined in the mask workspace for the masked block.

```
vars = p.getWorkspaceVariables;
```

See Also

“Block Masks” | `Simulink.Mask`

numParameters

Class: Simulink.Mask

Package: Simulink

Determine the number of parameters in a mask

Syntax

```
p = Simulink.Mask.get(blockName)
p.numParameters
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.numParameters` returns the number of parameters in the mask.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Examples

- 1 Get mask as an object using a masked block's path.

```
p = Simulink.Mask.get('myModel/Subsystem');
```

- 2 Get the number of parameters in the mask.

```
p.numParameters;
```

See Also

“Block Masks” | `Simulink.Mask`

removeDialogControl

Class: Simulink.Mask

Package: Simulink

Remove dialog control element from mask dialog box

Syntax

```
successIndicator = maskVariable.removeDialogControl(  
controlIdentifier)
```

Description

`successIndicator = maskVariable.removeDialogControl(controlIdentifier)` removes dialog control element, specified by `controlIdentifier`, like text, hyperlinks, or tabs from a mask dialog box. First get the mask object and assign it to the variable `maskVariable`.

Successful removal of a dialog control element returns a Boolean value of 1.

Input Arguments

controlIdentifier — Unique identifier for the element

character vector

Programmatic identifier for the dialog control element of mask dialog box, specified as a character vector.

Examples

Remove Dialog Control Element from Mask Dialog Box

```
% Get mask object on the Gain block in the model Engine.  
maskObj = Simulink.Mask.get('Engine/Gain');  
  
% Remove element named AllTab from mask dialog box.  
  
p = maskObj.removeDialogControl('AllTab');
```

See Also

“Block Masks” | Simulink.Mask

Introduced in R2013b

removeParameter

Class: Simulink.Mask

Package: Simulink

Remove parameter from mask dialog box

Syntax

```
successIndicator = maskVariable.removeParameter(controlIdentifier)
```

Description

`successIndicator = maskVariable.removeParameter(controlIdentifier)` removes parameter, specified by `controlIdentifier`, like edit, check box, popup from an existing mask dialog box. First get the mask object and assign it to the variable `maskVariable`.

Successful removal of a parameter returns a Boolean value of 1.

Input Arguments

controlIdentifier — Unique identifier for the parameter

character vector

Programmatic identifier for the parameter of mask dialog box, specified as a character vector.

Examples

Remove Parameter from Mask Dialog Box

```
% Get mask object on the Gain block in the model Engine.
```

```
maskObj = Simulink.Mask.get('Engine/Gain');  
  
% Remove parameter named checkbox1 from mask dialog box.  
  
p = maskObj.removeParameter('checkbox1');
```

Note You can also use the index number as the `controlIdentifier`.

See Also

“Block Masks” | `Simulink.Mask`

Introduced in R2012b

removeAllParameters

Class: Simulink.Mask

Package: Simulink

Remove all existing parameters from a mask

Syntax

```
p = Simulink.Mask.get(blockName)
p.removeAllParameters
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.removeAllParameters` deletes all existing parameters from the mask.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Examples

- 1 Get mask as an object using a masked block's path.

```
p = Simulink.Mask.get('myModel/Subsystem');
```

- 2 Delete all existing parameters from the mask.

```
p.removeAllParameters;
```

See Also

“Block Masks” | `Simulink.Mask`

set

Class: Simulink.Mask

Package: Simulink

Set the properties of an existing mask

Syntax

```
p = Simulink.Mask.get(blockName)
p.set(Name, Value)
```

Description

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.set(Name, Value)` sets mask properties that you specify using name–value pairs as arguments.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Type

Text used as title for mask documentation that user sees on clicking **Help** in the Mask Parameters dialog box.

Default: empty

Description

Text used as summary for mask documentation that user sees on clicking **Help** in the Mask Parameters dialog box.

Default: empty

Help

Text used as body text for mask documentation that user sees on clicking **Help** in the Mask Parameters dialog box.

Default: empty

Initialization

MATLAB code that initializes the mask.

Default: empty

SelfModifiable

Option to set whether the mask can modify itself during simulation.

Default: 'off'

Display

MATLAB code that draws the mask icon.

Default: empty

IconFrame

Option to specify whether the mask icon appears inside a visible block frame.

Default: 'on'

MaskIconOpaque

Option to set the mask icon as opaque or transparent.

Default: 'opaque'

RunInitForIconRedraw

Option to specify whether Simulink should run mask initialization before executing the mask icon commands.

Default: 'off'

IconRotate

Option to specify icon rotation.

Default: 'none'

PortRotate

Option to specify port rotation.

Default: 'default'

IconUnits

Option to specify whether mask icon is autoscaled, normalized, or scaled in pixels.

Default: 'autoscale'

Examples

- 1 Get mask as an object using a masked block's path.

```
p = Simulink.Mask.get('myModel/Subsystem');
```

- 2 Modify the mask so that its mask icon is transparent and its documentation summarizes what it does.

```
p.set('IconOpaque','off','Type','Random number generator','Description',...  
'This block generates random numbers.');
```

See Also

“Block Masks” | `Simulink.Mask`

Simulink.MaskParameter class

Package: Simulink

Control mask parameters programmatically

Description

Use an instance of `Simulink.MaskParameter` to set the properties of mask parameters.

Properties

Type

Specifies the mask parameter type.

Type: character vector

Values:

'edit'|'checkbox'|'popup'|'min'|'max'|'promote'|'combobox'|'radiobutton'|'unidt'|'slider'|'dial'|'spinbox'

Default: 'edit'

TypeOptions

Specifies the option for the parameter if it exists, otherwise, it is empty. Applicable for parameters of type popup, radio, Datatypestr, and promote .

Type: cell array of character vectors

Default: { '' }

Name

Specifies the name of the mask parameter. This name is assigned to the mask workspace variable created for this parameter.

Type: character vector

Default: Empty character vector

Prompt

Specifies a character vector that appears as the label associated with the parameter on the mask dialog.

Type: character vector

Default: Empty character vector

Value

Specifies the value of the mask parameter.

Default: Depends on the type of the parameter.

Evaluate

Indicates if the parameter value is to be evaluated in MATLAB or treated as a character vector when the block is evaluated.

Type: boolean

Values: 'on' | 'off'

Default: 'on'

Tunable

Indicates if the parameter value can be changed during simulation.

Type: boolean

Values: 'on' | 'off'

Default: 'on'

NeverSave

Indicates if the parameter value gets saved in the model file.

Type: boolean

Values: 'on' | 'off'

Default: 'off'

Hidden

Indicates if the parameter should never show on the mask dialog box.

Type: boolean

Values: 'on' | 'off'

Default: 'off'

ReadOnly

Indicates if the parameter on the mask dialog box is editable or is read-only.

Type: boolean

Values: 'on' | 'off'

Default: 'off'

Enabled

Indicates if the parameter is enabled in the mask dialog box.

Type: boolean

Values: 'on' | 'off'

Default: 'on'

Visible

Indicates if the parameter is visible in the mask dialog box.

Type: boolean

Values: 'on' | 'off'

Default: 'on'

ShowTooltip

Indicates if tool tip is enabled for the mask parameter.

Type: boolean

Values: 'on' | 'off'

Default: 'on'

Callback

Specifies the MATLAB code that executes when a user changes the parameter value from the mask dialog box.

Type: character vector

Default: Empty character vector

TabName

Specifies the tab name of the mask dialog box where the parameter is displayed.

Type: character vector

Default: Empty character vector

Alias

Specifies the alternate name for mask parameter.

Type: character vector

Default: Empty character vector

Methods

set Set properties of mask parameters

See Also

Topics

“Control Masks Programmatically”

“Block Masks”

set

Class: Simulink.MaskParameter

Package: Simulink

Set properties of mask parameters

Syntax

```
Simulink.MaskParameter.set(Name,Value)
```

Description

`Simulink.MaskParameter.set(Name,Value)` sets the properties of a mask parameter.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Type

Type of control that is used to specify the value of this parameter.

Default: edit

TypeOptions

The options that are displayed within a popup control or in a promoted parameter. This field is a cell array.

Default: empty

Name

The name of the mask parameter. This name is assigned to the mask workspace variable created for this parameter.

Default: empty

Prompt

Text that identifies the parameter on the Mask Parameters dialog.

Default: empty

Value

The default value of the mask parameter in the Mask Parameters dialog.

Default: Type specific; depends on the Type of the parameter

Evaluate

Option to specify whether parameter must be evaluated.

Default: 'on'

Tunable

Option to specify whether parameter is tunable.

Default: 'on'

Enabled

Option to specify whether user can set parameter value.

Default: 'on'

Visible

Option to set whether mask parameter is hidden or visible to the user.

Default: 'on'

Callback

Container for MATLAB code that executes when user makes a change in the Mask Parameters dialog and clicks **Apply**.

Default: empty

TabName

The name of the tab in the Mask Parameters dialog where the parameter appears.

Default: empty

Examples

- 1 Get mask as an object using a masked block's path.

```
p = Simulink.Mask.get('myModel/Subsystem');
```

- 2 Get a mask parameter.

```
a = p.Parameters(1);
```

- 3 Edit mask parameter so it is of type popup, cannot be evaluated, and appears on a tab named **Properties** in the Mask Parameters dialog.

```
a.set('Type','popup','TypeOptions',{'Red' 'Blue' 'Green'},...  
'Evaluate','off','TabName','Properties');
```

See Also

“Block Masks” | Simulink.Mask | Simulink.MaskParameter

Simulink.dialog.Control class

Package: Simulink.dialog

Create instances of dialog control

Description

Use an instance of `Simulink.dialog.Control` class to create, delete, or search dialog controls.

Properties

Name

Uniquely identifies the dialog control element and is a required field.

Type: character vector

See Also

`Simulink.dialog.Button` | `Simulink.dialog.Hyperlink` |
`Simulink.dialog.Image` | `Simulink.dialog.Text` |
`Simulink.dialog.parameter.Control` | `Simulink.dialog.Container` | “Block
Masks”

Simulink.dialog.Container class

Package: Simulink.dialog

Create instances of a container dialog control

Description

Use an instance of `Simulink.dialog.Container` class to add container type dialog control.

Properties

Name

Uniquely identifies the container dialog control and is a required field.

Type: character vector

Enabled

Indicates whether container is active on the mask dialog box.

Type: boolean

Value: 'on' | 'off'

Default: 'on'

Visible

Indicates whether container is displayed on the mask dialog box.

Type: boolean

Value: 'on' | 'off'

Default: 'on'

DialogControls

Specifies the child dialog controls contained in the container.

Type: `Simulink.dialog.Control`

Default: Empty array

Methods

<code>addDialogControl</code>	Add dialog control elements to mask dialog box
<code>removeDialogControl</code>	Remove dialog control element from mask dialog box
<code>getDialogControl</code>	Search for a specific dialog control on the mask

See Also

`Simulink.dialog.Group` | `Simulink.dialog.Panel` | `Simulink.dialog.Tab` | `Simulink.dialog.TabContainer` | `Simulink.dialog.Control` | “Block Masks”

addDialogControl

Class: Simulink.dialog.Container

Package: Simulink.dialog

Add dialog control elements to mask dialog box

Syntax

```
successIndicator = maskObj.addDialogControl(controlType,  
controlIdentifier)  
successIndicator = maskObj.addDialogControl(Name,Value)
```

Description

`successIndicator = maskObj.addDialogControl(controlType, controlIdentifier)` adds dialog control elements like text, hyperlinks, or tabs to mask dialog box. First get the mask object and assign it to the variable `maskObj`

`successIndicator = maskObj.addDialogControl(Name,Value)` specifies the Name and Value arguments for an element on the mask dialog box. You can specify multiple Name-Value pairs.

Input Arguments

controlType — Value type of dialog control element

character vector

Type of dialog control element, specified

- 'panel'
- 'group'
- 'tabcontainer'
- 'tab'

- 'collapsiblepanel'
- 'text'
- 'image'
- 'hyperlink'
- 'pushbutton'

controlIdentifier — Unique identifier for the element
character vector

Specifies the programmatic identifier for the element of mask dialog box. Use a name that is unique and does not have space between words. For more information, see “Variable Names” (MATLAB).

Name-Value Pair Arguments

Specify optional comma-separated pairs of *Name, Value* arguments. *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes (' ') and is case-sensitive. You can specify several name and value pair arguments in any order as *Name1, Value1, ..., NameN, ValueN*.

Type

Type of control that is used to specify the value of this dialog control element. *Type* is a required argument. The permitted values are 'panel', 'group', 'tabcontainer', 'tab', 'collapsiblepanel', 'text', 'image', 'hyperlink', and 'pushbutton'. If the parent dialog control type is 'tabcontainer', the child dialog control must be 'tab'.

Name

The identifier of the dialog control element. *Name* is a required argument. This field is available for all dialog control types.

Prompt

Text that is displayed in the dialog control element on the Mask dialog box. This field is available for all except for panel and image dialog control types.

Default: empty

Enabled

Option to specify whether you can set value for the dialog control element. This field is available for all dialog control types.

Default: 'on'

Visible

Option to set whether the dialog control element is hidden or visible to the user. This field is available for all dialog control types.

Default: 'on'

Callback

Container for MATLAB code that executes when you edit the dialog control element and click **Apply**. This field is available only for the hyperlink and pushbutton dialog control types.

Default: empty

Row

Option to set whether the dialog control is placed in the new row or the same row. This field is available for all dialog control types.

Default: empty

FilePath

Contains the path to an image file. This field is available for image, and pushbutton dialog control types.

Default: empty

Container

Option to specifies a container for the child dialog control. The permitted values are the names of 'panel', 'group', and 'tab' dialog controls.

Examples

Add Dialog Control Elements to Mask Dialog Box

Get mask object and add dialog control element to it.

```
% Get mask object on model Engine
maskObj = Simulink.Mask.get('Engine/Gain');

% Add hyperlink to mask dialog box

maskLink = maskObj.addDialogControl('hyperlink','link');
maskLink.Prompt = 'Mathworks Home Page';
maskLink.Callback = 'web(''www.mathworks.com'')'

% Alternative method to add hyperlink

maskLink = maskObj.addDialogControl('hyperlink','link');
maskLink.Prompt = 'www.mathworks.com';

% Add text to mask dialog box

maskText = maskObj.addDialogControl('text','text_tag');
maskText.Prompt = 'Enable range checking';

% Add button to mask dialog box

maskButton = maskObj.addDialogControl('pushbutton','button_tag');
maskButton.Prompt = 'Compute';
```

Add Dialog Control Elements to Mask Dialog Box Tabs

Create tabs on the mask dialog box and add elements to these tabs.

```
% Get mask object on a block named 'GainBlock'
maskObj = Simulink.Mask.get('GainBlock/Gain');

% Create a tab container
```

```

maskObj.addDialogControl('tabcontainer','allTabs');
tabs = maskObj.getDialogControl('allTabs');

% Create tabs and name them

maskTab1 = tabs.addDialogControl('tab','First');
maskTab1.Prompt = 'First tab';

maskTab2 = tabs.addDialogControl('tab','Second');
maskTab2.Prompt = 'Second tab';

% Add elements to one of the tabs

firstTab = tabs.getDialogControl('First');
firstTab.addDialogControl('text','textOnFirst');
firstTab.getDialogControl('textOnFirst').Prompt = 'Tab one';

```

Add Dialog Control Element Using Name-Value Pair

Add dialog control element and specify values for it

```

% Get mask object on model Engine
maskObj = Simulink.Mask.get('Engine/Gain');

% Add a dialog box and specify values for it

maskDialog = maskObj.addDialogControl('Type','text',...
'Prompt','hello','Visible','off');

```

See Also

[Simulink.dialog.Container](#) | “Block Masks”

Introduced in R2014a

removeDialogControl

Class: Simulink.dialog.Container

Package: Simulink.dialog

Remove dialog control element from mask dialog box

Syntax

```
successIndicator = maskVariable.removeDialogControl(  
controlIdentifier)
```

Description

`successIndicator = maskVariable.removeDialogControl(controlIdentifier)` removes dialog control element, specified by `controlIdentifier`, like text, hyperlinks, or tabs from a mask dialog box. First get the mask object and assign it to the variable `maskVariable`.

Successful removal of a dialog control element returns a Boolean value of 1.

Input Arguments

controlIdentifier — Unique identifier for the element

character vector

Programmatic identifier for the dialog control element of mask dialog box, specified as a character vector.

Examples

Remove Dialog Control Element from Mask Dialog Box

```
% Get mask object on the Gain block in the model Engine.  
maskObj = Simulink.Mask.get('Engine/Gain');  
  
% Remove element named AllTab from mask dialog box.  
maskTab = maskObj.removeDialogControl('AllTab');
```

See Also

`Simulink.dialog.Container` | “Block Masks”

Introduced in R2013b

getDialogControl

Class: Simulink.dialog.Container

Package: Simulink.dialog

Search for a specific dialog control on the mask

Syntax

```
[control, phandle] = handle.getDialogControl(controlIdentifier)
```

Description

[control, phandle] = handle.getDialogControl(controlIdentifier), search for a specific child dialog control recursively on the mask dialog box.

Input Arguments

controlIdentifier

Name of the dialog control being searched on the mask dialog box.

Default:

Output Arguments

control

Target dialog control being searched on the mask dialog box.

phandle

Parent of the dialog control being searched mask dialog box.

Examples

Find a dialog control

Find a text dialog control on the mask dialog box. `maskObj` is the handle to the mask object. The `getDialogControl` method returns the handle to the dialog control (`hdlgctrl`) and handle to the parent dialog control (`phandle`).

```
[hdlgctrl, phandle] = maskObj.getDialogControl('txt_var')
```

See Also

`Simulink.dialog.Container` | “Block Masks”

Simulink.dialog.Panel class

Package: Simulink.dialog

Create an instance of a panel dialog control

Description

Use an instance of `Simulink.dialog.Panel` class to create an instance of panel dialog control.

Properties

Name

Uniquely identifies the panel dialog control and is a required field.

Type: character vector

Row

Specifies whether panel is placed on the current row or on a new row.

Type: character vector

Values: 'current' | 'new'

Default: 'new'

Enabled

Specifies whether panel is active on the mask dialog box.

Type: boolean

Values: 'on' | 'off'

Default: 'on'

Visible

Specifies whether panel is displayed on the mask dialog box.

Type: boolean

Values: 'on' | 'off'

Default: 'on'

DialogControls

Specifies the child dialog controls contained in the panel.

Type: Simulink.dialog.Control

Default: Empty array

Methods

addDialogControl	Add dialog control elements to mask dialog box
removeDialogControl	Remove dialog control element from mask dialog box
getDialogControl	Search for a specific dialog control on the mask

See Also

Simulink.dialog.Group | Simulink.dialog.Tab |
Simulink.dialog.TabContainer | Simulink.dialog.Container |
Simulink.dialog.Control | “Block Masks”

Simulink.dialog.Group class

Package: Simulink.dialog

Create an instance of a group dialog control

Description

Use an instance of `Simulink.dialog.Group` class to create an instance of group dialog control.

Properties

Name

Uniquely identifies the group dialog control and is a required field.

Type: character vector

Prompt

Specifies the text displayed on the group.

Type: character vector

Default: Empty character vector

Row

Specifies whether group is placed on the current row or on a new row.

Type: character vector

Values: 'current' | 'new'

Default: 'new'

Enabled

Specifies whether group is active on the mask dialog box.

Type: boolean

Values: 'on' | 'off'

Default: 'on'

Visible

Specifies whether group is displayed on the mask dialog box.

Type: boolean

Values: 'on' | 'off'

Default: 'on'

DialogControls

Specifies the child dialog controls contained in the group.

Type: Simulink.dialog.Control

Default: Empty array

Methods

addDialogControl	Add dialog control elements to mask dialog box
removeDialogControl	Remove dialog control element from mask dialog box
getDialogControl	Search for a specific dialog control on the mask

See Also

`Simulink.dialog.Panel` | `Simulink.dialog.Tab` |
`Simulink.dialog.TabContainer` | `Simulink.dialog.Container` |
`Simulink.dialog.Control` | “Block Masks”

Simulink.dialog.Tab class

Package: Simulink.dialog

Create an instance of a tab dialog control

Description

Use an instance of `Simulink.dialog.Tab` class to create an instance of tab dialog control.

Properties

Name

Uniquely identifies the tab dialog control and is a required field.

Type: character vector

Prompt

Specifies the text displayed on the tab.

Type: character vector

Default: Empty character vector

Enabled

Specifies whether tab is active on the mask dialog box.

Type: boolean

Values: 'on' | 'off'

Default: 'on'

Visible

Specifies whether tab is displayed on the mask dialog box.

Type: boolean

Values: 'on' | 'off'

Default: 'on'

DialogControls

Specifies the child dialog controls contained in the tab dialog control.

Type: Simulink.dialog.Control

Default: Empty array

Methods

addDialogControl	Add dialog control elements to mask dialog box
removeDialogControl	Remove dialog control element from mask dialog box
getDialogControl	Search for a specific dialog control on the mask

See Also

Simulink.dialog.Group | Simulink.dialog.Panel |
Simulink.dialog.TabContainer | Simulink.dialog.Container |
Simulink.dialog.Control | “Block Masks”

Simulink.dialog.TabContainer class

Package: Simulink.dialog

Create an instance of a tab container dialog control

Description

Use an instance of `Simulink.dialog.TabContainer` class to create an instance of tab container dialog control. Tab container dialog box be used to group the tab dialog controls.

Properties

Name

Uniquely identifies the tab container dialog control and is a required field.

Type: character vector

Row

Specifies whether tab container is placed on the current row or on a new row.

Type: enumerated string

Values: 'current' | 'new'

Default: 'new'

Enabled

Specifies whether tab container is active on the mask dialog box.

Type: boolean

Values: 'on' | 'off'

Default: 'on'

Visible

Specifies whether tab container is displayed on the mask dialog box.

Type: boolean

Values: 'on' | 'off'

Default: 'on'

DialogControls

Specifies the child dialog controls contained in the group.

`Simulink.dialog.TabContainer` class can only contain `Simulink.dialog.Tab` dialog control.

Type: `Simulink.dialog.Tab`

Default: Empty array

Methods

<code>addDialogControl</code>	Add dialog control elements to mask dialog box
<code>removeDialogControl</code>	Remove dialog control element from mask dialog box
<code>getDialogControl</code>	Search for a specific dialog control on the mask

See Also

`Simulink.dialog.Group` | `Simulink.dialog.Panel` | `Simulink.dialog.Tab` | `Simulink.dialog.Container` | `Simulink.dialog.Control` | “Block Masks”

Simulink.dialog.Button class

Package: Simulink.dialog

Create a button dialog control

Description

Use an instance of `Simulink.dialog.Button` class to add a button dialog control.

Properties

Name

Uniquely identifies the dialog control and is a required field.

Type: character vector

Prompt

Specifies the text displayed on the button dialog control.

Type: character vector

Default: empty

FilePath

Specifies the path to the image file to be shown on the button dialog control.

Type: character vector

Default: empty

Callback

Specifies the MATLAB command (s) to be executed when the dialog control is invoked.

Type: character vector

Default: empty

Row

Specifies whether dialog control is placed on the current row or on a new row.

Type: character vector

Value: 'current' | 'new'

Default: 'current'

Enabled

Indicates whether container is active on the mask dialog box.

Type: boolean

Value: 'on' | 'off'

Default: 'on'

Visible

Indicates whether container is displayed on the mask dialog box.

Type: boolean

Value: 'on' | 'off'

Default: 'on'

See Also

`Simulink.dialog.Control` | “Block Masks”

Simulink.dialog.Hyperlink class

Package: Simulink.dialog

Create a hyperlink dialog control

Description

Use an instance of `Simulink.dialog.Hyperlink` class to add a hyperlink dialog control.

Properties

Name

Uniquely identifies the dialog control and is a required field.

Type: character vector

Prompt

Specifies the text displayed on the hyperlink.

Type: character vector

Default: empty

Callback

Specifies the MATLAB command (s) to be executed when the dialog control is invoked.

Type: character vector

Default: empty

Row

Specifies whether hyperlink is placed on the current row or on a new row.

Type: character vector

Value: 'current' | 'new'

Default: 'new'

Enabled

Indicates whether hyperlink is active on the mask dialog box.

Type: boolean

Value: 'on' | 'off'

Default: 'on'

Visible

Indicates whether hyperlink is displayed on the mask dialog box.

Type: boolean

Value: 'on' | 'off'

Default: 'on'

See Also

`Simulink.dialog.Control` | “Block Masks”

Simulink.dialog.Image class

Package: Simulink.dialog

Create an image dialog control

Description

Use an instance of `Simulink.dialog.Image` class to add an image dialog control.

Properties

Name

Uniquely identifies the dialog control and is a required field.

Type: character vector

FilePath

Specifies the path to the image file to be displayed on the dialog box.

Type: character vector

Default: empty

Row

Specifies whether dialog control is placed on the current row or on a new row.

Type: character vector

Value: 'current' | 'new'

Default: 'new'

Enabled

Indicates whether image is active on the mask dialog box.

Type: boolean

Value: 'on' | 'off'

Default: 'on'

Visible

Indicates whether image is displayed on the mask dialog box.

Type: boolean

Value: 'on' | 'off'

Default: 'on'

See Also

`Simulink.dialog.Control` | “Block Masks”

Simulink.dialog.Text class

Package: Simulink.dialog

Create a text dialog control

Description

Use an instance of `Simulink.dialog.Text` class to add a text dialog control.

Properties

Name

Uniquely identifies the dialog control element and is a required field.

Type: character vector

Prompt

Specifies the text displayed on the mask dialog box.

Type: character vector

Default: empty

WordWrap

Specifies whether to wrap long text to the next line on the mask dialog box.

Type: boolean

Value: 'on' | 'off'

Default: 'on'

Row

Specifies whether dialog control is placed on the current row or on a new row.

Type: character vector

Value: 'current' | 'new'

Default: 'new'

Enabled

Indicates whether dialog control is active on the mask dialog box.

Type: boolean

Value: 'on' | 'off'

Default: 'on'

Visible

Indicates whether dialog control is displayed on the mask dialog box.

Type: boolean

Value: 'on' | 'off'

Default: 'on'

See Also

`Simulink.dialog.Control` | “Block Masks”

Simulink.dialog.parameter.Control class

Package: Simulink.dialog.parameter

Create a parameter dialog control

Description

Use an instance of `Simulink.dialog.parameter.Control` class to add a parameter dialog control.

Properties

Name

Uniquely identifies the dialog control element. This is a required field and has the same value as its underlying parameter name.

Type: character vector

Row

Specifies whether the dialog control is placed on the current row or on a new row.

Type: character vector

Value: 'current' | 'new'

Default: 'new'

See Also

`Simulink.dialog.Control` | “Block Masks”

Simulink.sfunction.Analyzer class

Package: Simulink.sfunction

Create a Simulink S-function analyzer object

Description

This class enables you to perform checks on S-functions within a model or a library. These checks include MEX compiler setup check, source code check, MEX-file check, parameter robustness check for S-functions. The check result can be accessed either from a MATLAB structure or an HTML report.

The S-function analyzer checks the source code of the S-functions based on the S-function names. The S-function source code can be automatically included in the analysis if the source file is a single .c or .cpp file in MATLAB path that has same name as the S-function. Otherwise, the build information can be specified through the S-function Analyzer APIs. If no source code is available on the specified path, the analysis is skipped.

Construction

`sfunAnalyzer = Simulink.sfunction.Analyzer(model)` creates a `Simulink.sfunction.Analyzer` object with the model you specify. In this case, the source code for the S-function can be automatically included in the analysis if the source code file is a single .c or .cpp file in MATLAB path that has as the S-function. For example, if the specified model contains an S-function called `mysfun`, and the source file for `mysfun` is a single file `mysfun.c` in the MATLAB path, a `Simulink.sfunction.analyzer.BuildInfo` object is automatically created and included in the analysis.

`sfunAnalyzer = Simulink.sfunction.Analyzer(model, 'BuildInfo', {bdInfo})` creates a `Simulink.sfunction.Analyzer` object with the model and a `Simulink.sfunction.analyzer.BuildInfo` object named `bdInfo`.

```
sfunAnalyzer = Simulink.sfunction.Analyzer(model, 'Options', {opts})
```

creates an `Simulink.sfunction.Analyzer` object with the model and a `Simulink.sfunction.analyzer.Options` object named `opts`.

Input Arguments

model — Specify a model in the path

character vector

Names of the model in the path, specified as a character vector.

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'BuildInfo', {buildInfo}

BuildInfo — Specify the buildinfo object

cell array

A cell array of `buildinfo` objects. See `Simulink.sfunction.analyzer.BuildInfo` for more information.

Options — Specify the S-function analyzer running options

object

An object to specify S-function analyzer running options. These checks include Polyspace® Code Prover™ and parameter robustness checks. See `Simulink.sfunction.analyzer.Options` for more information.

Methods

<code>run</code>	Perform checks on S-functions
<code>generateReport</code>	Generate an HTML report of S-function checks

See Also

`Simulink.sfunction.Analyzer.generateReport` |
`Simulink.sfunction.Analyzer.run` |
`Simulink.sfunction.analyzer.BuildInfo` |
`Simulink.sfunction.analyzer.Options` |
`Simulink.sfunction.analyzer.findSfunctions`

Introduced in R2017b

run

Class: Simulink.sfunction.Analyzer

Package: Simulink.sfunction

Perform checks on S-functions

Syntax

```
result = run()
```

Description

`result = run()` returns a struct containing the result from the analyzer checks. An example result struct has the following fields:

```
TimeGenerated: '19-Jul-2017 19:25:32'  
Platform: 'win64'  
Release: '(R2017b)'  
SimulinkVersion: '9.0'  
ExemptedBlocks: {}  
MexConfiguration: [1x1 mex.CompilerConfiguration]  
Data: [4x4 struct]
```

Output Arguments

result — Sfunction.Analyzer structure

MATLAB structure

MATLAB structure containing the result of S-function analyzer.

See Also

`Simulink.sfunction.Analyzer` |

`Simulink.sfunction.Analyzer.generateReport` |

```
Simulink.sfunction.analyzer.BuildInfo |  
Simulink.sfunction.analyzer.Options |  
Simulink.sfunction.analyzer.findSfunctions
```

Introduced in R2017b

generateReport

Class: Simulink.sfunction.Analyzer

Package: Simulink.sfunction

Generate an HTML report of S-function checks

Syntax

```
generateReport ()
```

Description

`generateReport ()` generates an HTML report and launches the browser to display the report.

See Also

`Simulink.sfunction.Analyzer` | `Simulink.sfunction.Analyzer.run` |
`Simulink.sfunction.analyzer.BuildInfo` |
`Simulink.sfunction.analyzer.Options` |
`Simulink.sfunction.analyzer.findSfunctions`

Introduced in R2017b

Simulink.sfunction.analyzer.BuildInfo class

Package: Simulink.sfunction.analyzer

Create an object to represent build information

Description

`Simulink.sfunction.analyzer.BuildInfo` object captures the build information for S-functions, such as source files, header files, and linking libraries, for use with the `Simulink.sfunction.Analyzer` class.

Construction

`bdInfo = Simulink.sfunction.analyzer.BuildInfo(SfcnFile)` creates a `Simulink.sfunction.analyzer.BuildInfo` object.

`bdInfo = Simulink.sfunction.analyzer.BuildInfo(SfcnFile, 'SrcPath', {srcpaths}, 'ExtraSrcFileList', {srcfilelist})` creates a `Simulink.sfunction.analyzer.BuildInfo` object for a C-MEX S-function source file, a list of extra source files located in the specified path.

`bdInfo = Simulink.sfunction.analyzer.BuildInfo(SfcnFile, 'ObjFileList', {objfilelist})` creates a `Simulink.sfunction.analyzer.BuildInfo` object for C-MEX S-function source file and list of extra objective code files.

`bdInfo = Simulink.sfunction.analyzer.BuildInfo(SfcnFile, 'IncPaths', {incpathslist})` creates a `Simulink.sfunction.analyzer.BuildInfo` object for C-MEX S-function source file and paths to the folders including header files.

`bdInfo = Simulink.sfunction.analyzer.BuildInfo(SfcnFile, 'LibFileList', {libfilelist}, 'LibPaths', {libpaths})` creates a `Simulink.sfunction.analyzer.BuildInfo` object for C-MEX S-function source file and library files and library file paths used for building.

`bdInfo = Simulink.sfunction.analyzer.BuildInfo(SfcnFile, 'PreProcDefList', {preprocdir})` creates a `Simulink.sfunction.analyzer.BuildInfo` object for C-MEX S-function source file and pre-processor directives list.

Input Arguments

SfcnFile — S-function source file

character vector

S-function source file having the same name as the S-function.

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'SrcPath', {srcpath}`

SrcPath — Source file path

cell array of character vectors

Search paths to extra S-function source files that are referenced by `SfcnFile`, specified as a cell array.

ExtraSrcFileList — Extra source file list

cell array of character vectors

List of extra S-function source files, specified as a cell array.

ObjFileList — Extra objective code

cell array

List of objective files used for building, specified as a cell array.

IncPaths — Search paths for header files

cell array of character vectors

Specify paths to include folders for header files, specified as a cell array.

LibFileList — External libraries list

cell array of character vectors

List of external library files used for building, specified as a cell array.

LibPaths — Search paths for external libraries

cell array of character vectors

Search paths for external library files used for building, specified as a cell array.

PreProcDefList — Preprocessor directives

cell array of character vectors

List of the preprocessor directives, specified as a cell array.

Output Arguments

bdInfo — BuildInfo object

object

Build information for the S-functions supplied to the S-function analyzer. Returns a `simulink.sfunction.analyzer.BuildInfo` object.

Examples

Create a `bdInfo` object for an S-function `mysfun` that includes a source file `mysfun.c`:

Basic Use

```
bdInfo = simulink.sfunction.analyzer.BuildInfo('mysfun.c');
```

The output `bdInfo` has the following fields:

```
bdInfo =
```

```
BuildInfo with properties:
```

```
SfcnFile: 'mysfun.c'  
SfcnName: 'mysfun'
```

```

        SrcType: 'C'
        SrcPaths: {}
        ExtraSrcFileList: {}
        ObjFileList: {}
        IncPaths: {}
        LibFileList: {}
        LibPaths: {}
        PreProcDefList: {}

```

Advanced Use

Create a `bdInfo` object for an S-function `mysfun` that includes a source file `mysfun.c` and also includes:

- List of extra source files, `extra1.c` and `extra2.c`
- Paths to source file folders, `/path1` and `/path2`.
- List of objective files, `o1.obj` and `o2.obj`.
- List of library files, `l1.lib` and `l2.lib`.
- Library paths, `/libpath1`.
- Pre-processor running directives, `-DDEBUG`.

```

Simulink.sfunction.analyzer.BuildInfo('mysfun.c',...
    'ExtraSrcFileList', {extra1.c,extra2.c},... %speci
    'SrcPaths', {/path1,/path2},... %speci
    'ObjFileList', {o1.obj,o2.obj},... %speci
    'LibFileList', {l1.lib,l2.lib},... %speci
    'LibPaths', {/libpath1},... %speci
    'PreProcDefList', {DEBUG}); %speci

```

See Also

[Simulink.sfunction.Analyzer](#) |
[Simulink.sfunction.Analyzer.generateReport](#) |
[Simulink.sfunction.Analyzer.run](#) | [Simulink.sfunction.analyzer.Options](#)
| [Simulink.sfunction.analyzer.findSfunctions](#)

Introduced in R2017b

Simulink.sfunction.analyzer.Options class

Package: Simulink.sfunction.analyzer

Create an object to specify options for running S-function checks

Description

Simulink.sfunction.analyzer.Options object is created through the constructor `Simulink.sfunction.analyzer.Options()`.

Simulink.sfunction.analyzer.Options object captures the options for running S-function checks. These checks include whether to enable Polyspace and Parameter Robustness checks, maximum model simulation time and output path for result report.

Construction

`opts= Simulink.sfunction.analyzer.Options()` returns a options object with these property values:

```
EnablePolyspace: 0
EnableRobustness: 0
ReportPath: ''
ModelSimTimeOut: 10
```

Properties

EnablePolyspace — check

False (default) | True

Boolean type check to determine whether to include Polyspace Code Prover check.

Note These checks usually take some time to run.

EnableRobustness — Parameter robustness check

False (default) | True

Boolean type check to indicate whether to include Robutness checks.

Note These checks usually take some time to run.

ReportPath — Generated report directory

character array

Path to the generated report directory.

ModelSimTimeOut — Maximum model simulation time

positive scalar

Maximum model simulation time in seconds.

See Also

`Simulink.sfunction.Analyzer` |
`Simulink.sfunction.Analyzer.generateReport` |
`Simulink.sfunction.Analyzer.run` |
`Simulink.sfunction.analyzer.BuildInfo` |
`Simulink.sfunction.analyzer.findSfunctions`

Introduced in R2017b

findSfunctions

Find and return all eligible S-functions in a model

Syntax

```
sfuns = Simulink.sfunction.analyzer.findSfunctions(model)
```

Description

`sfuns = Simulink.sfunction.analyzer.findSfunctions(model)` returns all eligible S-functions in a model for the S-function checks. Rules are applied to filter out all ineligible S-functions.

Input Arguments

model — A Simulink model or library in path
Simulink model

A Simulink model or library in path.

Output Arguments

sfuns — A list of all eligible S-functions
cell array

Eligible s-functions in the model, specified as a cell array.

See Also

`Simulink.sfunction.Analyzer` |
`Simulink.sfunction.Analyzer.generateReport` |
`Simulink.sfunction.Analyzer.run` |

Simulink.sfunction.analyzer.BuildInfo |
Simulink.sfunction.analyzer.Options

Introduced in R2017b

addElement

Class: Simulink.SimulationData.Dataset

Package: Simulink.SimulationData

Add element to end of data set

Syntax

```
dataset = addElement(dataset,element)
dataset = addElement(dataset,element,name)
```

Description

`dataset = addElement(dataset,element)` adds an element to the `Simulink.SimulationData.Dataset` `dataset`.

`dataset = addElement(dataset,element,name)` adds an element to the `Simulink.SimulationData.Dataset` `data set` and gives the element the name that you specify with the `name` argument. If the object already has a name, the element instead uses the name you specify by using the `name` argument.

Input Arguments

dataset — Data set

`SimulationData.Dataset` object

The data set to which to add the element.

element — Element to add

`Simulink.SimulationData.Signal` object | `Simulink.SimulationData.State` object | `Simulink.SimulationData.DataStoreMemory` object | `timeseries` object | `matlab.io.datastore.SimulationDatastore` object

Element to add to the data set, specified as a `Simulink.SimulationData.Signal`, `Simulink.SimulationData.DataStoreMemory`, or `matlab.io.datastore.SimulationDatastore` object.

name — Name for element

character vector

Name for element, specified as a character vector.

Output Arguments

dataset — Data set

character vector

The data set to which you add the element, returned as a character vector. The new element is added to the end of the data set.

Examples

Create a Data Set

Create a data set and add three elements to it.

```
time = 0.1*(0:100)';
ds = Simulink.SimulationData.Dataset;
element1 = Simulink.SimulationData.Signal;
element1.Name = 'A';
element1.Values = timeseries(sin(time),time);
ds = addElement(ds,element1);
element2 = Simulink.SimulationData.Signal;
element2.Name = 'B';
element2.Values = timeseries(2*sin(time),time);
ds = addElement(ds,element2);
element3 = Simulink.SimulationData.Signal;
element3.Name = 'C';
element3.Values = timeseries(3*sin(time),time);
ds = addElement(ds,element3);
ds
```

```
ds =
```

```
Simulink.SimulationData.Dataset '' with 3 elements
```

		Name	BlockPath
1	[1x1 Signal]	A	''
2	[1x1 Signal]	B	''
3	[1x1 Signal]	C	''

- Use braces { } to access, modify, or add elements using index.

- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use Legacy ModelDataLogs API”

Alternative

To streamline indexing syntax, you can use curly braces ({}) to add an element to a dataset, instead of using `addElement`. For the index, use a scalar that is greater than the number of elements by one. The new element becomes the last element of the dataset.

```
time = 0.1*(0:100)';
ds = Simulink.SimulationData.Dataset;
element1 = Simulink.SimulationData.Signal;
element1.Name = 'A';
element1.Values = timeseries(sin(time),time);
ds{1} = element1;
element2 = Simulink.SimulationData.Signal;
element2.Name = 'B';
element2.Values = timeseries(2*sin(time),time);
ds{2} = element2;
element3 = Simulink.SimulationData.Signal;
element3.Name = 'C';
element3.Values = timeseries(3*sin(time),time);
ds{3} = element3;
```

See Also

`Simulink.SimulationData.BlockPath` |
`Simulink.SimulationData.DataStoreMemory` |
`Simulink.SimulationData.Dataset` |
`Simulink.SimulationData.Dataset.concat` |
`Simulink.SimulationData.Dataset.find` |
`Simulink.SimulationData.Dataset.get` |
`Simulink.SimulationData.Dataset.getElementNames` |
`Simulink.SimulationData.Dataset.numElements` |
`Simulink.SimulationData.Dataset.setElement` |
`Simulink.SimulationData.Signal` |
`matlab.io.datastore.SimulationDatastore`

Topics

“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”
“Log Data Stores”
“Convert Logged Data to Dataset Format”
“Migrate Scripts That Use Legacy ModelDataLogs API”

Introduced in R2011a

concat

Class: Simulink.SimulationData.Dataset

Package: Simulink.SimulationData

Concatenate dataset to another dataset

Syntax

```
dataset1 = concat(dataset1,dataset2)
```

Description

`dataset1 = concat(dataset1,dataset2)` concatenates the elements of `dataset2` to `dataset1`.

Input Arguments

dataset1 — Dataset to concatenate to

data set

Dataset to concatenate to with `dataset2`, returned as a cell array.

dataset2 — Dataset to concatenate

data set

Data set to concatenate to `dataset1`, specified as a cell array.

Output Arguments

dataset1 — Concatenated dataset

data set

Concatenated dataset from `dataset1` and `dataset2`.

Examples

Concatenate ds1 to ds

Convert output from two To Workspace blocks to Dataset format and concatenate them.

```
mdl = 'myvdp';  
open_system(mdl);  
sim(mdl)  
ds = Simulink.SimulationData.Dataset(simout);  
ds1 = Simulink.SimulationData.Dataset(simout1);  
dsfinal = concat(ds,ds1);
```

- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use Legacy ModelDataLogs API”

See Also

```
Simulink.SimulationData.BlockPath |  
Simulink.SimulationData.DataStoreMemory |  
Simulink.SimulationData.Dataset |  
Simulink.SimulationData.Dataset.addElement |  
Simulink.SimulationData.Dataset.find |  
Simulink.SimulationData.Dataset.get |  
Simulink.SimulationData.Dataset.getElementNames |  
Simulink.SimulationData.Dataset.numElements |  
Simulink.SimulationData.Dataset.setElement |  
Simulink.SimulationData.Signal
```

Topics

- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”

“Migrate Scripts That Use Legacy ModelDataLogs API”

Introduced in R2015a

get

Class: Simulink.SimulationData.Dataset

Package: Simulink.SimulationData

Get element or collection of elements from dataset

Syntax

```
element = get(dataset, index)
element = get(dataset, name)
element = get(dataset, {name})
```

Description

`element = get(dataset, index)` returns the element corresponding to the `index`. The `getElement` method uses the same syntax and behavior as the `get` method.

`element = get(dataset, name)` returns the element whose name matches `name`. When `name` is in a cell array, return the index of the element whose name matches `name`.

`element = get(dataset, {name})` returns a single element if only one element name matches, a `SimulationData.Dataset` if multiple elements with this name exist.

If you use **Log Dataset data to file** to create the MAT-file, use `Simulink.SimulationData.DatasetRef.getAsDatastore` for fast access to the data.

Input Arguments

dataset — Dataset

`SimulationData.Dataset` object

The data set from which to get the element.

index — Index value of element to get

scalar numeric

Index value of element to get. The index reflects the index value of a data set element.

name — Name for data set element

character array | cell array

Name for a data set element, specified as:

- A character array reflecting the name of the data set element
- A cell array containing one character vector. To return a `SimulationData.Dataset` object that can contain one element, use this format. Consider this form when writing scripts.

Output Arguments

element — Element

element | `SimulationData.Dataset` object | empty object

The element that the `get` method finds.

- If `index` is the first argument after the data set, the method returns the element at the `index`.
- If `name` is the first argument after the data set:
 - If the method finds one element, it returns the element.
 - If the method finds more than one element, return a `Dataset` that contains the elements.
 - If the method does not find an element, it returns an empty object.

Examples

Access Dataset Elements

Access `Simulink.SimulationData.Dataset` elements in the top model of the `ex_bus_logging` model. The signal logging dataset is `topOut`.

```

open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_bus_logging')));
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_mdhref_counter_bus')));
sim('ex_bus_logging')
topOut

topOut =

    Simulink.SimulationData.Dataset
    Package: Simulink.SimulationData

    Characteristics:
        Name: 'topOut'
        Total Elements: 4

    Elements:
        1: 'COUNTERBUS'
        2: 'OUTPUTBUS'
        3: 'INCREMENTBUS'
        4: 'inner_bus'

    -Use get or getElement to access elements by index, name or
    block path.
    -Use addElement or setElement to add or modify elements.

    Methods, Superclasses

```

Access Dataset Elements with Index

Access the element at index if the first argument is a numeric value.

```
e1 = logcout.get(1);
```

Access Dataset Elements with Characters

Access the element whose name matches name.

```
e1 = logcout.get('name');
```

Access Dataset Elements with Cell Array

Return a dataset if the first argument is a cell array with a character vector as the first element.

```
ds = logouts.get({'my_name'});
```

- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use Legacy ModelDataLogs API”

Alternatives

You can use curly braces to streamline indexing syntax to get an element in a dataset, instead of using `get` or `getElement`. The index must be a scalar that is not greater than the number of elements in the variable. For example, get the second element of the `logouts` dataset.

```
logouts{2}
```

Also, you can use the `find` method to get an element or collection of elements from a dataset.

See Also

```
Simulink.SimulationData.BlockPath |  
Simulink.SimulationData.DataStoreMemory |  
Simulink.SimulationData.Dataset |  
Simulink.SimulationData.Dataset.addElement |  
Simulink.SimulationData.Dataset.concat |  
Simulink.SimulationData.Dataset.find |  
Simulink.SimulationData.Dataset.getElementNames |  
Simulink.SimulationData.Dataset.numElements |  
Simulink.SimulationData.Dataset.setElement |  
Simulink.SimulationData.Signal
```

Topics

“Migrate Scripts That Use Legacy ModelDataLogs API”

“Export Signal Data Using Signal Logging”

“Log Data Stores”

“Convert Logged Data to Dataset Format”

“Migrate Scripts That Use Legacy ModelDataLogs API”

Introduced in R2011a

getElementNames

Class: Simulink.SimulationData.Dataset

Package: Simulink.SimulationData

Return names of all elements in dataset

Syntax

```
element_list = getElementNames(dataset)
```

Description

`element_list = getElementNames(dataset)` returns the names of all of the elements in the `Simulink.SimulationData.Dataset` object.

Input Arguments

dataset — Data set

`SimulationData.Dataset` object

The data set from which to the element name.

Output Arguments

element_list — Data set

cell array

Data set, returned as a cell array of the character vectors containing names of all of the elements of the dataset.

Examples

Return Names of Elements

Return the names of the elements for the `topOut` data set (the signal logging data).

```
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_bus_logging')));
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', ...
'examples', 'ex_mdhref_counter_bus')));
sim('ex_bus_logging')
el_names = topOut.getElementNames

el_names =

    'COUNTERBUS'
    'OUTPUTBUS'
    'INCREMENTBUS'
    'inner_bus'
```

- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use Legacy ModelDataLogs API”

See Also

```
Simulink.SimulationData.BlockPath |
Simulink.SimulationData.DataStoreMemory |
Simulink.SimulationData.Dataset |
Simulink.SimulationData.Dataset.addElement |
Simulink.SimulationData.Dataset.concat |
Simulink.SimulationData.Dataset.find |
Simulink.SimulationData.Dataset.get |
Simulink.SimulationData.Dataset.numElements |
Simulink.SimulationData.Dataset.setElement |
Simulink.SimulationData.Signal
```

Topics

“Migrate Scripts That Use Legacy ModelDataLogs API”

“Export Signal Data Using Signal Logging”
“Log Data Stores”
“Convert Logged Data to Dataset Format”
“Migrate Scripts That Use Legacy ModelDataLogs API”

Introduced in R2011a

find

Class: Simulink.SimulationData.Dataset

Package: Simulink.SimulationData

Get element or collection of elements from dataset

Syntax

```
[datasetOut,retIndex]=find(datasetIn,Name,Value,...)
```

```
[datasetOut,retIndex]=find(datasetIn,Name,Value,'-logicaloperator',...  
Name,Value,...)
```

```
[datasetOut,retIndex]=find(datasetIn,'-regexp',Name,Value,...)
```

Description

`[datasetOut,retIndex]=find(datasetIn,Name,Value,...)` returns a `Simulink.SimulationData.Dataset` object and indices of the elements whose property values match the specified property names and values. Specify optional comma-separated pairs of `Name,Value` properties. `Name` is the property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair properties in any order as `Name1,Value1,...,NameN,ValueN`.

`[datasetOut,retIndex]=find(datasetIn,Name,Value,'-logicaloperator',...
Name,Value,...)` applies the logical operator to the matching property value. You can combine multiple logical operators. Logical operator can be one of:

- `-or`
- `-and`

If you do not specify an operation, the method assumes `-and`.

`[datasetOut,retIndex]=find(datasetIn,'-regexp',Name,Value,...)` matches elements using regular expressions as if the value of the property is passed to the `regexp` function as:

```
regexp(element.Name, Value)
```

The method applies regular expression matching to the name-value pairs that appear after `-regexp`. If there is no `-regexp`, the method matches elements as if the value of the property is passed as:

```
isequal(element.Name, Value)
```

For more information on `-regexp`, see “`-regexp With Multiple Block Paths`” on page 5-916.

-regexp With Multiple Block Paths

`-regexp` works with properties of type `char`. To specify multiple block paths, you can use `Simulink.SimulationData.BlockPath` and `Simulink.BlockPath`. For example, when a signal is logged in a referenced model, you can use `Simulink.SimulationData.BlockPath` to specify multiple block paths.

The method returns elements that contain a **BlockPath** property where one or more of the individual block paths match the specified `Value` path when you use:

- `-regexp` with the **BlockPath** Name property.
- `Value` as a character vector or scalar object of type `Simulink.SimulationData.BlockPath` with one block path

Input Arguments

datasetIn — `SimulationData.Dataset`

`SimulationData.Dataset` object

`SimulationData.Dataset` object in which to search for matching elements.

Name — Name of property

character vector

Name of property to find in the element.

value — Value of property

character vector | double | `Simulink.SimulationData.BlockPath`

Value of property to find in the element.

Output Arguments

datasetOut — SimulationData.Dataset data set

SimulationData.Dataset

SimulationData.Dataset object that contains the elements that match the specified criteria. If there is no matching SimulationData.Dataset object, the returned SimulationData.Dataset object contains no elements.

retIndex — Indices

vector

Indices of the elements datasetIn that match the specified criteria.

Examples

Find Block Path

Find a specific block path (specified by character vector) and port index.

```
dsOut = find(dsIn, 'BlockPath', 'vdp/x1', 'PortIndex', 1)
```

Find Elements

Find elements that have either name or propagated name as InValve.

```
dsOut = find(dsIn, 'Name', 'InValve', '-or', 'PropagatedName', 'InValve')
dsOut = find(dsIn, '-regex', 'Name', 'In*', '-or', ...
             '-regex', 'PropagatedName', 'In*')
```

Find and Change Element

Find and replace all elements containing specified_name with a new_name.

```
[dsOut,idxInDs] = find(ds, 'specified_name');  
for idx=1: length(idxInDs)  
    % process each element  
    elm = get(dsOut, idx);  
    elm.Name= 'New_Name'  
    dsIn = setElement(dsIn, idxInDs(idx), elm);  
end
```

Find Signals in subSys Using -regexp

Find all signals logged in a subSys using -regexp.

```
dsOut = find(dsIn, '-regexp', 'BlockPath', 'mdl/subSys/.*')
```

Find Signals in Referenced Model

Find all signals logged in the Model block.

```
dsOut = find(dsIn, '-regexp', 'BlockPath', 'refmdl/ModelBlk')
```

- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Load Big Data for Simulations”

Alternative

You can use curly braces to streamline indexing syntax to get an element in a dataset, instead of using `find`. The index must be a scalar that is not greater than the number of elements in the variable. For example, get the second element of the `logouts` dataset.

```
logouts{2}
```

Also, you can use the `get` method to get an element or collection of elements from a dataset.

See Also

`Simulink.SimulationData.BlockPath` |
`Simulink.SimulationData.DataStoreMemory` |
`Simulink.SimulationData.Dataset` |
`Simulink.SimulationData.Dataset.addElement` |
`Simulink.SimulationData.Dataset.concat` |
`Simulink.SimulationData.Dataset.get` |
`Simulink.SimulationData.Dataset.getElementNames` |
`Simulink.SimulationData.Dataset.numElements` |
`Simulink.SimulationData.Dataset.setElement` |
`Simulink.SimulationData.DatasetRef.getDatasetVariableNames` |
`Simulink.SimulationData.Signal` | `findobj` | `regexp`

Topics

“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”
“Log Data Stores”
“Convert Logged Data to Dataset Format”
“Migrate Scripts That Use Legacy ModelDataLogs API”
“Load Big Data for Simulations”

Introduced in R2015b

numElements

Class: Simulink.SimulationData.Dataset

Package: Simulink.SimulationData

Get number of elements in data set

Syntax

```
length = numElements(dataset)
```

Description

`length = numElements(dataset)` gets the number of elements in the top-level dataset. To get the number of elements of a nested data set, use `numElements` with the nested data set.

Input Arguments

dataset — Data set

SimulationData.Dataset object

The data set from which to get the number of elements.

Output Arguments

length — Number of elements

double

Number of elements, returned as a double.

Examples

Get Number of Elements

Get the number of elements in the signal logging data set for the `ex_bus_logging`.

```
length = topOut.numElements()
```

- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use Legacy ModelDataLogs API”

See Also

```
Simulink.SimulationData.BlockPath |  
Simulink.SimulationData.DataStoreMemory |  
Simulink.SimulationData.Dataset |  
Simulink.SimulationData.Dataset.addElement |  
Simulink.SimulationData.Dataset.concat |  
Simulink.SimulationData.Dataset.find |  
Simulink.SimulationData.Dataset.get |  
Simulink.SimulationData.Dataset.getElementNames |  
Simulink.SimulationData.Dataset.setElement |  
Simulink.SimulationData.Signal
```

Topics

“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”
“Log Data Stores”
“Convert Logged Data to Dataset Format”
“Migrate Scripts That Use Legacy ModelDataLogs API”

Introduced in R2011a

plot

Class: Simulink.SimulationData.Dataset

Package: Simulink.SimulationData

Plot dataset elements in Signal Preview window or Simulation Data Inspector

Syntax

```
plot(ds)  
plot(ds,viewer)
```

Description

`plot(ds)` plots the `Simulink.SimulationData.Dataset` elements against time and interpolates values between samples by using either zero-order-hold or linear interpolation. The plot displays as a read-only plot in the Signal Preview window.

`plot(ds,viewer)` displays the plot in the Signal Preview window or Simulation Data Inspector, depending on the `viewer` value.

Input Arguments

ds — Data set

`SimulationData.Dataset` object

The data set that contains the elements to plot.

viewer — Viewer to display plot

`preview` (default) | `sdi`

Viewer to display the plot, specified as `preview` (Signal Preview window) or `sdi` (Simulation Data Inspector).

Examples

Plot a Data Set

Create a timeseries object `ts` and add elements to plot in Simulation Data Inspector.

```
% Create a timeseries object
ts = timeseries([0;20],[0;10]);
% Create a SimulationData.Dataset
ds = Simulink.SimulationData.Dataset();
% Place timeseries object in dataset
ds = ds.addElement(ts,'ts');
% Plot the element
plot(ds,'sdi');
```

- “View and Inspect Signal Data”
- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Load Big Data for Simulations”

See Also

`Simulink.SimulationData.Dataset` |
`Simulink.SimulationData.Dataset.addElement`

Topics

“View and Inspect Signal Data”
 “Migrate Scripts That Use Legacy ModelDataLogs API”
 “Export Signal Data Using Signal Logging”
 “Log Data Stores”
 “Convert Logged Data to Dataset Format”
 “Migrate Scripts That Use Legacy ModelDataLogs API”
 “Load Big Data for Simulations”

Introduced in R2016b

setElement

Class: Simulink.SimulationData.Dataset

Package: Simulink.SimulationData

Change element stored at specified index

Syntax

```
dataset = setElement(dataset,index,... element)
dataset = setElement(index,element, name)
```

Description

`dataset = setElement(dataset,index,... element)` changes the element stored at the specified index, for an existing index. If `index` is one greater than the number of elements in the data set, the function adds the element at the end of the data set.

`dataset = setElement(index,element, name)` changes the element stored at the specified index and gives it the name that you specify. You can use `name` to identify an element that does not have a name. If the signal already has a name, the element instead uses the name you specify by using the `name` argument.

Input Arguments

dataset — Data set

`SimulationData.Dataset` object

The data set for which to set the element.

index — Index

scalar

Index for the added element, specified as a scalar numeric value. The value must be between 1 and the number of elements plus 1.

element — Element to replace existing element

`Simulink.SimulationData.Signal` object |
`Simulink.SimulationData.DataStoreMemory` object

Element to replace existing element or to add to the data set, specified as a `Simulink.SimulationData.Signal` object or `Simulink.SimulationData.DataStoreMemory` object.

name — Element name

character vector

Element name, returned as a character vector.

Output Arguments

dataset — Data set

character vector

Data set in which you change or add an element, specified as a character vector.

Examples

Set Element Name

Set element name.

```
ds = Simulink.SimulationData.Dataset
element1 = Simulink.SimulationData.Signal
element1.Name = 'A'
ds = ds.addElement(element1)
element2 = Simulink.SimulationData.Signal
element2.Name = 'B'
elementNew = Simulink.SimulationData.Signal
ds = ds.setElement(2,elementNew,'B1')
ds

ds =

    Simulink.SimulationData.Dataset
```

```
Package: Simulink.SimulationData
```

```
Characteristics:
```

```
    Name: 'topOut'
    Total Elements: 2
```

```
Elements:
```

```
    1: 'A'
    2: 'B1'
```

Use `getElement` to access elements by index, name or block path.

Methods, Superclasses

- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Migrate Scripts That Use Legacy ModelDataLogs API”

Alternative

You can use curly braces to streamline indexing syntax to change an element in a dataset, instead of using `setElement`. The index must be a scalar that is not greater than the number of elements in the variable. For example, change the name of second element of the `logout` dataset.

```
logout{2}.Name = 'secondSignal'
```

See Also

```
Simulink.SimulationData.BlockPath |
Simulink.SimulationData.DataStoreMemory |
Simulink.SimulationData.Dataset |
Simulink.SimulationData.Dataset.addElement |
Simulink.SimulationData.Dataset.concat |
Simulink.SimulationData.Dataset.find |
Simulink.SimulationData.Dataset.get |
Simulink.SimulationData.Dataset.getElementNames |
```

```
Simulink.SimulationData.Dataset.numElements |  
Simulink.SimulationData.Signal
```

Topics

“Migrate Scripts That Use Legacy ModelDataLogs API”

“Export Signal Data Using Signal Logging”

“Log Data Stores”

“Migrate Scripts That Use Legacy ModelDataLogs API”

Introduced in R2011a

coder.BuildConfig class

Package: coder

Build context during code generation

Description

The code generator creates an object of this class to facilitate access to the build context. The build context encapsulates the settings used by the code generator including:

- Target language
- Code generation target
- Target hardware
- Build toolchain

Use `coder.BuildConfig` methods in the methods that you write for the `coder.ExternalDependency` class.

Construction

The code generator creates objects of this class.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Use `coder.BuildConfig` methods to access the build context in `coder.ExternalDependency` methods

This example shows how to use `coder.BuildConfig` methods to access the build context in `coder.ExternalDependency` methods. In this example, you use:

- `coder.BuildConfig.isMatlabHostTarget` to verify that the code generation target is the MATLAB host. If the host is not MATLAB report an error.
- `coder.BuildConfig.getStdLibInfo` to get the link-time and run-time library file extensions. Use this information to update the build information.

Write a class definition file for an external library that contains the function `adder`.

```
%=====
% This class abstracts the API to an external Adder library.
% It implements static methods for updating the build information
% at compile time and build time.
%=====

classdef AdderAPI < coder.ExternalDependency
    %#codegen

    methods (Static)

        function bName = getDescriptiveName(~)
            bName = 'AdderAPI';
        end

        function tf = isSupportedContext(ctx)
            if ctx.isMatlabHostTarget()
                tf = true;
            else
                error('adder library not available for this target');
            end
        end

        function updateBuildInfo(buildInfo, ctx)
            [~, linkLibExt, execLibExt, ~] = ctx.getStdLibInfo();

            % Header files
            hdrFilePath = fullfile(pwd, 'codegen', 'dll', 'adder');
            buildInfo.addIncludePaths(hdrFilePath);
        end
    end
end
```

```
% Link files
linkFiles = strcat('adder', linkLibExt);
linkPath = hdrFilePath;
linkPriority = '';
linkPrecompiled = true;
linkLinkOnly = true;
group = '';
buildInfo.addLinkObjects(linkFiles, linkPath, ...
    linkPriority, linkPrecompiled, linkLinkOnly, group);

% Non-build files
nbFiles = 'adder';
nbFiles = strcat(nbFiles, execLibExt);
buildInfo.addNonBuildFiles(nbFiles, '', '');
end

%API for library function 'adder'
function c = adder(a, b)
    if coder.target('MATLAB')
        % running in MATLAB, use built-in addition
        c = a + b;
    else
        % running in generated code, call library function
        coder.cinclude('adder.h');

        % Because MATLAB Coder generated adder, use the
        % housekeeping functions before and after calling
        % adder with coder.ceval.
        % Call initialize function before calling adder for the
        % first time.

        coder.ceval('adder_initialize');
        c = 0;
        c = coder.ceval('adder', a, b);

        % Call the terminate function after
        % calling adder for the last time.

        coder.ceval('adder_terminate');
    end
end
```

```
end  
end
```

See Also

`coder.ExternalDependency` | `coder.target`

Introduced in R2013b

coder.ExternalDependency class

Package: coder

Interface to external code

Description

`coder.ExternalDependency` is an abstract class for encapsulating the interface between external code and MATLAB code intended for code generation. You define classes that derive from `coder.ExternalDependency` to encapsulate the interface to external libraries, object files, and C/C++ source code. This encapsulation allows you to separate the details of the interface from your MATLAB code. The derived class contains information about external file locations, build information, and the programming interface to external functions.

To define a class, `myclass`, make the following line the first line of your class definition file:

```
classdef myclass < coder.ExternalDependency
```

You must define all of the methods listed in “Methods” on page 5-934. These methods are static and are not compiled. When you write these methods, use `coder.BuildConfig` methods to access build information.

You also define methods that call the external code. These methods are compiled. For each external function that you want to call, write a method to define the programming interface to the function. In the method, use `coder.ceval` to call the external function. Suppose you define the following method for a class named `AdderAPI`:

```
function c = adder(a, b)
    coder.cinclude('adder.h');
    c = 0;
    c = coder.ceval('adder', a, b);
end
```

This method defines the interface to a function `adder` which has two inputs `a` and `b`. In your MATLAB code, call `adder` this way:

```
y = AdderAPI.adder(x1, x2);
```

Methods

Examples

Encapsulate the interface to an external C dynamic linked library

This example shows how to encapsulate the interface to an external C dynamic linked library using `coder.ExternalDependency`.

Write a function `adder` that returns the sum of its inputs.

```
function c = adder(a,b)
    %#codegen
    c = a + b;
end
```

Generate a library that contains `adder`.

```
codegen('adder', '-args', {-2,5}, '-config:dll', '-report');
```

Write the class definition file `AdderAPI.m` to encapsulate the library interface.

```
%=====
% This class abstracts the API to an external Adder library.
% It implements static methods for updating the build information
% at compile time and build time.
%=====

classdef AdderAPI < coder.ExternalDependency
    %#codegen

    methods (Static)

        function bName = getDescriptiveName(~)
            bName = 'AdderAPI';
        end

        function tf = isSupportedContext(ctx)
            if ctx.isMatlabHostTarget()
                tf = true;
            else

```

```
        error('adder library not available for this target');
    end
end

function updateBuildInfo(buildInfo, ctx)
    [~, linkLibExt, execLibExt, ~] = ctx.getStdLibInfo();

    % Header files
    hdrFilePath = fullfile(pwd, 'codegen', 'dll', 'adder');
    buildInfo.addIncludePaths(hdrFilePath);

    % Link files
    linkFiles = strcat('adder', linkLibExt);
    linkPath = hdrFilePath;
    linkPriority = '';
    linkPrecompiled = true;
    linkLinkOnly = true;
    group = '';
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

    % Non-build files
    nbFiles = 'adder';
    nbFiles = strcat(nbFiles, execLibExt);
    buildInfo.addNonBuildFiles(nbFiles, '', '');
end

%API for library function 'adder'
function c = adder(a, b)
    if coder.target('MATLAB')
        % running in MATLAB, use built-in addition
        c = a + b;
    else
        % running in generated code, call library function
        coder.cinclude('adder.h');

        % Because MATLAB Coder generated adder, use the
        % housekeeping functions before and after calling
        % adder with coder.ceval.
        % Call initialize function before calling adder for the
        % first time.

        coder.ceval('adder_initialize');
        c = 0;
    end
end
```

```
        c = coder.ceval('adder', a, b);

        % Call the terminate function after
        % calling adder for the last time.

        coder.ceval('adder_terminate');
    end
end
end
end
```

Write a function `adder_main` that calls the external library function `adder`.

```
function y = adder_main(x1, x2)
    %#codegen
    y = AdderAPI.adder(x1, x2);
end
```

Generate a MEX function for `adder_main`. The MEX Function exercises the `coder.ExternalDependency` methods.

```
codegen('adder_main', '-args', {7,9}, '-report')
```

Copy the library to the current folder using the file extension for your platform.

For Windows, use:

```
copyfile(fullfile(pwd, 'codegen', 'dll', 'adder', 'adder.dll'));
```

For Linux, use:

```
copyfile(fullfile(pwd, 'codegen', 'dll', 'adder', 'adder.so'));
```

Run the MEX function and verify the result.

```
adder_main_mex(2,3)
```

See Also

`coder.BuildConfig` | `coder.ceval` | `coder.cinclude` | `coder.updateBuildInfo`

Topics

“Encapsulating the Interface to External Code”

“Best Practices for Using coder.ExternalDependency”

Introduced in R2013b

Scope Configuration

Control scope appearance and behavior

Description

Scope configuration properties control the appearance and behavior of a scope block. Create a scope configuration object with `get_param`, and then change property values using the object with dot notation.

Creation

`myScopeConfiguration = get_param(gcbh, 'ScopeConfiguration')` creates a scope configuration object for the selected scope block.

Scope Block Properties

Name — Title on a scope window

block name (default) | character vector

Title on a scope window, specified with a character vector.

NumInputPorts — Number of input ports

'1' (default) | character vector

Number of input ports on a scope block, specified by a character vector. Maximum number of input ports is 96.

Block Configuration Property: **Number of input ports**

ActiveDisplay — Display for setting display-specific properties

'1' (default) | character vector

Display for setting display-specific properties, specified as a character vector. The number of a display corresponds to its column-wise placement index.

Dependency: Setting this property selects the display for setting the properties ShowGrid, ShowLegend, Title, PlotAsMagnitudePhase, YLabel, and YLimits.

Block Configuration Property: **Active display**

AxesScaling — How to scale y-axes

'Manual' (default) | 'Auto' | 'Updates'

How to scale y-axes, specified as one of these values:

- 'Manual' — Manually scale y-axes with the **Scale Y-axis Limits** button.
- 'Auto' — Scale y-axes during and after simulation.
- 'Updates' — Scale y-axes after specified number of block updates (time intervals).

Dependency: If this property is set to 'Updates', also specify the property AxesScalingNumUpdates.

Block Configuration Property: **Axes scaling**

AxesScalingNumUpdates — Number of updates before scaling y-axes

'10' (default) | character vector

Number of updates before scaling y-axes. Specified as a character vector.

Dependency: Activate this property by setting AxesScaling to 'Updates'.

Block Configuration Property: **Number of updates**

DataLogging — Save scope data

false (default) | true

Save scope data to a variable in the MATLAB workspace, specified as one of these values:

- false — Inactivate logging and logging properties.
- true — Activate logging and logging properties.

This property does not apply to floating scopes and scope viewers.

Dependency: If this property is set to true, also specify the properties DataLoggingVariableName and DataLoggingSaveFormat.

Block Configuration Property: **Log data to workspace**

DataLoggingVariableName — Variable name for saving scope data`'ScopeData'` (default) | character vector

Variable name for saving scope data in the MATLAB workspace, specified as text. This property does not apply to floating scopes and scope viewers.

Dependency: Activate this property by setting `DataLogging` to `true`.

Block Configuration Property: **Variable name**

DataLoggingSaveFormat — Variable format for saving scope data`'Dataset'` (default) | `'Structure With Time'` | `'Structure'` | `'Array'`

Variable format for saving scope data to the MATLAB workspace, specified as one of these values:

- `'Dataset'` — Save data as a dataset object. This format does not support variable-size data, MAT-file logging, or external mode archiving. See `Simulink.SimulationData.Dataset`.
- `'StructureWithTime'` — Save data as a structure with associated time information. This format does not support single- or multiport frame-based data, or multirate data.
- `'Structure'` — Save data as a structure. This format does not support multi-rate data.
- `'Array'` — Save data as an array with associated time information. This format does not support multiport sample-based data, single or multiport frame-based data, variable-size data, or multi-rate data.

This property does not apply to floating scopes and scope viewers.

Dependency: Activate this property by setting `DataLogging` to `true`.

Block Configuration Property: **Save format**

DataLoggingLimitDataPoints — Limit buffered data`false` (default) | `true`

Limit buffered data before plotting and saving data, specified as one of these values:

- `false` — Save all the data. Setting this parameter to `false` can cause an out-of-memory error.

- `true` — Save signal data at the end of a simulation.

For simulations with **Stop time** set to `inf`, always set this parameter to `true`.

Dependency: If this property is set to `true`, also specify the number of data values to plot and save with the property `DataLoggingMaxPoints`.

Block Configuration Property: **Limit data points to last**

DataLoggingMaxPoints — Maximum number of data values

'5000' (default) | character vector

Maximum number of data values to plot and save, specified as a character vector. The data values are from the end of a simulation.

Dependency: Activate this property by setting `DataLoggingLimitDataPoints` to `true`. Specifying this property limits the data values a scope plots and the data values saved in the MATLAB variable specified in `DataLoggingVariableName`.

Block Configuration Property: Text box to the right of the **Limit data points to last** check box.

DataLoggingDecimateData — Reduce scope data

`false` (default) | `true`

Reduce scope data before plotting and saving, specified as one of these values:

- `false` — Do not reduce buffered data.
- `true` — Reduce buffered data.

Dependency: If this property is set to `true`, also specify `DataLoggingDecimation`.

Block Configuration Property: **Decimation**

DataLoggingDecimation — How signal data is reduced

'1' (default) | character vector

How signal data is reduced before plotting and saving, specified as a character vector. The scope buffers every *N*th data point, where *N* is the decimation factor you specify. A value of 1 buffers all data values.

Dependency: Activate this property by setting `DataLoggingDecimateData` to `true`.

Block Configuration Property: Text box to the right of the **Decimation** check box.

FrameBasedProcessing — Frame-based processing of signals

`false` (default for Time Scope block) | `true` (default for Scope block)

Frame-based processing of signals, specified as one of these values:

- `false` — Process signal values in a channel at each time interval (sample based).
- `true` — Process signal values in a channel as a group of values from multiple time intervals (frame based). Frame-based processing is available only with discrete input signals.

Block Configuration Property: **Input processing**

LayoutDimensions — Number of display rows and columns

`[1 1]` (default) | `[numberOfRows numberOfColumns]`

Number of display rows and columns, specified with as a two-element vector. The maximum layout dimension is four rows by four columns.

- If the number of displays is equal to the number of ports, signals from each port appear on separate displays.
- If the number of displays is less than the number of ports, signals from additional ports appear on the last *y*-axis.

Block Configuration Property: **Layout** button to the right of the **Number of input ports** text box

MaximizeAxes — Maximize size of signal plots

`'Auto'` (default) | `'On'` | `'Off'`

Maximize size of signal plots, specified as one of these values:

- `'Auto'` — If `Title` and `YLabel` are not specified, maximize all plots.
- `'On'` — Maximize all plots. Values in `Title` and `YLabel` are hidden.
- `'Off'` — Do not maximize plots.

Each of the plots expands to fit the full display. Maximizing the size of signal plots removes the background area around the plots.

Block Configuration Property: **Maximize axes**

MinimizeControls — Hide menu and toolbar`false` (default) | `true`

Hide menu and toolbar, specified by one of these values:

- `false` — Display menu and toolbar.
- `true` — Hide menu and toolbar.

If you dock the scope, this property is inactive.

OpenAtSimulationStart — Open scope window`true` (default for Time Scope block) | `false` (default for Scope block)

Open scope window, specified as one of these values:

- `true` — Open Scope when simulation starts.
- `false` — Do not open a closed Scope at the start of a simulation.

Block Configuration Property: **Open at simulation start**

PlotAsMagnitudePhase — Magnitude and phase plots`false` (default) | `true`

Magnitude and phase plots, specified by one of these values:

- `false` — Display signal plot.

If the signal is complex, plot the real and imaginary parts on the same *y*-axis (display).

- `true` — Display magnitude and phase plots.

If the signal is real, plot the absolute value of the signal for the magnitude. The phase is 0 degrees for positive values and 180 degrees for negative values.

Dependency: Set `ActiveDisplay` before setting this property.

Block Configuration Property: **Plot signals as magnitude and phase**

Position — Size and location of Scope`[left bottom width height]`

Size and location of Scope window, specified as a four-element vector consisting of the left, bottom, width, and height positions, in pixels.

By default, a scope window appears in the center of your screen with a width of 560 pixels and height of 420 pixels.

ShowGrid — Vertical and horizontal grid lines

`true` (default) | `false`

Vertical and horizontal grid lines, specified as one of these values:

- `true` — Display grid lines.
- `false` — Hide grid lines.

Dependency: Set `ActiveDisplay` property before setting this property.

Block Configuration Property: **Show grid**

SampleTime — Time interval

'-1' for inherited | character vector

Time interval between Scope block updates during a simulation, specified as a character vector. This property does not apply to floating scopes and scope viewers.

Block Configuration Property: **Sample Time**

ShowLegend — Signal legend

`false` (default) | `true`

Signal legend, specified as one of these values:

- `false` — Hide legend.
- `true` — Show legend on active display.

Names listed in the legend are the signal names from the model. For signals with multiple channels, a channel index is appended after the signal name. See the Scope block reference for an example.

Dependency: Set `ActiveDisplay` property before setting this property.

Block Configuration Property: **Show legend**

ShowTimeAxisLabel — Display or hide x-axis labels

`true` (default for Time Scope block) | `false` (default for Scope block)

Display or hide *x*-axis labels, specified as one of these values:

- `true` — Display *x*-axis labels for the active display.
- `false` — Hide *x*-axis labels.

Dependency: Set `ActiveDisplay` property before setting this property. If this property is set to `true`, also set `TimeAxisLabels`. If `TimeAxisLabels` is set to `'None'`, this property is inactive.

Block Configuration Property: **Show time-axis label**

TimeAxisLabels — How x-axis labels display

`'All'` (default for Time Scope block) | `'Bottom'` (default for Scope block) | `'None'`

How *x*-axis labels display, specified as one of these values:

- `'All'` — Display *x*-axis labels on all *y*-axes.
- `'Bottom'` — Display *x*-axis label only on the bottom *y*-axis.
- `'None'` — Do not display labels and deactivate `ShowTimeAxisLabel` property.

Dependency: Set `ActiveDisplay` before specifying this property. Activate this property by setting `ShowTimeAxisLabel` to `true` and setting `Maximize axes` to `'Off'`.

Block Configuration Property: **Time-axis labels**

TimeDisplayOffset — X-axis range offset

`'0'` (default) | character vector

X-axis range offset, specified as a character vector or vector of character vectors. For input signals with multiple channels, enter a scaler or vector of offsets.

- Scaler — Offset all channels of an input signal by the same value.
- Vector — Independently offset the channels.

Block Configuration Property: **Time display offset**

TimeSpan — Length of x-axis range to display

`'0'` (default) | character vector | `'Auto'`

Length of x -axis range to display, specified as one of these values:

- Positive real number — Value less than the total simulation time.
- 'Auto' — Difference between the simulation start and stop times.

The block calculates the beginning and end times of the x -axis range using the `TimeDisplayOffset` and `TimeSpan` properties. For example, if you set `TimeDisplay` to 10 and the `TimeSpan` to 20, the scope sets the x -axis range from 10 to 30.

Block Configuration Property: **Time span**

TimeSpanOverrunAction — How to display data

'Wrap' (default) | 'Scroll'

How to display data beyond the visible x -axis range, specified as one of these values:

- 'Wrap' — Draw a full screen of data from left to right, clear the screen, and then restart drawing of data.
- 'Scroll' — Move data to the left as new data is drawn on the right. This mode is graphically intensive and can affect run-time performance.

You can see the effects of this option only when plotting is slow with large models or small step sizes.

Block Configuration Property: **Time span overrun action**

TimeUnits — Units to display on the x -axis

'Metric' (default for Time Scope block) | 'None' (default for Scope block) | 'Seconds'

Units to display on the x -axis, specified as one of these values:

- 'Metric' — Display time units based on the length of the `TimeSpan` property.
- 'None' — Display Time on the x -axis.
- 'Seconds' — Display Time (seconds) on the x -axis.

Block Configuration Property: **Time units**

Title — Title for display

'%<SignalLabel>' (default) | character vector

Title for a display, specified as text. The default value %<SignalLabel> uses the input signal name for the title.

Dependency: Set `ActiveDisplay` before setting this property.

Block Configuration Property: **Title**

visible — Visibility of scope window

`true` (default) | `false`

Visibility of scope window, specified as one of these values:

- `true` — Scope window visible.
- `false` — Scope window hidden.

Block Configuration Property: No corresponding property

YLabel — Y-axis label

`' '` (default) | character vector

Y-axis label for active display, specified as a character vector.

Dependency: Set `ActiveDisplay` before setting this property. If `PlotAsMagnitudePhase` is `true`, the value of `YLabel` is hidden and plots are labeled **Magnitude and Phase**.

Block Configuration Property: **Y-label**

YLimits — Minimum and maximum values of y-axis

`[-10 10]` (default) | `[ymin ymax]`

Minimum and maximum values of y-axis, specified as a two-element numeric vector.

Dependency: Set `ActiveDisplay` before setting this property. When `PlotAsMagnitudePhase` is `true`, this property specifies the y-axis limits for the magnitude plot. The y-axis limits of the phase plot are always `[-180 180]`.

Block Configuration Property: **Y-limits (Minimum)** and **Y-limits (Maximum)**

Examples

Create Scope Configuration Object

This example creates a scope configuration object using the 'vdp' model.

```
open_system('vdp')  
myScopeConfiguration = get_param('vdp/Scope','ScopeConfiguration');  
myScopeConfiguration.NumInputPorts = '2';
```

See Also

[Floating Scope](#) | [Scope](#) | [Time Scope](#)

Topics

“Control Scopes Programmatically”

Introduced in R2013a

Model and Block Parameters

- “Model Parameters” on page 6-2
- “Common Block Properties” on page 6-109
- “Block-Specific Parameters” on page 6-128
- “Mask Parameters” on page 6-278

Model Parameters

In this section...
“About Model Parameters” on page 6-2
“Examples of Setting Model Parameters” on page 6-108

About Model Parameters

You can query and/or modify the properties (parameters) of a Simulink model from the command line. Parameters that describe a model are model parameters, and parameters that describe a Simulink block are block parameters. Block parameters that are common to Simulink blocks are called common block parameters. There are also block-specific parameters. Masks also have parameters, that is, parameters that describe a masked block.

The model and block properties can also include callbacks, which are commands that execute when certain model or block events occur. These events include opening a model, simulating a model, copying a block, opening a block, and so on.

This table lists, in alphabetical order, parameters that describe a model. You can set these parameters using the `set_param` command. The **Description** column indicates where you can set the value on a dialog box.

For examples, see “Examples of Setting Model Parameters” on page 6-108. The **Values** column shows the type of value required, the possible values (separated with a vertical line), and the default value enclosed in braces.

The table also includes model callback parameters (see “Callbacks for Customized Model Behavior”). Do not use model parameters in a `PreLoadFcn` callback. Instead, use them in a `PostLoadFcn` callback.

Model Parameters in Alphabetical Order

Parameter	Description	Values
AbsTol	Specify the largest acceptable solver error, as the value of the measured state approaches zero. Set by Absolute tolerance on the Solver pane of the Configuration Parameters dialog box.	{'auto'}
AccelVerboseBuild	Controls the verbosity level during code generation for Simulink Accelerator mode, model reference Accelerator mode, and Rapid Accelerator mode. Set by Verbose accelerator builds on the Configuration Parameters dialog box.	{'off'} 'on'
AlgebraicLoopMsg	Specifies diagnostic action to take when there is an algebraic loop. Set by Algebraic loop on the Solver Diagnostics pane of the Configuration Parameters dialog box.	'none' {'warning'} 'error'
ArrayBoundsChecking	Select the diagnostic action to take when blocks write data to locations outside the memory allocated to them. Set by Array bounds exceeded on the Configuration Parameters dialog box.	{'none'} 'warning' 'error'

Parameter	Description	Values
ArtificialAlgebraic-LoopMsg	<p>Specifies diagnostic action to take if algebraic loop minimization cannot be performed for a subsystem because an input port of that subsystem has direct feedthrough.</p> <p>Set by Minimize algebraic loop on the Solver Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
AssertControl	<p>Enable model verification blocks in the current model either globally or locally.</p> <p>Set by Model Verification block enabling on the Configuration Parameters dialog box.</p>	{'UseLocalSettings'} 'EnableAll' 'DisableAll'
AutoInsertRateTranBlk	<p>Specify whether Simulink software inserts hidden Rate Transition blocks between blocks that have different sample rates.</p> <p>Set by Automatically handle rate transition for data transfer on the Solver pane of the Configuration Parameters dialog box.</p>	'on' {'off'}

Parameter	Description	Values
BlockDescription-StringDataTip	<p>Specifies whether to display the user description for a block as a data tip.</p> <p>In the Simulink Editor, set by Description on the Display > Blocks > Block Tool Tip Options menu.</p>	'on' {'off'}
BlockNameDataTip	<p>Specifies whether to display the block name as a data tip.</p> <p>In the Simulink Editor, set by Block Name on the Display > Blocks > Block Tool Tip Options menu.</p>	'on' {'off'}
BlockParametersDataTip	<p>Specifies whether to display a block parameter in a data tip.</p> <p>In the Simulink Editor, set by Parameter Names & Values on the Display > Blocks > Block Tool Tip Options menu.</p>	'on' {'off'}
BlockPriority-ViolationMsg	<p>Select the diagnostic action to take if Simulink software detects a block priority specification error.</p> <p>Set by Block priority violation on the Solver Diagnostics pane of the Configuration Parameters dialog box.</p>	{'warning'} 'error'

Parameter	Description	Values
BlockReduction	<p>Enables block reduction optimization.</p> <p>Set by Block reduction on the Configuration Parameters dialog box.</p>	{'on'} 'off'
BlockReductionOpt	See BlockReduction on page 6-0 parameter for more information.	
BooleanDataType	<p>Enable Boolean mode.</p> <p>Set by Implement logic signals as Boolean data (vs. double) on the Configuration Parameters dialog box.</p>	{'on'} 'off'
BrowserLookUnderMasks	<p>Show masked subsystems in the Model Browser.</p> <p>In the Simulink Editor, set by Include Systems with Mask Parameters on the View > Model Browser menu.</p>	'on' {'off'}
BrowserShowLibraryLinks	<p>Show library links in the Model Browser.</p> <p>In the Simulink Editor, set by Include Library Links on the View > Model Browser menu.</p>	'on' {'off'}
BufferReusableBoundary	Insert buffers at reusable subsystem boundaries if needed.	'on' {'off'}

Parameter	Description	Values
BufferReuse	<p>Enable reuse of block I/O buffers.</p> <p>Set by Reuse block outputs on the Optimization > Signals and Parameters pane of the Configuration Parameters dialog box.</p>	{'on'} 'off'
BusNameAdapt	<p>Repair broken selections in the Bus Selector and Bus Assignment block parameters dialog boxes that are due to upstream bus hierarchy changes.</p> <p>Set by “Repair bus selections” on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box.</p>	{'WarnAndRepair'} 'ErrorWithoutRepair'
BusObjectLabelMismatch	<p>Select the diagnostic action to take if the name of a bus element does not match the name specified by the corresponding bus object.</p> <p>Set by Element name mismatch on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'

Parameter	Description	Values
CheckExecutionContext- RuntimeOutputMsg	Specify whether to display a warning if Simulink software detects potential output differences from previous releases. Set by Check runtime output of execution context on the Configuration Parameters dialog box.	'on' {'off'}
CheckForMatrix- Singularity	See CheckMatrixSingularityMsg on page 6-0 parameter for more information.	
CheckMatrix- SingularityMsg	Select the diagnostic action to take if the Product block detects a singular matrix while inverting one of its inputs in matrix multiplication mode. Set by Division by singular matrix on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.	{'none'} 'warning' 'error'
CheckModelReference- TargetMessage	Select the diagnostic action to take if Simulink software detects a target that needs to be rebuilt. Set by “Never rebuild diagnostic” on the Model Referencing pane of the Configuration Parameters dialog box.	'none' 'warning' {'error'}

Parameter	Description	Values
CheckSSInitialOutputMsg	<p>Enable checking for undefined initial subsystem output.</p> <p>Set by Check undefined subsystem initial output on the Configuration Parameters dialog box.</p>	{ 'on' } 'off'
CloseFcn	<p>Set the close callback function, which can be a command or a variable.</p> <p>Set by Model close function on the Callbacks pane of the Model Properties dialog box.</p> <p>See “Create Model Callbacks” for more information.</p>	{ '' }
CompiledBusType	<p>Return information about whether the signal connected to a port is not a bus, or whether it is a virtual or nonvirtual bus.</p> <p>(Read-only) Get with the <code>get_param</code> command. Specify a port or line handle. See “Display Information About Buses”.</p>	Return values are 'NOT_BUS', VIRTUAL_BUS, and NON_VIRTUAL_BUS
CompiledModelBlockNormalModeVisibility	<p>For a top model that is being simulated or that is in a compiled state, return information about which Model blocks have normal mode visibility enabled.</p>	Return values indicate which Model blocks have normal mode visibility enabled.

Parameter	Description	Values
ConditionallyExecute-Inputs	<p>Enable conditional input branch execution optimization.</p> <p>Set by Conditional input branch execution on the Configuration Parameters dialog box.</p>	{'on'} 'off'
ConsecutiveZCsStepRelTol	<p>Relative tolerance associated with the time difference between zero-crossing events.</p> <p>Set by Time tolerance on the Solver pane of the Configuration Parameters dialog box.</p>	{'10*128*eps'}
ConsistencyChecking	<p>Select the diagnostic action to take if S-functions have continuous sample times, but do not produce consistent results when executed multiple times.</p> <p>Set by Solver data inconsistency on the Solver Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
ContinueFcn	<p>Continue simulation callback.</p> <p>Set by Simulation continue function on the Callbacks pane of the Model Properties dialog box.</p>	{''}

Parameter	Description	Values
CovCompData	<p>If CovHtmlReporting is set to on and CovCumulativeReport is set to on, this parameter specifies cvdata objects containing additional model coverage data to include in the model coverage report.</p> <p>Set by Additional data to include in report (cvdata objects) on the Reporting pane of the Configuration Parameters dialog box.</p>	' '
CovCumulativeReport	<p>If CovHtmlReporting is set to on, this parameter allows the CovCumulativeReport and CovCompData parameters to specify the number of coverage results displayed in the model coverage report.</p> <p>If set to on, the Simulink Coverage™ software displays the coverage results from successive simulations in the report.</p> <p>If set to off, the software displays the coverage results for the last simulation in the report.</p> <p>Set by the Cumulative runs (on) / Last run (off) options on the Reporting pane of the Configuration Parameters dialog box.</p>	'on' {'off'}

Parameter	Description	Values
CovDataFileName	If CovEnable is set to on, specifies the name of the file to which Simulink Coverage saves the coverage data results.	{ '' }
CovCumulativeVarName	If CovSaveCumulativeToWorkspaceVar is set to on, the Simulink Coverage software saves the results of successive simulations in the workspace variable specified by this property.	{ 'covCumulativeData' }
CovEnable	Enables coverage analysis for Simulink Coverage. Set by Enable coverage analysis on the Coverage pane of the Configuration Parameters dialog box.	'on' {'off'}
CovEnableCumulative	Accumulates model coverage results for Simulink Coverage from successive simulations. Set this and CovSaveCumulativeToWorkspaceVar to on to collect model coverage results for multiple simulations in one cvdata object.	'on' {'off'}

Parameter	Description	Values
CovExternalEMLEnable	Enables coverage for any external MATLAB functions that MATLAB functions for code generation call in your model. The functions can be defined in a MATLAB Function block or in a Stateflow chart. Enable this feature by checking MATLAB Files on the Coverage pane of the Configuration Parameters dialog box.	{'on'} 'off'
CovForceBlockReductionOff	If CovForceBlockReductionOff is set to on, the Simulink Coverage software ignores the value of the Simulink Block reduction parameter. The software provides coverage data for every block in the model that collects coverage.	{'on'} 'off'
CovHighlightResults	Enable model coloring for coverage results. Enabled by selecting Display coverage results using model coloring on the Coverage > Results pane of the Configuration Parameters dialog box.	'on' {'off'}

Parameter	Description	Values
CovHTMLOptions	<p>If CovHtmlReporting is set to on, use this parameter to select from a set of display options for the resulting model coverage report.</p> <p>Select these options in the Results Explorer settings after you record coverage for a model.</p>	<p>Character vector of appended character sets separated by a space. HTML options are enabled or disabled through a value of 1 or 0, respectively, in the following character sets (default values shown):</p> <ul style="list-style-type: none"> • '-sRT=1' — Show report • '-sVT=0' — Web view mode • '-aTS=1' — Include each test in the model summary • '-bRG=1' — Produce bar graphs in the model summary • '-bTC=0' — Use two color bar graphs (red, blue) • '-hTR=0' — Display hit/count ratio in the model summary • '-nFC=0' — Exclude fully covered model objects from report • '-nFD=1' — Exclude fully covered model object details from report • '-scm=1' — Include cyclomatic complexity numbers in summary • '-bcm=1' — Include cyclomatic complexity numbers in block details • '-xEV=0' — Filter Stateflow events from report

Parameter	Description	Values
		<ul style="list-style-type: none"> '-xEM=0' — Filter Execution metric from report
CovIncludeTopModel	Option to include the top-level model in the coverage analysis.	{'on'} {'off'}
CovHtmlReporting	<p>Set to on to tell the Simulink Coverage software to create an HTML report containing the coverage data at the end of simulation.</p> <p>Set by Generate report automatically after analysis on the Coverage > Results pane of the Configuration Parameters dialog box.</p>	'on' {'off'}
CovLogicBlockShortCircuit	<p>Enables the option to treat Simulink logic blocks as short-circuited for coverage analysis. Enabled by selecting Treat Simulink logic blocks as short-circuited on the Coverage pane of the Configuration Parameters dialog box.</p>	'on' {'off'}

Parameter	Description	Values
CovMcdcMode	<p>Determines the definition of Modified Condition Decision Coverage (MCDC) to use during coverage analysis.</p> <p>To record model coverage using the masking MCDC definition, setCovMcdcMode to 'Masking'. To record model coverage using the unique-cause MCDC definition, setCovMcdcMode to 'UniqueCause'.</p>	{'Masking'} 'UniqueCause'

Parameter	Description	Values
CovMetricSettings	<p>Selects coverage metrics for a coverage report.</p> <p>Coverage metrics are enabled by selecting the check boxes for individual coverages in the Coverage metrics section of the Coverage pane of the Configuration Parameters dialog box.</p> <p>Enable options 's' and 'w' by selecting Treat Simulink Logic blocks as short-circuited and Warn when unsupported blocks exist in model, respectively, on the Coverage pane of the Configuration Parameters dialog box.</p> <p>Disable option 'e' by selecting Display coverage results using model coloring in the Results Explorer settings after you record coverage for a model.</p> <hr/> <p>Note The metrics and options set by this parameter can also be set by the following parameters:</p> <ul style="list-style-type: none"> • CovHighlightResults • CovLogicBlockShortCircuit • CovMetricStructuralLevel • CovMetricLookupTable 	<p>{ 'dwe' }</p> <p>Each order-independent value enables a coverage metric or option as follows:</p> <ul style="list-style-type: none"> • 'd' — Enable decision coverage • 'c' — Enable condition coverage and decision coverage • 'm' — Enable MCDC coverage, condition coverage, and decision coverage • 't' — Enable lookup table coverage • 'r' — Enable signal range coverage • 'z' — Enable signal size coverage • 'o' — Enable coverage for Simulink Design Verifier blocks • 'i' — Enable saturation on integer overflow coverage • 'b' — Enable relational boundary coverage • 's' — Treat Simulink logic blocks as short-circuited • 'w' — Warn when unsupported blocks exist in model • 'e' — Eliminate model coloring for coverage results

Parameter	Description	Values
	<ul style="list-style-type: none"> • CovMetricSignalRange • CovMetricSignalSize • CovMetricObjectiveConstraint • CovMetricSaturateOnIntegerOverflow • CovMetricRelationalBoundary • CovUnsupportedBlockWarning 	
CovMetricLookupTable	Enable lookup table coverage. Enabled by selecting Lookup Table in the Coverage metrics section of the Coverage pane of the Configuration Parameters dialog box.	'on' {'off'}
CovMetricObjectiveConstraint	Enable Simulink Design Verifier objectives and constraints coverage. Enabled by selecting Objectives and Constraints in the Coverage metrics section of the Coverage pane of the Configuration Parameters dialog box.	'on' {'off'}
CovMetricRelationalBoundary	Enable relational boundary coverage. Enabled by selecting Relational Boundary in the Coverage metrics section of the Coverage pane of the Configuration Parameters dialog box.	'on' {'off'}

Parameter	Description	Values
CovMetricSaturateOnIntegerOverflow	Enable saturate on integer overflow coverage. Enabled by selecting Saturate on Integer Overflow in the Coverage metrics section of the Coverage pane of the Configuration Parameters dialog box.	'on' {'off'}
CovMetricSignalRange	Enable signal range coverage. Enabled by selecting Signal Range in the Coverage metrics section of the Coverage pane of the Configuration Parameters dialog box.	'on' {'off'}
CovMetricSignalSize	Enable signal size coverage. Enabled by selecting Signal Size in the Coverage metrics section of the Coverage pane of the Configuration Parameters dialog box.	'on' {'off'}
CovMetricStructuralLevel	Define the level of structural coverage. Set by Structural coverage level on the Coverage pane of the Configuration Parameters dialog box.	'BlockExecution' {'Decision'} 'ConditionDecision' 'MCDC'

Parameter	Description	Values
CovModelRefEnable	<p>If CovModelRefEnable is set to on or all, the Simulink Coverage software generates coverage data for all referenced models. If CovModelRefEnable is set to filtered, coverage data is collected for all referenced models except those specified by the parameter CovModelRefExcluded.</p> <p>Set by Coverage for referenced models on the Coverage pane of the Configuration Parameters dialog box.</p>	'on' {'off'} 'all' 'filtered'
CovModelRefExcluded	<p>If CovModelRefEnable is set to filtered, this parameter stores a comma-separated list of referenced models for which coverage is disabled.</p> <p>Set by selecting Coverage for referenced models on the Coverage pane of the Configuration Parameters dialog box and then clicking Select Models.</p>	{' '}

Parameter	Description	Values
CovNameIncrementing	<p>If CovSaveSingleToWorkspaceVar is set to on, setting CovNameIncrementing to on causes the Simulink Coverage software to append numerals to the workspace variable names for results so that earlier results are not overwritten (for example, covdata1, covdata2, etc.)</p> <p>Set by Increment variable name with each simulation below the selected Save last run in workspace variable check box on the Coverage > Results pane of the Configuration Parameters dialog box.</p>	'on' {'off'}
CovOutputDir	<p>If CovEnable is set to on, specifies the directory in which Simulink Coverage saves the coverage output files.</p>	{' '}
CovPath	<p>Model path of the subsystem for which the Simulink Coverage software gathers and reports coverage data.</p> <p>Set by selecting Subsystem on the Coverage pane of the Configuration Parameters dialog box and then clicking Select Subsystem.</p>	{'/'}

Parameter	Description	Values
CovReportOnPause	Specifies that when you pause during simulation, the model coverage report appears in updated form, with coverage results up to the current pause or stop time.	{'on'} 'off'
CovSaveCumulativeTo-WorkspaceVar	If set to on, the Simulink Coverage software accumulates and saves the results of successive simulations in the workspace variable specified by CovCumulativeVarName.	'on' {'off'}
CovSaveName	<p>If CovSaveSingleToWorkspaceVar is set to on, the Simulink Coverage software saves the results of the last simulation run in the workspace variable specified by this property.</p> <p>Set by cvdata object name below the selected Save last run in workspace variable check box on the Coverage > Results pane of the Configuration Parameters dialog box.</p>	{'covdata'}

Parameter	Description	Values
CovSaveSingleTo-WorkspaceVar	<p>If set to on, the Simulink Coverage software saves the results of the last simulation run in the workspace variable specified by CovSaveName.</p> <p>Set by Save last run in workspace variable on the Coverage > Results pane of the Configuration Parameters dialog box.</p>	'on' {'off'}
CovScope	<p>Sets the scope of analysis for coverage recording. Set by the Scope of analysis section of the Coverage pane in the Configuration Parameters dialog box.</p>	{'EntireSystem'} 'ReferencedModels' 'Subsystem'
CovSFcnEnable	<p>Enables coverage for C/C++ S-Function blocks in your model. Enable this feature by checking C/C++ S-Functions on the Coverage pane of the Configuration Parameters dialog box. For more information, see “Coverage for C and C++ S-Functions” (Simulink Coverage).</p>	'on' {'off'}
CovShowResultsExplorer	<p>Option to shows the results explorer after simulation. Enabled by selecting Show Results Explorer on the Coverage > Results pane of the Configuration Parameters dialog box.</p>	{'on'} 'off'

Parameter	Description	Values
CovUnsupportedBlockWarning	Warn when unsupported blocks exist in model. Enabled by selecting Warn when unsupported blocks exist in model on the Coverage pane of the Configuration Parameters dialog box.	'on' {'off'}
Created	Date and time model was created. Set by Created on on the History pane of the Model Properties dialog box. See “Model Information and History” for more information.	character vector
Creator	Name of model creator. Set by Created by on the History pane of the Model Properties dialog box. See “Model Information and History” for more information.	character vector
CurrentBlock	For internal use.	
CurrentOutputPort	For internal use.	

Parameter	Description	Values
DataDictionary	<p>Simulink data dictionary to which this model is linked.</p> <p>Set by Data Dictionary and Base Workspace on the Data pane of the Model Properties dialog box.</p> <p>For basic information about data dictionaries, see “What Is a Data Dictionary?”. To use this parameter programmatically, see “Store Data in Dictionary Programmatically”.</p>	{ '' }
DataLoggingOverride	<p>A Simulink.SimulationData.ModelLoggingInfo object that specifies the signal logging override settings for a model.</p> <p>See “Override Signal Logging Settings”.</p>	Simulink.SimulationData.ModelLoggingInfo — { 'OverrideSignals' } 'LogAllAsSpecifiedInModel'
DatasetSignalFormat	<p>Format for logged Dataset leaf elements.</p> <p>For details, see “Dataset signal format”.</p>	'timetable' { 'timeseries' }
DataTransfer	<p>A Simulink.GlobalDataTransfer object that configures data transfers for models configured for concurrent execution.</p>	'on' { 'off' }

Parameter	Description	Values
DataTypeOverride	<p>Specifies data type used to override fixed-point data types.</p> <p>Set by Data type override on the Fixed-Point Tool.</p>	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'
Decimation	<p>Specify that Simulink software output only every N points, where N is the specified decimation factor.</p> <p>Set by “Decimation” on the Data Import/Export pane of the Configuration Parameters dialog box.</p>	{'1'}
DefaultParameterBehavior	<p>Enable inlining of block parameters in generated code.</p> <p>Set by Default parameter behavior on the Optimization > Signals and Parameters pane of the Configuration Parameters dialog box. For more information, see Default parameter behavior.</p>	'Inlined' {'Tunable'}
DefaultAnnotationFontName DefaultBlockFontName DefaultLineFontName	<p>Name of font to use for new annotation text, block text, or signal line labels and on existing annotations, block names, or signal lines whose FontName property is set to 'auto'.</p> <p>Set with Diagram > Format > Font Styles for Model dialog box.</p>	character vector

Parameter	Description	Values
DefaultAnnotationFontSize DefaultBlockFontSize DefaultLineFontSize	Size of font to use for new annotation text, block text, or signal line labels and on existing annotations, blocks, or signal lines whose <code>FontSize</code> property is set to <code>-1</code> . Set with Diagram > Format > Font Styles for Model dialog box.	positive integer
DefaultAnnotationFontAngle DefaultBlockFontAngle DefaultLineFontAngle	Angle of font for new annotation text, block text, or signal line labels and on existing annotations, blocks, or signal lines whose <code>FontAngle</code> property is set to <code>'auto'</code> . Set with Diagram > Format > Font Styles for Model dialog box.	{'normal'} 'italic'
DefaultAnnotationFontWeight DefaultBlockFontWeight DefaultLineFontWeight	Weight of font for new annotation text, block text, or signal line labels and on existing annotations, blocks, or signal lines whose <code>FontWeight</code> property is set to <code>'auto'</code> . Set with Diagram > Format > Font Styles for Model dialog box.	{'normal'} 'bold'

Parameter	Description	Values
DefaultUnderspecifiedData Type	Specify data type to use if Simulink cannot infer the type of a signal during data type propagation. Set by “ Default for underspecified data type ” on the Optimization pane of the Configuration Parameters dialog box.	{'double'} 'single'
DeleteChildFcn	Delete child callback function. Created on the Callbacks pane of the Block Properties dialog box. See “Specify Block Callbacks” for more information.	{''}
Description	Description of this model. Set by Model description on the Description pane of the Model Properties dialog box.	{''}
Dirty	If the parameter is on, the model has unsaved changes.	'on' {'off'}
DiscreteInherit- ContinuousMsg	For internal use.	
DisplayBdSearchResults	For internal use.	
DisplayBlockIO	For internal use.	
DisplayCallgraph- Dominators	For internal use	
DisplayCompileStats	For internal use.	
DisplayCondInputTree	For internal use.	
DisplayCondStIdTree	For internal use.	

Parameter	Description	Values
DisplayErrorDirections	For internal use.	
DisplayInvisibleSources	For internal use.	
DisplaySortedLists	For internal use.	
DisplayVectorAnd-FunctionCounts	For internal use.	
DisplayVect-PropagationResults	For internal use.	
ExecutionContextIcon	<p>Show execution context bars on conditional subsystems that do not propagate execution context across the subsystem boundaries.</p> <p>In the Simulink Editor, set by Execution Context Indicator on the Display > Signals & Ports menu.</p>	'on' {'off'}
ExplicitPartitioning	Specifies whether or not to manually map tasks (explicit mapping) or use the rate-based tasks.	'on' {'off'}
ExpressionFolding	<p>Enables expression folding.</p> <p>Set by Eliminate superfluous local variables (Expression folding) on the Configuration Parameters dialog box.</p>	{'on'} 'off'

Parameter	Description	Values
ExternalInput	<p>Names of MATLAB workspace variables used to designate data and times to be loaded from the workspace.</p> <p>Set by the Input field on the Data Import/Export pane of the Configuration Parameters dialog box.</p>	{'t, u'}
ExtMode...	<p>Parameters whose names start with ExtMode apply to Simulink External Mode.</p> <p>For more information, see External Mode (Simulink Coder).</p>	
ExtrapolationOrder	<p>Extrapolation order of the ode14x implicit fixed-step solver.</p> <p>Set by Extrapolation order on the Solver pane of the Configuration Parameters dialog box.</p>	integer — 1 2 3 {4}
FastRestart	<p>Enable or disable fast restart mode.</p> <p>In the Simulink Editor toolbar, click the Fast restart button on or off.</p>	{'on'} 'off'

Parameter	Description	Values
FcnCallInpInside-ContextMsg	<p>Specifies diagnostic action to take when Simulink software must compute any function-call subsystem inputs directly or indirectly during execution of a call to a function-call subsystem.</p> <p>Set by Context-dependent inputs on the Connectivity Diagnostics pane of the Configuration Parameters dialog box.</p>	{'Error'} 'Warning'
FileName	For internal use.	
FinalStateName	<p>Names of final states to save to the workspace after a simulation ends.</p> <p>Set by the Final states field on the Data Import/Export pane of the Configuration Parameters dialog box.</p>	{'xFinal'}
FixedStep	<p>Fixed-step size.</p> <p>Set by Fixed step size (fundamental sample time) on the Solver pane of the Configuration Parameters dialog box.</p>	{'auto'}

Parameter	Description	Values
FixptConstOverflowMsg	<p>Specifies diagnostic action to take when a fixed-point constant overflow occurs during simulation.</p> <p>Set by Detect overflow on the Type Conversion Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
FixptConstPrecisionLossMsg	<p>Specifies diagnostic action to take when a fixed-point constant precision loss occurs during simulation.</p> <p>Set by Detect precision loss on the Type Conversion Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
FixptConstUnderflowMsg	<p>Specifies diagnostic action to take when a fixed-point constant underflow occurs during simulation.</p> <p>Set by Detect underflow on the Type Conversion Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
FixPtInfo	For internal use.	
FollowLinksWhenOpeningFromGotoBlocks	Specifies whether to search for Goto tags in libraries referenced by the model when opening the From block dialog box.	'on' {'off'}
ForceArrayBoundsChecking	For internal use.	

Parameter	Description	Values
ForceConsistencyChecking	For internal use.	
ForceModelCoverage	For internal use.	
ForwardingTable	Specifies the forwarding table for this library. See “Create Forwarding Table” for more information.	{{'old_path_1', 'new_path_1'} ... {'old_path_n', 'new_path_n'}}
ForwardingTableString	For internal use.	
GeneratePreprocessorConditionals	When generating code for an ERT target, this parameter determines whether variant choices are enclosed within C preprocessor conditional statements (<code>#if</code>). When you select this option, Simulink analyzes all variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness of all variant choices.	{'off'} 'on'
GridSpacing	Has no effect in Simulink Editor. This parameter will be removed in a future release.	integer — {20}
Handle	Handle of the block diagram for this model.	double
HardwareBoard	Select the type of hardware on which to run your model. Set by “ Hardware board ” on the Hardware Implementation pane of the Configuration Parameters dialog box.	{'none'}

Parameter	Description	Values
HideAutomaticNames	Hides block names given automatically by the Simulink Editor. See “Hide or Display Block Names”.	{'on'} 'off'
HiliteAncestors	For internal use.	
IgnoreBidirectionalLines	For internal use.	
IgnoredZcDiagnostic	Control diagnostic messages related to zero-crossings that are being ignored.	'none' {'warning'} 'error'
InheritedTsInSrcMsg	<p>Message behavior when the sample time is inherited.</p> <p>Set by Source block specifies -1 sample time on the Sample Time Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
InitFcn	<p>Function that is called when this model is first compiled for simulation.</p> <p>Set by Model initialization function on the Callbacks pane of the Model Properties dialog box.</p> <p>See “Create Model Callbacks” for more information.</p>	{''}
InitialState	<p>Initial state name or values.</p> <p>Set by the Initial state field on the Data Import/Export pane of the Configuration Parameters dialog box.</p>	variable or vector — {'xInitial'}

Parameter	Description	Values
InitialStep	Initial step size. Set by Initial step size on the Solver pane of the Configuration Parameters dialog box.	{'auto'}
InitialStep	Initial step size. Set by Initial step size on the Solver pane of the Configuration Parameters dialog box.	{'auto'}
InitialStep	Initial step size. Set by Initial step size on the Solver pane of the Configuration Parameters dialog box.	{'auto'}

Parameter	Description	Values
InitInArrayFormatMsg	<p>Message behavior when the initial state is an array. You set with the initial state with the Initial state configuration parameter.</p> <p>Avoid using an array for the initial state. If the order of the elements in the array does not match the order in which blocks initialize, the simulation can produce unexpected results. To promote deterministic simulation results, use the default setting or set the diagnostic to <code>error</code>.</p> <p>Alternatively, you can set the message behavior using the “InitInArrayFormatMsg” on the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
InsertRTBMode	<p>Control whether the Rate Transition block parameter Ensure deterministic data transfer (maximum delay) is set for auto-inserted Rate Transition blocks.</p> <p>Set by Deterministic data transfer on the Solver pane of the Configuration Parameters dialog box.</p>	'Always' {'Whenever possible'} 'Never (minimum delay)'

Parameter	Description	Values
InspectSignalLogs	<p>Enable Simulink software to display logged signals in the Simulation Data Inspector tool at the end of a simulation or whenever you pause the simulation.</p> <p>Set by “Record logged workspace data in Simulation Data Inspector” on the Data Import/Export pane of the Configuration Parameters dialog box.</p>	'on' {'off'}
InstrumentedSignals	<p>Returns a <code>Simulink.HMI.InstrumentedSignals</code> object with the properties of model name and the number of signals that are marked for streaming. From this object, you can control signal streaming using the block path and output port index.</p>	object — <code>Simulink.HMI.InstrumentedSignals</code>
Int32ToFloatConvMsg	<p>Specify message behavior when a 32-bit integer is converted to a single-precision float.</p> <p>Set by 32-bit integer to single precision float conversion on the Type Conversion Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' {'warning'}

Parameter	Description	Values
IntegerOverflowMsg	Specify message behavior when an integer overflow occurs. Set by “Wrap on overflow” in the Signals section on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.	'none' {'warning'} 'error'
IntegerSaturationMsg	Specify message behavior when an integer saturation occurs. Set by “Saturate on overflow” in the Signals section on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.	'none' {'warning'} 'error'
InvalidFcnCallConnMsg	Specify message behavior when an invalid function-call connection exists. Set by Invalid function-call connection on the Connectivity Diagnostics pane of the Configuration Parameters dialog box.	'warning' {'error'}
Jacobian	For internal use.	
LastModifiedBy	User name of the person who last modified this model.	character vector
LastModifiedDate	Date when the model was last saved.	character vector

Parameter	Description	Values
LibraryLinkDisplay	<p>Displays the blocks in the model that are linked or have disabled or modified links.</p> <p>In the Simulink Editor, set by Library Links on the Display menu.</p>	<p>'none' {'disabled'} 'user' 'all'</p> <p>Set to none, does not display the link badge on the block.</p> <p>Set to disabled, displays the disabled link badge on the block.</p> <p>Set to user, displays only links to the user libraries.</p> <p>Set to all, displays all links.</p>
LibraryType	For internal use.	
LifeSpan	<p>Specify how long (in days) an application that contains blocks depending on elapsed or absolute time should be able to execute before timer overflow.</p> <p>Set by Application lifespan (days) on the Optimization pane of the Configuration Parameters dialog box.</p>	{'auto'} any positive, nonzero scalar value
LimitDataPoints	<p>Specify that the number of data points exported to the MATLAB workspace be limited to the number specified.</p> <p>Set by the Limit data points configuration parameter</p>	{'on'} 'off'
LinearizationMsg	For internal use.	
Lines	For internal use.	

Parameter	Description	Values
LoadExternalInput	Load input from workspace. Set by the Input check box on the Data Import/Export pane of the Configuration Parameters dialog box.	'on' {'off'}
LoadInitialState	Load initial state from workspace. Set by the Initial state check box on the Data Import/Export pane of the Configuration Parameters dialog box.	'on' {'off'}
Location	For internal use.	
Lock	Lock or unlock a block library. Setting this parameter to on prevents a user from inadvertently changing a library.	'on' {'off'}
LockLinksToLibrary	Lock or unlock links to a library. Setting this parameter to on prevents a user from inadvertently changing linked blocks from the Simulink Editor.	'on' {'off'}
LoggingFileName	Use when you enable LoggingToFile parameter for logging to persistent storage. Specify the destination MAT-file for data logging.	{'out.mat'} Do not use a file name from one locale in a different locale.

Parameter	Description	Values
LoggingToFile	<p>Store logging data that uses Dataset format to persistent storage (MAT-file). Using a Simulink.SimulationData.DatasetRef object to access signal logging and states logging data loads data into the model workspace incrementally. Accessing data for other kinds of logging loads all of the data at once.</p> <p>Use this feature when logging large amounts of data that can cause memory issues. For details, see “Log Data to Persistent Storage”.</p>	'on' {'off'}
MAModelExclusionFile	<p>Specifies the location of the Model Advisor exclusion file.</p> <p>Set by the File Name field on the Model Advisor Exclusion Editor dialog box.</p>	{' '}
MaskedZcDiagnostic	Control diagnostic messages related to zero-crossings that are being masked.	'none' {'warning'} 'error'

Parameter	Description	Values
MaxConsecutiveMinStep	<p>Maximum number of minimum step size violations allowed during simulation. This option appears when the solver type is <code>Variable-step</code> and the solver is an ode one.</p> <p>Set by Number of consecutive min steps on the Solver pane of the Configuration Parameters dialog box.</p>	{'1'}
MaxConsecutiveZCs	<p>Maximum number of consecutive zero crossings allowed during simulation. This option appears when the solver type is <code>Variable-step</code> and the solver is an ode one.</p> <p>Set by Number of consecutive zero crossings on the Solver pane of the Configuration Parameters dialog box.</p>	{'1000'}
MaxConsecutiveZCsMsg	<p>Specifies diagnostic action to take when Simulink software detects the maximum number of consecutive zero crossings allowed. This option appears when the solver type is <code>Variable-step</code> and the solver is an ode one.</p> <p>Set by Consecutive zero crossings violation on the Solver Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' 'warning' 'error'

Parameter	Description	Values
MaxDataPoints	<p>Maximum number of output data points to save.</p> <p>Set by the Limit data points to last field on the Data Import/Export pane of the Configuration Parameters dialog box.</p>	{ '1000' }
MaxMDLFileLineLength	<p>Controls the line lengths in the model file. Use this to avoid line-wrapping, which can be important for source control tools.</p> <p>Specifies the maximum length in bytes, which may differ from the number of characters in Japanese, and is different from the number of columns when tabs are present.</p>	<p>integer — -1 (unlimited) or >= 80.</p> <p>Default is 120.</p>
MaxNumMinSteps	Maximum number of times the solver uses the minimum step size.	{ '-1' }
MaxOrder	<p>Maximum order for ode15s.</p> <p>Set by Maximum order on the Solver pane of the Configuration Parameters dialog box.</p>	'1' '2' '3' '4' { '5' }
MaxStep	<p>Maximum step size.</p> <p>Set by Max step size on the Solver pane of the Configuration Parameters dialog box.</p>	{ 'auto' }
MdlSubVersion	For internal use	

Parameter	Description	Values
MergeDetectMultiDriving-BlocksExec	<p>Select the diagnostic action to take when the software detects a Merge block with more than one driving block executing at the same time step.</p> <p>Set by Detect multiple driving blocks executing at the same time step on the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
Metadata	<p>Names and attributes of arbitrary data associated with the model. To extract this metadata structure without needing to load the model, use the method <code>Simulink.MDLInfo.getMetadata</code>.</p>	Structure. Fields can be character vectors, numeric matrices of type "double", or more structures.
MinMaxOverflow-ArchiveData	For internal use	
MinMaxOverflow-ArchiveMode	<p>Logging type for fixed-point logging.</p> <p>Set by Overwrite or merge model simulation results in the Fixed-Point Tool.</p>	{'Overwrite'} 'Merge'
MinMaxOverflowLogging	<p>Setting for fixed-point logging.</p> <p>Set by Fixed-point instrumentation mode in the Fixed-Point Tool.</p>	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'ForceOff'

Parameter	Description	Values
MinStep	<p>Minimum step size for the solver.</p> <p>Set by Min step size on the Solver pane of the Configuration Parameters dialog box.</p>	{'auto'}
MinStepSizeMsg	<p>Message shown when minimum step size is violated.</p> <p>Set by Min step size violation on the Solver Diagnostics pane of the Configuration Parameters dialog box.</p>	{'warning'} {'error'}
ModelBlockNormalModeVisibility	<p>Use with <code>set_param</code> to set normal mode visibility on for the specified Model blocks.</p> <p>You can set this parameter with the Model Block Normal Mode Visibility dialog box. For details, see “Specify the Instance Having Normal Mode Visibility”.</p>	<p>With <code>set_param</code>, use an array of <code>Simulink.BlockPath</code> objects or cell array of cell arrays of character vectors of paths to blocks or models.</p> <p>With <code>set_param</code>, an empty array specifies to use the Simulink default selection for the instance to have normal mode visibility enabled.</p>
ModelBlockNormalModeVisibilityBlockPath	<p>Return information about which Model blocks have normal mode visibility enabled. Use with a model that you are editing.</p>	<p>Return values indicate which Model blocks have normal mode visibility enabled. See “Simulate Models with Multiple Referenced Model Instances”.</p>
ModelBrowserVisibility	<p>Show the Model Browser.</p> <p>In the Simulink Editor, set by Model Browser on the View menu.</p>	'on' {'off'}

Parameter	Description	Values
ModelBrowserWidth	Width of the Model Browser pane in the model window. To display the Model Browser pane, see the ModelBrowserVisibility on page 6-0 parameter.	integer — {200}
ModelDataFile	For internal use.	{ '' }
ModelDependencies	List of model dependencies. Set by Model dependencies on the Model Referencing pane of the Configuration Parameters dialog box.	{ '' }

Parameter	Description	Values
ModelReferenceCS-MismatchMessage	<p>This parameter is maintained for compatibility purposes only. Do not use this parameter.</p> <p>You can use the Model Advisor to identify models referenced in Accelerator mode for which Simulink ignores certain configuration parameters.</p> <ol style="list-style-type: none"> 1 In the Simulink Editor, select Analysis > Model Advisor. 2 Select By Task. 3 Run the Check diagnostic settings ignored during accelerated model reference simulation check. <p>For more information, see “Diagnostic Configuration Parameters Ignored in Accelerator Mode”.</p>	<p>{'none'} 'warning' 'error'</p> <p>Simulink ignores this parameter if you set it to warning or error.</p>
ModelReferenceData-LoggingMessage	<p>Message shown when there is unsupported data logging.</p> <p>Set by Unsupported data logging on the Model Referencing Diagnostics pane of the Configuration Parameters dialog box.</p>	<p>'none' {'warning'} 'error'</p>

Parameter	Description	Values
ModelReferenceExtra-NoncontSigs	<p>Specifies diagnostic action to take when a discrete signal appears to pass through a Model block to the input of a block with continuous states.</p> <p>Set by Extraneous discrete derivative signals on the Solver Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' 'warning' {'error'}
ModelReferenceIO-MismatchMessage	<p>Message shown when there is a port and parameter mismatch.</p> <p>Set by Port and parameter mismatch on the Model Referencing Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
ModelReferenceIOMsg	<p>Message shown when there is an invalid root Inport or Outport block connection.</p> <p>Set by Invalid root Inport/Outport block connection on the Model Referencing Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'

Parameter	Description	Values
ModelReferenceMin- AlgLoopOccurrences	<p>Toggles the minimization of algebraic loop occurrences.</p> <p>Set by Minimize algebraic loop occurrences on the Model Referencing pane of the Configuration Parameters dialog box.</p>	'on' {'off'}
ModelReferenceNum- InstancesAllowed	<p>Total number of model reference instances allowed per top model.</p> <p>Set by Total number of instances allowed per top model on the Model Referencing pane of the Configuration Parameters dialog box.</p>	'Zero' 'Single' {'Multi'}
ModelReferencePass- RootInputsByReference	<p>Toggles the passing of scalar root inputs by value.</p> <p>Set by “Pass fixed-size scalar root inputs by value for code generation” on the Model Referencing pane of the Configuration Parameters dialog box.</p>	{'on'} 'off'
ModelReferenceSim- TargetVerbose	This parameter is deprecated and has no effect. Use <code>AccelVerboseBuild</code> instead.	
ModelReferenceSymbol- NameMessage	For referenced models, specifies diagnostic action to take when the Maximum identifier length does not provide enough space to make global identifiers unique across models.	'none' {'warning'} 'error'

Parameter	Description	Values
ModelReferenceTargetType	For internal use.	
ModelReferenceVersionMismatchMessage	<p>Message shown when there is a model block version mismatch.</p> <p>Set by Model block version mismatch on the Model Referencing Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
ModelVersion	Version number of model.	{'1.1'}
ModelVersionFormat	<p>Format of model's version number.</p> <p>Set by Model version on the History pane of the Model Properties dialog box.</p> <p>See “Model Information and History” for more information.</p>	{'1.%<AutoIncrement: 0>'}
ModelWorkspace	References this model's model workspace object.	an instance of the Simulink.ModelWorkspace class
ModifiedByFormat	<p>Format for the display of last modifier.</p> <p>Set by Last saved by on the History pane of the Model Properties dialog box.</p> <p>See “Model Information and History” for more information.</p> <p>Can also be set by Last saved by on the Model history field on the History pane of the Model Explorer.</p>	{'%<Auto>'}

Parameter	Description	Values
ModifiedComment	Field for user comments.	{ '' }
ModifiedDateFormat	<p>Format used to generate the value of the <code>LastModifiedDate</code> parameter.</p> <p>Set by Last saved on on the History pane of the Model Properties dialog box.</p> <p>See “Model Information and History” for more information.</p>	{ '%<Auto>' }
ModifiedHistory	<p>Area for keeping notes about the history of the model.</p> <p>Set by the Model history field on the History pane of the Model Properties dialog box.</p> <p>See “Model Information and History” for more information.</p> <p>Can also be set by the Model history field on the History pane of the Model Explorer.</p>	{ '' }
MultiTaskCondExecSysMsg	<p>Select the diagnostic action to take if Simulink software detects a subsystem that might cause data corruption or nondeterministic behavior.</p> <p>Set by Multitask conditionally executed subsystem on the Sample Time Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' 'warning' {'error'}

Parameter	Description	Values
MultiTaskDSMMsg	<p>Specifies diagnostic action to take when one task reads data from a Data Store Memory block to which another task writes data.</p> <p>Set by Multitask data store on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' 'warning' {'error'}
MultiTaskRateTransMsg	<p>Specifies diagnostic action to take when an invalid rate transition takes place between two blocks operating in multitasking mode.</p> <p>Set by Multitask rate transition on the Sample Time Diagnostics pane of the Configuration Parameters dialog box.</p>	'warning' {'error'}
Name	Model name.	character vector
NonBusSignalsTreatedAsBuses	<p>Detect when Simulink implicitly converts a non-bus signal to a bus signal to support connecting the signal to a block expecting a bus signal.</p> <p>“Non-bus signals treated as bus signals” on the Diagnostics > Connectivity pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'

Parameter	Description	Values
NumberNewtonIterations	<p>Number of Newton's method iterations performed by the ode14x implicit fixed-step solver.</p> <p>Set by Number Newton's iterations on the Solver pane of the Configuration Parameters dialog box.</p>	integer — {1}
NumStatesForStiffnessChecking	<p>Threshold value of number of continuous states in model for stiffness calculation. If the number of continuous states in the model exceeds the NumStatesForStiffnessChecking value, auto solver uses ode15s.</p> <p>For more information, see “Use Auto Solver to Select a Solver”.</p>	{' '}
ObjectParameters	Names and attributes of model parameters.	structure
Open	For internal use.	
OptimizeBlockIOStorage	<p>Enables signal storage reuse optimization.</p> <p>Set by Signal storage reuse on the Configuration Parameters dialog box.</p>	{'on'} 'off'

Parameter	Description	Values
OutputOption	<p>Time step output options for variable-step solvers.</p> <p>Set by Output options parameter under Configuration Parameters > Data Import/Export > Additional parameters.</p>	'AdditionalOutputTimes' {'RefineOutputTimes'} 'SpecifiedOutputTimes'
OutputSaveName	<p>Workspace variable to store the model outputs.</p> <p>Set by the Output field on the Data Import/Export pane of the Configuration Parameters dialog box.</p>	{'yout'}
OutputTimes	<p>Output times set when Set by Output options parameter under Configuration Parameters > Data Import/Export > Additional parameters is set to Produce additional output.</p> <p>Set using the Output times parameter.</p>	{'[]'}
PaperOrientation	Printing paper orientation.	'portrait' {'landscape'}
PaperPosition	When PaperPositionMode is set to manual, this parameter determines the position and size of a diagram on paper and the size of the diagram exported as a graphic file in the units specified by PaperUnits.	vector — [left, bottom, width, height]

Parameter	Description	Values
PaperPositionMode	<p>Paper position mode.</p> <ul style="list-style-type: none"> • auto <p>When printing, Simulink software sizes the diagram to fit the printed page. When exporting a diagram as a graphic image, Simulink software sizes the exported image to be the same size as the diagram's normal size on screen.</p> <ul style="list-style-type: none"> • manual <p>When printing, Simulink software positions and sizes the diagram on the page as indicated by PaperPosition. When exporting a diagram as a graphic image, Simulink software sizes the exported graphic to have the height and width specified by PaperPosition.</p> <ul style="list-style-type: none"> • tiled <p>Enables tiled printing.</p> <p>See “Tiled Printing” for more information.</p>	<p>{'auto'} 'manual' 'tiled'</p>
PaperSize	<p>Size of PaperType in PaperUnits.</p>	<p>vector — [width height] (read only)</p>

Parameter	Description	Values
PaperType	Printing paper type.	'usletter' 'uslegal' 'a0' 'a1' 'a2' 'a3' 'a4' 'a5' 'b0' 'b1' 'b2' 'b3' 'b4' 'b5' 'arch-A' 'arch-B' 'arch-C' 'arch-D' 'arch-E' 'A' 'B' 'C' 'D' 'E' 'tabloid'
PaperUnits	Printing paper size units.	'normalized' {'inches'} 'centimeters' 'points'

Parameter	Description	Values
ParallelModelReferenceErrorOnInvalidPool	<p>Specify if you want the Simulink software to perform a consistency check on the parallel pool before starting a parallel build.</p> <p>If you set the parameter to <code>on</code>, the client and the remote workers must meet the following criteria for the parallel build to initiate:</p> <ul style="list-style-type: none"> • The parallel pool is open. • The pool is <code>spmd</code> compatible. • The platform is consistent between workers and client. • The workers have a Simulink Real-Time license. • A common compiler exists across workers and client. <p>If you set the parameter to <code>off</code>, the software displays a warning for the first condition that fails and then performs a sequential build.</p>	{'on'} 'off'

Parameter	Description	Values
ParameterArgumentNames	<p>List of parameters used as arguments when this model is called as a reference.</p> <p>Set by checking the Argument column for variables in the model workspace of the referenced model. See “Parameterize Instances of a Reusable Referenced Model”.</p>	{' '}
ParameterDowncastMsg	<p>Specifies diagnostic action to take when a parameter downcast occurs during simulation.</p> <p>Set by Detect downcast on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' 'warning' {'error'}
ParameterOverflowMsg	<p>Specifies diagnostic action to take when a parameter overflow occurs during simulation.</p> <p>Set by Detect overflow on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' 'warning' {'error'}

Parameter	Description	Values
ParameterPrecision-LossMsg	<p>Specifies diagnostic action to take when parameter precision loss occurs during simulation.</p> <p>Set by Detect precision loss on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
ParameterTunabilityLossMsg	<p>Specifies diagnostic action to take when a parameter cannot be tuned because it uses unsupported functions or operators.</p> <p>Set by Detect loss of tunability on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
ParameterUnderflowMsg	<p>Specifies diagnostic action to take when a parameter underflow occurs during simulation.</p> <p>Set by Detect underflow on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
ParamWorkspaceSource	For internal use.	
Parent	Name of the model or subsystem that owns this object. The value of this parameter for a model is an empty character vector.	{''}

Parameter	Description	Values
Pause	<p>Pause simulation callback.</p> <p>Set by Simulation pause function on the Callbacks pane of the Model Properties dialog box.</p>	{ '' }
PortDataTypeDisplayFormat	<p>When you display port data types in a model by selecting Display > Signals and Ports > Port Data Types, choose whether to display data type aliases, base data types, or both.</p> <p>In the Simulink Editor, set by Display > Signals and Ports > Port Data Type Display Format.</p>	{ 'AliasTypeOnly' 'BaseTypeOnly' 'BaseAndAliasTypes' }
PositivePriorityOrder	<p>Choose the appropriate priority ordering for the real-time system targeted by this model. The Simulink Coder software uses this information to implement asynchronous data transfers.</p> <p>Set by Higher priority value indicates higher task priority on the Solver pane of the Configuration Parameters dialog box.</p>	'on' {'off'}

Parameter	Description	Values
PostLoadFcn	<p>Function invoked just after this model is loaded.</p> <p>Set by Model post-load function on the Callbacks pane of the Model Properties dialog box.</p> <p>See “Create Model Callbacks” for more information.</p>	{ ' ' }
PostSaveFcn	<p>Function invoked just after this model is saved to disk. Not executed for blocks inside library links.</p> <p>Set by Model post-save function on the Callbacks pane of the Model Properties dialog box.</p> <p>See “Create Model Callbacks” for more information.</p>	{ ' ' }
PreLoadFcn	<p>Preload callback.</p> <p>Set by Model pre-load function on the Callbacks pane of the Model Properties dialog box.</p> <p>See “Create Model Callbacks” for more information.</p>	{ ' ' }

Parameter	Description	Values
PreSaveFcn	<p>Function invoked just before this model is saved to disk. Not executed for blocks inside library links, except when you are breaking the link, e.g., with <code>save_system(A, B, 'BreakUserLinks', 'on')</code>.</p> <p>Set by Model pre-save function on the Callbacks pane of the Model Properties dialog box.</p> <p>See “Create Model Callbacks” for more information.</p>	{ '' }
ProdBitPerChar	<p>Describes the length in bits of the C <code>char</code> data type supported by the hardware board to be used by this model.</p> <p>Set by Number of bits: char on the Hardware Implementation pane of the Configuration Parameters dialog box.</p>	integer — { 8 }
ProdBitPerInt	<p>Describes the length in bits of the C <code>int</code> data type supported by the hardware board to be used by this model.</p> <p>Set by Number of bits: int on the Hardware Implementation pane of the Configuration Parameters dialog box.</p>	integer — { 32 }

Parameter	Description	Values
ProdBitPerLong	<p>Describes the length in bits of the C long data type supported by the hardware board to be used by this model.</p> <p>Set by Number of bits: long on the Hardware Implementation pane of the Configuration Parameters dialog box.</p>	integer — {32}
ProdBitPerLongLong	<p>Describes the length in bits of the C long data type supported by the hardware board to be used by this model.</p> <p>Set by “Number of bits: long long” on the Hardware Implementation pane of the Configuration Parameters dialog box.</p> <p>The value of this parameter must be greater than or equal to the value of ProdBitPerLong.</p>	integer — {64}
ProdBitPerShort	<p>Describes the length in bits of the C short data type supported by the hardware board to be used by this model.</p> <p>Set by Number of bits: short on the Hardware Implementation pane of the Configuration Parameters dialog box.</p>	integer — {16}

Parameter	Description	Values
ProdEndianness	<p>Describes the significance of the first byte of a data word of the hardware board to be used by this model.</p> <p>Set by Byte ordering on the Hardware Implementation pane of the Configuration Parameters dialog box.</p>	{'Unspecified'} 'LittleEndian' 'BigEndian'
ProdEqTarget	<p>Specifies that the hardware used to test the code generated from this model is the same as the production hardware or has the same characteristics.</p>	{'on'} 'off'
ProdHWDeviceType	<p>Predefined hardware device to specify the C language constraints for your microprocessor.</p> <p>Set by “Device vendor” and Device type on the Hardware Implementation pane of the Configuration Parameters dialog box.</p>	{'Generic->Unspecified (assume 32-bit Generic)'}
ProdIntDivRoundTo	<p>Describes how the C compiler that creates production code for this model rounds the result of dividing one signed integer by another to produce a signed integer quotient.</p> <p>Set by Signed integer division rounds to on the Hardware Implementation pane of the Configuration Parameters dialog box.</p>	'Floor' 'Zero' {'Undefined'}

Parameter	Description	Values
ProdLargestAtomicFloat	<p>Specify the largest floating-point data type that can be atomically loaded and stored on the hardware board.</p> <p>Set by “Largest atomic size: floating-point” on the Hardware Implementation pane of the Configuration Parameters dialog box.</p>	'Float' 'Double' {'None'}
ProdLargestAtomicInteger	<p>Specify the largest integer data type that can be atomically loaded and stored on the hardware board.</p> <p>Set this parameter to 'LongLong' only if the production hardware supports the C long long data type and you have set ProdLongLongMode to 'on'.</p> <p>Set by “Largest atomic size: integer” on the Hardware Implementation pane of the Configuration Parameters dialog box.</p>	{'Char'} 'Short' 'Int' 'Long' 'LongLong'
ProdLongLongMode	<p>Specify that your C compiler supports the C long long data type. Most C99 compilers support long long.</p> <p>Set by “Support long long” on the Hardware Implementation pane of the Configuration Parameters dialog box.</p>	'on' {'off'}

Parameter	Description	Values
ProdShiftRightIntArith	<p>Describes whether the C compiler that creates production code for this model implements a signed integer right shift as an arithmetic right shift.</p> <p>Set by Shift right on a signed integer as arithmetic shift on the Hardware Implementation pane of the Configuration Parameters dialog box.</p>	{'on'} 'off'
ProdWordSize	<p>Describes the word length in bits of the hardware board to be used by this model.</p> <p>Set by Number of bits: native on the Hardware Implementation pane of the Configuration Parameters dialog box.</p>	integer — {32}
Profile	<p>Enables the simulation profiler for this model.</p> <p>In the Simulink Editor, set by Show Profiler Report on the Analysis menu.</p>	'on' {'off'}

Parameter	Description	Values
PropagateSignalLabelsOutOfModel	<p>Pass propagated signal names to output signals of Model block.</p> <p>Set by Propagate all signal labels out of the model on the Model Referencing pane of the Configuration Parameters dialog box.</p> <p>See “Propagate all signal labels out of the model” for more information.</p>	{'on'} 'off'
PropagateVarSize	<p>Select how variable-size signals propagate through referenced models.</p> <p>Set by Propagate sizes of variable-size signals on the Model Referencing pane of the Configuration Parameters dialog box.</p> <p>See “Model Configuration Parameters: Model Referencing” for more information.</p>	'Infer from blocks in model' 'Only when enabling' 'During execution'
ReadBeforeWriteMsg	<p>Specifies diagnostic action to take when the model attempts to read data from a data store before it has stored data at the current time step.</p> <p>Set by Detect read before write on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	{'UseLocalSettings'} 'DisableAll' 'EnableAllAsWarning' 'EnableAllAsError'

Parameter	Description	Values
RecordCoverage	<p>If RecordCoverage is set to on, Simulink collects and reports model coverage data during simulation. The format of this report is controlled by the values of the following parameters:</p> <p>CovCompData</p> <p>CovCumulativeReport</p> <p>CovCumulativeVarName</p> <p>CovHTMLOptions</p> <p>CovHtmlReporting</p> <p>CovMetricSettings</p> <p>CovModelRefEnable</p> <p>CovModelRefExcluded</p> <p>CovNameIncrementing</p> <p>CovPath</p> <p>CovReportOnPause</p> <p>CovSaveCumulativeToWorkSpaceVar</p> <p>CovSaveName</p> <p>CovSaveSingleToWorkspaceVar</p>	'on' {'off'}

Parameter	Description	Values
	<p>If set to <code>off</code>, no model coverage data is collected or reported.</p> <p>Set by Entire System on the Coverage pane of the Configuration Parameters dialog box.</p>	
Refine	<p>Refine factor.</p> <p>Set by Refine factor parameter under parameter under Configuration Parameters > Data Import/Export > Additional parameters.</p>	{ '1' }
RelTol	<p>Relative error tolerance.</p> <p>Set by Relative tolerance on the Solver pane of the Configuration Parameters dialog box.</p>	{ '1e-3' }
RemoveDisableFunc	<p>For model referencing contexts for ERT targets, remove the generated disable functions that cannot be reached from anywhere in the generated code.</p> <p>Set by the “Remove disable function” (Embedded Coder) configuration parameter.</p>	'on' {'off'}

Parameter	Description	Values
RemoveResetFunc	<p>For model referencing contexts for ERT targets, remove the generated reset functions that cannot be reached from anywhere in the generated code.</p> <p>Set by the “Remove reset function” (Embedded Coder) configuration parameter.</p>	{ 'on' } 'off'
ReportName	Name of the associated file for the Report Generator.	{ 'simulink-default.rpt' }
ReqHilite	<p>Highlights all the blocks in the Simulink diagram that have requirements associated with them.</p> <p>In the Simulink Editor, set by Highlight Model on the Analysis > Requirements menu.</p>	'on' { 'off' }
RequirementInfo	For internal use.	
RootOutputRequire- BusObject	<p>Specifies diagnostic action to take when a bus enters a root model Output block for which a bus object has not been specified.</p> <p>Set by Unspecified bus object at root Output block on the Connectivity Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' { 'warning' } 'error'

Parameter	Description	Values
RTPrefix	<p>Specifies diagnostic action to take when Simulink software encounters an object name that begins with <code>rt</code>.</p> <p>Set by "rt" prefix for identifiers on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' 'warning' {'error'}
RTW...	For information about model parameters beginning with RTW, see Configuration Parameters for Simulink Models and Parameter Reference in the Simulink Coder documentation.	
SampleTimeAnnotations	In the Simulink Editor, set by Annotations on the Display > Sample Time menu.	'on' {'off'}
SampleTimeColors	In the Simulink Editor, set by Colors on the Display > Sample Time Display menu.	'on' {'off'}
SampleTimeConstraint	<p>This option appears when the solver type is Fixed-step.</p> <p>Set by Periodic sample time constraint on the Solver pane of the Configuration Parameters dialog box.</p>	{'Unconstrained'} 'STIndependent' 'Specified'

Parameter	Description	Values
SampleTimeProperty	Specifies and assigns priorities to the sample times implemented by the model. This option appears when Periodic sample time constraint is set to Specified. Set by Sample time properties on the Solver pane of the Configuration Parameters dialog box.	Structure containing the fields SampleTime, Offset, and Priority
SavedCharacterEncoding	Specifies the character set used to encode this model. See the <code>slCharacterEncoding</code> command for more information.	character vector
SaveDefaultBlockParams	For internal use.	
SavedSinceLoaded	Indicates whether the model has been saved since it was loaded. 'on' indicates the model has been saved.	'on' 'off'
SaveFinalState	Save final states to workspace. Set by the Final states check box on the Data Import/Export pane of the Configuration Parameters dialog box.	'on' {'off'}
SaveFormat	Format used to save data to the MATLAB workspace. Set by Format on the Data Import/Export pane of the Configuration Parameters dialog box.	{'Dataset'} 'Structure' 'StructureWithTime' 'Array'

Parameter	Description	Values
SaveOutput	<p>Save simulation output to workspace.</p> <p>Set by the Output check box on the Data Import/Export pane of the Configuration Parameters dialog box.</p> <p>Do not use a variable name that is the same as a <code>Simulink.SimulationOutput</code> object function name or property name.</p>	{'on'} 'off'
SaveState	<p>Save states to workspace.</p> <p>Set by the States check box on the Data Import/Export pane of the Configuration Parameters dialog box.</p>	'on' {'off'}
SaveTime	<p>Save simulation time to workspace.</p> <p>Set by the Time check box on the Data Import/Export pane of the Configuration Parameters dialog box.</p> <p>Do not use a variable name that is the same as a <code>Simulink.SimulationOutput</code> object function name or property name.</p>	{'on'} 'off'

Parameter	Description	Values
SaveWithDisabledLinksMsg	<p>Specifies diagnostic action to take when saving a block diagram having disabled library links.</p> <p>Set by Block diagram contains disabled library links on the Saving Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
SaveWithParameterized-LinksMsg	<p>Specifies diagnostic action to take when saving a block diagram having parameterized library links.</p> <p>Set by Block diagram contains parameterized library links on the Saving Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
ScreenColor	<p>Background color of the model window.</p> <p>In the Simulink Editor, set by Canvas Color on the Diagram > Format menu.</p>	'black' {'white'} 'red' 'green' 'blue' 'cyan' 'magenta' 'yellow' 'gray' 'lightBlue' 'orange' 'darkGreen' [r,g,b,a] where r, g, b, and a are the red, green, blue, and alpha values of the color normalized to the range 0.0 to 1.0. The alpha value is ignored.
ScrollbarOffset	For internal use.	

Parameter	Description	Values
SFcnCompatibilityMsg	<p>Specifies diagnostic action to take when S-function upgrades are needed.</p> <p>Set by S-function upgrades needed on the Compatibility Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
SFExecutionAtInitializationDiag	<p>Select the diagnostic action to take when Stateflow detects triggered or enabled charts that are not running at initialization.</p> <p>Set by “Execute-at-Initialization disabled in presence of input events” on the Compatibility Diagnostics pane of the Configuration Parameters dialog box.</p>	

Parameter	Description	Values
SFInvalidInputDataAccess- InChartInitDiag	<p>Select the diagnostic action to take when a chart:</p> <ul style="list-style-type: none"> • Has the <code>ExecuteAtInitialization</code> property set to true • Accesses input data on a default transition or associated state entry actions, which execute at chart initialization <p>Set by Invalid input data access in chart initialization on the Diagnostics > Stateflow pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
SFMachineParented- DataDiag	<p>Select the diagnostic action to take when Stateflow detects machine-parented data that you can replace with chart-parented data of scope Data Store Memory.</p> <p>Set by “Use of machine-parented data instead of Data Store Memory” on the Diagnostics > Stateflow pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'

Parameter	Description	Values
SFNoUnconditionalDefault-TransitionDiag	<p>Select the diagnostic action to take when a chart does not have an unconditional default transition to a state or a junction.</p> <p>Set by No unconditional default transitions on the Diagnostics > Stateflow pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
SFSelfTransitionDiag	<p>Select the diagnostic action to take when you can remove a self-transition on a leaf state.</p> <p>Set by “Self transition on leaf state” on the Diagnostics > Stateflow pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
SFSimEcho	<p>Enables output to appear in the MATLAB Command Window during simulation of a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</p> <p>Set by Echo expressions without semicolons on the Simulation Target pane of the Configuration Parameters dialog box.</p>	{'on'} 'off'

Parameter	Description	Values
SFTemporalDelaySmallerThanSampleTimeDiag	<p>Select the diagnostic action to take when a state or transition absolute time operator uses a time value that is shorter than the sample time for the Stateflow block.</p> <p>Set by “Absolute time temporal value shorter than sampling period” on the Simulation Target pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
SFTransitionActionBeforeConditionDiag	<p>Select the diagnostic action to take when a transition action is specified before a condition action in a transition path containing multiple segmented transitions.</p> <p>Set by “Transition action specified before condition action” on the Diagnostics > Stateflow pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
SFTransitionOutsideNaturalParentDiag	<p>Select the diagnostic action to take when a chart contains a transition that loops outside the parent state or junction.</p> <p>Set by Transition outside natural parent on the Diagnostics > Stateflow pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'

Parameter	Description	Values
SFUndirectedBroadcast-EventsDiag	<p>Select the diagnostic action to take when a chart contains undirected local event broadcasts.</p> <p>Set by Undirected event broadcasts on the Diagnostics > Stateflow pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
SFUnexpectedBacktracking-Diag	<p>Select the diagnostic action to take when a chart junction:</p> <ul style="list-style-type: none"> • Does not have an unconditional transition path to a state or a terminal junction • Has multiple transition paths leading to it <p>Set by Unexpected backtracking on the Diagnostics > Stateflow pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
SFUnreachableExecution-PathDiag	<p>Select the diagnostic action to take when there are chart constructs not on a valid execution path.</p> <p>Set by “Unreachable execution path” on the Diagnostics > Stateflow pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'

Parameter	Description	Values
SFUnusedDataAndEventsDialog	<p>Select the diagnostic action to take for detection of unused data and events in a chart.</p> <p>Set by Unused data and events on the Diagnostics > Stateflow pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
ShapePreserveControl	<p>At each time step, use derivative information to improve integration accuracy.</p> <p>Set by Shape preservation on the Solver pane of the Configuration Parameters dialog box.</p>	'EnableAll' {'DisableAll'}
ShowGrid	Has no effect in Simulink Editor. This parameter will be removed in a future release.	'on' {'off'}
ShowLinearization-Annotations	Toggles linearization icons in the model.	{'on'} 'off'
ShowLineDimensions	<p>Show signal dimensions on this model's block diagram.</p> <p>In the Simulink Editor, set by Signal Dimensions on the Display > Signal & Ports menu.</p>	'on' {'off'}
ShowLineDimensions-OnError	For internal use.	
ShowLineWidths	Deprecated. Use ShowLineDimensions on page 6-0 instead.	
ShowLoopsOnError	Highlight invalid loops graphically.	{'on'} 'off'

Parameter	Description	Values
ShowModelReference-BlockIO	<p>Toggles display of I/O mismatch on block.</p> <p>In the Simulink Editor, set by Block I/O Mismatch for Referenced Model on the Display > Blocks menu.</p>	'on' {'off'}
ShowModelReference-BlockVersion	<p>Toggles display of version on block.</p> <p>In the Simulink Editor, set by Block Version for Referenced Models on the Display > Blocks menu.</p>	'on' {'off'}
Shown	For internal use.	
ShowPageBoundaries	<p>Toggles display of page boundaries on the Simulink Editor canvas.</p> <p>In the Simulink Editor, set by Show Page Boundaries on the File > Print menu.</p>	'on' {'off'}
ShowPortDataTypes	<p>Show data types of ports on this model's block diagram.</p> <p>In the Simulink Editor, set by Port Data Types on the Display > Signals & Ports menu.</p>	'on' {'off'}
ShowPortDataTypesOnError	For internal use.	

Parameter	Description	Values
ShowPortUnits	<p>Show units of ports, subsystem, and model block icons on the model block diagram.</p> <p>In the Simulink Editor, set Port Units on the Display > Signals & Ports menu.</p>	'on' {'off'}
ShowStorageClass	<p>Show storage classes of signals on this model's block diagram.</p> <p>In the Simulink Editor, set by Storage Class on the Format > Signals & Ports menu.</p>	'on' {'off'}
ShowTestPointIcons	<p>Show test point icons on this model's block diagram.</p> <p>In the Simulink Editor, set by Testpoint & Logging Indicators on the Display > Signals & Ports menu.</p>	{'on'} 'off'
ShowViewerIcons	<p>Show viewer icons on this model's block diagram.</p> <p>In the Simulink Editor, set by Viewer Indicator on the Display > Signals & Ports menu.</p>	{'on'} 'off'

Parameter	Description	Values
SignalHierarchy	<p>If the signal is a bus, returns the name and hierarchy of the signals in the bus.</p> <p>(Read-only) Get with the <code>get_param</code> command. Specify a port or line handle. See “Display Information About Buses”.</p>	Return values reflect the structure of the signal that you specify.
SignalInfNanChecking	<p>Specifies diagnostic action to take when the value of a block output is <code>Inf</code> or <code>NaN</code> at the current time step.</p> <p>Set by Inf or NaN block output on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
SignalLabelMismatchMsg	<p>Specifies diagnostic action to take when a signal label mismatch occurs.</p> <p>Set by Signal label mismatch on the Connectivity Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
SignalLogging	<p>Globally enable signal logging for this model.</p> <p>Set by the Signal logging check box on the Data Import/Export pane of the Configuration Parameters dialog box.</p>	{'on'} 'off'

Parameter	Description	Values
SignalLoggingName	<p>Name for saving signal logging data to a workspace.</p> <p>Set by the Signal logging field on the Data Import/Export pane of the Configuration Parameters dialog box.</p> <p>Do not use a variable name that is the same as a <code>Simulink.SimulationOutput</code> object function name or property name.</p>	{ 'logouts' }
SignalLoggingSaveFormat	<p>Format for saving signal logging data.</p>	<p>{ 'Dataset' }</p> <p>'ModelDataLogs' is supported for backward compatibility. However, when you open a model in R2016a or later, signal logging uses <code>Dataset</code> format, regardless of the setting of this parameter.</p>
SignalNameFromLabel	<p>Propagate signal names for Bus Creator block input signals whenever you change the name of an input signal programmatically.</p> <p>Set with the <code>set_param</code> command, using either a port or line handle and a character vector specifying the signal name to propagate.</p>	{ '' }

Parameter	Description	Values
SignalRangeChecking	<p>Select the diagnostic action to take when signals exceed specified minimum or maximum values.</p> <p>Set by Simulation range checking on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
SignalResolutionControl	<p>Control which named states and signals get resolved to Simulink signal objects.</p> <p>Set by Signal resolution on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	'None' {'UseLocalSettings'} 'TryResolveAll' 'TryResolveAllWithWarning'
SigSpecEnsureSample-TimeMsg	<p>Specifies diagnostic action to take when the sample time of the source port of a signal specified by a Signal Specification block differs from the signal's destination port.</p> <p>Set by Enforce sample times specified by Signal Specification blocks on the Sample Time Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'

Parameter	Description	Values
SimBuildMode	<p>Specifies how you build the simulation target for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</p> <p>Set by Simulation target build mode on the Simulation Target pane of the Configuration Parameters dialog box.</p>	<p>{'sf_incremental_build'} 'sf_nonincremental_build' 'sf_make' 'sf_make_clean' 'sf_make_clean_objects'</p>
SimCompilerOptimization	<p>Specifies the compiler optimization level during acceleration code generation.</p> <p>Set by Compiler optimization level on the Configuration Parameters dialog box.</p>	<p>'on' {'off'}</p>
SimCtrlC	<p>Enables responsiveness checks in code generated for MATLAB Function blocks.</p> <p>Set by “Ensure responsiveness” on the Simulation Target pane of the Configuration Parameters dialog box.</p>	<p>{'on'} 'off'</p>

Parameter	Description	Values
SimCustomHeaderCode	<p>Enter code lines to appear near the top of a generated header file for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</p> <p>Set by Header file on the Simulation Target > Custom Code pane of the Configuration Parameters dialog box.</p>	{ ' ' }
SimCustomInitializer	<p>Enter code statements that execute once at the start of simulation for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</p> <p>Set by Initialize function on the Simulation Target > Custom Code pane of the Configuration Parameters dialog box.</p>	{ ' ' }
SimCustomSourceCode	<p>Enter code lines to appear near the top of a generated source code file for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</p> <p>Set by Source file on the Simulation Target > Custom Code pane of the Configuration Parameters dialog box.</p>	{ ' ' }

Parameter	Description	Values
SimCustomTerminator	<p>Enter code statements that execute at the end of simulation for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</p> <p>Set by Terminate function on the Simulation Target > Custom Code pane of the Configuration Parameters dialog box.</p>	{ '' }
SimIntegrity	<p>Detects violations of memory integrity while building MATLAB Function blocks and stops simulation with a diagnostic.</p> <p>Set by “Ensure memory integrity” on the Simulation Target pane of the Configuration Parameters dialog box.</p>	{ 'on' } 'off'
SimParseCustomCode	<p>Specify whether or not to parse the custom code and report unresolved symbols in the model.</p> <p>Set by Parse custom code symbols on the Simulation Target > Custom Code pane of the Configuration Parameters dialog box.</p>	{ 'on' } 'off'

Parameter	Description	Values
SimReservedNameArray	<p>Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code. This action prevents naming conflicts between identifiers in the generated code and in custom code for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</p> <p>Set by Reserved names on the Simulation Target > Symbols pane of the Configuration Parameters dialog box.</p>	character vector array — <code>{{}}</code>
SimulationCommand	<p>Executes a simulation command.</p> <hr/> <p>Note You cannot use <code>set_param</code> to run a simulation in a MATLAB session that does not have a display, i.e., if you used <code>matlab -nodisplay</code> to start the session.</p>	<code>'start' 'stop' 'pause' 'continue' 'step' 'update' 'WriteDataLogs' 'SimParamDialog' 'connect' 'disconnect' 'WriteExtModeParamVect' 'AccelBuild'</code>
SimulationMode	<p>Indicates whether Simulink software should run in Normal, Accelerator, Rapid Accelerator, SIL, PIL, or External mode.</p> <p>In the Simulink Editor, set by the Simulation > Mode menu.</p>	<code>{'normal'} 'accelerator' 'rapid-accelerator' 'external' 'Software-in-the-loop (SIL)' 'Processor-in-the-loop (PIL)'</code>

Parameter	Description	Values
SimulationStatus	Indicates simulation status.	{'stopped'} 'updating' 'initializing' 'running' 'compiled' 'paused' 'terminating' 'external'
SimulationTime	Current time value for the simulation.	double — {0}
SimStateInterfaceChecksumMismatchMsg	Check to ensure that the interface checksum is identical to the model checksum before loading the SimState.	'none' 'warning' 'error'
SimStateOlderReleaseMsg	Check to report that the SimState was generated by an earlier version of Simulink. In the Diagnostics pane of the Configuration Parameters dialog box, configure the diagnostic to allow Simulink to report the message as error or warning.	'error' 'warning'
SimUserDefines	Enter a space-separated list of preprocessor macro definitions to be added to the generated code for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Set by Defines on the Simulation Target > Custom Code pane of the Configuration Parameters dialog box.	{ '' }

Parameter	Description	Values
SimUserIncludeDirs	<p>Enter a space-separated list of directory paths that contain files you include in the compiled target for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</p> <p>Set by Include directories on the Simulation Target > Custom Code pane of the Configuration Parameters dialog box.</p>	<p>{ ' ' }</p> <hr/> <p>Note If your list includes any Windows path names that contain spaces, each instance must be enclosed in double quotes within the argument, for example,</p> <p>'C:\Project "C:\Custom Files"'</p> <hr/>
SimUserLibraries	<p>Enter a space-separated list of static libraries that contain custom object code to link into the target for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</p> <p>Set by Libraries on the Simulation Target > Custom Code pane of the Configuration Parameters dialog box.</p>	{ ' ' }
SimUserSources	<p>Enter a space-separated list of source files to compile and link into the target for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</p> <p>Set by Source files on the Simulation Target > Custom Code pane of the Configuration Parameters dialog box.</p>	{ ' ' }

Parameter	Description	Values
SingleTaskRateTransMsg	<p>Specifies diagnostic action to take when a rate transition takes place between two blocks operating in single-tasking mode.</p> <p>Set by Single task rate transition on the Sample Time Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
Solver	<p>Solver used for the simulation.</p> <p>Set by the Solver drop-down list on the Solver pane of the Configuration Parameters dialog box.</p>	'VariableStepDiscrete' {'ode45'} 'ode23' 'ode113' 'ode15s' 'ode23s' 'ode23t' 'ode23tb' 'FixedStepDiscrete' 'ode8' 'ode5' 'ode4' 'ode3' 'ode2' 'ode1' 'ode14x'
EnableMultiTasking	<p>Solver mode for this model. This option appears when the solver type is Fixed-step.</p> <p>Set by “Treat each discrete rate as a separate task” on the Solver pane of the Configuration Parameters dialog box.</p>	{'On'} 'Off'
SolverName	<p>Solver used for the simulation. See Solver on page 6-0 parameter for more information.</p>	

Parameter	Description	Values
SolverPrmCheckMsg	<p>Enables diagnostics to control when Simulink software automatically selects solver parameters. This option notifies you if:</p> <ul style="list-style-type: none"> • Simulink software changes a user-modified parameter to make it consistent with other model settings • Simulink software automatically selects solver parameters for the model, such as FixedStepSize <p>Set by Automatic solver parameter selection on the Solver Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
SolverResetMethod	<p>This option appears when the solver type is Variable-step and the solver is ode15s (stiff/NDF), ode23t (Mod. stiff/Trapezoidal), or ode23tb (stiff/TR-BDF2).</p> <p>Set by Solver reset method on the Solver pane of the Configuration Parameters dialog box.</p>	{'Fast'} 'Robust'

Parameter	Description	Values
SolverType	<p>Solver type used for the simulation.</p> <p>Set by Type on the Solver pane of the Configuration Parameters dialog box.</p>	{'Variable-step'} 'Fixed-step'
SortedOrder	<p>Show the sorted order of this model's blocks on the block diagram.</p> <p>In the Simulink Editor, set by Sorted Execution Order on the Display > Blocks menu.</p>	'on' {'off'}
StartFcn	<p>Start simulation callback.</p> <p>Set by Simulation start function on the Callbacks pane of the Model Properties dialog box.</p> <p>See “Create Model Callbacks” for more information.</p>	{''}
StartTime	<p>Simulation start time.</p> <p>Set by Start time on the Solver pane of the Configuration Parameters dialog box.</p>	{'0.0'}
StateNameClashWarn	<p>Select the diagnostic action to take when a name is used for more than one state in the model.</p> <p>Set by State name clash on the Solver Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' {'warning'}

Parameter	Description	Values
StateSaveName	<p>State output name to be saved to workspace.</p> <p>Set by the States field on the Data Import/Export pane of the Configuration Parameters dialog box.</p>	{ 'xout' }
StatusBar	<p>Has no effect in Simulink Editor. This parameter will be removed in a future release.</p> <p>In the Simulink Editor, set by Status Bar on the View menu.</p>	{ 'on' } 'off'
StiffnessThreshold	<p>Threshold value to determine if the model is stiff.</p> <p>A model is stiff if the stiffness exceeds the <code>StiffnessThreshold</code> value. The default value for this parameter is 1000. For more information, see “Use Auto Solver to Select a Solver”.</p>	{ '' }
StopFcn	<p>Stop simulation callback.</p> <p>Set by Simulation stop function on the Callbacks pane of the Model Properties dialog box.</p> <p>See “Create Model Callbacks” for more information.</p>	{ '' }

Parameter	Description	Values
StopTime	Simulation stop time. Set by Stop time on the Solver pane of the Configuration Parameters dialog box.	{'10.0'}
StrictBusMsg	Specifies diagnostic action to take when Simulink software detects a signal that some blocks treat as a mux or vector, while other blocks treat the signal as a bus. To be enabled, several options in MathWorks products require this parameter be set to 'ErrorLevel1' or 'ErrorOnBusTreatedAsVector'. Set by Bus signal treated as vector on the Diagnostics Connectivity pane of the Configuration Parameters dialog box.	{'ErrorLevel1' 'WarnOnBusTreatedAsVector' 'ErrorOnBusTreatedAsVector' '
Tag	User-specified text that is assigned to the model's Tag parameter and saved with the model.	{''}
TargetBitPerChar	Describes the length in bits of the C char data type supported by the hardware used to test generated code.	integer — {8}
TargetBitPerInt	Describes the length in bits of the C int data type supported by the hardware used to test generated code.	integer — {32}

Parameter	Description	Values
TargetBitPerLong	Describes the length in bits of the C <code>long</code> data type supported by the hardware used to test generated code.	integer — {32}
TargetBitPerLongLong	Describes the length in bits of the C <code>long long</code> data type supported by the hardware used to test generated code. The value of this parameter must be greater than or equal to the value of <code>TargetBitPerLong</code> .	integer — {64}
TargetBitPerShort	Describes the length in bits of the C <code>short</code> data type supported by the hardware used to test generated code.	integer — {16}
TargetEndianness	Describes the significance of the first byte of a data word of the hardware used to test generated code.	{'Unspecified' 'LittleEndian' 'BigEndian'}
TargetHWDeviceType	Describes the characteristics of the hardware used to test generated code.	{'Generic->Unspecified (assume 32-bit Generic)'}
TargetIntDivRoundTo	Describes how the C compiler that creates test code for this model rounds the result of dividing one signed integer by another to produce a signed integer quotient.	'Floor' 'Zero' {'Undefined'}
TargetLargestAtomicFloat	Specify the largest floating-point data type that can be atomically loaded and stored on the hardware used to test code.	'Float' 'Double' {'None'}

Parameter	Description	Values
TargetLargestAtomicInteger	Specify the largest integer data type that can be atomically loaded and stored on the hardware used to test code. Set this parameter to 'LongLong' only if the test hardware supports the C long long data type and you have set TargetLongLongMode to 'on'.	{'Char'} 'Short' 'Int' 'Long' 'LongLong'
TargetLongLongMode	Specify that your C compiler supports the C long long data type. Most C99 compilers support long long.	'on' {'off'}
TargetShiftRightIntArith	Describes whether the C compiler that creates test code for this model implements a signed integer right shift as an arithmetic right shift.	{'on'} 'off'
TargetTypeEmulationWarnSuppressLevel	Specifies whether Simulink Coder software displays or suppresses warning messages when emulating integer sizes in rapid prototyping environments.	integer — {0}
TargetWordSize	Describes the word length in bits of the hardware used to test generated code.	integer — {32}

Parameter	Description	Values
TasksWithSamePriorityMsg	<p>Specifies diagnostic action to take when tasks have equal priority.</p> <p>Set by Tasks with equal priority on the Sample Time Diagnostics pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
TiledPageScale	Scales the size of the tiled page relative to the model.	{'1'}
TiledPaperMargins	Controls the size of the margins associated with each tiled page. Each element in the vector represents a margin at the particular edge.	vector — [left, top, right, bottom]
TimeAdjustmentMsg	<p>Specifies diagnostic action to take if Simulink software makes a minor adjustment to a sample hit time while running the model.</p> <p>Set by Sample hit time adjusting on the Configuration Parameters dialog box.</p>	{'none'} 'warning'
TimeSaveName	<p>Simulation time name.</p> <p>Set by the Time field on the Data Import/Export pane of the Configuration Parameters dialog box.</p>	variable — {'tout'}

Parameter	Description	Values
TLC...	Parameters whose names begin with TLC are used for code generation. See the Simulink Coder documentation for more information.	
ToolBar	<p>Has no effect in Simulink Editor. This parameter will be removed in a future release.</p> <p>In the Simulink Editor, hide or display all toolbars with Toolbars on the View menu or, hide or display specific toolbars using File > Simulink Preferences > Editor Default toolbar options.</p>	{'on'} 'off'
TryForcingSFcnDF	This flag is used for backward compatibility with user S-functions that were written prior to R12.	'on' {'off'}
TunableVars	<p>List of global (tunable) parameters.</p> <p>Set in the Model Parameter Configuration dialog box.</p>	{''}
TunableVarsStorageClass	<p>List of storage classes for their respective tunable parameters.</p> <p>Set in the Model Parameter Configuration dialog box.</p>	{''}

Parameter	Description	Values
TunableVarsTypeQualifier	List of storage type qualifiers for their respective tunable parameters. Set in the Model Parameter Configuration dialog box.	{ '' }
Type	Simulink object type (read only).	{ 'block_diagram' }
UnconnectedInputMsg	Unconnected input ports diagnostic. Set by Unconnected block input ports on the Connectivity Diagnostics pane of the Configuration Parameters dialog box.	'none' {'warning'} 'error'
UnconnectedLineMsg	Unconnected lines diagnostic. Set by Unconnected line on the Connectivity Diagnostics pane of the Configuration Parameters dialog box.	'none' {'warning'} 'error'
UnconnectedOutputMsg	Unconnected block output ports diagnostic. Set by Unconnected block output ports on the Connectivity Diagnostics pane of the Configuration Parameters dialog box.	'none' {'warning'} 'error'

Parameter	Description	Values
UnderSpecifiedData- TypeMsg	<p>Detect usage of heuristics to assign signal data types.</p> <p>Set by Underspecified data types on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'
UnderspecifiedInitial- izationDetection	<p>Select how Simulink software handles initialization of initial conditions for conditionally executed subsystems, Merge blocks, subsystem elapsed time, and Discrete-Time Integrator blocks.</p> <p>Set by Underspecified initialization detection on the Configuration Parameters dialog box.</p>	{'classic'} 'simplified'
UniqueDataStoreMsg	<p>Specifies diagnostic action to take when the model contains multiple Data Store Memory blocks that specify the same data store name.</p> <p>Set by Duplicate data store names on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning' 'error'

Parameter	Description	Values
UnknownTsInhSupMsg	<p>Detect blocks that have not set whether they allow the model containing them to inherit a sample time.</p> <p>Set by Unspecified inheritability of sample time on the Diagnostics > Sample Time pane of the Configuration Parameters dialog box.</p>	'none' {'warning'} 'error'
UnnecessaryDatatype-ConvMsg	<p>Detect unnecessary data type conversion blocks.</p> <p>Set by Unnecessary type conversions on the Type Conversion Diagnostics pane of the Configuration Parameters dialog box.</p>	{'none'} 'warning'
UpdateHistory	<p>Specifies when to prompt the user about updating the model history.</p> <p>Set by Prompt to update model history on the History pane of the Model Properties dialog box or Prompt to update model history on the History pane of the Model Explorer.</p> <p>See “Model Information and History” for more information.</p>	{'UpdateHistoryNever'} 'UpdateHistoryWhenSave'

Parameter	Description	Values
UpdateModelReference-Targets	Specify whether to rebuild simulation and Simulink Coder targets for referenced models before updating, simulating, or generating code for this model. Set by Rebuild options on the Model Referencing pane of the Configuration Parameters dialog box.	'IfOutOfDate' 'Force' 'AssumeUpToDate' {'IfOutOfDateOrStructuralChange'}
UseAnalysisPorts	For internal use.	
UseDivisionForNetSlopeComputation	Use division to handle net slope computations when simplicity and accuracy conditions are met.	{'off'} 'on' 'UseDivisionForReciprocalsOfIntegersOnly'
VectorMatrix-ConversionMsg	Detect vector-to-matrix or matrix-to-vector conversions. Set by Vector/matrix block input conversion on the Type Conversion Diagnostics pane of the Configuration Parameters dialog box.	{'none'} 'warning' 'error'
Version	Simulink version you are currently running, e.g., '7.6'. If you are using a service pack, the ver function returns an additional digit, e.g., 7.4.1 (R2009bSP1). To get version information without loading the block diagram into memory, see Simulink.MDLInfo.	double (read only)

Parameter	Description	Values
VersionLoaded	<p>Simulink version that last saved the model, e.g., '7.6'. If you are using a service pack, the <code>ver</code> function returns an additional digit, e.g., 7.4.1 (R2009bSP1). See also <code>SavedSinceLoaded</code>.</p> <p>To get version information without loading the block diagram into memory, see <code>Simulink.MDLInfo</code>.</p>	double (read only)
VisualizeLoggedSignalsWhenLoggingToFile	<p>Control visualization of logged signals when <code>LoggingToFile</code> is enabled. The default is 'off', which logs to a MAT-file without the overhead of visualizing signals. Set the value to 'on' to stream logged data to these visualization tools also:</p> <ul style="list-style-type: none"> • Simulation Data Inspector • Message Viewer blocks • Dashboard blocks • Logic Analyzer <p>Logging big data to visualization tools can cause memory and disk space overhead.</p>	'on' {'off'}

Parameter	Description	Values
WideLines	<p>Draws lines that carry vector or matrix signals wider than lines that carry scalar signals.</p> <p>In the Simulink Editor, set by Wide Nonscalar Lines on the Display > Signals & Ports menu.</p>	'on' {'off'}
WideVectorLines	<p>Deprecated. Use WideLines on page 6-0 instead.</p>	
WriteAfterReadMsg	<p>Specifies diagnostic action to take when the model attempts to store data in a data store after previously reading data from it in the current time step.</p> <p>Set by Detect write after read on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	{'UseLocalSettings'} 'DisableAll' 'EnableAllAsWarning' 'EnableAllAsError'
WriteAfterWriteMsg	<p>Specifies diagnostic action to take when the model attempts to store data in a data store twice in succession in the current time step.</p> <p>Set by Detect write after write on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</p>	{'UseLocalSettings'} 'DisableAll' 'EnableAllAsWarning' 'EnableAllAsError'

Parameter	Description	Values
ZCThreshold	<p>Specifies the deadband region used during the detection of zero crossings. Signals falling within this region are defined as having crossed through zero.</p> <p>Set by Signal threshold on the Solver pane of the Configuration Parameters dialog box.</p>	{'auto'} any real number greater than or equal to zero
ZeroCross	For internal use.	
ZeroCrossAlgorithm	<p>Specifies the algorithm to detect zero crossings when you select a variable-step solver.</p> <p>Set by Algorithm on the Solver pane of the Configuration Parameters dialog box.</p>	{'Nonadaptive'} 'Adaptive'
ZeroCrossControl	<p>Enable zero-crossing detection.</p> <p>Set by Zero-crossing control on the Solver pane of the Configuration Parameters dialog box.</p>	{'UseLocalSettings'} 'EnableAll' 'DisableAll'
ZoomFactor	<p>Zoom factor of the Simulink Editor window expressed as a percentage of normal (100%) or by the keywords FitSystem or FitSelection.</p> <p>In the Simulink Editor, set by the zoom commands on the View menu.</p>	{'100'} 'FitSystem' 'FitSelection'

Examples of Setting Model Parameters

These examples show how to set model parameters for the `mymodel` system.

This command sets the simulation start and stop times.

```
set_param('mymodel','StartTime','5','StopTime','100')
```

This command sets the solver to `ode15s` and changes the maximum order.

```
set_param('mymodel','Solver','ode15s','MaxOrder','3')
```

This command associates a `PostSaveFcn` callback.

```
set_param('mymodel','PostSaveFcn','my_save_cb')
```

Common Block Properties

In this section...
“About Common Block Properties” on page 6-109
“Examples of Setting Block Properties” on page 6-126

About Common Block Properties

This table lists the properties common to all Simulink blocks, including block callback properties (see “Callbacks for Customized Model Behavior”). Examples of commands that change these properties follow this table (see “Examples of Setting Block Properties” on page 6-126).

Common Block Properties

Property	Description	Values
AncestorBlock	Name of the library block that the block is linked to (for blocks with a disabled link).	character vector
AttributesFormatString	Block annotation text (corresponds to block properties).	character vector
BackgroundColor	Block background color.	color value '[r,g,b]' '[r,g,b,a]' r, g, and b, are the red, green, blue values of the color in the range 0.0 to 1.0. If specified, the alpha value (a) is ignored. Possible color values are 'black', 'white', 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'gray', 'lightBlue', 'orange', 'darkGreen'.
BlockDescription	Block description shown in the Block Properties dialog box.	character array
BlockDiagramType	Returns <code>model</code> if it is in an open Simulink block diagram. Returns <code>library</code> if it is a Simulink library.	'model' 'library'
BlockType	Block type (read only).	character array
ClipboardFcn	Function called when block is copied to the clipboard (Ctrl+C) or when the menu item Copy is selected.	function character vector
CloseFcn	Function called when <code>close_system</code> is run on block.	function character vector

Property	Description	Values
Commented	Exclude block from simulation.	{'off'} 'on' 'through'
CompiledPortComplexSignals	<p>Complexity of port signals after updating diagram. You must compile the model before querying this property. For example:</p> <pre>vdp([], [], [], 'compile'); d = get_param(gcb, 'CompiledPortComplexSignals'); vdp([], [], [], 'term');</pre>	structure array
CompiledIsActive	<p>Specifies whether the block status is active or not at compile time.</p> <p>CompiledIsActive returns off if any one of these conditions is true at compile time:</p> <ul style="list-style-type: none"> • Block is inactive path of Inline Variant. • Block is inactive choice of Variant Subsystem. • Block is commented out is within a Subsystem block that is commented out. • Block is inactive due to condition propagated from Variant Subsystem block. 	'off' 'on'
CompiledPortDataTypes	Data types of port signals after updating diagram. You must compile the model before querying this property. See CompiledPortComplexSignals on page 6-0 .	structure array

Property	Description	Values
CompiledPortDesignMin	<p>Design minimum of port signals after updating diagram. You must compile the model before querying this property. For example:</p> <pre>feval(gcs, [], [], [], 'compile'); ports = get_param(gcb, 'PortHandles'); min = get_param(ports.Outport, 'CompiledPortDesignMin'); feval(model, [], [], [], 'term');</pre>	structure array
CompiledPortDesignMax	<p>Design maximum of port signals at compile time. You must compile the model before querying this property. For example:</p> <pre>feval(gcs, [], [], [], 'compile'); ports = get_param(gcb, 'PortHandles'); max = get_param(ports.Outport, 'CompiledPortDesignMax'); feval(model, [], [], [], 'term');</pre>	structure array
CompiledPortDimensions	<p>Dimensions of port signals after updating diagram. You must compile the model before querying this property. See CompiledPortComplexSignals on page 6-0 .</p>	numeric array
CompiledPortDimensionsMode	<p>Indication whether the port signal has a variable size (after updating diagram). You must compile the model before querying this property. See “Programmatically Determine Whether Signal Line Has Variable Size”.</p>	double number. 0 indicates the signal does not have a variable size. 1 indicates the signal has a variable size.

Property	Description	Values
CompiledPortFrameData	Frame mode of port signals after updating diagram. You must compile the model before querying this property. See CompiledPortComplexSignals on page 6-0 .	structure array
CompiledPortWidths	Structure of port widths after updating diagram. You must compile the model before querying this property. See CompiledPortComplexSignals on page 6-0 .	structure array
CompiledSampleTime	Block sample time after updating diagram. You must compile the model before querying this property. See CompiledPortComplexSignals on page 6-0 .	vector [sample time, offset time] or cell {[sample time 1, offset time 1]; [sample time 2, offset time 2];....[sample time n, offset time n]}
ContinueFcn	Function called at the restart of a simulation (after a pause).	function character vector
CopyFcn	Function called when block is copied. See “Block Callback Parameters” for details.	function character vector
DataTypeOverrideCompiled	For internal use.	
DeleteFcn	Function called when block is deleted. See “Block Callback Parameters” for details.	MATLAB expression
DestroyFcn	Function called when block is destroyed. See “Block Callback Parameters” for details.	MATLAB expression

Property	Description	Values
Description	Description of block. Set by the Description field in the General pane of the Block Properties dialog box.	text and tokens
Diagnostics	For internal use.	
DialogParameters	List of names/attributes of block-specific parameters for an unmasked block, or mask parameters for a masked block.	structure
DropShadow	Display drop shadow.	{'off'} 'on'
ExtModeLoggingSupported	Enable a block to support uploading of signal data in external mode (for example, with a scope block).	{'off'} 'on'
ExtModeLoggingTrig	Enable a block to act as the trigger block for external mode signal uploading.	{'off'} 'on'
ExtModeUploadOption	Enable a block to upload signal data in external mode when the Select all check box on the External Signal & Triggering dialog box is not selected. A value of <code>log</code> indicates the block uploads signals. A value of <code>none</code> indicates the block does not upload signals. The value <code>monitor</code> is currently not in use. If the Select all check box on the External Signal & Triggering dialog box is selected, it overrides this parameter setting.	{'none'} 'log' 'monitor'
FontAngle	Font angle.	'normal' 'italic' 'oblique' {'auto'}
FontName	Font name.	character array

Property	Description	Values
FontSize	Font size. A value of -1 specifies that this block inherits the font size specified by the DefaultBlockFontSize model parameter.	real {'-1'}
FontWeight	Font weight.	'light' 'normal' 'demi' 'bold' {'auto'}
ForegroundColor	Foreground color of block icon.	<p>color value '[r,g,b]' '[r,g,b,a]'</p> <p>r, g, and b, are the red, green, blue values of the color in the range 0.0 to 1.0. The value changes if it is too similar to the canvas color (ScreenColor parameter). Use get_param to return the actual value. If specified, the alpha value (a) is ignored.</p> <p>Possible color values are 'black', 'white', 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'gray', 'lightBlue', 'orange', 'darkGreen'.</p>
Handle	Block handle.	real

Property	Description	Values
HideAutomaticName	Specify whether the block name given automatically by the Simulink Editor displays in the model. To hide automatic names, use the default setting of 'on'. (The HideAutomaticNames parameter for the model must also be set to 'on'.) Set to 'off' to display the name, and also set the block ShowName parameter to 'on'. Blocks whose ShowName parameter is 'off' are hidden regardless of this setting. For more information on how the parameters interact, see “Hide or Display Block Names”.	{'on'} 'off'
HiliteAncestors	For internal use.	
InitFcn	Initialization function for a block. Created on the Callbacks pane of the Model Properties dialog box. For more information, see “Create Model Callbacks”. On non-masked blocks, updating the diagram or running the simulation call this function.	MATLAB expression
InputSignalNames	Names of input signals.	cell array

Property	Description	Values
IntrinsicDialogParameters	List of names/attributes of block-specific parameters (regardless of whether the block is masked or unmasked). Use instead of DialogParameters if you want block-specific parameters for masked or unmasked blocks.	structure
IOSignalStrings	Block paths to objects that are connected to the Signal & Scope Manager. Simulink software saves these paths when the model is saved.	list
IOType	Signal & Scope Manager type. For internal use.	
LibraryVersion	For a linked block, the initial value of this property is the ModelVersion of the library at the time the link was created. The value updates with increments in the model version of the library.	character vector — <code>{'1.1'}</code>
LineHandles	Handles of lines connected to block.	structure

Property	Description	Values
LinkData	Array of details about changes to the blocks inside the link that differ between a parameterized link and its library, listing the block names and parameter values. Use [] to reset to deparameterized, e.g., <code>set_param(gcb, 'linkData', [])</code> . See “Restore Disabled or Parameterized Links”.	cell array
LinkStatus	Link status of block. Updates out-of-date linked blocks when queried using <code>get_param</code> . See “Control Linked Block Programmatically”.	'none' 'resolved' 'unresolved' 'implicit' 'inactive' 'restore' 'propagate' 'propagateHierarchy' 'restoreHierarchy'
LoadFcn	Function called when block is loaded.	MATLAB expression
MinMaxOverflow-Logging_Compiled	For internal use.	
ModelCloseFcn	Function called when model is closed. The <code>ModelCloseFcn</code> is called prior to the block's <code>DeleteFcn</code> and <code>DestroyFcn</code> callbacks, if either are set.	MATLAB expression
ModelParamTableInfo	For internal use.	
MoveFcn	Function called when block is moved.	MATLAB expression
Name	Block name.	character vector
NameChangeFcn	Function called when block name is changed.	MATLAB expression

Property	Description	Values
NamePlacement	Position of block name.	{'normal'} 'alternate'
ObjectParameters	Names/attributes of block's parameters.	structure
OpenFcn	Function called when this Block Parameters dialog box opens.	MATLAB expression
Orientation	Where block faces.	{'right'} 'left' 'up' 'down'
OutputSignalNames	Names of output signals.	cell array
Parent	Name of the system that owns the block.	character vector {'untitled'}
ParentCloseFcn	Function called when parent subsystem is closed. The ParentCloseFcn of blocks at the root model level is not called when the model is closed.	MATLAB expression
PauseFcn	Function called at the pause of a simulation.	function character vector

Property	Description	Values
PortConnectivity	<p>The value of this property is an array of structures, each of which describes one of the block's input or output ports. Each port structure has the following fields:</p> <ul style="list-style-type: none"> • Type <p>Specifies the port's type and/or number. The value of this field can be:</p> <ul style="list-style-type: none"> • n, where n is the number of the port for data ports • 'enable' if the port is an enable port • 'trigger' if the port is a trigger port • 'state' for state ports • 'ifaction' for action ports • 'LConn#' for a left connection port where # is the port's number • 'RConn#' for a right connection port where # is the port's number • Position <p>The value of this field is a two-element vector, $[x\ y]$, that specifies the port's position.</p> • SrcBlock 	structure array

Property	Description	Values
	<p>Handle of the block connected to this port. This field is null for output ports and -1 for unconnected input ports. <code>SrcBlock</code> property is a valid source handle for Variant Subsystem blocks.</p> <ul style="list-style-type: none"> • <code>SrcPort</code> <p>Number of the port connected to this port, starting at zero. This field is null for both output ports and unconnected input ports.</p> <ul style="list-style-type: none"> • <code>DstBlock</code> <p>Handle of the block to which this port is connected. This field is null for input ports and contains a 1-by-0 empty matrix for unconnected output ports.</p> <ul style="list-style-type: none"> • <code>DstPort</code> <p>Number of the port to which this port is connected, starting at zero. This field is null for input ports and contains a 1-by-0 empty matrix for unconnected output ports.</p>	

Property	Description	Values
PortHandles	<p>The value of this property is a structure that specifies the handles of the block's ports. The structure has the following fields:</p> <ul style="list-style-type: none"> • Inport Handles of the block's input ports. • Outport Handles of the block's output ports. • Enable Handle of the block's enable port. • Trigger Handle of the block's trigger port. • State Handle of the block's state port. • LConn Handles of the block's left connection ports (for blocks that support Physical Modeling tools). • RConn Handles of the block's right connection ports (for blocks that support Physical Modeling tools). 	structure array

Property	Description	Values
	<ul style="list-style-type: none"> • Ifaction <p>Handle of the block's action port.</p> <ul style="list-style-type: none"> • Reset <p>Handle of the block's reset port.</p>	
PortRotationType	Type of port rotation used by this block. This is a read-only property.	'default' 'physical'
Ports	<p>A vector that specifies the number of each kind of port this block has. The order of the vector's elements corresponds to the following port types:</p> <ul style="list-style-type: none"> • Inport • Outport • Enable • Trigger • State • LConn • RConn • Ifaction • Reset 	vector

Property	Description	Values
Position	<p>Position of block in model window.</p> <p>To help with block alignment, the position you set can differ from the actual block position by a few pixels. Use <code>get_param</code> to return the actual position.</p>	<p>vector of coordinates, in pixels: [left top right bottom]</p> <p>The origin is the upper-left corner of the Simulink Editor canvas before any canvas resizing. The maximum absolute value for a coordinate is 32767. Positive values are to the right of and down from the origin. Negative values are to the left of and up from the origin.</p>
PostSaveFcn	Function called after the block is saved.	MATLAB expression
PreCopyFcn	Function called before the block is copied. See “Block Callback Parameters” for details.	MATLAB expression
PreDeleteFcn	Function called before the block is deleted. See “Block Callback Parameters” for details.	MATLAB expression
PreSaveFcn	Function called before the block is saved. See “Block Callback Parameters” for details.	MATLAB expression
Priority	Specifies the block's order of execution relative to other blocks in the same model. Set by the Priority field on the General pane of the Block Properties dialog box.	character vector { ' ' }
ReferenceBlock	Name of the library block to which this block links.	character vector { ' ' }
RequirementInfo	For internal use.	

Property	Description	Values
RTWData	User specified data, used by Simulink Coder software. Intended only for use with user written S-functions. See the section “S-Function RTWdata” (Simulink Coder) for details.	
SampleTime	Value of the sample time parameter. See “Specify Sample Time” for more details.	character vector
Selected	Status of whether or not block is selected.	{ 'on' } 'off'
ShowName	Display or hide block name. To display a block name given by the Simulink Editor (automatic names), set the block 'HideAutomaticName' parameter to 'off' and ShowName to 'on'. To hide an automatic block name given by the Editor, set ShowName to 'on', HideAutomaticName to 'on', and HideAutomaticNames on the model to 'on'. For more information on how the parameters interact, see “Hide or Display Block Names”.	{ 'on' } 'off'
StartFcn	Function called at the start of a simulation.	MATLAB expression
StatePerturbation-ForJacobian	State perturbation size to use during linearization. See “Change Perturbation Level of Blocks Perturbed During Linearization” (Simulink Control Design) for details.	character vector

Property	Description	Values
StaticLinkStatus	Link status of block. Does not update out-of-date linked blocks when queried using <code>get_param</code> . See also <code>LinkStatus</code> .	'none' 'resolved' 'unresolved' 'implicit' 'inactive' 'restore' 'propagate' 'propagateHierarchy' 'restoreHierarchy'
StopFcn	Function called at the termination of a simulation.	MATLAB expression
Tag	Text that appears in the block label that Simulink software generates. Set by the Tag field on the General pane of the Block Properties dialog box.	character vector {''}
Type	Simulink object type (read only).	'block'
UndoDeleteFcn	Function called when block deletion is undone.	MATLAB expression
UserData	User-specified data that can have any MATLAB data type.	{'[]'}
UserDataPersistent	Status of whether or not <code>UserData</code> will be saved in the model file.	'on' {'off'}

Examples of Setting Block Properties

These examples illustrate how to change common block properties.

This command changes the orientation of the Gain block in the `mymodel` system so it faces the opposite direction (right to left).

```
set_param('mymodel/Gain','Orientation','left')
```

This command associates an `OpenFcn` callback with the Gain block in the `mymodel` system.

```
set_param('mymodel/Gain','OpenFcn','my_open_cb')
```

This command sets the `Position` property of the Gain block in the `mymodel` system. The block is 75 pixels wide by 25 pixels high.

```
set_param('mymodel/Gain','Position',[50 250 125 275])
```

Block-Specific Parameters

To write scripts that create and modify models, you can use the `get_param` and `set_param` functions to query and modify the properties and parameters of a block or diagram. Use the tables to determine the programmatic name of a parameter or property in a block dialog box.

- Continuous Library Block Parameters on page 6-0
- Discontinuities Library Block Parameters on page 6-0
- Discrete Library Block Parameters on page 6-0
- Logic and Bit Operations Library Block Parameters on page 6-0
- Lookup Tables Block Parameters on page 6-0
- Math Operations Library Block Parameters on page 6-0
- Model Verification Library Block Parameters on page 6-0
- Model-Wide Utilities Library Block Parameters on page 6-0
- Ports & Subsystems Library Block Parameters on page 6-0
- Signal Attributes Library Block Parameters on page 6-0
- Signal Routing Library Block Parameters on page 6-0
- Sinks Library Block Parameters on page 6-0
- Sources Library Block Parameters on page 6-0
- User-Defined Functions Library Block Parameters on page 6-0
- Additional Discrete Block Library Parameters on page 6-0
- Additional Math: Increment - Decrement Block Parameters on page 6-0

Programmatic Parameters of Blocks and Models

Programmatic parameters that describe a model are model parameters (see “Model Parameters” on page 6-2). Parameters that describe a block are block parameters. Parameters that are common to all Simulink blocks are common block parameters (see “Common Block Properties” on page 6-109). Many blocks also have unique block-specific parameters. A masked block can have mask parameters (see “Mask Parameters” on page 6-278).

The model and block properties also include callbacks, which are commands that execute when a certain model or block event occurs. These events include opening a model,

simulating a model, copying a block, opening a block, etc. See “Model, Block, and Port Callbacks”.

Tip For block parameters that accept array values, the number of elements in the array cannot exceed what `int_T` can represent. This limitation applies to both simulation and Simulink Coder code generation.

The maximum number of characters that a parameter edit field can contain is 49,000.

Block-Specific Parameters and Programmatic Equivalents

The tables list block-specific parameters for Simulink blocks. The type of the block appears in parentheses after the block name. Some Simulink blocks work as masked subsystems. The tables indicate masked blocks by adding the designation "masked subsystem" after the block type.

The type listed for nonmasked blocks is the value of the `BlockType` parameter (see “Common Block Properties” on page 6-109). The type listed for masked blocks is the value of the `MaskType` parameter (see “Mask Parameters” on page 6-278).

The **Dialog Box Prompt** column indicates the text of the prompt for the parameter in the block dialog box. The **Values** column shows the type of value required (scalar, vector, variable), the possible values (separated with a vertical line), and the default value (enclosed in braces).

Continuous Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Derivative (Derivative)		
CoefficientInTFapproximation	Coefficient c in the transfer function approximation $s/(c*s+1)$ used for linearization	{'inf'}
Integrator (Integrator)		
ExternalReset	External reset	{'none'} 'rising' 'falling' 'either' 'level' 'level hold'
InitialConditionSource	Initial condition source	{'internal'} 'external'
InitialCondition	Initial condition	scalar or vector — {'0'}
LimitOutput	Limit output	{'off'} 'on'
UpperSaturationLimit	Upper saturation limit	scalar or vector — {'inf'}
LowerSaturationLimit	Lower saturation limit	scalar or vector — {'-inf'}
ShowSaturationPort	Show saturation port	{'off'} 'on'
ShowStatePort	Show state port	{'off'} 'on'
AbsoluteTolerance	Absolute tolerance	character vector, scalar, or vector — {'auto'} {'-1'} any real scalar or vector
IgnoreLimit	Ignore limit and reset when linearizing	{'off'} 'on'
ZeroCross	Enable zero-crossing detection	'off' {'on'}
ContinuousStateAttributes	State Name	{''} user-defined
WrapState	Enable wrapping of states	{'off'} 'on'
WrappedStateUpperValue	Upper value of wrapped state	scalar or vector — {'pi'}
WrappedStateLowerValue	Lower value of wrapped state	scalar or vector — {'-pi'}
Second-Order Integrator (SecondOrderIntegrator)		

Block (Type)/Parameter	Dialog Box Prompt	Values
ICSourceX	Initial condition source x	{'internal'} 'external'
ICX	Initial condition x	scalar or vector — {'0'}
LimitX	Limit x	{'off'} 'on'
UpperLimitX	Upper limit x	scalar or vector — {'inf'}
LowerLimitX	Lower limit x	scalar or vector — {'-inf'}
WrapStateX	Enable wrapping of x	{'off'} 'on'
WrappedUpperValueX	Upper value for wrapping x	scalar or vector — {'pi'}
WrappedLowerValueX	Lower value for wrapping x	scalar or vector — {'-pi'}
AbsoluteToleranceX	Absolute tolerance x	character vector, scalar, or vector — {'auto'} {'-1'} any real scalar or vector
StateNameX	State name x	{ } user-defined
ICSourceDXDT	Initial condition source dx/dt	{'internal'} 'external'
ICDXDT	Initial condition dx/dt	scalar or vector — {'0'}
LimitDXDT	Limit dx/dt	{'off'} 'on'
UpperLimitDXDT	Upper limit dx/dt	scalar or vector — {'inf'}
LowerLimitDXDT	Lower limit dx/dt	scalar or vector — {'-inf'}
AbsoluteToleranceDXDT	Absolute tolerance dx/dt	character vector, scalar, or vector — {'auto'} {'-1'} any real scalar or vector
StateNameDXDT	State name dx/dt	{ } user-defined
ExternalReset	External reset	{'none'} 'rising' 'falling' 'either'
ZeroCross	Enable zero-crossing detection	{'on'} 'off'
ReinitDXDTwhenXreachesSaturation	Reinitialize dx/dt when x reaches saturation	{'off'} 'on'
IgnoreStateLimitsAndResetForLinearization	Ignore state limits and the reset for linearization	{'off'} 'on'
ShowOutput	Show output	{'both'} 'x' 'dxdt'

Block (Type)/Parameter	Dialog Box Prompt	Values
VariableDelayType	Select delay type	'Variable transport delay' {'Variable time delay'}
MaximumDelay	Maximum delay	scalar or vector — {'10'}
InitialOutput	Initial output	scalar or vector — {'0'}
MaximumPoints	Initial buffer size	scalar — {'1024'}
FixedBuffer	Use fixed buffer size	{'off'} 'on'
ZeroDelay	Handle zero delay	{'off'} 'on'
TransDelayFeedthrough	Direct feedthrough of input during linearization	{'off'} 'on'
PadeOrder	Pade order (for linearization)	{'0'}
ContinuousStateAttributes	State Name	{''} user-defined
Variable Transport Delay (VariableTransportDelay)		
VariableDelayType	Select delay type	{'Variable transport delay'} 'Variable time delay'
MaximumDelay	Maximum delay	scalar or vector — {'10'}
InitialOutput	Initial output	scalar or vector — {'0'}
MaximumPoints	Initial buffer size	scalar — {'1024'}
FixedBuffer	Use fixed buffer size	{'off'} 'on'
TransDelayFeedthrough	Direct feedthrough of input during linearization	{'off'} 'on'
PadeOrder	Pade order (for linearization)	{'0'}
AbsoluteTolerance	Absolute tolerance	character vector, scalar, or vector — {'auto'} {'-1'} any positive real scalar or vector
ContinuousStateAttributes	State Name	{''} user-defined

Block (Type)/Parameter	Dialog Box Prompt	Values
Zero-Pole (ZeroPole)		
Zeros	Zeros	vector — {'[1]'} vector — {'[0 -1]'} vector — {'[1]'} character vector, scalar, or vector — {'auto'} {'-1'} any positive real scalar or vector
Poles	Poles	
Gain	Gain	
AbsoluteTolerance	Absolute tolerance	
ContinuousStateAttributes	State Name	{''} user-defined

Discontinuities Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Backlash (Backlash)		
BacklashWidth	Deadband width	scalar or vector — {'1'}
InitialOutput	Initial output	scalar or vector — {'0'}
ZeroCross	Enable zero-crossing detection	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Coulomb & Viscous Friction (Coulombic and Viscous Friction) (masked subsystem)		
offset	Coulomb friction value (Offset)	{'[1 3 2 0]'}
gain	Coefficient of viscous friction (Gain)	{'1'}
Dead Zone (DeadZone)		
LowerValue	Start of dead zone	scalar or vector — {'-0.5'}
UpperValue	End of dead zone	scalar or vector — {'0.5'}
SaturateOnIntegerOverflow	Saturate on integer overflow	'off' {'on'}
LinearizeAsGain	Treat as gain when linearizing	'off' {'on'}
ZeroCross	Enable zero-crossing detection	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Dead Zone Dynamic (Dead Zone Dynamic) (masked subsystem)		
Hit Crossing (HitCross)		
HitCrossingOffset	Hit crossing offset	scalar or vector — {'0'}
HitCrossingDirection	Hit crossing direction	'rising' 'falling' {'either'}
ShowOutputPort	Show output port	'off' {'on'}
ZeroCross	Enable zero-crossing detection	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Quantizer (Quantizer)		
QuantizationInterval	Quantization interval	scalar or vector — {'0.5'}

Block (Type)/Parameter	Dialog Box Prompt	Values
LinearizeAsGain	Treat as gain when linearizing	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Rate Limiter (RateLimiter)		
RisingSlewLimit	Rising slew rate	{'1'}
FallingSlewLimit	Falling slew rate	{'-1'}
SampleTimeMode	Sample time mode	'continuous' {'inherited'}
InitialCondition	Initial condition	{'0'}
LinearizeAsGain	Treat as gain when linearizing	'off' {'on'}
Rate Limiter Dynamic (Rate Limiter Dynamic) (masked subsystem)		
Relay (Relay)		
OnSwitchValue	Switch on point	{'eps'}
OffSwitchValue	Switch off point	{'eps'}
OnOutputValue	Output when on	{'1'}
OffOutputValue	Output when off	{'0'}
ZeroCross	Enable zero-crossing detection	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output data type	'Inherit: Inherit via back propagation' {'Inherit: All ports same datatype'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16)' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
Saturation (Saturate)		
UpperLimit	Upper limit	scalar or vector — {'0.5'}
LowerLimit	Lower limit	scalar or vector — {'-0.5'}
LinearizeAsGain	Treat as gain when linearizing	'off' {'on'}
ZeroCross	Enable zero-crossing detection	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	'Inherit: Inherit via back propagation' {'Inherit: Same as input'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'

Block (Type)/Parameter	Dialog Box Prompt	Values
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
Saturation Dynamic (Saturation Dynamic) (masked subsystem)		
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	{'Inherit: Same as second input'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
OutputDataTypeScalingMode	Deprecated in R2007b	
OutDataType	Deprecated in R2007b	
OutScaling	Deprecated in R2007b	
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
DoSatur	Saturate on integer overflow	{'off'} 'on'
Wrap To Zero (Wrap To Zero) (masked subsystem)		
Threshold	Threshold	{'255'}

Discrete Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Delay (Delay)		
DelayLengthSource	Delay length > Source	{'Dialog'} 'Input port'
DelayLength	Delay length > Value	{'2'}
DelayLengthUpperLimit	Delay length > Upper limit	{'100'}
InitialConditionSource	Initial condition > Source	{'Dialog'} 'Input port'
InitialCondition	Initial condition > Value	{'0.0'}
ExternalReset	External reset	{'None'} 'Rising' 'Falling' 'Either' 'Level' 'Level hold'
InputProcessing	Input processing	'Columns as channels (frame based)' {'Elements as channels (sample based)'} 'Inherited'
UseCircularBuffer	Use circular buffer for state	{'off'} 'on'
PreventDirectFeedthrough	Prevent direct feedthrough	{'off'} 'on'
RemoveDelayLengthCheckInGeneratedCode	Remove delay length check in generated code	{'off'} 'on'
DiagnosticForDelayLength	Diagnostic for delay length	{'None'} 'Warning' 'Error'
SampleTime	Sample time (-1 for inherited)	{'-1'}
StateName	State name	{''}
StateMustResolveToSignalObject	State name must resolve to Simulink signal object	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
StateSignalObject	Signal object class	Simulink.Signal object
	Code generation storage class	Object of a class that is derived from Simulink.Signal
StateStorageClass	Code generation storage class	{'Auto'} 'SimulinkGlobal' 'ExportedGlobal' 'ImportedExtern' 'ImportedExternPointer' 'Custom'
CodeGenStateStorageType Qualifier	Code generation storage type qualifier	{''}
Difference (Difference) (masked subsystem)		
ICPrevInput	Initial condition for previous input	{'0.0'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
OutputDataTypeScalingMode	Deprecated in R2007b	
OutDataType	Deprecated in R2007b	
OutScaling	Deprecated in R2007b	

Block (Type)/Parameter	Dialog Box Prompt	Values
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
DoSatur	Saturate to max or min when overflows occur	{'off'} 'on'
Discrete Derivative (Discrete Derivative) (masked subsystem)		
gainval	Gain value	{'1.0'}
ICPrevScaledInput	Initial condition for previous weighted input $K*u/Ts$	{'0.0'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
OutputDataTypeScaling Mode	Deprecated in R2007b	
OutDataType	Deprecated in R2007b	
OutScaling	Deprecated in R2007b	
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
DoSatur	Saturate to max or min when overflows occur	{'off'} 'on'
Discrete FIR Filter (Discrete FIR Filter)		
CoefSource	Coefficient source	{'Dialog parameters'} 'Input port'
FilterStructure	Filter structure	{'Direct form'} 'Direct form symmetric' 'Direct form antisymmetric' 'Direct form transposed' 'Lattice MA' Note You must have a DSP System Toolbox license to use a filter structure other than Direct form.
Coefficients	Coefficients	vector — {'[0.5 0.5]'} <hr/>
InputProcessing	Input processing	'Columns as channels (frame based)' {'Elements as channels (sample based)'} <hr/>
InitialStates	Initial states	scalar or vector — {'0'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
CoefMin	Coefficients minimum	{'[]'}
CoefMax	Coefficients maximum	{'[]'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
TapSumDataTypeStr	Tap sum data type	{'Inherit: Same as input'} 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)'
CoefDataTypeStr	Coefficients data type	{'Inherit: Same word length as input'} 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16)' 'fixdt(1,16,0)'
ProductDataTypeStr	Product output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as input' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)'
AccumDataTypeStr	Accumulator data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as input' 'Inherit: Same as product output' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)'
StateDataTypeStr	State data type	'Inherit: Same as input' {'Inherit: Same as accumulator'} 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)'

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output data type	'Inherit: Same as input' {'Inherit: Same as accumulator'} 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16)' 'fixdt(1,16,0)'
LockScale	Lock data type settings against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnIntegerOverflow	Saturate on integer overflow	{'off'} 'on'
Discrete Filter (DiscreteFilter)		
Numerator	Numerator coefficients	vector — {'[1]'}
Denominator	Denominator coefficients	vector — {'[1 0.5]'}
IC	Initial states	{'0'}
SampleTime	Sample time (-1 for inherited)	{'1'}
a0EqualsOne	Optimize by skipping divide by leading denominator coefficient (a0)	{'off'} 'on'
NumCoefMin	Numerator coefficient minimum	{'[]'}
NumCoefMax	Numerator coefficient maximum	{'[]'}
DenCoefMin	Denominator coefficient minimum	{'[]'}
DenCoefMax	Denominator coefficient maximum	{'[]'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
StateDataTypeStr	State data type	{'Inherit: Same as input'} 'int8' 'int16' 'int32' 'fixdt(1,16,0)'
NumCoefDataTypeStr	Numerator coefficient data type	{'Inherit: Inherit via internal rule'} 'int8' 'int16' 'int32' 'fixdt(1,16)' 'fixdt(1,16,0)'
DenCoefDataTypeStr	Denominator coefficient data type	{'Inherit: Inherit via internal rule'} 'int8' 'int16' 'int32' 'fixdt(1,16)' 'fixdt(1,16,0)'
NumProductDataTypeStr	Numerator product output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as input' 'int8' 'int16' 'int32' 'fixdt(1,16,0)'
DenProductDataTypeStr	Denominator product output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as input' 'int8' 'int16' 'int32' 'fixdt(1,16,0)'
NumAccumDataTypeStr	Numerator accumulator data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as input' 'Inherit: Same as product output' 'int8' 'int16' 'int32' 'fixdt(1,16,0)'

Block (Type)/Parameter	Dialog Box Prompt	Values
DenAccumDataTypeStr	Denominator accumulator data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as input' 'Inherit: Same as product output' 'int8' 'int16' 'int32' 'fixdt(1,16,0)'
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as input' 'int8' 'int16' 'int32' 'fixdt(1,16,0)'
LockScale	Lock data type settings against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnIntegerOverflow	Saturate on integer overflow	{'off'} 'on'
StateName	State name	{''}
StateMustResolveToSignalObject	State name must resolve to Simulink signal object	{'off'} 'on'
StateSignalObject	Signal object class	Simulink.Signal object
	Code generation storage class	Object of a class that is derived from Simulink.Signal
StateStorageClass	Code generation storage class	{'Auto'} 'SimulinkGlobal' 'ExportedGlobal' 'ImportedExtern' 'ImportedExternPointer' 'Custom'

Block (Type)/Parameter	Dialog Box Prompt	Values
RTWStateStorageTypeQualifier	Code generation storage type qualifier	{ '' }
Discrete State-Space (DiscreteStateSpace)		
A	A	matrix — {'1'}
B	B	matrix — {'1'}
C	C	matrix — {'1'}
D	D	matrix — {'1'}
InitialCondition	Initial conditions	vector — {'0'}
SampleTime	Sample time (-1 for inherited)	{ '1' }
StateName	State name	{ '' }
StateMustResolveToSignalObject	State name must resolve to Simulink signal object	{ 'off' } 'on'
StateSignalObject	Signal object class	Simulink.Signal object
	Code generation storage class	Object of a class that is derived from Simulink.Signal
StateStorageClass	Code generation storage class	{ 'Auto' } 'SimulinkGlobal' 'ExportedGlobal' 'ImportedExtern' 'ImportedExternPointer' 'Custom'
RTWStateStorageTypeQualifier	Code generation storage type qualifier	{ '' }
Discrete Transfer Fcn (DiscreteTransferFcn)		
Numerator	Numerator coefficients	vector — {'[1]'} { '1' }
Denominator	Denominator coefficients	vector — {'[1 0.5]'} { '1' }
InitialStates	Initial states	{ '0' }
SampleTime	Sample time (-1 for inherited)	{ '1' }

Block (Type)/Parameter	Dialog Box Prompt	Values
a0EqualsOne	Optimize by skipping divide by leading denominator coefficient (a0)	{'off'} 'on'
NumCoefMin	Numerator coefficient minimum	{'[]'}
NumCoefMax	Numerator coefficient maximum	{'[]'}
DenCoefMin	Denominator coefficient minimum	{'[]'}
DenCoefMax	Denominator coefficient maximum	{'[]'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
StateDataTypeStr	State data type	{'Inherit: Same as input'} 'int8' 'int16' 'int32' 'fixdt(1,16,0)'
NumCoefDataTypeStr	Numerator coefficient data type	{'Inherit: Inherit via internal rule'} 'int8' 'int16' 'int32' 'fixdt(1,16,0)'
DenCoefDataTypeStr	Denominator coefficient data type	{'Inherit: Inherit via internal rule'} 'int8' 'int16' 'int32' 'fixdt(1,16,0)'
NumProductDataTypeStr	Numerator product output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as input' 'int8' 'int16' 'int32' 'fixdt(1,16,0)'

Block (Type)/Parameter	Dialog Box Prompt	Values
DenProductDataTypeStr	Denominator product output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as input' 'int8' 'int16' 'int32' 'fixdt(1,16,0)'
NumAccumDataTypeStr	Numerator accumulator data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as input' 'Inherit: Same as product output' 'int8' 'int16' 'int32' 'fixdt(1,16,0)'
DenAccumDataTypeStr	Denominator accumulator data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as input' 'Inherit: Same as product output' 'int8' 'int16' 'int32' 'fixdt(1,16,0)'
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as input' 'int8' 'int16' 'int32' 'fixdt(1,16,0)'
LockScale	Lock data type settings against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnIntegerOverflow	Saturate on integer overflow	{'off'} 'on'
StateName	State name	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
StateMustResolveToSignalObject	State name must resolve to Simulink signal object	{'off'} 'on'
StateSignalObject	Signal object class Code generation storage class	Simulink.Signal object Object of a class that is derived from Simulink.Signal
StateStorageClass	Code generation storage class	{'Auto'} 'SimulinkGlobal' 'ExportedGlobal' 'ImportedExtern' 'ImportedExternPointer' 'Custom'
RTWStateStorageTypeQualifier	Code generation storage type qualifier	{''}
Discrete Zero-Pole (DiscreteZeroPole)		
Zeros	Zeros	vector — {'[1]'} vector — {'[0 0.5]'} {'1'}
Poles	Poles	
Gain	Gain	
SampleTime	Sample time (-1 for inherited)	
StateName	State name	{''}
StateMustResolveToSignalObject	State name must resolve to Simulink signal object	{'off'} 'on'
StateSignalObject	Signal object class Code generation storage class	Simulink.Signal object Object of a class that is derived from Simulink.Signal
StateStorageClass	Code generation storage class	{'Auto'} 'SimulinkGlobal' 'ExportedGlobal' 'ImportedExtern' 'ImportedExternPointer' 'Custom'

Block (Type)/Parameter	Dialog Box Prompt	Values
RTWStateStorageType Qualifier	Code generation storage type qualifier	{''}
Discrete-Time Integrator (DiscreteIntegrator)		
IntegratorMethod	Integrator method	{'Integration: Forward Euler'} 'Integration: Backward Euler' 'Integration: Trapezoidal' 'Accumulation: Forward Euler' 'Accumulation: Backward Euler' 'Accumulation: Trapezoidal'
gainval	Gain value	{'1.0'}
ExternalReset	External reset	{'none'} 'rising' 'falling' 'either' 'level' 'sampled level'
InitialConditionSource	Initial condition source	{'internal'} 'external'
InitialCondition	Initial condition	scalar or vector — {'0'}
InitialConditionSetting	Initial condition setting	{'State (most efficient)'} 'Output' 'Compatibility'
SampleTime	Sample time (-1 for inherited)	{'1'}
LimitOutput	Limit output	{'off'} 'on'
UpperSaturationLimit	Upper saturation limit	scalar or vector — {'inf'}
LowerSaturationLimit	Lower saturation limit	scalar or vector — {'-inf'}
ShowSaturationPort	Show saturation port	{'off'} 'on'
ShowStatePort	Show state port	{'off'} 'on'
IgnoreLimit	Ignore limit and reset when linearizing	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
StateName	State name	{''}
StateMustResolveTo SignalObject	State name must resolve to Simulink signal object	{'off'} 'on'
StateSignalObject	Signal object class	Simulink.Signal object
	Code generation storage class	Object of a class that is derived from Simulink.Signal
StateStorageClass	Code generation storage class	{'Auto'} 'SimulinkGlobal' 'ExportedGlobal' 'ImportedExtern' 'ImportedExternPointer' 'Custom'

Block (Type)/Parameter	Dialog Box Prompt	Values
RTWStateStorageType Qualifier	Code generation storage type qualifier	{''}
First-Order Hold (First-Order Hold) (masked subsystem)		
Ts	Sample time	{'1'}
Memory (Memory)		
InitialCondition	Initial condition	scalar or vector — {'0'}
InheritSampleTime	Inherit sample time	{'off'} 'on'
LinearizeMemory	Direct feedthrough of input during linearization	{'off'} 'on'
LinearizeAsDelay	Treat as a unit delay when linearizing with discrete sample time	{'off'} 'on'
StateName	State name	{''}
StateMustResolveTo SignalObject	State name must resolve to Simulink signal object	{'off'} 'on'
StateSignalObject	Signal object class	Simulink.Signal object
	Code generation storage class	Object of a class that is derived from Simulink.Signal
StateStorageClass	Code generation storage class	{'Auto'} 'SimulinkGlobal' 'ExportedGlobal' 'ImportedExtern' 'ImportedExternPointer' 'Custom'
RTWStateStorageType Qualifier	Code generation storage type qualifier	{''}
Tapped Delay (S-Function) (Tapped Delay Line) (masked subsystem)		
vinit	Initial condition	{'0.0'}
samptime	Sample time	{'-1'}
NumDelays	Number of delays	{'4'}

Block (Type)/Parameter	Dialog Box Prompt	Values
DelayOrder	Order output vector starting with	{'Oldest'} 'Newest'
includeCurrent	Include current input in output vector	{'off'} 'on'
Transfer Fcn First Order (First Order Transfer Fcn) (masked subsystem)		
PoleZ	Pole (in Z plane)	{'0.95'}
ICPrevOutput	Initial condition for previous output	{'0.0'}
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
DoSatur	Saturate to max or min when overflows occur	{'off'} 'on'
Transfer Fcn Lead or Lag (Lead or Lag Compensator) (masked subsystem)		
PoleZ	Pole of compensator (in Z plane)	{'0.95'}
ZeroZ	Zero of compensator (in Z plane)	{'0.75'}
ICPrevOutput	Initial condition for previous output	{'0.0'}
ICPrevInput	Initial condition for previous input	{'0.0'}
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
DoSatur	Saturate to max or min when overflows occur	{'off'} 'on'
Transfer Fcn Real Zero (Transfer Fcn Real Zero) (masked subsystem)		
ZeroZ	Zero (in Z plane)	{'0.75'}
ICPrevInput	Initial condition for previous input	{'0.0'}

Block (Type)/Parameter	Dialog Box Prompt	Values
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
DoSatur	Saturate to max or min when overflows occur	{'off'} 'on'
Unit Delay (UnitDelay)		
InitialCondition	Initial condition	scalar or vector — {'0'}
InputProcessing	Input processing	'Columns as channels (frame based)' {'Elements as channels (sample based)'} 'Inherited'
SampleTime	Sample time (-1 for inherited)	{'-1'}
StateName	State name	{''}
StateMustResolveToSignalObject	State name must resolve to Simulink signal object	{'off'} 'on'
StateSignalObject	Signal object class	Simulink.Signal object
	Code generation storage class	Object of a class that is derived from Simulink.Signal
StateStorageClass	Code generation storage class	{'Auto'} 'SimulinkGlobal' 'ExportedGlobal' 'ImportedExtern' 'ImportedExternPointer' 'Custom'
CodeGenStateStorageTypeQualifier	Code generation storage type qualifier	{''}
Zero-Order Hold (ZeroOrderHold)		
SampleTime	Sample time (-1 for inherited)	{'1'}

Logic and Bit Operations Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Bit Clear (Bit Clear) (masked subsystem)		
iBit	Index of bit (0 is least significant)	{'0'}
Bit Set (Bit Set) (masked subsystem)		
iBit	Index of bit (0 is least significant)	{'0'}
Bitwise Operator (S-Function) (Bitwise Operator) (masked subsystem)		
logicop	Operator	{'AND'} {'OR'} {'NAND'} {'NOR'} {'XOR'} {'NOT'}
UseBitMask	Use bit mask ...	'off' {'on'}
NumInputPorts	Number of input ports	{'1'}
BitMask	Bit Mask	{'bin2dec('11011001')'}
BitMaskRealWorld	Treat mask as	'Real World Value' {'Stored Integer'}
Combinatorial Logic (CombinatorialLogic)		
TruthTable	Truth table	{'[0 0;0 1;0 1;1 0;0 1;1 0;1 0;1 1]'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Compare To Constant (Compare To Constant) (masked subsystem)		
relop	Operator	'==' '~=' '<' {'<='} '>' {'>='}
const	Constant value	{'3.0'}
OutDataTypeStr	Output data type	{'boolean'} 'uint8'
ZeroCross	Enable zero-crossing detection	'off' {'on'}
Compare To Zero (Compare To Zero) (masked subsystem)		
relop	Operator	'==' '~=' '<' {'<='} '>' {'>='}
OutDataTypeStr	Output data type	{'boolean'} 'uint8'

Block (Type)/Parameter	Dialog Box Prompt	Values
ZeroCross	Enable zero-crossing detection	'off' {'on'}
Detect Change (Detect Change) (masked subsystem)		
vinit	Initial condition	{'0'}
OutDataTypeStr	Output data type	{'boolean'} 'uint8'
Detect Decrease (Detect Decrease) (masked subsystem)		
vinit	Initial condition	{'0.0'}
OutDataTypeStr	Output data type	{'boolean'} 'uint8'
Detect Fall Negative (Detect Fall Negative) (masked subsystem)		
vinit	Initial condition	{'0'}
OutDataTypeStr	Output data type	{'boolean'} 'uint8'
Detect Fall Nonpositive (Detect Fall Nonpositive) (masked subsystem)		
vinit	Initial condition	{'0'}
OutDataTypeStr	Output data type	{'boolean'} 'uint8'
Detect Increase (Detect Increase) (masked subsystem)		
vinit	Initial condition	{'0.0'}
OutDataTypeStr	Output data type	{'boolean'} 'uint8'
Detect Rise Nonnegative (Detect Rise Nonnegative) (masked subsystem)		
vinit	Initial condition	{'0'}
OutDataTypeStr	Output data type	{'boolean'} 'uint8'
Detect Rise Positive (Detect Rise Positive) (masked subsystem)		
vinit	Initial condition	{'0'}
OutDataTypeStr	Output data type	{'boolean'} 'uint8'
Extract Bits (Extract Bits) (masked subsystem)		

Block (Type)/Parameter	Dialog Box Prompt	Values
bitsToExtract	Bits to extract	{'Upper half'} 'Lower half' 'Range starting with most significant bit' 'Range ending with least significant bit' 'Range of bits'
numBits	Number of bits	{'8'}
bitIdxRange	Bit indices ([start end], 0-based relative to LSB)	{'[0 7]'}
outScalingMode	Output scaling mode	{'Preserve fixed-point scaling'} 'Treat bit field as an integer'
Interval Test (Interval Test) (masked subsystem)		
IntervalClosedRight	Interval closed on right	'off' {'on'}
uplimit	Upper limit	{'0.5'}
IntervalClosedLeft	Interval closed on left	'off' {'on'}
lowlimit	Lower limit	{'-0.5'}
OutDataTypeStr	Output data type	{'boolean'} 'uint8'
Interval Test Dynamic (Interval Test Dynamic) (masked subsystem)		
IntervalClosedRight	Interval closed on right	'off' {'on'}
IntervalClosedLeft	Interval closed on left	'off' {'on'}
OutDataTypeStr	Output data type	{'boolean'} 'uint8'
Logical Operator (Logic)		
Operator	Operator	{'AND'} 'OR' 'NAND' 'NOR' 'XOR' 'NXOR' 'NOT'
Inputs	Number of input ports	{'2'}
IconShape	Icon shape	{'rectangular'} 'distinctive'
SampleTime	Sample time (-1 for inherited)	{'-1'}

Block (Type)/Parameter	Dialog Box Prompt	Values
AllPortsSameDT	Require all inputs and output to have the same data type	{'off'} 'on'
OutDataTypeStr	Output data type	'Inherit: Logical (see Configuration Parameters: Optimization)' {'boolean'} 'fixdt(1,16)'
Relational Operator (RelationalOperator)		
Operator	Relational operator	'==' '~=' '<' {'<='} '>=' '>' 'isInf' 'isNaN' 'isFinite'
ZeroCross	Enable zero-crossing detection	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
InputSameDT	Require all inputs to have the same data type	{'off'} 'on'
OutDataTypeStr	Output data type	'Inherit: Logical (see Configuration Parameters: Optimization)' {'boolean'} 'fixdt(1,16)'
Shift Arithmetic (ArithShift)		
BitShiftNumberSource	Bits to shift > Source	{'Dialog'} 'Input port'
BitShiftDirection	Bits to shift > Direction	'Left' 'Right' {'Bidirectional'}
BitShiftNumber	Bits to shift > Number	{'8'}
BinPtShiftNumber	Binary points to shift > Number	{'0'}
DiagnosticForOORShift	Diagnostic for out-of-range shift value	{'None'} 'Warning' 'Error'

Block (Type)/Parameter	Dialog Box Prompt	Values
CheckOORBitShift	Check for out-of-range 'Bits to shift' in generated code	{'off'} 'on'
nBitShiftRight	Deprecated in R2011a	
nBinPtShiftRight	Deprecated in R2011a	

Lookup Tables Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Cosine (Cosine) (masked subsystem)		
Formula	Output formula	'sin(2*pi*u)' {'cos(2*pi*u)'} 'exp(j*2*pi*u)' 'sin(2*pi*u) and cos(2*pi*u)'
NumDataPoints	Number of data points for lookup table	{'(2^5)+1'}
OutputWordLength	Output word length	{'16'}
InternalRulePriority	Internal rule priority for lookup table	{'Speed'} 'Precision'
Direct Lookup Table (n-D) (LookupNDDirect)		
NumberOfTableDimensions	Number of table dimensions	'1' {'2'} '3' '4'
InputsSelectThisObjectFromTable	Inputs select this object from table	{'Element'} 'Column' '2-D Matrix'
TableIsInput	Make table an input	{'off'} 'on'
Table	Table data	{'[4 5 6;16 19 20;10 18 23]}'
DiagnosticForOutOfRangeInput	Diagnostic for out-of-range input	'None' {'Warning'} 'Error'
SampleTime	Sample time (-1 for inherited)	{'-1'}
TableMin	Table minimum	{'[]'}
TableMax	Table maximum	{'[]'}
TableDataTypeStr	Table data type	{'Inherit: Inherit from Table data'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16)' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'

Block (Type)/Parameter	Dialog Box Prompt	Values
LockScale	Lock data type settings against changes by the fixed-point tools	{'off'} 'on'
maskTabDims	Deprecated in R2009b	
explicitNumDims	Deprecated in R2009b	
outDims	Deprecated in R2009b	
tabIsInput	Deprecated in R2009b	
mxTable	Deprecated in R2009b	
clipFlag	Deprecated in R2009b	
samptime	Deprecated in R2009b	
Interpolation Using Prelookup (Interpolation_n-D)		
NumberOfTableDimensions	Number of table dimensions	'1' {'2'} '3' '4'
Table	Table data > Value	{'sqrt([1:11]' * [1:11])'}
TableSource	Table data > Source	{'Dialog'} 'Input port'
TableSpecification	Specification	{'Explicit values'} 'Lookup table object' To set this parameter from 'Explicit values' to 'Lookup table object', use the same call to set_param to set the parameter LookupTableObject. For example: <pre>set_param('myModel/myInterpBlock', ... 'TableSpecification', ... 'Lookup table object', ... 'LookupTableObject', 'myLUTObject')</pre>
LookupTableObject	Name of lookup table object	{''}
InterpMethod	Interpolation method	'Flat' {'Linear'}
ExtrapMethod	Extrapolation method	'Clip' {'Linear'}

Block (Type)/Parameter	Dialog Box Prompt	Values
ValidIndexMayReachLast	Valid index input may reach last index	{'off'} 'on'
DiagnosticForOutOfRangeInput	Diagnostic for out-of-range input	{'None'} 'Warning' 'Error'
RemoveProtectionIndex	Remove protection against out-of-range index in generated code	{'off'} 'on'
NumSelectionDims	Number of sub-table selection dimensions	{'0'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
TableDataTypeStr	Table data > Data Type	'Inherit: Inherit from 'Table data'' {'Inherit: Same as output'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16)' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
TableMin	Table data > Minimum	{'[]'}
TableMax	Table data > Maximum	{'[]'}
IntermediateResultsDataStr	Intermediate results > Data Type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as output' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output > Data Type	'Inherit: Inherit via back propagation' {'Inherit: Inherit from table data'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
OutMin	Output > Minimum	{'[]'}
OutMax	Output > Maximum	{'[]'}
InternalRulePriority	Internal rule priority	{'Speed'} 'Precision'
LockScale	Lock data type settings against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnIntegerOverflow	Saturate on integer overflow	{'off'} 'on'
CheckIndexInCode	Deprecated in R2011a	
n-D Lookup Table, 1-D Lookup Table, 2-D Lookup Table (Lookup_n-D)		
NumberOfTableDimensions	Number of table dimensions	'1' '2' '3' '4'. Default is '1' for 1-D Lookup Table, '2' for 2-D Lookup Table, '3' for n-D Lookup Table.

Block (Type)/Parameter	Dialog Box Prompt	Values
DataSpecification	(n-D Lookup Table) Data specification	<p>{'Table and breakpoints'} 'Lookup table object'</p> <p>To set this parameter from 'Table and breakpoints' to 'Lookup table object', use the same call to set_param to set the parameter LookupTableObject. For example:</p> <pre>set_param('myModel/myLookupBlock', ... 'DataSpecification', 'Lookup table... object', ... 'LookupTableObject', 'myLUTObject')</pre>
LookupTableObject	Name of lookup table object.	{''}
Table	Table data	{'reshape repmat([4 5 6;16 19 20;10 18 23],1,2), [3,3,2])'}
BreakpointsSpecification	Breakpoints specification	{'Explicit values'} 'Even spacing'
BreakpointsForDimension1FirstPoint	First point	{'1'}
BreakpointsForDimension2FirstPoint	First point	{'1'}
BreakpointsForDimension3FirstPoint	First point	{'1'}
...
BreakpointsForDimension30FirstPoint	First point	{'1'}
BreakpointsForDimension1Spacing	Spacing	{'1'}
BreakpointsForDimension2Spacing	Spacing	{'1'}
BreakpointsForDimension3Spacing	Spacing	{'1'}

Block (Type)/Parameter	Dialog Box Prompt	Values
...
BreakpointsForDimension30Spacing	Spacing	{'1'}
BreakpointsForDimension1	Breakpoints 1	{'[10,22,31]'}
BreakpointsForDimension2	Breakpoints 2	{'[10,22,31]'}
BreakpointsForDimension3	Breakpoints 3	{'[5, 7]'}
...
BreakpointsForDimension30	Breakpoints 30	{'[1:3]'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
InterpMethod	Interpolation method	'Flat' {'Linear'} 'Cubic spline'
ExtrapMethod	Extrapolation method	'Clip' {'Linear'} 'Cubic spline'
UseLastTableValue	Use last table value for inputs at or above last breakpoint	{'off'} 'on'
DiagnosticForOutOfRangeInput	Diagnostic for out-of-range input	{'None'} 'Warning' 'Error'
RemoveProtectionInput	Remove protection against out-of-range input in generated code	{'off'} 'on'
IndexSearchMethod	Index search method	'Evenly spaced points' 'Linear search' {'Binary search'}
BeginIndexSearchUsingPreviousIndexResult	Begin index search using previous index result	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
UseOneInputPortForAll InputData	Use one input port for all input data	{'off'} 'on'
SupportTunableTableSize	Support tunable table size in code generation	{'off'} 'on'
MaximumIndicesForEach Dimension	Maximum indices for each dimension	{'[]'}
TableDataTypeStr	Table data > Data Type	'Inherit: Inherit from 'Table data'' {'Inherit: Same as output'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16)' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
TableMin	Table data > Minimum	{'[]'}
TableMax	Table data > Maximum	{'[]'}
BreakpointsForDimension1 DataTypeStr	Breakpoints 1 > Data Type	{'Inherit: Same as corresponding input'} 'Inherit: Inherit from 'Breakpoint data'' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16)' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
BreakpointsForDimension1 Min	Breakpoints 1 > Minimum	{'[]'}
BreakpointsForDimension1 Max	Breakpoints 1 > Maximum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
BreakpointsForDimension2 DataTypeStr	Breakpoints 2 > Data Type	{'Inherit: Same as corresponding input'} 'Inherit: Inherit from 'Breakpoint data' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16)' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
BreakpointsForDimension2 Min	Breakpoints 2 > Minimum	{'[]'}
BreakpointsForDimension2 Max	Breakpoints 2 > Maximum	{'[]'}
...
BreakpointsForDimension30 DataTypeStr	Breakpoints 30 > Data Type	{'Inherit: Same as corresponding input'} 'Inherit: Inherit from 'Breakpoint data' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16)' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
BreakpointsForDimension30 Min	Breakpoints 30 > Minimum	{'[]'}
BreakpointsForDimension30 Max	Breakpoints 30 > Maximum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
FractionDataTypeStr	Fraction > Data Type	{'Inherit: Inherit via internal rule'} 'double' 'single' 'fixdt(1,16,0)'
IntermediateResults DataTypeStr	Intermediate results > Data Type	'Inherit: Inherit via internal rule' {'Inherit: Same as output'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
OutDataTypeStr	Output > Data Type	'Inherit: Inherit via back propagation' 'Inherit: Inherit from table data' {'Inherit: Same as first input'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
OutMin	Output > Minimum	{'[]'}
OutMax	Output > Maximum	{'[]'}
InternalRulePriority	Internal rule priority	{'Speed'} 'Precision'
InputSameDT	Require all inputs to have the same data type	'off' {'on'}
LockScale	Lock data type settings against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' 'Floor' 'Nearest' 'Round' {'Simplest'} 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
ProcessOutOfRangeInput	Deprecated in R2009b	

Block (Type)/Parameter	Dialog Box Prompt	Values
Lookup Table Dynamic (Lookup Table Dynamic) (masked subsystem)		
LookUpMeth	Lookup Method	'Interpolation-Extrapolation' {'Interpolation-Use End Values'} 'Use Input Nearest' 'Use Input Below' 'Use Input Above'
OutDataTypeStr	Output data type	{'fixdt('double')} 'Inherit: Inherit via back propagation' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
OutputDataTypeScaling Mode	Deprecated in R2007b	
OutDataType	Deprecated in R2007b	
OutScaling	Deprecated in R2007b	
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
DoSatur	Saturate to max or min when overflows occur	{'off'} 'on'
Prelookup (PreLookup)		

Block (Type)/Parameter	Dialog Box Prompt	Values
BreakpointsSpecification	Specification	<p>{'Explicit values'} 'Even spacing' 'Breakpoint object'</p> <p>To set this parameter from 'Explicit values' or 'Even spacing' to 'Breakpoint object', use the same call to set_param to set the parameter BreakpointObject. For example:</p> <pre>set_param('myModel/myPrelookupBlock', ... 'BreakpointsSpecification', ... 'Breakpoint object', ... 'BreakpointObject', 'myBPObject')</pre>
BreakpointObject	Name of breakpoint object	{''}
BreakpointsFirstPoint	First point	{'10'}
BreakpointsSpacing	Spacing	{'10'}
BreakpointsNumPoints	Number of points	{'11'}
BreakpointsData	Value	{'[10:10:110]'}
BreakpointsDataSource	Source	{'Dialog'} 'Input port'
IndexSearchMethod	Index search method	'Evenly spaced points' 'Linear search' {'Binary search'}
BeginIndexSearchUsingPreviousIndexResult	Begin index search using previous index result	{'off'} 'on'
OutputOnlyTheIndex	Output only the index	{'off'} 'on'
ExtrapMethod	Extrapolation method	'Clip' {'Linear'}
UseLastBreakpoint	Use last breakpoint for input at or above upper limit	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
DiagnosticForOutOfRange Input	Diagnostic for out-of-range input	{'None'} 'Warning' 'Error'
RemoveProtectionInput	Remove protection against out-of-range input in generated code	{'off'} 'on'
SampleTime	Sample time (-1 for inherited)	{'-1'}
BreakpointDataTypeStr	Breakpoint > Data Type	{'Inherit: Same as input'} 'Inherit: Inherit from 'Breakpoint data' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16)' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
BreakpointMin	Breakpoint > Minimum	{'[]'}
BreakpointMax	Breakpoint > Maximum	{'[]'}
IndexDataTypeStr	Index > Data Type	'int8' 'uint8' 'int16' 'uint16' 'int32' {'uint32'} 'fixdt(1,16)'
FractionDataTypeStr	Fraction > Data Type	{'Inherit: Inherit via internal rule'} 'double' 'single' 'fixdt(1,16,0)'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
ProcessOutOfRangeInput	Deprecated in R2011a	
Sine (Sine) (masked subsystem)		

Block (Type)/Parameter	Dialog Box Prompt	Values
Formula	Output formula	{ 'sin(2*pi*u)' } 'cos(2*pi*u)' 'exp(j*2*pi*u)' 'sin(2*pi*u) and cos(2*pi*u)'
NumDataPoints	Number of data points for lookup table	{ '(2^5)+1' }
OutputWordLength	Output word length	{ '16' }
InternalRulePriority	Internal rule priority for lookup table	{ 'Speed' } 'Precision'

Math Operations Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Abs (Abs)		
ZeroCross	Enable zero-crossing detection	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	'Inherit: Inherit via internal rule' 'Inherit: Inherit via back propagation' {'Inherit: Same as input'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
Add (Sum)		
IconShape	Icon shape	{'rectangular'} 'round'
Inputs	List of signs	{'++'}

Block (Type)/Parameter	Dialog Box Prompt	Values
CollapseMode	Sum over	{'All dimensions'} 'Specified dimension'
CollapseDim	Dimension	{'1'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
InputSameDT	Require all inputs to have the same data type	{'off'} 'on'
AccumDataTypeStr	Accumulator data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as first input' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'Inherit: Same as first input' 'Inherit: Same as accumulator' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock data type settings against changes by the fixed-point tools	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnIntegerOverflow	Saturate on integer overflow	{'off'} 'on'
Algebraic Constraint (Algebraic Constraint)		
Constraint	Constraint on input signal	{'f(z) = 0'} 'f(z) = z'
Solver	Algebraic Loop Solver	{'auto'} 'Trust Region' 'Line Search'
Tolerance	Solver Tolerance	{'auto'}
InitialGuess	Initial guess	{'0'}
Assignment (Assignment)		
NumberOfDimensions	Number of output dimensions	{'1'}
IndexMode	Index mode	'Zero-based' {'One-based'}
OutputInitialize	Initialize output (Y)	{'Initialize using input port <Y0>'} 'Specify size for each dimension in table'
IndexOptionArray	Index Option	'Assign all' {'Index vector (dialog)'} 'Index vector (port)' 'Starting index (dialog)' 'Starting index (port)'
IndexParamArray	Index	cell array
OutputSizeArray	Output Size	cell array
DiagnosticForDimensions	Action if any output element is not assigned	'Error' 'Warning' {'None'}
SampleTime	Sample time (-1 for inherited)	{'-1'}

Block (Type)/Parameter	Dialog Box Prompt	Values
IndexOptions	See IndexOptionArray on page 6-0 parameter for more information.	
Indices	See IndexParamArray on page 6-0 parameter for more information.	
OutputSizes	See OutputSizeArray on page 6-0 parameter for more information.	
Bias (Bias)		
Bias	Bias	{'0.0'}
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
Complex to Magnitude-Angle (ComplexToMagnitudeAngle)		
Output	Output	'Magnitude' 'Angle' {'Magnitude and angle'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Complex to Real-Imag (ComplexToRealImag)		
Output	Output	'Real' 'Imag' {'Real and imag'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Divide (Product)		
Inputs	Number of inputs	{'*/'}
Multiplication	Multiplication	{'Element-wise(*)'} 'Matrix(*)'
SampleTime	Sample time (-1 for inherited)	{'-1'}
InputSameDT	Require all inputs to have same data type	{'off'} 'on'
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'Inherit: Same as first input' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
Dot Product (DotProduct)		
SampleTime	Sample time (-1 for inherited)	{'-1'}
InputSameDT	Require all inputs to have same data type	'off' {'on'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'Inherit: Same as first input' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
OutputDataTypeScaling Mode	Deprecated in R2007b	
OutDataType	Deprecated in R2007b	
OutScaling	Deprecated in R2007b	
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
Find (Find)		
IndexOutputFormat	Index output format	{'Linear indices'} 'Subscripts'
NumberOfInputDimensions	Number of input dimensions	integer — {'1'}
IndexMode	Index mode	{'Zero-based'} 'One-based'

Block (Type)/Parameter	Dialog Box Prompt	Values
ShowOutputForNonzero InputValues	Show output port for nonzero input values	{'off'} 'on'
SampleTime	Sample time (-1 for inherited)	{'-1'}
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16)'
Gain (Gain)		
Gain	Gain	{'1'}
Multiplication	Multiplication	{'Element-wise (K.*u)'} 'Matrix(K*u)' 'Matrix(u*K)' 'Matrix(K*u) (u vector)'
SampleTime	Sample time (-1 for inherited)	{'-1'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'Inherit: Same as input' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock output data type setting against changes by the fixed- point tools	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnIntegerOverflow	Saturate on integer overflow	{'off'} 'on'
ParamMin	Parameter minimum	{'[]'}
ParamMax	Parameter maximum	{'[]'}
ParamDataTypeStr	Parameter data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as input' 'Inherit: Inherit from Gain' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16)' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
Magnitude-Angle to Complex (MagnitudeAngleToComplex)		
Input	Input	'Magnitude' 'Angle' {'Magnitude and angle'}
ConstantPart	Magnitude or Angle	{'0'}
ApproximationMethod	Approximation method	{'None'} 'CORDIC'
NumberOfIterations	Number of iterations	{'11'}
ScaleReciprocalGainFactor	Scale output by reciprocal of gain factor	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Math Function (Math)		

Block (Type)/Parameter	Dialog Box Prompt	Values
Operator	Function	{'exp'} 'log' '10^u' 'log10' 'magnitude^2' 'square' 'pow' 'conj' 'reciprocal' 'hypot' 'rem' 'mod' 'transpose' 'hermitian'
OutputSignalType	Output signal type	{'auto'} 'real' 'complex'
SampleTime	Sample time (-1 for inherited)	{'-1'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	'Inherit: Inherit via internal rule' 'Inherit: Inherit via back propagation' {'Inherit: Same as first input'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock output data type setting against changes by the fixed- point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	'off' {'on'}

Block (Type)/Parameter	Dialog Box Prompt	Values
Matrix Concatenate (Concatenate)		
NumInputs	Number of inputs	{'2'}
Mode	Mode	'Vector' {'Multidimensional array'}
ConcatenateDimension	Concatenate dimension	{'2'}
MinMax (MinMax)		
Function	Function	{'min'} 'max'
Inputs	Number of input ports	{'1'}
ZeroCross	Enable zero-crossing detection	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
InputSameDT	Require all inputs to have the same data type	{'off'} 'on'
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'

Block (Type)/Parameter	Dialog Box Prompt	Values
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
MinMax Running Resettable (MinMax Running Resettable) (masked subsystem)		
Function	Function	{'min'} 'max'
vinit	Initial condition	{'0.0'}
Permute Dimensions (PermuteDimensions)		
Order	Order	{'[2,1]}'}
Polynomial (Polyval)		
coefs	Polynomial Coefficients	{'[+2.081618890e-019, -1.441693666e-014, +4.719686976e-010, -8.536869453e-006, +1.621573104e-001, -8.087801117e+001]'}
Product (Product)		
Inputs	Number of inputs	{'2'}
Multiplication	Multiplication	{'Element-wise(*)'} 'Matrix(*)'
CollapseMode	Multiply over	{'All dimensions'} 'Specified dimension'
CollapseDim	Dimension	{'1'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
InputSameDT	Require all inputs to have same data type	{'off'} 'on'
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'Inherit: Same as first input' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' 'Floor' 'Nearest' 'Round' 'Simplest' {'Zero'}
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
Product of Elements (Product)		
Inputs	Number of inputs	{ '*' }
Multiplication	Multiplication	{ 'Element-wise(.*)' } 'Matrix(*)'
CollapseMode	Multiply over	{ 'All dimensions' } 'Specified dimension'
CollapseDim	Dimension	{ '1' }
SampleTime	Sample time (-1 for inherited)	{ '-1' }
InputSameDT	Require all inputs to have same data type	{ 'off' } 'on'
OutMin	Output minimum	{ '[]' }
OutMax	Output maximum	{ '[]' }

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'Inherit: Same as first input' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
Real-Imag to Complex (RealImagToComplex)		
Input	Input	'Real' 'Imag' {'Real and imag'}
ConstantPart	Real part or Imag part	{'0'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Reciprocal Sqrt (Sqrt)		
Operator	Function	'sqrt' 'signedSqrt' {'rSqrt'}
OutputSignalType	Output signal type	{'auto'} 'real' 'complex'
SampleTime	Sample time (-1 for inherited)	{'-1'}
OutMin	Output minimum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	'Inherit: Inherit via internal rule' 'Inherit: Inherit via back propagation' {'Inherit: Same as first input'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	'off' {'on'}
IntermediateResults DataTypeStr	Intermediate results data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit from input' 'Inherit: Inherit from output' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
AlgorithmType	Method	'Exact' {'Newton-Raphson'}

Block (Type)/Parameter	Dialog Box Prompt	Values
Iterations	Number of iterations	{'3'}
Reshape (Reshape)		
OutputDimensionality	Output dimensionality	{'1-D array'} 'Column vector (2-D)' 'Row vector (2-D)' 'Customize' 'Derive from reference input port'
OutputDimensions	Output dimensions	{'[1,1]}'
Rounding Function (Rounding)		
Operator	Function	{'floor'} 'ceil' 'round' 'fix'
SampleTime	Sample time (-1 for inherited)	{'-1'}
Sign (Signum)		
ZeroCross	Enable zero-crossing detection	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Signed Sqrt (Sqrt)		
Operator	Function	'sqrt' {'signedSqrt'} 'rSqrt'
OutputSignalType	Output signal type	{'auto'} 'real' 'complex'
SampleTime	Sample time (-1 for inherited)	{'-1'}
OutMin	Output minimum	{'[]}'
OutMax	Output maximum	{'[]}'

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output data type	'Inherit: Inherit via internal rule' 'Inherit: Inherit via back propagation' {'Inherit: Same as first input'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	'off' {'on'}
IntermediateResults DataTypeStr	Intermediate results data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit from input' 'Inherit: Inherit from output' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
AlgorithmType	Method	{'Exact'} 'Newton-Raphson'
Iterations	Number of iterations	{'3'}

Block (Type)/Parameter	Dialog Box Prompt	Values
Sine Wave Function (Sin)		
SineType	Sine type	{'Time based'} 'Sample based'
TimeSource	Time	'Use simulation time' {'Use external signal'}
Amplitude	Amplitude	{'1'}
Bias	Bias	{'0'}
Frequency	Frequency	{'1'}
Phase	Phase	{'0'}
Samples	Samples per period	{'10'}
Offset	Number of offset samples	{'0'}
SampleTime	Sample time	{'0'}
VectorParams1D	Interpret vector parameters as 1-D	'off' {'on'}
Slider Gain (Slider Gain) (masked subsystem)		
low	Low	{'0'}
gain	Gain	{'1'}
high	High	{'2'}
Sqrt (Sqrt)		
Operator	Function	{'sqrt'} 'signedSqrt' 'rSqrt'
OutputSignalType	Output signal type	{'auto'} 'real' 'complex'
SampleTime	Sample time (-1 for inherited)	{'-1'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output data type	'Inherit: Inherit via internal rule' 'Inherit: Inherit via back propagation' {'Inherit: Same as first input'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	'off' {'on'}
IntermediateResults DataTypeStr	Intermediate results data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit from input' 'Inherit: Inherit from output' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
AlgorithmType	Method	{'Exact'} 'Newton-Raphson'
Iterations	Number of iterations	{'3'}

Block (Type)/Parameter	Dialog Box Prompt	Values
Squeeze (Squeeze) (masked subsystem)		
None	None	None
Subtract (Sum)		
IconShape	Icon shape	{'rectangular'} 'round'
Inputs	List of signs	{'+-'}
CollapseMode	Sum over	{'All dimensions'} 'Specified dimension'
CollapseDim	Dimension	{'1'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
InputSameDT	Require all inputs to have the same data type	{'off'} 'on'
AccumDataTypeStr	Accumulator data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as first input' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'Inherit: Same as first input' 'Inherit: Same as accumulator' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock data type settings against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
Sum (Sum)		
IconShape	Icon shape	'rectangular' {'round'}
Inputs	List of signs	{' ++'}
CollapseMode	Sum over	{'All dimensions'} 'Specified dimension'
CollapseDim	Dimension	{'1'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
InputSameDT	Require all inputs to have the same data type	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
AccumDataTypeStr	Accumulator data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as first input' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'Inherit: Same as first input' 'Inherit: Same as accumulator' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock data type settings against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
Sum of Elements (Sum)		

Block (Type)/Parameter	Dialog Box Prompt	Values
IconShape	Icon shape	{'rectangular'} 'round'
Inputs	List of signs	{'+'}
CollapseMode	Sum over	{'All dimensions'} 'Specified dimension'
CollapseDim	Dimension	{'1'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
InputSameDT	Require all inputs to have the same data type	{'off'} 'on'
AccumDataTypeStr	Accumulator data type	{'Inherit: Inherit via internal rule'} 'Inherit: Same as first input' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'Inherit: Same as first input' 'Inherit: Same as accumulator' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'

Block (Type)/Parameter	Dialog Box Prompt	Values
LockScale	Lock data type settings against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
Trigonometric Function (Trigonometry)		
Operator	Function	{'sin'} 'cos' 'tan' 'asin' 'acos' 'atan' 'atan2' 'sinh' 'cosh' 'tanh' 'asinh' 'acosh' 'atanh' 'sincos' 'cos + jsin'
ApproximationMethod	Approximation method	{'None'} 'CORDIC'
NumberOfIterations	Number of iterations	{'11'}
OutputSignalType	Output signal type	{'auto'} 'real' 'complex'
SampleTime	Sample time (-1 for inherited)	{'-1'}
Unary Minus (UnaryMinus)		
SampleTime	Sample time (-1 for inherited)	{'-1'}
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
Vector Concatenate (Concatenate)		
NumInputs	Number of inputs	{'2'}
Mode	Mode	{'Vector'} 'Multidimensional array'
Weighted Sample Time Math (SampleTimeMath)		

Block (Type)/Parameter	Dialog Box Prompt	Values
TsampMathOp	Operation	{ '+' } '-' '*' '/' 'Ts Only' '1/Ts Only'
weightValue	Weight value	{ '1.0' }
TsampMathImp	Implement using	{ 'Online Calculations' } 'Offline Scaling Adjustment'
OutDataTypeStr	Output data type	{ 'Inherit via internal rule' } 'Inherit via back propagation'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' { 'Floor' } 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{ 'off' } 'on'
OutputDataTypeScaling Mode	Deprecated in R2009b	
DoSatur	Deprecated in R2009b	

Model Verification Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Assertion (Assertion)		
Enabled	Enable assertion	'off' {'on'}
AssertionFailFcn	Simulation callback when assertion fails	{''}
StopWhenAssertionFail	Stop simulation when assertion fails	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Check Dynamic Gap (Checks_DGap) (masked subsystem)		
enabled	Enable assertion	'off' {'on'}
callback	Simulation callback when assertion fails (optional)	{''}
stopWhenAssertionFail	Stop simulation when assertion fails	'off' {'on'}
export	Output assertion signal	{'off'} 'on'
icon	Select icon type	{'graphic'} 'text'
Check Dynamic Range (Checks_DRange) (masked subsystem)		
enabled	Enable assertion	'off' {'on'}
callback	Simulation callback when assertion fails (optional)	{''}
stopWhenAssertionFail	Stop simulation when assertion fails	'off' {'on'}
export	Output assertion signal	{'off'} 'on'
icon	Select icon type	{'graphic'} 'text'
Check Static Gap (Checks_SGap) (masked subsystem)		
max	Upper bound	{'100'}
max_included	Inclusive upper bound	'off' {'on'}
min	Lower bound	{'0'}
min_included	Inclusive lower bound	'off' {'on'}

Block (Type)/Parameter	Dialog Box Prompt	Values
enabled	Enable assertion	'off' {'on'}
callback	Simulation callback when assertion fails (optional)	{''}
stopWhenAssertionFail	Stop simulation when assertion fails	'off' {'on'}
export	Output assertion signal	{'off'} 'on'
icon	Select icon type	{'graphic'} 'text'
Check Static Range (Checks_SRange) (masked subsystem)		
max	Upper bound	{'100'}
max_included	Inclusive upper bound	'off' {'on'}
min	Lower bound	{'0'}
min_included	Inclusive lower bound	'off' {'on'}
enabled	Enable assertion	'off' {'on'}
callback	Simulation callback when assertion fails (optional)	{''}
stopWhenAssertionFail	Stop simulation when assertion fails	'off' {'on'}
export	Output assertion signal	{'off'} 'on'
icon	Select icon type	{'graphic'} 'text'
Check Discrete Gradient (Checks_Gradient) (masked subsystem)		
gradient	Maximum gradient	{'1'}
enabled	Enable assertion	'off' {'on'}
callback	Simulation callback when assertion fails (optional)	{''}
stopWhenAssertionFail	Stop simulation when assertion fails	'off' {'on'}
export	Output assertion signal	{'off'} 'on'
icon	Select icon type	{'graphic'} 'text'
Check Dynamic Lower Bound (Checks_DMin) (masked subsystem)		

Block (Type)/Parameter	Dialog Box Prompt	Values
Enabled	Enable assertion	'off' {'on'}
callback	Simulation callback when assertion fails (optional)	{''}
stopWhenAssertionFail	Stop simulation when assertion fails	'off' {'on'}
export	Output assertion signal	{'off'} 'on'
icon	Select icon type	{'graphic'} 'text'
Check Dynamic Upper Bound (Checks_DMax) (masked subsystem)		
enabled	Enable assertion	'off' {'on'}
callback	Simulation callback when assertion fails (optional)	{''}
stopWhenAssertionFail	Stop simulation when assertion fails	'off' {'on'}
export	Output assertion signal	{'off'} 'on'
icon	Select icon type	{'graphic'} 'text'
Check Input Resolution (Checks_Resolution) (masked subsystem)		
resolution	Resolution	{'1'}
enabled	Enable assertion	'off' {'on'}
callback	Simulation callback when assertion fails (optional)	{''}
stopWhenAssertionFail	Stop simulation when assertion fails	'off' {'on'}
export	Output assertion signal	{'off'} 'on'
Check Static Lower Bound (Checks_SMin) (masked subsystem)		
min	Lower bound	{'0'}
min_included	Inclusive boundary	'off' {'on'}
enabled	Enable assertion	'off' {'on'}
callback	Simulation callback when assertion fails (optional)	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
stopWhenAssertionFail	Stop simulation when assertion fails	'off' {'on'}
export	Output assertion signal	{'off'} 'on'
icon	Select icon type	{'graphic'} 'text'
Check Static Upper Bound (Checks_SMax) (masked subsystem)		
max	Upper bound	{'0'}
max_included	Inclusive boundary	'off' {'on'}
enabled	Enable assertion	'off' {'on'}
callback	Simulation callback when assertion fails (optional)	{''}
stopWhenAssertionFail	Stop simulation when assertion fails	'off' {'on'}
export	Output assertion signal	{'off'} 'on'
icon	Select icon type	{'graphic'} 'text'

Model-Wide Utilities Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Block Support Table (Block Support Table) (masked subsystem)		
DocBlock (DocBlock) (masked subsystem)		
ECoderFlag	Embedded Coder Flag	{ '' }
DocumentType	Document Type	{ 'Text' } 'RTF' 'HTML'
Model Info (CMBlock) (masked subsystem)		
InitialSaveTempField	InitialSaveTempField	{ '' }
InitialBlockCM	InitialBlockCM	{ 'None' }
BlockCM	BlockCM	{ 'None' }
Frame	Show block frame	'off' { 'on' }
SaveTempField	SaveTempField	{ '' }
DisplayStringWithTags	DisplayStringWithTags	{ 'Model Info' }
MaskDisplayString	MaskDisplayString	{ 'Model Info' }
HorizontalTextAlignment	Horizontal text alignment	{ 'Center' }
LeftAlignmentValue	LeftAlignmentValue	{ '0.5' }
SourceBlockDiagram	SourceBlockDiagram	{ 'untitled' }
TagMaxNumber	TagMaxNumber	{ '20' }
CMTag1	CMTag1	{ '' }
CMTag2	CMTag2	{ '' }
CMTag3	CMTag3	{ '' }
CMTag4	CMTag4	{ '' }
CMTag5	CMTag5	{ '' }
CMTag6	CMTag6	{ '' }
CMTag7	CMTag7	{ '' }
CMTag8	CMTag8	{ '' }
CMTag9	CMTag9	{ '' }

Block (Type)/Parameter	Dialog Box Prompt	Values
CMTag10	CMTag10	{''}
CMTag11	CMTag11	{''}
CMTag12	CMTag12	{''}
CMTag13	CMTag13	{''}
CMTag14	CMTag14	{''}
CMTag15	CMTag15	{''}
CMTag16	CMTag16	{''}
CMTag17	CMTag17	{''}
CMTag18	CMTag18	{''}
CMTag19	CMTag19	{''}
CMTag20	CMTag20	{''}
Timed-Based Linearization (Timed Linearization) (masked subsystem)		
LinearizationTime	Linearization time	{'1'}
SampleTime	Sample time (of linearized model)	{'0'}
Trigger-Based Linearization (Triggered Linearization) (masked subsystem)		
TriggerType	Trigger type	{'rising'} 'falling' 'either' 'function-call'
SampleTime	Sample time (of linearized model)	{'0'}

Ports & Subsystems Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Action Port (ActionPort)		
InitializeStates	States when execution is resumed	{'held'} 'reset'
PropagateVarSize	Propagate sizes of variable-size signals	{'Only when execution is resumed'} 'During execution'
Atomic Subsystem (SubSystem)		
ShowPortLabels	Show port labels Note The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.	'none' {'FromPortIcon'} 'FromPortBlockName' 'SignalName' 'off' 'on'
BlockChoice	Block choice	{''}
TemplateBlock	Template block	{''}
MemberBlocks	Member blocks	{''}
Permissions	Read/Write permissions	{'ReadWrite'} 'ReadOnly' 'NoReadOrWrite'
ErrorFcn	Name of error callback function	{''}
PermitHierarchicalResolution	Permit hierarchical resolution	{'All'} 'ExplicitOnly' 'None'
TreatAsAtomicUnit	Treat as atomic unit	'off' {'on'}
TreatAsGroupedWhenPropagatingVariantConditions	Treat as grouped when propagating variant conditions	'off' {'on'}
MinAlgLoopOccurrences	Minimize algebraic loop occurrences	{'off'} 'on'
PropExecContextOutsideSubsystem	Propagate execution context across subsystem boundary	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
CheckFcnCallInp InsideContextMsg	Warn if function-call inputs are context-specific	{'off'} 'on'
SystemSampleTime	Sample time (-1 for inherited)	{'-1'}
RTWSystemCode	Function packaging	{'Auto'} 'Inline' 'Nonreusable function' 'Reusable function'
RTWFcnNameOpts	Function name options	{'Auto'} 'Use subsystem name' 'User specified'
RTWFcnName	Function name	{''}
RTWFileNameOpts	File name options	{'Auto'} 'Use subsystem name' 'Use function name' 'User specified'
RTWFileName	File name (no extension)	{''}
FunctionInterfaceSpec	Function interface This parameter requires a license for Embedded Coder software and an ERT-based system target file.	{'void_void'} 'Allow arguments'
FunctionWithSeparateData	“Function with separate data” on page 1-0 This parameter requires a license for Embedded Coder software and an ERT-based system target file.	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
RTWMemSecFuncInitTerm	<p>“Memory section for initialize/terminate functions” on page 1-0</p> <p>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</p>	<p>{'Inherit from model'} 'Default' list of memory sections from model's package</p>
RTWMemSecFuncExecute	<p>“Memory section for execution functions” on page 1-0</p> <p>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</p>	<p>{'Inherit from model'} 'Default' list of memory sections from model's package</p>
RTWMemSecDataConstants	<p>“Memory section for constants” on page 1-0</p> <p>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</p>	<p>{'Inherit from model'} 'Default' list of memory sections from model's package</p>
RTWMemSecDataInternal	<p>“Memory section for internal data” on page 1-0</p> <p>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</p>	<p>{'Inherit from model'} 'Default' list of memory sections from model's package</p>
RTWMemSecDataParameters	<p>“Memory section for parameters” on page 1-0</p> <p>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</p>	<p>{'Inherit from model'} 'Default' list of memory sections from model's package</p>

Block (Type)/Parameter	Dialog Box Prompt	Values
DataTypeOverride	No dialog box prompt Specifies data type used to override fixed-point data types. Set by Data type override on the Fixed-Point Tool.	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'
MinMaxOverflowLogging	No dialog box prompt Setting for fixed-point instrumentation. Set by Fixed-point instrumentation mode on the Fixed-Point Tool.	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'ForceOff'
SimViewingDevice	No dialog box prompt If set to 'on', designates the block as a Signal Viewing Subsystem — an atomic subsystem that encapsulates processing and viewing of signals received from the target system in External mode. For more information, see “Signal Viewing Subsystems” (Simulink Coder).	{'off'} 'on'
IsSubsystemVirtual		boolean — {'off'} 'on' Read-only
Code Reuse Subsystem (SubSystem)		
ShowPortLabels	Show port labels Note The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.	'none' {'FromPortIcon'} 'FromPortBlockName' 'SignalName' 'off' 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
BlockChoice	Block choice	{''}
TemplateBlock	Template block	{''}
MemberBlocks	Member blocks	{''}
Permissions	Read/Write permissions	{'ReadWrite'} 'ReadOnly' 'NoReadOrWrite'
ErrorFcn	Name of error callback function	{''}
PermitHierarchical Resolution	Permit hierarchical resolution	{'All'} 'ExplicitOnly' 'None'
TreatAsAtomicUnit	Treat as atomic unit	'off' {'on'}
MinAlgLoopOccurrences	Minimize algebraic loop occurrences	{'off'} 'on'
PropExecContext OutsideSubsystem	Propagate execution context across subsystem boundary	{'off'} 'on'
CheckFcnCallInp InsideContextMsg	Warn if function-call inputs are context-specific	{'off'} 'on'
SystemSampleTime	Sample time (-1 for inherited)	{'-1'}
RTWSystemCode	Function packaging	'Auto' 'Inline' 'Nonreusable function' {'Reusable function'}
RTWFcnNameOpts	Function name options	'Auto' {'Use subsystem name'} 'User specified'
RTWFcnName	Function name	{''}
RTWFileNameOpts	File name options	'Auto' {'Use subsystem name'} 'Use function name' 'User specified'
RTWFileName	File name (no extension)	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
RTWMemSecFuncInitTerm	<p>“Memory section for initialize/terminate functions” on page 1-0</p> <p>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</p>	<p>{'Inherit from model'} 'Default' list of memory sections from model's package</p>
RTWMemSecFuncExecute	<p>“Memory section for execution functions” on page 1-0</p> <p>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</p>	<p>{'Inherit from model'} 'Default' list of memory sections from model's package</p>
RTWMemSecDataConstants	<p>“Memory section for constants” on page 1-0</p> <p>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</p>	<p>{'Inherit from model'} 'Default' list of memory sections from model's package</p>
RTWMemSecDataInternal	<p>“Memory section for internal data” on page 1-0</p> <p>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</p>	<p>{'Inherit from model'} 'Default' list of memory sections from model's package</p>
RTWMemSecDataParameters	<p>“Memory section for parameters” on page 1-0</p> <p>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</p>	<p>{'Inherit from model'} 'Default' list of memory sections from model's package</p>

Block (Type)/Parameter	Dialog Box Prompt	Values
DataTypeOverride	No dialog box prompt Specifies data type used to override fixed-point data types. Set by Data type override on the Fixed-Point Tool.	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'
MinMaxOverflowLogging	No dialog box prompt Setting for fixed-point instrumentation. Set by Fixed-point instrumentation mode on the Fixed-Point Tool.	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'ForceOff'
IsSubsystemVirtual		boolean — {'off'} 'on' Read-only
Configurable Subsystem (SubSystem)		
ShowPortLabels	Show port labels Note The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.	'none' {'FromPortIcon'} 'FromPortBlockName' 'SignalName' 'off' 'on'
BlockChoice	Block choice	{''}
TemplateBlock	Template block	{'self'}
MemberBlocks	Member blocks	{''}
Permissions	Read/Write permissions	{'ReadWrite'} 'ReadOnly' 'NoReadOrWrite'
ErrorFcn	Name of error callback function	{''}
PermitHierarchicalResolution	Permit hierarchical resolution	{'All'} 'ExplicitOnly' 'None'
TreatAsAtomicUnit	Treat as atomic unit	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
MinAlgLoopOccurrences	Minimize algebraic loop occurrences	{'off'} 'on'
PropExecContext OutsideSubsystem	Propagate execution context across subsystem boundary	{'off'} 'on'
CheckFcnCallInp InsideContextMsg	Warn if function-call inputs are context-specific	{'off'} 'on'
SystemSampleTime	Sample time (-1 for inherited)	{'-1'}
RTWSystemCode	Function packaging	{'Auto'} 'Inline' 'Nonreusable function' 'Reusable function'
RTWFcnNameOpts	Function name options	{'Auto'} 'Use subsystem name' 'User specified'
RTWFcnName	Function name	{''}
RTWFileNameOpts	File name options	{'Auto'} 'Use subsystem name' 'Use function name' 'User specified'
RTWFileName	File name (no extension)	{''}
DataTypeOverride	No dialog box prompt Specifies data type used to override fixed-point data types. Set by Data type override on the Fixed-Point Tool.	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'
MinMaxOverflowLogging	No dialog box prompt Setting for fixed-point instrumentation. Set by Fixed-point instrumentation mode on the Fixed-Point Tool.	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'ForceOff'

Block (Type)/Parameter	Dialog Box Prompt	Values
SimViewingDevice	No dialog box prompt If set to 'on', designates the block as a Signal Viewing Subsystem — an atomic subsystem that encapsulates processing and viewing of signals received from the target system in External mode. For more information, see “Signal Viewing Subsystems” (Simulink Coder).	{'off'} 'on'
DataTypeOverride	No dialog box prompt Specifies data type used to override fixed-point data types. Set by Data type override on the Fixed-Point Tool.	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'
MinMaxOverflowLogging	No dialog box prompt Setting for fixed-point instrumentation. Set by Fixed-point instrumentation mode on the Fixed-Point Tool.	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'ForceOff'
IsSubsystemVirtual	No dialog box prompt	boolean — {'on'} 'off' Read-only
Enable (EnablePort)		
StatesWhenEnabling	States when enabling	{'held'} 'reset'
PropagateVarSize	Propagate sizes of variable-size signals	{'Only when enabling'} 'During execution'
ShowOutputPort	Show output port	{'off'} 'on'
ZeroCross	Enable zero-crossing detection	'off' {'on'}
Enabled and Triggered Subsystem (SubSystem)		

Block (Type)/Parameter	Dialog Box Prompt	Values
ShowPortLabels	Show port labels Note The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.	'none' {'FromPortIcon'} 'FromPortBlockName' 'SignalName' 'off' 'on'
BlockChoice	Block choice	{''}
TemplateBlock	Template block	{''}
MemberBlocks	Member blocks	{''}
Permissions	Read/Write permissions	{'ReadWrite'} 'ReadOnly' 'NoReadOrWrite'
ErrorFcn	Name of error callback function	{''}
PermitHierarchicalResolution	Permit hierarchical resolution	{'All'} 'ExplicitOnly' 'None'
TreatAsAtomicUnit	Treat as atomic unit	'off' {'on'}
MinAlgLoopOccurrences	Minimize algebraic loop occurrences	{'off'} 'on'
PropExecContextOutsideSubsystem	Propagate execution context across subsystem boundary	{'off'} 'on'
CheckFcnCallInpInsideContextMsg	Warn if function-call inputs are context-specific	{'off'} 'on'
SystemSampleTime	Sample time (-1 for inherited)	{'-1'}
RTWSystemCode	Function packaging	{'Auto'} 'Inline' 'Nonreusable function' 'Reusable function'
RTWFcnNameOpts	Function name options	{'Auto'} 'Use subsystem name' 'User specified'
RTWFcnName	Function name	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
RTWFileNameOpts	File name options	{'Auto'} 'Use subsystem name' 'Use function name' 'User specified'
RTWFileName	File name (no extension)	{''}
DataTypeOverride	No dialog box prompt Specifies data type used to override fixed-point data types. Set by Data type override on the Fixed-Point Tool.	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'
MinMaxOverflowLogging	No dialog box prompt Setting for fixed-point instrumentation. Set by Fixed-point instrumentation mode on the Fixed-Point Tool.	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'ForceOff'
IsSubsystemVirtual		boolean — {'off'} 'on' Read-only
Enabled Subsystem (SubSystem)		
ShowPortLabels	Show port labels Note The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.	'none' {'FromPortIcon'} 'FromPortBlockName' 'SignalName' 'off' 'on'
BlockChoice	Block choice	{''}
TemplateBlock	Template block	{''}
MemberBlocks	Member blocks	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
Permissions	Read/Write permissions	{'ReadWrite'} 'ReadOnly' 'NoReadOrWrite'
ErrorFcn	Name of error callback function	{''}
PermitHierarchical Resolution	Permit hierarchical resolution	{'All'} 'ExplicitOnly' 'None'
TreatAsAtomicUnit	Treat as atomic unit	'off' {'on'}
MinAlgLoopOccurrences	Minimize algebraic loop occurrences	{'off'} 'on'
PropExecContext OutsideSubsystem	Propagate execution context across subsystem boundary	{'off'} 'on'
CheckFcnCallInp InsideContextMsg	Warn if function-call inputs are context-specific	{'off'} 'on'
SystemSampleTime	Sample time (-1 for inherited)	{'-1'}
RTWSystemCode	Function packaging	{'Auto'} 'Inline' 'Nonreusable function' 'Reusable function'
RTWFcnNameOpts	Function name options	{'Auto'} 'Use subsystem name' 'User specified'
RTWFcnName	Function name	{''}
RTWFileNameOpts	File name options	{'Auto'} 'Use subsystem name' 'Use function name' 'User specified'
RTWFileName	File name (no extension)	{''}
DataTypeOverride	No dialog box prompt Specifies data type used to override fixed-point data types. Set by Data type override on the Fixed-Point Tool.	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'

Block (Type)/Parameter	Dialog Box Prompt	Values
MinMaxOverflowLogging	No dialog box prompt Setting for fixed-point instrumentation. Set by Fixed-point instrumentation mode on the Fixed-Point Tool.	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'ForceOff'
IsSubsystemVirtual		boolean — {'off'} 'on' Read-only
For Each(ForEach)		
InputPartition	Partition	cell array of character vectors
InputPartitionDimension	Partition dimension for input signal	cell array of character vectors
InputPartitionWidth	Width of partition for input signal	cell array of character vectors
OutputConcatenationDimension	Concatenation dimension of output signal	cell array of character vectors
For Iterator (ForIterator)		
ResetStates	States when starting	{'held'} 'reset'
IterationSource	Iteration limit source	{'internal'} 'external'
IterationLimit	Iteration limit	{'5'}
ExternalIncrement	Set next i (iteration variable) externally	{'off'} 'on'
ShowIterationPort	Show iteration variable	'off' {'on'}
IndexMode	Index mode	'Zero-based' {'One-based'}
IterationVariableDataType	Iteration variable data type	{'int32'} 'int16' 'int8' 'double'
For Iterator Subsystem (SubSystem)		

Block (Type)/Parameter	Dialog Box Prompt	Values
ShowPortLabels	Show port labels Note The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.	'none' {'FromPortIcon'} 'FromPortBlockName' 'SignalName' 'off' 'on'
BlockChoice	Block choice	{''}
TemplateBlock	Template block	{''}
MemberBlocks	Member blocks	{''}
Permissions	Read/Write permissions	{'ReadWrite'} 'ReadOnly' 'NoReadOrWrite'
ErrorFcn	Name of error callback function	{''}
PermitHierarchicalResolution	Permit hierarchical resolution	{'All'} 'ExplicitOnly' 'None'
TreatAsAtomicUnit	Treat as atomic unit	'off' {'on'}
MinAlgLoopOccurrences	Minimize algebraic loop occurrences	{'off'} 'on'
PropExecContextOutsideSubsystem	Propagate execution context across subsystem boundary	{'off'} 'on'
CheckFcnCallInpInsideContextMsg	Warn if function-call inputs are context-specific	{'off'} 'on'
SystemSampleTime	Sample time (-1 for inherited)	{'-1'}
RTWSystemCode	Function packaging	{'Auto'} 'Inline' 'Nonreusable function' 'Reusable function'
RTWFcnNameOpts	Function name options	{'Auto'} 'Use subsystem name' 'User specified'
RTWFcnName	Function name	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
RTWFileNameOpts	File name options	{'Auto'} 'Use subsystem name' 'Use function name' 'User specified'
RTWFileName	File name (no extension)	{''}
DataTypeOverride	No dialog box prompt Specifies data type used to override fixed-point data types. Set by Data type override on the Fixed-Point Tool.	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'
MinMaxOverflowLogging	No dialog box prompt Setting for fixed-point instrumentation . Set by Fixed-point instrumentation mode on the Fixed-Point Tool.	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'ForceOff'
IsSubsystemVirtual		boolean — {'off'} 'on' Read-only
Function-Call Generator (Function-Call Generator) (masked subsystem)		
sample_time	Sample time	{'1'}
numberOfIterations	Number of iterations	{'1'}
Function-Call Subsystem (SubSystem)		
ShowPortLabels	Show port labels Note The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.	'none' {'FromPortIcon'} 'FromPortBlockName' 'SignalName' 'off' 'on'
BlockChoice	Block choice	{''}
TemplateBlock	Template block	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
MemberBlocks	Member blocks	{''}
Permissions	Read/Write permissions	{'ReadWrite'} 'ReadOnly' 'NoReadOrWrite'
ErrorFcn	Name of error callback function	{''}
PermitHierarchical Resolution	Permit hierarchical resolution	{'All'} 'ExplicitOnly' 'None'
TreatAsAtomicUnit	Treat as atomic unit	'off' {'on'}
MinAlgLoopOccurrences	Minimize algebraic loop occurrences	{'off'} 'on'
PropExecContext OutsideSubsystem	Propagate execution context across subsystem boundary	{'off'} 'on'
CheckFcnCallInp InsideContextMsg	Warn if function-call inputs are context-specific	{'off'} 'on'
SystemSampleTime	Sample time (-1 for inherited)	{'-1'}
RTWSystemCode	Function packaging	{'Auto'} 'Inline' 'Nonreusable function' 'Reusable function'
RTWFcnNameOpts	Function name options	{'Auto'} 'Use subsystem name' 'User specified'
RTWFcnName	Function name	{''}
RTWFileNameOpts	File name options	{'Auto'} 'Use subsystem name' 'Use function name' 'User specified'
RTWFileName	File name (no extension)	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
DataTypeOverride	No dialog box prompt Specifies data type used to override fixed-point data types. Set by Data type override on the Fixed-Point Tool.	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'
MinMaxOverflowLogging	No dialog box prompt Setting for fixed-point instrumentation. Set by Fixed-point instrumentation mode on the Fixed-Point Tool.	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'ForceOff'
IsSubsystemVirtual		boolean — {'off'} 'on' Read-only
If (If)		
NumInputs	Number of inputs	{'1'}
IfExpression	If expression (e.g., u1 ~= 0)	{'u1 > 0'}
ElseIfExpressions	Elseif expressions (comma-separated list, e.g., u2 ~= 0, u3(2) < u2)	{''}
ShowElse	Show else condition	'off' {'on'}
ZeroCross	Enable zero-crossing detection	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
If Action Subsystem (SubSystem)		
ShowPortLabels	Show port labels Note The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.	'none' {'FromPortIcon'} 'FromPortBlockName' 'SignalName' 'off' 'on'
BlockChoice	Block choice	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
TemplateBlock	Template block	{''}
MemberBlocks	Member blocks	{''}
Permissions	Read/Write permissions	{'ReadWrite'} 'ReadOnly' 'NoReadOrWrite'
ErrorFcn	Name of error callback function	{''}
PermitHierarchical Resolution	Permit hierarchical resolution	{'All'} 'ExplicitOnly' 'None'
TreatAsAtomicUnit	Treat as atomic unit	'off' {'on'}
MinAlgLoopOccurrences	Minimize algebraic loop occurrences	{'off'} 'on'
PropExecContext OutsideSubsystem	Propagate execution context across subsystem boundary	{'off'} 'on'
CheckFcnCallInp InsideContextMsg	Warn if function-call inputs are context-specific	{'off'} 'on'
SystemSampleTime	Sample time (-1 for inherited)	{'-1'}
RTWSystemCode	Function packaging	{'Auto'} 'Inline' 'Nonreusable function' 'Reusable function'
RTWFcnNameOpts	Function name options	{'Auto'} 'Use subsystem name' 'User specified'
RTWFcnName	Function name	{''}
RTWFileNameOpts	File name options	{'Auto'} 'Use subsystem name' 'Use function name' 'User specified'
RTWFileName	File name (no extension)	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
DataTypeOverride	No dialog box prompt Specifies data type used to override fixed-point data types. Set by Data type override on the Fixed-Point Tool.	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'
MinMaxOverflowLogging	No dialog box prompt Setting for fixed-point instrumentation. Set by Fixed-point instrumentation mode on the Fixed-Point Tool.	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'Off'
IsSubsystemVirtual		boolean — {'off'} 'on' Read-only
In1 (Inport)		
Port	Port number	{'1'}
IconDisplay	Icon display	'Signal name' {'Port number'} 'Port number and signal name'
LatchByDelayingOutsideSignal	Latch input by delaying outside signal	{'off'} 'on'
LatchInputForFeedbackSignals	Latch input for feedback signals of function-call subsystem outputs	{'off'} 'on'
Interpolate	Interpolate data	'off' {'on'}
UseBusObject	Specify properties via bus object	{'off'} 'on'
BusObject	Bus object for specifying bus properties	{'BusObject'}
BusOutputAsStruct	Output as nonvirtual bus	{'off'} 'on'
PortDimensions	Port dimensions (-1 for inherited)	{'-1'}
SampleTime	Sample time (-1 for inherited)	{'-1'}

Block (Type)/Parameter	Dialog Box Prompt	Values
OutMin	Minimum	{'[]'}
OutMax	Maximum	{'[]'}
OutDataTypeStr	Data type	{'Inherit: auto'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
Unit	Specify physical unit of the input signal to the block.	{'inherit'} '<Enter unit>'
UnitNoProp	Specify physical unit of the input signal to the block without propagation. For a list of acceptable units, see Allowed Units.	'<Enter unit>'
SignalType	Signal type	{'auto'} 'real' 'complex'
Model (ModelReference)		
ModelNameDialog	The name of the referenced model exactly as you typed it in, with any surrounding whitespace removed. When you set ModelNameDialog programmatically or with the GUI, Simulink automatically sets the values of ModelName and ModelFile based on the value of ModelNameDialog.	{'<Enter Model Name>'}

Block (Type)/Parameter	Dialog Box Prompt	Values
<p>ModelName</p>	<p>The value of ModelNameDialog stripped of any filename extension that you provided. For backward compatibility, setting ModelName programmatically actually sets ModelNameDialog, which then sets ModelName as described. You cannot use get_param to obtain the ModelName of a protected model, because the name without a suffix would be ambiguous. Use get_param on ModelFile instead. You can test ProtectedModel to determine programmatically whether a referenced model is protected.</p>	<p>character vector — Set automatically when ModelNameDialog is set.</p>
<p>ModelFile</p>	<p>The value of ModelNameDialog with a filename extension. The suffix of the first match Simulink finds becomes the suffix of ModelFile. Setting ModelFile programmatically actually sets ModelNameDialog, which then sets ModelFile as described.</p>	<p>character vector — Set automatically when ModelNameDialog is set.</p>
<p>ProtectedModel</p>	<p>Read-only boolean indicating whether the model referenced by the block is protected (on) or unprotected (off).</p>	<p>boolean — 'off' 'on' — Set automatically when ModelNameDialog is set.</p>

Block (Type)/Parameter	Dialog Box Prompt	Values
ParameterArgumentNames	Names of model arguments that the referenced model defines. Corresponds to the Name column in the table under Arguments > Model arguments .	{ '' }
ParameterArgumentValues	Values for model arguments. Corresponds to the Value column in the table under Arguments > Model arguments .	structure with no fields
SimulationMode	Specifies whether to simulate the model by generating and executing code or by interpreting the model in Simulink software.	'Normal' 'Accelerator' 'Software-in-the-loop (SIL)' 'Processor-in-the-loop (PIL)'

Block (Type)/Parameter	Dialog Box Prompt	Values
Variant	<p>Specifies whether the Model block references variant models.</p> <hr/> <p>Note For new models, use a Model block for model variants only if you need to use variants that are conditionally executed models (models with control ports). Model variants are supported for backward compatibility. However, support for model variants will be removed in a future release.</p> <p>To convert a Model block that contains variant models to a Variant Subsystem block, right-click the Model block and select Subsystems & Model Reference > Convert to > Variant Subsystem. This conversion can require that you update scripts that use the Variants command-line parameter.</p>	{'off'} 'on'
Variants	<p>An array of variant structures where each element specifies one variant. The structure fields are as follows:</p> <p>variant.Name – The name of the Simulink.Variant object that represents the variant to which this element applies.</p>	array — []

Block (Type)/Parameter	Dialog Box Prompt	Values
	<code>variant.ModelName</code> – The name of the referenced model associated with the specified variant object in the Model block.	{ '' }
	<code>variant.ParameterArgument</code> Names – Read-only character vector containing the names of the model arguments for which the Model block must supply values.	{ '' }
	<code>variant.ParameterArgument</code> Values – The values to supply for the model arguments when this variant is the active variant.	{ '' }
	<code>variant.SimulationMode</code> – The execution mode to use when this variant is the active variant.	{ 'Accelerator' } 'Normal' 'Processor-in-the-loop (PIL)'
<code>VariantConfigurationObject</code>	Specifies the variant configuration object that is associated with the model.	{ '' } The value is an empty character vector if no configuration object is associated; otherwise, it is the name of a <code>Simulink.VariantConfigurationData</code> object.
<code>OverrideUsingVariant</code>	Whether to override the variant conditions and make a specified variant the active variant, and if so, the name of that variant.	{ '' } The value is the empty character vector if no overriding variant object is specified; or the name of the overriding object.

Block (Type)/Parameter	Dialog Box Prompt	Values
ActiveVariant	The variant that is currently active, either because its variant condition is true or OverrideUsingVariant has overridden the variant conditions and specified this variant.	{''} The value is the empty character vector if no variant is active; or the name of the active variant.
GeneratePreprocessorConditionals	Locally controls whether generated code contains preprocessor conditionals. This parameter applies only to Simulink Coder code generation and has no effect on the behavior of a model in Simulink. The parameter is available only for ERT targets. For more information, see “Variant Systems” (Embedded Coder).	{'off'} 'on'
DefaultDataLogging		{'off'} 'on'
Out1 (Outport)		
Port	Port number	{'1'}
IconDisplay	Icon display	'Signal name' {'Port number'} 'Port number and signal name'
UseBusObject	Specify properties via bus object	{'off'} 'on'
BusObject	Bus object for validating input bus	{'BusObject'}
BusOutputAsStruct	Output as nonvirtual bus in parent model	{'off'} 'on'
PortDimensions	Port dimensions (-1 for inherited)	{'-1'}

Block (Type)/Parameter	Dialog Box Prompt	Values
VarSizeSig	Variable-size signal	{'Inherit'} 'No' 'Yes'
SampleTime	Sample time (-1 for inherited)	{'-1'}
OutMin	Minimum	{'[]'}
OutMax	Maximum	{'[]'}
OutDataTypeStr	Data type	{'Inherit: auto'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>' 'Bus: <object name>'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
Unit	Specify physical unit of the input signal to the block. For a list of acceptable units, see Allowed Units.	{'inherit'} '<Enter unit>'
UnitNoProp	Specify physical unit of the input signal to the block without propagation. For a list of acceptable units, see Allowed Units.	'<Enter unit>'
SignalObject	This parameter does not appear in the block dialog box. Use the Model Data Editor instead. See “Design Data Interface by Configuring Inport and Outport Blocks” (Simulink Coder).	Simulink.Signal object Object of a class that is derived from Simulink.Signal

Block (Type)/Parameter	Dialog Box Prompt	Values
StorageClass	This parameter does not appear in the block dialog box. Use the Model Data Editor instead. See “Design Data Interface by Configuring Inport and Outport Blocks” (Simulink Coder).	{'Auto'} 'SimulinkGlobal' 'ExportedGlobal' 'ImportedExtern' 'ImportedExternPointer' 'Custom'
SignalName	Signal name	character vector
SignalType	Signal type	{'auto'} 'real' 'complex'
OutputWhenDisabled	Output when disabled	{'held'} 'reset'
InitialOutput	Initial output	{'[]'}
MustResolveToSignalObject	This parameter does not appear in the block dialog box. Use the Model Data Editor instead. See “For Signals”.	{'off'} 'on'
Subsystem (SubSystem)		
ShowPortLabels	Show port labels Note The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.	'none' {'FromPortIcon'} 'FromPortBlockName' 'SignalName' 'off' 'on'
BlockChoice	Block choice	{''}
TemplateBlock	Template block	{''}
MemberBlocks	Member blocks	{''}
Permissions	Read/Write permissions	{'ReadWrite'} 'ReadOnly' 'NoReadOrWrite'
ErrorFcn	Name of error callback function	{''}
PermitHierarchicalResolution	Permit hierarchical resolution	{'All'} 'ExplicitOnly' 'None'

Block (Type)/Parameter	Dialog Box Prompt	Values
TreatAsAtomicUnit	Treat as atomic unit	{'off'} 'on'
TreatAsGroupedWhenPropagatingVariantConditions	Treat as grouped when propagating variant conditions	'off' {'on'}
VariantControl	Variant control	{'Variant'} '(default)'
MinAlgLoopOccurrences	Minimize algebraic loop occurrences	{'off'} 'on'
PropExecContext OutsideSubsystem	Propagate execution context across subsystem boundary	{'off'} 'on'
CheckFcnCallInp InsideContextMsg	Warn if function-call inputs are context-specific	{'off'} 'on'
SystemSampleTime	Sample time (-1 for inherited)	{'-1'}
RTWSystemCode	Code generation function packaging	{'Auto'} 'Inline' 'Nonreusable function' 'Reusable function'
RTWFcnNameOpts	Code generation function name options	{'Auto'} 'Use subsystem name' 'User specified'
RTWFcnName	Code generation function name	{''}
RTWFileNameOpts	Code generation file name options	{'Auto'} 'Use subsystem name' 'Use function name' 'User specified'
RTWFileName	Code generation file name (no extension)	{''}
DataTypeOverride	Specifies data type used to override fixed-point data types. Set by Data type override on the Fixed-Point Tool.	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'

Block (Type)/Parameter	Dialog Box Prompt	Values
MinMaxOverflowLogging	No dialog box prompt Setting for fixed-point instrumentation. Set by Fixed-point instrumentation mode on the Fixed-Point Tool.	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'ForceOff'
IsSubsystemVirtual		boolean — {'on'} 'off' Read-only
Virtual	For internal use	
Switch Case (SwitchCase)		
CaseConditions	Case conditions (e.g., {1,[2,3]})	{' {1}'}
ShowDefaultCase	Show default case	'off' {'on'}
ZeroCross	Enable zero-crossing detection	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
CaseShowDefault	Deprecated in R2009b	
Switch Case Action Subsystem (SubSystem)		
ShowPortLabels	Show port labels Note The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.	'none' {'FromPortIcon'} 'FromPortBlockName' 'SignalName' 'off' 'on'
BlockChoice	Block choice	{''}
TemplateBlock	Template block	{''}
MemberBlocks	Member blocks	{''}
Permissions	Read/Write permissions	{'ReadWrite'} 'ReadOnly' 'NoReadOrWrite'
ErrorFcn	Name of error callback function	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
PermitHierarchicalResolution	Permit hierarchical resolution	{'All'} 'ExplicitOnly' 'None'
TreatAsAtomicUnit	Treat as atomic unit	'off' {'on'}
MinAlgLoopOccurrences	Minimize algebraic loop occurrences	{'off'} 'on'
PropExecContextOutsideSubsystem	Propagate execution context across subsystem boundary	{'off'} 'on'
CheckFcnCallInpInsideContextMsg	Warn if function-call inputs are context-specific	{'off'} 'on'
SystemSampleTime	Sample time (-1 for inherited)	{'-1'}
RTWSystemCode	Code generation function packaging	{'Auto'} 'Inline' 'Nonreusable function' 'Reusable function'
RTWFcnNameOpts	Code generation function name options	{'Auto'} 'Use subsystem name' 'User specified'
RTWFcnName	Code generation function name	{''}
RTWFileNameOpts	Code generation file name options	{'Auto'} 'Use subsystem name' 'Use function name' 'User specified'
RTWFileName	Code generation file name (no extension)	{''}
DataTypeOverride	No dialog box prompt Specifies data type used to override fixed-point data types. Set by Data type override on the Fixed-Point Tool.	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'

Block (Type)/Parameter	Dialog Box Prompt	Values
MinMaxOverflowLogging	No dialog box prompt Setting for fixed-point instrumentation. Set by Fixed-point instrumentation mode on the Fixed-Point Tool.	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'ForceOff'
IsSubsystemVirtual		boolean — {'off'} 'on' Read-only
Trigger (TriggerPort)		
TriggerType	Trigger type	{'rising'} 'falling' 'either' 'function-call'
IsSimulinkFunction	Configure the Function-call subsystem to be a Simulink Function	{'off'} 'on'
StatesWhenEnabling	States when enabling	{'held'} 'reset' 'inherit'
PropagateVarSize	Propagate sizes of variable-size signals	{'During execution'} 'Only when enabling'
ShowOutputPort	Show output port	{'off'} 'on'
OutputDataType	Output data type	{'auto'} 'double' 'int8'
SampleTimeType	Sample time type	{'triggered'} 'periodic'
SampleTime	Sample time	{'1'}
ZeroCross	Enable zero-crossing detection	'off' {'on'}
PortDimensions	Port dimensions (-1 for inherited)	{'-1'}
TriggerSignalSampleTime	Trigger signal sample time	{'-1'}
OutMin	Minimum	{'[]'}
OutMax	Maximum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Data type	{'Inherit: auto'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>'
Interpolate	Interpolate data	'off' {'on'}
Triggered Subsystem (SubSystem)		
ShowPortLabels	Show port labels Note The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.	'none' {'FromPortIcon'} 'FromPortBlockName' 'SignalName' 'off' 'on'
BlockChoice	Block choice	{''}
TemplateBlock	Template block	{''}
MemberBlocks	Member blocks	{''}
Permissions	Read/Write permissions	{'ReadWrite'} 'ReadOnly' 'NoReadOrWrite'
ErrorFcn	Name of error callback function	{''}
PermitHierarchicalResolution	Permit hierarchical resolution	{'All'} 'ExplicitOnly' 'None'
TreatAsAtomicUnit	Treat as atomic unit	'off' {'on'}
MinAlgLoopOccurrences	Minimize algebraic loop occurrences	{'off'} 'on'
PropExecContextOutsideSubsystem	Propagate execution context across subsystem boundary	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
CheckFcnCallInp InsideContextMsg	Warn if function-call inputs are context-specific	{'off'} 'on'
SystemSampleTime	Sample time (-1 for inherited)	{'-1'}
RTWSystemCode	Code generation function packaging	{'Auto'} 'Inline' 'Nonreusable function' 'Reusable function'
RTWFcnNameOpts	Code generation function name options	{'Auto'} 'Use subsystem name' 'User specified'
RTWFcnName	Code generation function name	{''}
RTWFileNameOpts	Code generation file name options	{'Auto'} 'Use subsystem name' 'Use function name' 'User specified'
RTWFileName	Code generation file name (no extension)	{''}
DataTypeOverride	No dialog box prompt Specifies data type used to override fixed-point data types. Set by Data type override on the Fixed-Point Tool.	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'
MinMaxOverflowLogging	No dialog box prompt Setting for fixed-point instrumentation. Set by Fixed-point instrumentation mode on the Fixed-Point Tool.	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'ForceOff'
IsSubsystemVirtual		boolean — {'off'} 'on' Read-only
Unit Conversion		

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output data type	{'Inherit via internal rule'} 'Inherit via back propagation'
Unit System Configuration		
AllowAllUnitSystems	Allow or restrict unit systems.	boolean — {'on'} 'off'
UnitSystems	Displays allowed unit system.	cell array of character vectors — {'SI', 'English', SI (extended)', 'CGS'}
While Iterator (WhileIterator)		
MaxIters	Maximum number of iterations (-1 for unlimited)	{'5'}
WhileBlockType	While loop type	{'while'} 'do-while'
ResetStates	States when starting	{'held'} 'reset'
ShowIterationPort	Show iteration number port	{'off'} 'on'
OutputDataType	Output data type	{'int32'} 'int16' 'int8' 'double'
While Iterator Subsystem (SubSystem)		
ShowPortLabels	Show port labels Note The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.	'none' {'FromPortIcon'} 'FromPortBlockName' 'SignalName' 'off' 'on'
BlockChoice	Block choice	{''}
TemplateBlock	Template block	{''}
MemberBlocks	Member blocks	{''}
Permissions	Read/Write permissions	{'ReadWrite'} 'ReadOnly' 'NoReadOrWrite'

Block (Type)/Parameter	Dialog Box Prompt	Values
ErrorFcn	Name of error callback function	{''}
PermitHierarchicalResolution	Permit hierarchical resolution	{'All'} 'ExplicitOnly' 'None'
TreatAsAtomicUnit	Treat as atomic unit	'off' {'on'}
MinAlgLoopOccurrences	Minimize algebraic loop occurrences	{'off'} 'on'
PropExecContextOutsideSubsystem	Propagate execution context across subsystem boundary	{'off'} 'on'
CheckFcnCallInpInsideContextMsg	Warn if function-call inputs are context-specific	{'off'} 'on'
SystemSampleTime	Sample time (-1 for inherited)	{'-1'}
RTWSystemCode	Code generation function packaging	{'Auto'} 'Inline' 'Nonreusable function' 'Reusable function'
RTWFcnNameOpts	Code generation function name options	{'Auto'} 'Use subsystem name' 'User specified'
RTWFcnName	Code generation function name	{''}
RTWFileNameOpts	Code generation file name options	{'Auto'} 'Use subsystem name' 'Use function name' 'User specified'
RTWFileName	Code generation file name (no extension)	{''}
DataTypeOverride	No dialog box prompt Specifies data type used to override fixed-point data types. Set by Data type override on the Fixed-Point Tool.	{'UseLocalSettings'} 'ScaledDouble' 'Double' 'Single' 'Off'

Block (Type)/Parameter	Dialog Box Prompt	Values
MinMaxOverflowLogging	No dialog box prompt Setting for fixed-point instrumentation. Set by Fixed-point instrumentation mode on the Fixed-Point Tool.	{'UseLocalSettings'} 'MinMaxAndOverflow' 'OverflowOnly' 'ForceOff'
IsSubsystemVirtual		boolean — {'off'} 'on' Read-only

Signal Attributes Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Bus to Vector (BusToVector)		
Data Type Conversion (DataTypeConversion)		
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	{'Inherit: Inherit via back propagation'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
ConvertRealWorld	Input and output to have equal	{'Real World Value (RWV)'} 'Stored Integer (SI)'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
SampleTime	Sample time (-1 for inherited)	{'-1'}
Data Type Conversion Inherited (Conversion Inherited) (masked subsystem)		
ConvertRealWorld	Input and Output to have equal	{'Real World Value'} 'Stored Integer'

Block (Type)/Parameter	Dialog Box Prompt	Values
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
DoSatur	Saturate to max or min when overflows occur	{'off'} 'on'
Data Type Duplicate (DataTypeDuplicate)		
NumInputPorts	Number of input ports	{'2'}
Data Type Propagation (Data Type Propagation) (masked subsystem)		
PropDataTypeMode	1. Propagated data type	'Specify via dialog' {'Inherit via propagation rule'}
PropDataType	1.1. Propagated data type (e.g., fixdt(1,16), fixdt('single'))	{'fixdt(1,16)'} {'single'}
IfRefDouble	1.1. If any reference input is double, output is	{'double'} 'single'
IfRefSingle	1.2. If any reference input is single, output is	'double' {'single'}
IsSigned	1.3. Is-Signed	'IsSigned1' 'IsSigned2' {'IsSigned1 or IsSigned2'} 'TRUE' 'FALSE'
NumBitsBase	1.4.1. Number-of-Bits: Base	'NumBits1' 'NumBits2' {'max([NumBits1 NumBits2])'} 'min([NumBits1 NumBits2])' 'NumBits1+NumBits2'
NumBitsMult	1.4.2. Number-of-Bits: Multiplicative adjustment	{'1'}

Block (Type)/Parameter	Dialog Box Prompt	Values
NumBitsAdd	1.4.3. Number-of-Bits: Additive adjustment	{'0'}
NumBitsAllowFinal	1.4.4. Number-of-Bits: Allowable final values	{'1:128'}
PropScalingMode	2. Propagated scaling	'Specify via dialog' 'Inherit via propagation rule' 'Obtain via best precision'
PropScaling	2.1. Propagated scaling: Slope or [Slope Bias] ex. 2 ⁻⁹	{'2 ⁻¹⁰ '}
ValuesUsedBestPrec	2.1. Values used to determine best precision scaling	{'[5 -7]'} {'1:128'}
SlopeBase	2.1.1. Slope: Base	'Slope1' 'Slope2' 'max([Slope1 Slope2])' 'min([Slope1 Slope2])' 'Slope1*Slope2' 'Slope1/Slope2' 'PosRange1' 'PosRange2' 'max([PosRange1 PosRange2])' 'min([PosRange1 PosRange2])' 'PosRange1*PosRange2' 'PosRange1/PosRange2'
SlopeMult	2.1.2. Slope: Multiplicative adjustment	{'1'}
SlopeAdd	2.1.3. Slope: Additive adjustment	{'0'}

Block (Type)/Parameter	Dialog Box Prompt	Values
BiasBase	2.2.1. Bias: Base	{'Bias1'} 'Bias2' 'max([Bias1 Bias2])' 'min([Bias1 Bias2])' 'Bias1*Bias2' 'Bias1/Bias2' 'Bias1+Bias2' 'Bias1-Bias2'
BiasMult	2.2.2. Bias: Multiplicative adjustment	{'1'}
BiasAdd	2.2.3. Bias: Additive adjustment	{'0'}
Data Type Scaling Strip (Scaling Strip) (masked subsystem)		
IC (InitialCondition)		
Value	Initial value	{'1'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Probe (Probe)		
ProbeWidth	Probe width	'off' {'on'}
ProbeSampleTime	Probe sample time	'off' {'on'}
ProbeComplexSignal	Detect complex signal	'off' {'on'}
ProbeSignalDimensions	Probe signal dimensions	'off' {'on'}
ProbeFramedSignal	Detect framed signal	'off' {'on'}
ProbeWidthDataType	Data type for width	{'double'} 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'Same as input'
ProbeSampleTimeDataType	Data type for sample time	{'double'} 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'Same as input'

Block (Type)/Parameter	Dialog Box Prompt	Values
ProbeComplexityDataType	Data type for signal complexity	{'double'} 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'Same as input'
ProbeDimensionsDataType	Data type for signal dimensions	{'double'} 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'Same as input'
ProbeFrameDataType	Data type for signal frames	{'double'} 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'Same as input'
Rate Transition (RateTransition)		
Integrity	Ensure data integrity during data transfer	'off' {'on'}
Deterministic	Ensure deterministic data transfer (maximum delay)	'off' {'on'}
InitialCondition	Initial conditions	{'0'}
OutPortSampleTimeOpt	Output port sample time options	{'Specify'} 'Inherit' 'Multiple of input port sample time'
OutPortSampleTimeMultiple	Sample time multiple (>0)	{'1'}
OutPortSampleTime	Output port sample time	{'-1'}
Signal Conversion (SignalConversion)		
ConversionOutput	Output	{'Signal copy'} 'Virtual bus' 'Nonvirtual bus'

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Data type	{'Inherit: auto'} 'Bus: <object name>'
OverrideOpt	Exclude this block from 'Block reduction' optimization	{'off'} 'on'
Signal Specification (SignalSpecification)		
Dimensions	Dimensions (-1 for inherited)	{'-1'}
VarSizeSig	Variable-size signal	{'Inherit'} 'No' 'Yes'
SampleTime	Sample time (-1 for inherited)	{'-1'}
OutMin	Minimum	{'[]'}
OutMax	Maximum	{'[]'}
OutDataTypeStr	Data type	{'Inherit: auto'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>' 'Bus: <object name>'
BusOutputAsStruct	Require nonvirtual bus	{'off'} 'on'
Unit	Specify physical unit of the input signal to the block. For a list of acceptable units, see Allowed Units.	{'inherit'} '<Enter unit>'
UnitNoProp	Specify physical unit of the input signal to the block without propagation. For a list of acceptable units, see Allowed Units.	'<Enter unit>'

Block (Type)/Parameter	Dialog Box Prompt	Values
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
SignalType	Signal type	{'auto'} 'real' 'complex'
Weighted Sample Time (SampleTimeMath)		
TsampMathOp	Operation	'+' '-' '*' '/' {'Ts Only'} '1/Ts Only'
weightValue	Weight value	{'1.0'}
TsampMathImp	Implement using	{'Online Calculations'} 'Offline Scaling Adjustment'
OutDataTypeStr	Output data type	{'Inherit via internal rule'} 'Inherit via back propagation'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
OutputDataTypeScaling Mode	Deprecated in R2009b	
DoSatur	Deprecated in R2009b	
Width (width)		
OutputDataTypeScaling Mode	Output data type mode	{'Choose intrinsic data type'} 'Inherit via back propagation' 'All ports same datatype'

Block (Type)/Parameter	Dialog Box Prompt	Values
DataType	Output data type	{ 'double' } 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32'

Signal Routing Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Bus Assignment (BusAssignment)		
AssignedSignals	Signals that are being assigned	{ '' }
InputSignals	Signals in the bus	matrix — { '{ }' }
Bus Creator (BusCreator)		
InheritFromInputs	Override bus signal names from inputs	{ 'on' } 'off' If set to 'on', overrides bus signal names from inputs. Otherwise, inherits bus signal names from a bus object.
Inputs	Number of inputs	{ '2' }
DisplayOption		'none' 'signals' { 'bar' }
NonVirtualBus	Output as nonvirtual bus	{ 'off' } 'on'
OutDataTypeStr	Data type	{ 'Inherit: auto' } 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>' 'Bus: <object name>'
Bus Selector (BusSelector)		
OutputSignals	Selected signals	character vector — in the form 'signal1, signal2'
OutputAsBus	Output as bus	{ 'off' } 'on'
InputSignals	Signals in bus	matrix — { '{ }' }
Data Store Memory (DataStoreMemory)		
DataStoreName	Data store name	{ 'A' }

Block (Type)/Parameter	Dialog Box Prompt	Values
ReadBeforeWriteMsg	Detect read before write	'none' {'warning'} 'error'
WriteAfterWriteMsg	Detect write after write	'none' {'warning'} 'error'
WriteAfterReadMsg	Detect write after read	'none' {'warning'} 'error'
InitialValue	Initial value	{'0'}
StateMustResolveToSignalObject	Data store name must resolve to Simulink signal object	{'off'} 'on'
DataLogging	Log Signal Data	'off' {'on'}
DataLoggingNameMode	Logging Name	{'SignalName'} 'Custom'
DataLoggingName	Logging Name	{''}
DataLoggingLimitDataPoints	Limit data points to last	'off' {'on'}
DataLoggingMaxPoints	Limit data points to last	non-zero integer {5000}
DataLoggingDecimateData	Decimation	'off' {'on'}
DataLoggingLimitDataPoints	Decimation	non-zero integer {2}
StateStorageClass	Storage class	{'Auto'} 'SimulinkGlobal' 'ExportedGlobal' 'ImportedExtern' 'ImportedExternPointer' 'Custom'
StateSignalObject	Signal object class Storage class	Simulink.Signal object Object of a class that is derived from Simulink.Signal
RTWStateStorageTypeQualifier	Code generation type qualifier	{''}
VectorParams1D	Interpret vector parameters as 1-D	'off' {'on'}

Block (Type)/Parameter	Dialog Box Prompt	Values
ShowAdditionalParam	Show additional parameters	{'off'} 'on'
OutMin	Minimum	{'[]'}
OutMax	Maximum	{'[]'}
OutDataTypeStr	Data type	{'Inherit: auto'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
SignalType	Signal type	{'auto'} 'real' 'complex'
Data Store Read (DataStoreRead)		
DataStoreName	Data store name	{'A'}
SampleTime	Sample time	{'0'}
Data Store Write (DataStoreWrite)		
DataStoreName	Data store name	{'A'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
Demux (Demux)		
Outputs	Number of outputs	{'2'}
DisplayOption	Display option	'none' {'bar'}
BusSelectionMode	Bus selection mode	{'off'} 'on'
Environment Controller (Environment Controller) (masked subsystem)		
From (From)		
GotoTag	Goto tag	{'A'}

Block (Type)/Parameter	Dialog Box Prompt	Values
IconDisplay	Icon display	'Signal name' {'Tag'} 'Tag and signal name'
Goto (Goto)		
GotoTag	Tag	{'A'}
IconDisplay	Icon display	'Signal name' {'Tag'} 'Tag and signal name'
TagVisibility	Tag visibility	{'local'} 'scoped' 'global'
Goto Tag Visibility (GotoTagVisibility)		
GotoTag	Goto tag	{'A'}
Index Vector (MultiPortSwitch)		
DataPortOrder	Data port order	{'Zero-based contiguous'} 'One-based contiguous' 'Specify indices'
Inputs	Number of data ports	{'1'}
zeroidx	Deprecated in R2010a	
SampleTime	Sample time (-1 for inherited)	{'-1'}
InputSameDT	Require all data port inputs to have the same data type	{'off'} 'on'
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'

Block (Type)/Parameter	Dialog Box Prompt	Values
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
AllowDiffInputSizes	Allow different data input sizes (Results in variable-size output signal)	{'off'} 'on'
Manual Switch (Manual Switch) (masked subsystem)		
varsize	Allow different input sizes (Results in variable-size output signal)	{'off'} 'on'
SampleTime	Sample time (-1 for inherited)	{'-1'}
Merge (Merge)		
Inputs	Number of inputs	{'2'}
InitialOutput	Initial output	{'[]'}
AllowUnequalInput PortWidths	Allow unequal port widths	{'off'} 'on'
InputPortOffsets	Input port offsets	{'[]'}
Multiport Switch (MultiPortSwitch)		
DataPortOrder	Data port order	'Zero-based contiguous' {'One-based contiguous'} 'Specify indices'
Inputs	Number of data ports	{'3'}
zeroidx	Deprecated in R2010a	
DataPortIndices	Data port indices	{' {1,2,3}' }

Block (Type)/Parameter	Dialog Box Prompt	Values
DataPortForDefault	Data port for default case	{'Last data port'} 'Additional data port'
DiagnosticForDefault	Diagnostic for default case	'None' 'Warning' {'Error'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
InputSameDT	Require all data port inputs to have the same data type	{'off'} 'on'
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock output data type setting against changes by the fixed- point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
AllowDiffInputSizes	Allow different data input sizes (Results in variable-size output signal)	{'off'} 'on'
Mux (Mux)		
Inputs	Number of inputs	{'2'}

Block (Type)/Parameter	Dialog Box Prompt	Values
DisplayOption	Display option	'none' 'signals' {'bar'}
UseBusObject	For internal use	
BusObject	For internal use	
NonVirtualBus	For internal use	
Selector (Selector)		
NumberOfDimensions	Number of input dimensions	{'1'}
IndexMode	Index mode	'Zero-based' {'One-based'}
IndexOptionArray	Index Option	'Select all' {'Index vector (dialog)'} 'Index vector (port)' 'Starting index (dialog)' 'Starting index (port)'
IndexParamArray	Index	cell array
OutputSizeArray	Output Size	cell array
InputPortWidth	Input port size	{'1'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
IndexOptions	See IndexOptionArray on page 6-0 parameter for more information.	
Indices	See IndexParamArray on page 6-0 parameter for more information.	
OutputSizes	See OutputSizeArray on page 6-0 parameter for more information.	
Switch (Switch)		
Criteria	Criteria for passing first input	{'u2 >= Threshold'} 'u2 > Threshold' 'u2 ~= 0'
Threshold	Threshold	{'0'}

Block (Type)/Parameter	Dialog Box Prompt	Values
ZeroCross	Enable zero-crossing detection	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
InputSameDT	Require all data port inputs to have the same data type	{'off'} 'on'
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	{'Inherit: Inherit via internal rule'} 'Inherit: Inherit via back propagation' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
SaturateOnInteger Overflow	Saturate on integer overflow	{'off'} 'on'
AllowDiffInputSizes	Allow different input sizes (Results in variable-size output signal)	{'off'} 'on'
Variant Source (tVariantSource)		
VariantControls	Variant control	{'Variant'} '(default)'
OverrideUsingVariant	Override variant conditions and use the following variant	{''}

Block (Type)/Parameter	Dialog Box Prompt	Values
AllowZeroVariantControls	Allow zero active variant controls	{'off'} 'on'
ShowConditionOnBlock	Show variant condition on block	{'off'} 'on'
GeneratePreprocessorConditionals	Analyze all choices during update diagram and generate preprocessor conditionals	{'off'} 'on'
CompiledActiveVariantControl		string - {''} The value is a empty string if no variant is active; or the name of the active variant. Compile the model before querying this property.
CompiledActiveVariantPort		string - {'-1'} The value is -1 if no variant is active; or the index of the active variant. Compile the model before querying this property.
Variant Sink (VariantSink)		
VariantControls	Variant control	{'Variant'} '(default)'
OverrideUsingVariant	Override variant conditions and use the following variant	{''}
AllowZeroVariantControls	Allow zero active variant controls	{'off'} 'on'
ShowConditionOnBlock	Show variant condition on block	{'off'} 'on'
GeneratePreprocessorConditionals	Analyze all choices during update diagram and generate preprocessor conditionals	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
CompiledActiveVariantControl		string – {''} The value is an empty string if no variant is active; or the name of the active variant. Compile the model before querying this property.
CompiledActiveVariantPort		string – {'-1'} The value is -1 if no variant is active; or the index of the active variant. Compile the model before querying this property.
Vector Concatenate (Concatenate)		
NumInputs	Number of inputs	{'2'}
Mode	Mode	{'Vector'} 'Multidimensional array'

Sinks Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Display (Display)		
Format	Format	{'short'} 'long' 'short_e' 'long_e' 'bank' 'hex (Stored Integer)' 'binary (Stored Integer)' 'decimal (Stored Integer)' 'octal (Stored Integer)'
Decimation	Decimation	{'1'}
Floating	Floating display	{'off'} 'on'
SampleTime	Sample time (-1 for inherited)	{'-1'}
Floating Scope (Scope)		
Floating		'off' {'on'}
Location		vector — {'[376 294 700 533]'}
Open		{'off'} 'on'
NumInputPorts		Do not change this parameter with the command-line. Instead, use the Number of axes parameter in the Scope parameters dialog.
TickLabels		'on' 'off' {'OneTimeTick'}
ZoomMode		{'on'} 'xonly' 'yonly'
AxesTitles		character vector
Grid		'off' {'on'} 'xonly' 'yonly'
TimeRange		{'auto'}

Block (Type)/Parameter	Dialog Box Prompt	Values
YMin		{'-5'}
YMax		{'5'}
SaveToWorkspace		{'off'} 'on'
SaveName		{'ScopeData'}
DataFormat		{'StructureWithTime'} 'Structure' 'Array'
LimitDataPoints		'off' {'on'}
MaxDataPoints		{'5000'}
Decimation		{'1'}
SampleInput		{'off'} 'on'
SampleTime		{'0'}
Out1 (Outport)		
Port	Port number	{'1'}
IconDisplay	Icon display	'Signal name' {'Port number'} 'Port number and signal name'
BusOutputAsStruct	Output as nonvirtual bus in parent model	{'off'} 'on'
PortDimensions	Port dimensions (-1 for inherited)	{'-1'}
VarSizeSig	Variable-size signal	{'Inherit'} 'No' 'Yes'
SampleTime	Sample time (-1 for inherited)	{'-1'}
OutMin	Minimum	{'[]'}
OutMax	Maximum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Data type	{'Inherit: auto'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>' 'Bus: <object name>'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
SignalObject	This parameter does not appear in the block dialog box. Use the Model Data Editor instead. See “Design Data Interface by Configuring Inport and Output Blocks” (Simulink Coder).	Simulink.Signal object Object of a class that is derived from Simulink.Signal
StorageClass	This parameter does not appear in the block dialog box. Use the Model Data Editor instead. See “Design Data Interface by Configuring Inport and Output Blocks” (Simulink Coder).	{'Auto'} 'SimulinkGlobal' 'ExportedGlobal' 'ImportedExtern' 'ImportedExternPointer' 'Custom'
SignalName	Signal name	character vector
SignalType	Signal type	{'auto'} 'real' 'complex'
OutputWhenDisabled	Output when disabled	{'held'} 'reset'
InitialOutput	Initial output	{'[]'}
MustResolveToSignalObject	This parameter does not appear in the block dialog box. Use the Model Data Editor instead. See “For Signals”.	{'off'} 'on'

Block (Type)/Parameter	Dialog Box Prompt	Values
Scope (Scope)		
Floating		{'off'} 'on'
Location		vector — {'[188 390 512 629]'} 629]'
Open		{'off'} 'on'
NumInputPorts		Do not change this parameter with the command-line. Instead, use the Number of axes parameter in the Scope parameters dialog.
TickLabels		'on' 'off' {'OneTimeTick'}
ZoomMode		{'on'} 'xonly' 'yonly'
AxesTitles		character vector
Grid		'off' {'on'} 'xonly' 'yonly'
TimeRange		{'auto'}
YMin		{'-5'}
YMax		{'5'}
SaveToWorkspace		{'off'} 'on'
SaveName		{'ScopeData1'}
DataFormat		{'StructureWithTime'} 'Structure' 'Array'
LimitDataPoints		'off' {'on'}
MaxDataPoints		{'5000'}
Decimation		{'1'}
SampleInput		{'off'} 'on'
SampleTime		{'0'}
Stop Simulation		

Block (Type)/Parameter	Dialog Box Prompt	Values
Terminator		
To File (ToFile)		
FileName	File name	{'untitled.mat'}
MatrixName	Variable name	{'ans'}
SaveFormat	Save format	{'Timeseries'} 'Array'
Decimation	Decimation	{'1'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
To Workspace (ToWorkspace)		
VariableName	Variable name	{'simout'}
MaxDataPoints	Limit data points to last	{'inf'}
Decimation	Decimation	{'1'}
SampleTime	Sample time (-1 for inherited)	{'-1'}
SaveFormat	Save format	{'Timeseries'} 'Structure With Time' 'Structure' 'Array'
FixptAsFi	Log fixed-point data as an fi object	{'off'} 'on'
XY Graph (XY scope) (masked subsystem)		
xmin	x-min	{'-1'}
xmax	x-max	{'1'}
ymin	y-min	{'-1'}
ymax	y-max	{'1'}
st	Sample time	{'-1'}

Sources Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Band-Limited White Noise (Band-Limited White Noise) (masked subsystem)		
Cov	Noise power	{' [0.1] '}
Ts	Sample time	{' 0.1 '}
seed	Seed	{' [23341] '}
VectorParams1D	Interpret vector parameters as 1-D	'off' {'on'}
Chirp Signal (chirp) (masked subsystem)		
f1	Initial frequency	{' 0.1 '}
T	Target time	{' 100 '}
f2	Frequency at target time	{' 1 '}
VectorParams1D	Interpret vectors parameters as 1-D	'off' {'on'}
Clock (Clock)		
DisplayTime	Display time	{'off'} 'on'
Decimation	Decimation	{' 10 '}
Constant (Constant)		
Value	Constant value	{' 1 '}
VectorParams1D	Interpret vector parameters as 1-D	'off' {'on'}
SampleTime	Sampling time	{'Sample based'} 'Frame based'
OutMin	Output minimum	{' [] '}
OutMax	Output maximum	{' [] '}

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output data type	{'Inherit: Inherit from 'Constant value''} 'Inherit: Inherit via back propagation' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16)' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>'
LockScale	Lock output data type setting against changes by the fixed- point tools	{'off'} 'on'
SampleTime	Sample time	{'inf'}
FramePeriod	Frame period	{'inf'}
Counter Free-Running (Counter Free-Running) (masked subsystem)		
NumBits	Number of Bits	{'16'}
tsamp	Sample time	{'-1'}
Counter Limited (Counter Limited) (masked subsystem)		
uplimit	Upper limit	{'7'}
tsamp	Sample time	{'-1'}
Digital Clock (DigitalClock)		
SampleTime	Sample time	{'1'}
Enumerated Constant (EnumeratedConstant)		
OutDataTypeStr	Output data type	{'Enum: S1DemoSign'}
Value	Value	{'S1DemoSign.Positive' 'S1DemoSign.Zero' 'S1DemoSign.Negative'}
SampleTime	Sample time	{'inf'}

Block (Type)/Parameter	Dialog Box Prompt	Values
From File (FromFile)		
FileName	File name	{'untitled.mat'}
ExtrapolationBeforeFirstDataPoint	Data extrapolation before first data point	{'Linear extrapolation' 'Hold first value' 'Ground value'}
InterpolationWithinTimeRange	Data interpolation within time range	{'Linear interpolation' 'Zero order hold'}
ExtrapolationAfterLastDataPoint	Data extrapolation after last data point	{'Linear extrapolation' 'Hold last value' 'Ground value'}
SampleTime	Sample time	{'0'}
From Workspace (FromWorkspace)		
VariableName	Data	{'simin'}
OutDataTypeStr	Output Data type	{'Inherit: auto' 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>' 'Bus: <object name>'}
SampleTime	Sample time	{'0'}
Interpolate	Interpolate data	'off' {'on'}
ZeroCross	Enable zero-crossing detection	'off' {'on'}
OutputAfterFinalValue	Form output after final data value by	{'Extrapolation' 'Setting to zero' 'Holding final value' 'Cyclic repetition'}
Ground		
In1 (Inport)		

Block (Type)/Parameter	Dialog Box Prompt	Values
Port	Port number	{'1'}
IconDisplay	Icon display	'Signal name' {'Port number'} 'Port number and signal name'
BusOutputAsStruct	Output as nonvirtual bus	{'off'} 'on'
PortDimensions	Port dimensions (-1 for inherited)	{'-1'}
VarSizeSig	Variable-size signal	{'Inherit'} 'No' 'Yes'
SampleTime	Sample time (-1 for inherited)	{'-1'}
OutMin	Minimum	{'[]'}
OutMax	Maximum	{'[]'}
OutDataTypeStr	Data type	{'Inherit: auto'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>' 'Bus: <object name>'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
Unit	Specify physical unit of the input signal to the block. For a list of acceptable units, see Allowed Units.	{'inherit'} '<Enter unit>'
UnitNoProp	Specify physical unit of the input signal to the block without propagation. For a list of acceptable units, see Allowed Units.	'<Enter unit>'

Block (Type)/Parameter	Dialog Box Prompt	Values
SignalType	Signal type	{'auto'} 'real' 'complex'
LatchByDelaying OutsideSignal	Latch input by delaying outside signal	{'off'} 'on'
LatchInputFor FeedbackSignals	Latch input for feedback signals of function-call subsystem outputs	{'off'} 'on'
OutputFunctionCall	Output a function-call trigger signal	{'off'} 'on'
Interpolate	Interpolate data	'off' {'on'}
Pulse Generator (DiscretePulseGenerator)		
PulseType	Pulse type	{'Time based'} 'Sample based'
TimeSource	Time (t)	{'Use simulation time'} 'Use external signal'
Amplitude	Amplitude	{'1'}
Period	Period	{'10'}
PulseWidth	Pulse width	{'5'}
PhaseDelay	Phase delay	{'0'}
SampleTime	Sample time	{'1'}
VectorParams1D	Interpret vector parameters as 1-D	'off' {'on'}
Ramp (Ramp) (masked subsystem)		
slope	Slope	{'1'}
start	Start time	{'0'}
X0	Initial output	{'0'}
VectorParams1D	Interpret vector parameters as 1-D	'off' {'on'}
Random Number (RandomNumber)		
Mean	Mean	{'0'}

Block (Type)/Parameter	Dialog Box Prompt	Values
Variance	Variance	{'1'}
Seed	Seed	{'0'}
SampleTime	Sample time	{'0.1'}
VectorParams1D	Interpret vector parameters as 1-D	'off' {'on'}
Repeating Sequence (Repeating table) (masked subsystem)		
rep_seq_t	Time values	{'[0 2]'}
rep_seq_y	Output values	{'[0 2]'}
Repeating Sequence Interpolated (Repeating Sequence Interpolated) (masked subsystem)		
OutValues	Vector of output values	{'[3 1 4 2 1].''}
TimeValues	Vector of time values	{'[0 0.1 0.5 0.6 1].''}
LookUpMeth	Lookup Method	{'Interpolation-Use End Values'} 'Use Input Nearest' 'Use Input Below' 'Use Input Above'
tsamp	Sample time	{'0.01'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	'Inherit: Inherit via back propagation' {'double'} 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
OutputDataTypeScaling Mode	Deprecated in R2007b	
OutDataType	Deprecated in R2007b	
OutScaling	Deprecated in R2007b	

Block (Type)/Parameter	Dialog Box Prompt	Values
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
Repeating Sequence Stair (Repeating Sequence Stair) (masked subsystem)		
OutValues	Vector of output values	{'[3 1 4 2 1].''}
tsamp	Sample time	{'-1'}
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}
OutDataTypeStr	Output data type	'Inherit: Inherit via back propagation' {'double'} 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16)' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)' 'Enum: <class name>'
OutputDataTypeScaling Mode	Deprecated in R2007b	
OutDataType	Deprecated in R2007b	
ConRadixGroup	Deprecated in R2007b	
OutScaling	Deprecated in R2007b	
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
Signal Builder (Sigbuilder block) (masked subsystem)		
Signal Editor (SignalEditor)		
Filename	File name	{'untitled.mat'}
ActiveScenario	Active scenario	{'Scenario'}

Block (Type)/Parameter	Dialog Box Prompt	Values
ActiveSignal	Signals	{'Signal 1'}
IsBus	Output a bus signal	'on' {'off'}
OutputBusObjectStr	Select bus object	{'Bus: BusObject'}
SampleTime	Sample time	{'0'}
Interpolate	Interpolate data	{'off'} 'on'
ZeroCross	Enable zero-crossing detection	{'off'} 'on'
OutputAfterFinalValue	Form output after final data value by	{'Setting to zero'} 'Extrapolation' 'Holding final value'
Unit	Unit	{'inherit'}
Signal Generator (SignalGenerator)		
WaveForm	Wave form	{'sine'} 'square' 'sawtooth' 'random'
TimeSource	Time (t)	{'Use simulation time'} 'Use external signal'
Amplitude	Amplitude	{'1'}
Frequency	Frequency	{'1'}
Units	Units	'rad/sec' {'Hertz'}
VectorParams1D	Interpret vector parameters as 1-D	'off' {'on'}
Sine Wave (Sin)		
SineType	Sine type	{'Time based'} 'Sample based'
TimeSource	Time	{'Use simulation time'} 'Use external signal'
Amplitude	Amplitude	{'1'}
Bias	Bias	{'0'}
Frequency	Frequency	{'1'}
Phase	Phase	{'0'}

Block (Type)/Parameter	Dialog Box Prompt	Values
Samples	Samples per period	{'10'}
Offset	Number of offset samples	{'0'}
SampleTime	Sample time	{'0'}
VectorParams1D	Interpret vector parameters as 1-D	'off' {'on'}
Step (Step)		
Time	Step time	{'1'}
Before	Initial value	{'0'}
After	Final value	{'1'}
SampleTime	Sample time	{'0'}
VectorParams1D	Interpret vector parameters as 1-D	'off' {'on'}
ZeroCross	Enable zero-crossing detection	'off' {'on'}
Uniform Random Number (UniformRandomNumber)		
Minimum	Minimum	{'-1'}
Maximum	Maximum	{'1'}
Seed	Seed	{'0'}
SampleTime	Sample time	{'0.1'}
VectorParams1D	Interpret vector parameters as 1-D	'off' {'on'}
Waveform Generator (WaveformGenerator)		
OutMin	Output minimum	{'[]'}
OutMax	Output maximum	{'[]'}

Block (Type)/Parameter	Dialog Box Prompt	Values
OutDataTypeStr	Output data type	'Inherit: Inherit via back propagation' {'Inherit: Inherit from table data'} 'double' 'single' 'int8' 'uint8' 'int16' 'uint16' 'int32' 'uint32' 'boolean' 'fixdt(1,16)' 'fixdt(1,16,0)' 'fixdt(1,16,2^0,0)'
LockScale	Lock output data type setting against changes by the fixed-point tools	{'off'} 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' 'Floor' {'Nearest'} 'Round' 'Simplest' 'Zero'
SaturateOnIntegerOverflow	Saturate on integer overflow	{'off'} 'on'
SelectedSignal	Output signal	{'1'}
SampleTime	Sample time	{'0'}

User-Defined Functions Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
MATLAB Function (Stateflow) (masked subsystem)		
MATLAB System (MATLABSystem)		
System	System object class name	{ '' }
Fcn (Fcn)		
Expr	Expression	{ 'sin(u(1)*exp(2.3*(-u(2))))' }
SampleTime	Sample time (-1 for inherited)	{ '-1' }
Level-2 MATLAB S-Function (M-S-Function)		
FunctionName	S-function name	{ 'mlfile' }
Parameters	Parameters	{ '' }
Interpreted MATLAB Function (MATLABFcn)		
MATLABFcn	MATLAB function	{ 'sin' }
OutputDimensions	Output dimensions	{ '-1' }
OutputSignalType	Output signal type	{ 'auto' } 'real' 'complex'
Output1D	Collapse 2-D results to 1-D	'off' {'on'}
SampleTime	Sample time (-1 for inherited)	{ '-1' }
S-Function (S-Function)		
FunctionName	S-function name	{ 'system' }
Parameters	S-function parameters	{ '' }
SFunctionModules	S-function modules	{ '' }
S-Function Builder (S-Function Builder) (masked subsystem)		
FunctionName	S-function name	{ 'system' }
Parameters	S-function parameters	{ '' }
SFunctionModules	S-function modules	{ '' }

Additional Discrete Block Library Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Fixed-Point State-Space (Fixed-Point State-Space) (masked subsystem)		
A	State Matrix A	{ '[2.6020 -2.2793 0.6708; 1 0 0; 0 1 0]'
B	Input Matrix B	{ '[1; 0; 0]'
C	Output Matrix C	{ '[0.0184 0.0024 0.0055]'
D	Direct Feedthrough Matrix D	{ '[0.0033]'
X0	Initial condition for state	{ '0.0'
InternalDataType	Data type for internal calculations	{ 'fixdt('double')'
StateEqScaling	Scaling for State Equation AX +BU	{ '2^0'
OutputEqScaling	Scaling for Output Equation CX +DU	{ '2^0'
LockScale	Lock output data type setting against changes by the fixed-point tools	{ 'off' } 'on'
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
DoSatur	Saturate to max or min when overflows occur	{ 'off' } 'on'
Transfer Fcn Direct Form II (Transfer Fcn Direct Form II) (masked subsystem)		
NumCoefVec	Numerator coefficients	{ '[0.2 0.3 0.2]'
DenCoefVec	Denominator coefficients excluding lead (which must be 1.0)	{ '[-0.9 0.6]'
vinit	Initial condition	{ '0.0'

Block (Type)/Parameter	Dialog Box Prompt	Values
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
DoSatur	Saturate to max or min when overflows occur	{'off'} 'on'
Transfer Fcn Direct Form II Time Varying (Transfer Fcn Direct Form II Time Varying) (masked subsystem)		
vinit	Initial condition	{'0.0'}
RndMeth	Integer rounding mode	'Ceiling' 'Convergent' {'Floor'} 'Nearest' 'Round' 'Simplest' 'Zero'
DoSatur	Saturate to max or min when overflows occur	{'off'} 'on'
Unit Delay Enabled (Unit Delay Enabled) (masked subsystem)		
vinit	Initial condition	{'0.0'}
tsamp	Sample time	{'-1'}
Unit Delay Enabled External IC (Unit Delay Enabled External Initial Condition) (masked subsystem)		
tsamp	Sample time	{'-1'}
Unit Delay Enabled Resettable (Unit Delay Enabled Resettable) (masked subsystem)		
vinit	Initial condition	{'0.0'}
tsamp	Sample time	{'-1'}
Unit Delay Enabled Resettable External IC (Unit Delay Enabled Resettable External Initial Condition) (masked subsystem)		
tsamp	Sample time	{'-1'}
Unit Delay External IC (Unit Delay External Initial Condition) (masked subsystem)		
tsamp	Sample time	{'-1'}
Unit Delay Resettable (Unit Delay Resettable) (masked subsystem)		

Block (Type)/Parameter	Dialog Box Prompt	Values
vinit	Initial condition	{'0.0'}
tsamp	Sample time	{'-1'}
Unit Delay Resettable External IC (Unit Delay Resettable External Initial Condition) (masked subsystem)		
tsamp	Sample time	{'-1'}
Unit Delay With Preview Enabled (Unit Delay With Preview Enabled) (masked subsystem)		
vinit	Initial condition	{'0.0'}
tsamp	Sample time	{'-1'}
Unit Delay With Preview Enabled Resettable (Unit Delay With Preview Enabled Resettable) (masked subsystem)		
vinit	Initial condition	{'0.0'}
tsamp	Sample time	{'-1'}
Unit Delay With Preview Enabled Resettable External RV (Unit Delay With Preview Enabled Resettable External RV) (masked subsystem)		
vinit	Initial condition	{'0.0'}
tsamp	Sample time	{'-1'}
Unit Delay With Preview Resettable (Unit Delay With Preview Resettable) (masked subsystem)		
vinit	Initial condition	{'0.0'}
tsamp	Sample time	{'-1'}
Unit Delay With Preview Resettable External RV (Unit Delay With Preview Resettable External RV) (masked subsystem)		
vinit	Initial condition	{'0.0'}
tsamp	Sample time	{'-1'}

Additional Math: Increment - Decrement Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
Decrement Real World (Real World Value Decrement) (masked subsystem)		
Decrement Stored Integer (Stored Integer Value Decrement) (masked subsystem)		
Decrement Time To Zero (Decrement Time To Zero) (masked subsystem)		
Decrement To Zero (Decrement To Zero) (masked subsystem)		
Increment Real World (Real World Value Increment) (masked subsystem)		
Increment Stored Integer (Stored Integer Value Increment) (masked subsystem)		

Mask Parameters

About Mask Parameters

This section lists parameters that describe masked blocks. You can use these descriptive parameters with `get_param` and `set_param` to obtain and specify the properties of a block mask.

The descriptive mask parameters listed in this section apply to all masks, and provide access to all mask properties. Be careful not to confuse these descriptive mask parameters with the mask-specific parameters defined for an individual mask in the Mask Editor **Parameters** pane.

See “Masking Fundamentals” and “Mask Editor Overview” for information about block masks and the Mask Editor.

Mask Parameters

Parameter	Description/Prompt	Values
Mask	Turns mask on or off.	{ 'on' } 'off'
MaskCallbackString	Mask parameter callbacks that are executed when the respective parameter is changed on the dialog. Set by the Dialog callback field on the Parameters pane of the Mask Editor dialog box. For more information, see “Mask Callback Code”.	pipe-delimited character vector { '' }
MaskCallbacks	Cell array version of MaskCallbackString.	cell array { ' [] ' }
MaskDescription	Block description. Set by the Mask description field on the Documentation pane of the Mask Editor dialog box.	character vector { '' }
MaskDisplay	Drawing commands for the block icon. Set by the Icon Drawing commands field on the Icon & Ports pane of the Mask Editor dialog box.	character vector { '' }
MaskEditorHandle	For internal use only.	
MaskEnableString	Option that determines whether a parameter is greyed out in the dialog. Set by the Enable parameter check box on the Parameters pane of the Mask Editor dialog box.	pipe-delimited character vector { '' }
MaskEnables	Cell array version of MaskEnableString.	cell array of character vectors, each either 'on' or 'off' { ' [] ' }

Parameter	Description/Prompt	Values
MaskHelp	Block help. Set by the Mask help field on the Documentation pane of the Mask Editor dialog box.	character vector {''}
MaskIconFrame	Set the visibility of the icon frame (Visible is on, Invisible is off). Set by the Block Frame option on the Icon & Ports pane of the Mask Editor dialog box.	{'on'} 'off'
MaskIconOpaque	Set the transparency of the icon. Set by the Icon Transparency option on the Icon & Ports pane of the Mask Editor dialog box.	{'opaque'} 'transparent' 'opaque-with-ports'
MaskIconRotate	Set the rotation of the icon (Rotates is on, Fixed is off). Set by the Icon Rotation option on the Icon & Ports pane of the Mask Editor dialog box.	'on' {'off'}
MaskIconUnits	Set the units for the drawing commands. Set by the Icon Units option on the Icon & Ports pane of the Mask Editor dialog box.	'pixel' {'autoscale'} 'normalized'
MaskInitialization	Initialization commands. Set by the Initialization commands field on the Initialization pane of the Mask Editor dialog box.	MATLAB command {''}
MaskNames	Cell array of mask dialog parameter names. Set inside the Variable column in the Parameters pane of the Mask Editor dialog box.	matrix {'[]'}

Parameter	Description/Prompt	Values
MaskPortRotate	Specify the port rotation policy for the masked block. Set in the Port Rotation area on the Icon & Ports pane of the Mask Editor dialog box. For more information, see “Adjust Visual Presentation to Improve Model Readability”.	{'default'} 'physical'
MaskPrompts	List of dialog parameter prompts (see below). Set inside the Dialog parameters area on the Parameters pane of the Mask Editor dialog box.	cell array of character vectors {'[]'}
MaskPromptString	List of dialog parameter prompts (see below). Set inside the Dialog parameters area on the Parameters pane of the Mask Editor dialog box.	character vector {''}
MaskPropertyName	Pipe-delimited version of MaskNames.	character vector {''}
MaskRunInitForIconRedraw	Specifies whether Simulink must run mask initialization before executing the mask icon commands.	{'auto'} 'on' 'off'
MaskSelfModifiable	Indicates that the block can modify itself. Set by the Allow library block to modify its contents check box on the Initialization pane of the Mask Editor dialog box.	'on' {'off'}

Parameter	Description/Prompt	Values
MaskStyles	Determines whether the dialog parameter is a check box, edit field, or pop-up list. Set by the Type column in the Parameters pane of the Mask Editor dialog box.	cell array { ' [] ' }
MaskStyleString	Comma-separated version of MaskStyles.	character vector { ' ' }
MaskTabNameString	For internal use only.	
MaskTabNames	For internal use only.	
MaskToolTipsDisplay	Determines which mask dialog parameters to display in the tooltip for this masked block. Specify as a cell array of 'on' or 'off' values, each of which indicates whether to display the parameter named at the corresponding position in the cell array returned by MaskNames.	cell array of 'on' and 'off' {}
MaskToolTipString	Comma-delimited version of MaskToolTipsDisplay.	character vector { ' ' }
MaskTunableValues	Allows the changing of mask dialog values during simulation. Set by the Tunable column in the Parameters pane of the Mask Editor dialog box.	cell array of character vectors { ' [] ' }
MaskTunableValueString	Comma-delimited character vector version of MaskTunableValues.	delimited character vector { ' ' }
MaskType	Mask type. Set by the Mask type field on the Documentation pane of the Mask Editor dialog box.	character vector { 'Stateflow' }

Parameter	Description/Prompt	Values
MaskValues	Dialog parameter values.	cell array { ' [] ' }
MaskValueString	Delimited character vector version of MaskValues.	delimited character vector { ' ' }
MaskVarAliases	Specify aliases for a block's mask parameters. The aliases must appear in the same order as the parameters appear in the block's MaskValues parameter.	cell array { ' [] ' }
MaskVarAliasString	For internal use only.	
MaskVariables	List of the dialog parameters' variables (see below). Set inside the Dialog parameters area on the Parameters pane of the Mask Editor dialog box.	character vector { ' ' }
MaskVisibilities	Specifies visibility of parameters. Set with the Show parameter check box in the Options for selected parameter area on the Parameters pane of the Mask Editor dialog box.	matrix { ' [] ' }
MaskVisibilityString	Delimited character vector version of MaskVisibilities.	character vector { ' ' }
MaskWSVariables	List of the variables defined in the mask workspace (read only).	matrix { ' [] ' }

See *Control Masks Programmatically*, for more information on setting the mask parameters from the MATLAB command line.

Fixed-Point Tool

- “Fixed-Point Tool Parameters and Dialog Box” on page 7-2
- “Advanced Settings” on page 7-20

Fixed-Point Tool Parameters and Dialog Box

The Fixed-Point Tool includes the following components:

- **Main** toolbar
- **Model Hierarchy** pane
- **Contents** pane
- **Dialog** pane

Main Toolbar

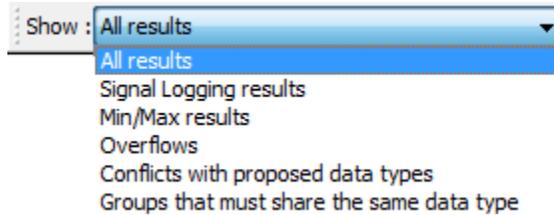
The Fixed-Point Tool's main toolbar appears near the top of the Fixed-Point Tool window under the Fixed-Point Tool's menu.

The toolbar contains the following buttons that execute commonly used Fixed-Point Tool commands:

Button	Usage
	Open the Fixed-Point Advisor to prepare the model for conversion to fixed point.
	Simulate a model and store the run results.
	Pause a simulation.
	Stop a simulation.
	Analyze model and store derived minimum and maximum results.
	Propose data types. Propose fraction lengths for specified word lengths or propose word lengths for specified fraction lengths.
	Apply accepted data types.
	Compare selected runs.
	Create a difference plot for the selected signals.
	Plot the selected signal.

Button	Usage
	Create a histogram plot for the selected signal.

The toolbar also contains the **Show** option:



The **Show** option specifies the type of results to display in the **Contents** pane. The **Contents** pane displays information only after you simulate a system or propose fraction lengths. If there are no results that satisfy a particular filter option, the list will be blank.

Show Option	Result
All results	Displays all results for the selected tree node.
Signal Logging results	For the selected tree node, displays blocks whose output ports have logged signal data. The Fixed-Point tool marks these blocks with the logged signal icon  . Note You can plot simulation results associated with logged signal data using the Simulation Data Inspector.
Min/Max results	For the selected tree node, displays blocks that record design Min/Max, simulation Min/Max, and overflow data. Prerequisites: Fixed-point instrumentation mode should not be set to <code>Force Off</code> .
Overflows	For the selected tree node, displays blocks that have non-zero overflows recorded. If a block has its Saturate on integer overflow option selected, overflow information appears in the Saturations column, otherwise it appears in the OverflowWraps column.

Show Option	Result
<p>Conflicts with proposed data types</p>	<p>For the selected tree node, displays results that have potential data typing or scaling issues.</p> <p>Prerequisites: This information is available only after you propose data types.</p> <p>The Fixed-Point Tool marks these results with a yellow or red icon, as shown here:</p> <div style="border: 1px solid black; padding: 5px;"> <p> The proposed data type poses potential issues for this object. Open the Result Details tab to review these issues.</p> <p> The proposed data type will introduce errors if applied to this object. Open the Result Details tab for details about how to resolve these issues.</p> </div>
<p>Groups that must share the same data type</p>	<p>For the selected tree node, displays blocks that must share the same data type because of data type propagation rules.</p> <p>Prerequisites: This information is available only after you propose fraction lengths.</p> <p>The Fixed-Point Tool allocates an identification tag to blocks that must share the same data type. This identification tag is displayed in the DTGroup column as follows:</p> <ul style="list-style-type: none"> • If the selected tree node is the model root <p>All results for the model are listed. The DTGroup column is sorted by default so that you can easily view all blocks in a group.</p> • If the selected tree node is a subsystem <p>The identification tags have a suffix that indicates the total number of results in each group. For example, G2 (2) means group G2 has 2 members. This information enables you to see how many members of a group belong to the selected subsystem and which groups share data types across subsystem boundaries.</p>

Model Hierarchy Pane

The **Model Hierarchy** pane displays a tree-structured view of the Simulink model hierarchy. The first node in the pane represents a Simulink model. Expanding the root node displays subnodes that represent the model's subsystems, MATLAB Function blocks, Stateflow charts, and referenced models.

The Fixed-Point Tool's **Contents** pane displays elements that comprise the object selected in the **Model Hierarchy** pane. The **Dialog** pane provides parameters for specifying the selected object's data type override and fixed-point instrumentation mode. You can also specify an object's data type override and fixed-point instrumentation mode by right-clicking on the object. The **Model Hierarchy** pane indicates the value of these parameters by displaying the following abbreviations next to the object name:

Abbreviation	Parameter Value
Fixed-point instrumentation mode	
mno	Minimums, maximums and overflows
o	Overflows only
fo	Force off
Data type override	
scl	Scaled double
dbl	Double
sgl	Single
off	Off

Contents Pane

The **Contents** pane displays a tabular view of objects that log fixed-point data in the system or subsystem selected in the **Model Hierarchy** pane. The table rows correspond to model objects, such as blocks, block parameters, and Stateflow data. The table columns correspond to attributes of those objects, such as the data type, design minimum and maximum values, and simulation minimum and maximum values.

The **Contents** pane displays information only after you simulate a system, analyze the model to derive minimum and maximum values, or propose fraction lengths.

You can control which of the following columns the Fixed-Point Tool displays in this pane. For more information, see “Customizing the Contents Pane View” on page 7-8.

Column Label	Description
Accept	Check box that enables you to selectively accept the Fixed-Point Tool's data type proposal.
CompiledDesignMax	Compile-time information for DesignMax .
CompiledDesignMin	Compile-time information for DesignMin .
CompiledDT	Compile-time data type. This data type appears on the signal line in <code>sfix</code> format. See “Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer).
DerivedMax	Maximum value the Fixed-Point tool derives for this signal from design ranges specified for blocks.
DerivedMin	Minimum value the Fixed-Point tool derives for this signal from design ranges specified for blocks.
DesignMax	Maximum value the block specifies in its parameter dialog box, for example, the value of its Output maximum parameter.
DesignMin	Minimum value the block specifies in its parameter dialog box, for example, the value of its Output minimum parameter.
DivByZero	Number of divide-by-zero instances that occur during simulation.
DTGroup	Identification tag associated with objects that share data types.
InitValueMax	<p>Maximum initial value for a signal or parameter. Some model objects provide parameters that allow you to specify the initial values of their signals. For example, the Constant block includes a Constant value that initializes the block output signal.</p> <hr/> <p>Note The Fixed-Point Tool uses this parameter when it proposes data types.</p>

Column Label	Description
InitValueMin	<p>Minimum initial value for a signal or parameter. Some model objects provide parameters that allow you to specify the initial values of their signals. For example, the Constant block includes a Constant value that initializes the block output signal.</p> <hr/> <p>Note The Fixed-Point Tool uses this parameter when it proposes data types.</p>
LogSignal	<p>Check box that allows you to enable or disable signal logging for an object.</p>
ModelRequiredMin	<p>Minimum value of a parameter used during simulation. For example, the n-D Lookup Table block uses the Breakpoints and Table data parameters to perform its lookup operation and generate output. In this example, the block uses more than one parameter so the Fixed-Point Tool sets ModelRequiredMin to the minimum of the minimum values of all these parameters.</p> <hr/> <p>Note The Fixed-Point Tool uses this parameter when it proposes data types.</p>
ModelRequiredMax	<p>Maximum value of a parameter used during simulation. For example, the n-D Lookup Table block uses the Breakpoints and Table data parameters to perform its lookup operation and generate output. In this example, the block uses more than one parameter so the Fixed-Point Tool sets ModelRequiredMax to the maximum of the maximum values of all these parameters.</p> <hr/> <p>Note The Fixed-Point Tool uses this parameter when it proposes data types.</p>
Name	<p>Identifies path and name of block.</p>
OverflowWraps	<p>Number of overflows that wrap during simulation.</p>
ProposedDT	<p>Data type that the Fixed-Point Tool proposes.</p>

Column Label	Description
ProposedMax	Maximum value that results from the data type the Fixed-Point Tool proposes.
ProposedMin	Minimum value that results from the data type the Fixed-Point Tool proposes.
Run	Indicates the run name for these results.
Saturations	Number of overflows that saturate during simulation.
SimDT	Data type the block uses during simulation. This data type appears on the signal line in <code>sfix</code> format. See “Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer).
SimMax	Maximum value that occurs during simulation.
SimMin	Minimum value that occurs during simulation.
SpecifiedDT	Data type the block specifies in its parameter dialog box, for example, the value of its Output data type parameter.

Customizing the Contents Pane View

You can customize the **Contents** pane in the following ways:

- “Using Column Views” on page 7-8
- “Changing Column Order and Width” on page 7-10
- “Sorting by Columns” on page 7-10

Using Column Views

The Fixed-Point Tool provides the following standard Column Views:

View Name	Columns Provided	When Does the Fixed-Point Tool Display this View?
Simulation View (default)	Name, Run, CompiledDT, SpecifiedDT, SimMin, SimMax, DesignMin, DesignMax, OverflowWraps, Saturations	After a simulating minimum and maximum values.

View Name	Columns Provided	When Does the Fixed-Point Tool Display this View?
Automatic Data Typing View	Name, Run, CompiledDT, CompiledDesignMax, CompiledDesignMin, Accept, ProposedDT, SpecifiedDT, DesignMin, DesignMax, DerivedMin, DerivedMax, SimMin, SimMax, OverflowWraps, Saturations, ProposedMin, ProposedMax	After proposing data types if proposal is based on simulation, derived, and design min/max.
Automatic Data Typing With Simulation Min/Max View	Name, Run, CompiledDT, Accept, ProposedDT, SpecifiedDT, SimMin, SimMax, DesignMin, DesignMax, OverflowWraps, Saturations, ProposedMin, ProposedMax	After proposing data types if the proposal is based on simulation and design min/max.
Automatic Data Typing With Derived Min/Max View	Name, Run, CompiledDesignMax, CompiledDesignMin, Accept, ProposedDT, SpecifiedDT, DerivedMin, DerivedMax, ProposedMin, ProposedMax	After proposing data types if the proposal is based on design min/max and/or derived min/max.
Data Collection View	Name, Run, CompiledDT, SpecifiedDT, DerivedMin, DerivedMax, SimMin, SimMax, OverflowWraps, Saturations	After simulating or deriving minimum and maximum values if the results have simulation min/max, derived min/max, and design min/max.
Derived Min/Max View	Name, Run, CompiledDesignMax, CompiledDesignMin, DerivedMin, DerivedMax	After deriving minimum and maximum values.

By selecting **Show Details**, you can:

- Customize the standard column views

- Create your own column views
- Export and import column views saved in MAT-files, which you can share with other users
- Reset views to factory settings

If you upgrade to a new release of Simulink, and the column views available in the Fixed-Point Tool do not match the views described in the documentation, reset your views to factory settings. When you reset all views, the Model Explorer removes all the custom views you have created. Before you reset views to factory settings, export any views that you will want to use in the future.

You can prevent the Fixed-Point Tool from automatically changing the column view of the contents pane by selecting **View > Lock Column View** in the Fixed-Point Tool menu. For more information on controlling views, see “Customize Model Explorer Views”.

Changing Column Order and Width

You can alter the order and width of columns that appear in the **Contents** pane as follows:

- To move a column, click and drag the head of a column to a new location among the column headers.
- To make a column wider or narrower, click and drag the right edge of a column header. If you double-click the right edge of a column header, the column width changes to fit its contents.

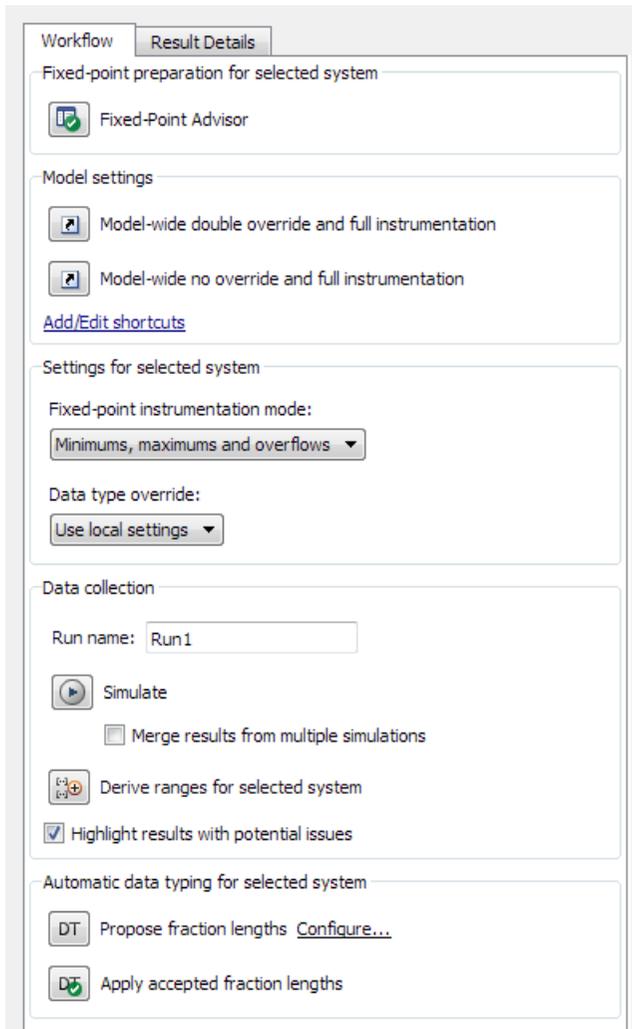
Sorting by Columns

By default, the **Contents** pane displays its contents in ascending order of the **Name** column. You can alter the order in which the **Contents** pane displays its rows as follows:

- To sort all the rows in ascending order of another column, click the head of that column.
- To change the order from ascending to descending, simply click again on the head of that column.

Dialog Pane

Use the Dialog pane to view and change properties associated with the system under design.



The Dialog pane includes the following components:

Component	Description
System under design	Displays the system under design for conversion. You can change the selected system by clicking Change .

Component	Description
Fixed-point preparation	Contains the Fixed-Point Advisor button. Use this button to open the Fixed-Point Advisor to guide you through the tasks to prepare your floating-point model for conversion to fixed point. For more information, see “Fixed-Point Advisor” on page 7-12.
Configure model settings	Contains default configurations that set up run parameters, such as the run name and data type override settings, by clicking a button. For more information, see “Configure model settings” on page 7-13.
Range collection	Contains controls to collect simulation or derived minimum and maximum data for your model.
Automatic data typing	Contains controls to propose and, optionally, accept data type proposals.
Result Details tab	Use this tab to view data type information about the object selected in the Contents pane.

Tips

From the Fixed-Point Tool **View** menu, you can customize the layout of the **Dialog** pane. Select:

- **Show Fixed-Point Preparation** to show/hide the **Fixed-Point Advisor** button. By default, the Fixed-Point Tool displays this button.
- **Show Dialog View** to show/hide the **Dialog** pane. By default, the Fixed-Point Tool displays this pane.
- **Settings for selected system** to show/hide the **Settings for selected system** pane. By default, the Fixed-Point Tool displays this pane.

Fixed-Point Advisor

Open the Fixed-Point Advisor to guide you through the tasks to prepare a floating-point model for conversion to fixed point. Use the Fixed-Point Advisor if your model contains blocks that do not support fixed-point data types.

Configure model settings

Use the configurations to set up model-wide data type override and instrumentation settings prior to simulation. The Fixed-Point Tool provides:

- Frequently-used factory default configurations
- The ability to add and edit custom configurations

Note The factory default configurations apply to the whole model. You cannot use these shortcuts to configure subsystems.

Factory Defaults

Factory Default Configuration	Description
<p>Range collection using double override</p>	<p>Use this configuration to observe ideal numeric behavior of the model and collect ranges for data type proposals.</p> <p>This configuration sets:</p> <ul style="list-style-type: none"> • Run name to DoubleOverride • Fixed-point instrumentation mode to Minimums, maximums and overflows • Data type override to Double • Data type override applies to to All numeric types <p>By default, a button for this configuration appears in the Configure model settings pane.</p>

Factory Default Configuration	Description
Range collection with specified data types	<p>Use this configuration to collect ranges of actual model and to validate current behavior.</p> <p>This configuration sets:</p> <ul style="list-style-type: none"> • Run name to NoOverride • Fixed-point instrumentation mode to Minimums, maximums and overflows • Data type override to Use local settings <p>By default, a button for this shortcut appears in the Configure model settings pane.</p>
Remove overrides and disable range collection	<p>Use this configuration to cleanup settings after finishing fixed-point conversion and to restore maximum simulation speed.</p> <p>This configuration sets:</p> <ul style="list-style-type: none"> • Fixed-point instrumentation mode to Off • Data type override to Use local settings <p>By default, a button for this shortcut appears in the Configure model settings pane.</p>

Advanced settings

Use **Advanced settings** to add new configurations or edit existing user-defined configurations.

Run name

Specifies the run name

If you use a default configuration to set up a run, the Fixed-Point Tool uses the run name associated with this configuration. You can override the run name by entering a new name in this field.

Tips

- To store data for multiple runs, provide a different run name for each run. Running two simulations with the same run name overwrites the original run unless you select **Merge results from multiple simulations**.
- You can edit the run name in the Contents pane **Run** column.

For more information, see “Run Management” (Fixed-Point Designer).

Simulate

Simulates model and stores results.

Action

Simulates the model and stores the results with the run name specified in **Run name**. The Fixed-Point Tool displays the run name in the **Run** column of the **Contents** pane.

Merge instrumentation results from multiple simulations

Control how simulation results are stored

Settings

Default: Off

On

Merges new simulation minimum and maximum results with existing simulation results in the run specified by the run name parameter. Allows you to collect complete range information from multiple test benches. Does not merge signal logging results.

Off

Clears all existing simulation results from the run specified by the run name parameter before displaying new simulation results.

Command-Line Alternative

Parameter: 'MinMaxOverflowArchiveMode'

Type: string

Value: 'Overwrite' | 'Merge'

Default: 'Overwrite'

Tip

Select this parameter to log simulation minimum and maximum values captured over multiple simulations. For more information, see “Propose Data Types For Merged Simulation Ranges” (Fixed-Point Designer).

Derive ranges for selected system

Derive minimum and maximum values for signals for the selected system.

The Fixed-Point Tool analyzes the selected system to compute derived minimum and maximum values based on design minimum and maximum values specified on blocks. For example, using the **Output minimum** and **Output maximum** for block outputs.

Action

Analyzes the selected system to compute derived minimum and maximum information based on the design minimum and maximum values specified on blocks.

By default, the Fixed-Point Tool displays the `Derived Min/Max View` with the following information in the **Contents** pane.

Command-Line Alternative

No command line alternative available.

Dependencies

Range analysis:

- Requires a Fixed-Point Designer license.

Propose

Signedness

Select whether you want The Fixed-Point Tool to propose signedness for results in your model. The Fixed-Point Tool proposes signedness based on collected range data and block constraints. By default, the **Signedness** check box is selected.

When the check box is selected, signals that are always strictly positive get an unsigned data type proposal. If you clear the check box, the Fixed-Point Tool proposes a signed data type for all results that currently specify a floating-point or an inherited output data type unless other constraints are present. If a result specifies a fixed-point output data type, the Fixed-Point Tool will propose a data type with the same signedness as the currently specified data type unless other constraints are present.

Word length or fraction length

Select whether you want the Fixed-Point Tool to propose word lengths or fraction lengths for the objects in your system.

- If you select **Word length**, the Fixed-Point Tool proposes a data type with the specified fraction length and the minimum word length to avoid overflows.
- If you select **Fraction length**, the Fixed-Point Tool proposes a data type with the specified word length and best-precision fraction length while avoiding overflows.

If a result currently specifies a fixed-point data type, that information will be used in the proposal. If a result specifies a floating-point or inherited output data type, and the **Inherited** and **Floating point** check boxes are selected, the Fixed-Point Tool uses the settings specified under **Automatic data typing** to make a data type proposal.

Propose for

Inherited

Propose data types for results that specify one of the inherited output data types.

Floating-point

Propose data types for results that specify floating-point output data types.

Default fraction length

Specify the default fraction length for objects in your model. The Fixed-Point Tool proposes a data type with the specified fraction length and the minimum word length that avoids overflows.

Command-Line Alternative

No command line alternative available.

Default word length

Specify the default word length for objects in your model. The Fixed-Point Tool will propose best-precision fraction lengths based on the specified default word length.

Command-Line Alternative

No command line alternative available.

When proposing types use

Specify the types of ranges to use for data type proposals.

Design and derived ranges

The Fixed-Point Tool uses the design ranges in conjunction with derived ranges to propose data types. Design ranges take precedence over derived ranges.

Design and simulation ranges

The Fixed-Point Tool uses the design ranges in conjunction with collected simulation ranges to propose data types. Design ranges take precedence over simulation ranges.

The **Safety margin for simulation min/max (%)** parameter specifies a range that differs from that defined by the simulation range. For more information, see “Safety margin for simulation min/max (%)” on page 7-18

All collected ranges

The Fixed-Point Tool uses design ranges in addition to derived and simulation ranges to propose data types.

Design minimum and maximum values take precedence over simulation and derived ranges.

Command-Line Alternative

No command line alternative available.

Safety margin for simulation min/max (%)

Specify safety factor for simulation minimum and maximum values.

Settings

Default: 0

The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. The specified safety margin must be a real number greater than -100. For example, a value of 55 specifies that a range *at least* 55 percent larger is desired. A value of -15 specifies that a range *up to* 15 percent smaller is acceptable.

Dependencies

Before performing automatic data typing, you must specify design minimum and maximum values or run a simulation to collect simulation minimum and maximum data, or collect derived minimum and maximum values.

Command-Line Alternative

No command line alternative available.

Advanced Settings

In this section...
“Advanced Settings Overview” on page 7-20
“Fixed-point instrumentation mode” on page 7-20
“Data type override” on page 7-21
“Data type override applies to” on page 7-23
“Name of shortcut” on page 7-25
“Allow modification of fixed-point instrumentation settings” on page 7-25
“Allow modification of data type override settings” on page 7-26
“Allow modification of run name” on page 7-27
“Run name” on page 7-27
“Capture system settings” on page 7-27
“Fixed-point instrumentation mode” on page 7-27
“Data type override” on page 7-28
“Data type override applies to” on page 7-29

Advanced Settings Overview

Use the Advanced Settings dialog to control the fixed-point instrumentation mode, and data type override settings. You can also use the Advanced Settings dialog to add or edit user-defined configurations. You cannot modify the factory default configurations. If you add a new configuration and want it to appear as a button on the Fixed-Point Tool **Configure model settings** pane, use the controls in the **Shortcuts** tab.

Fixed-point instrumentation mode

Control which objects log minimum, maximum and overflow data during simulation.

Settings

Default: Use local settings

Use local settings

Logs data according to the value of this parameter set for each subsystem. Otherwise, settings for parent systems always override those of child systems.

Minimums, maximums and overflows

Logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem during simulation.

Overflows only

Logs only overflow data for all blocks in the current system or subsystem.

Force off

Does not log data for any block in the current system or subsystem. Use this selection to work with models containing fixed-point enabled blocks if you do not have a Fixed-Point Designer license.

Tips

- You cannot change the instrumentation mode for linked subsystems or referenced models.

Dependencies

The value of this parameter for parent systems controls min/max logging for all child subsystems, unless Use local settings is selected.

Command-Line Alternative

Parameter: 'MinMaxOverflowLogging'

Type: string

Value: 'UseLocalSettings' | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'

Default: 'UseLocalSettings'

Data type override

Control data type override of objects that allow you to specify data types in their dialog boxes.

Settings

Default: Use local settings

The value of this parameter for parent systems controls data type override for all child subsystems, unless `Use local settings` is selected.

`Use local settings`

Overrides data types according to the setting of this parameter for each subsystem.

`Scaled double`

Overrides the data type of all blocks in the current system and subsystem with doubles; however, the scaling and bias specified in the dialog box of each block is maintained.

`Double`

Overrides the output data type of all blocks in the current system or subsystem with doubles. The overridden values have no scaling or bias.

`Single`

Overrides the output data type of all blocks in the current system or subsystem with singles. The overridden values have no scaling or bias.

`Off`

No data type override is performed on any block in the current system or subsystem. The settings on the blocks are used.

Tips

- Set this parameter to `Double` or `Single` and the **Data type override applies to** parameter to `All numeric types` to work with models containing fixed-point enabled blocks if you do not have a Fixed-Point Designer license.
- You cannot change the **Data type override** setting on linked subsystems or referenced models.
- Data type override never applies to `boolean` data types.
- When you set the **Data type override** parameter of a parent system to `Double`, `Single`, `Scaled double` or `Off`, this setting also applies to all child subsystems and you cannot change the data type override setting for these child subsystems. When the **Data type override** parameter of a parent system is `Use local settings`, you can set the **Data type override** parameter for individual children.
- Use this parameter with the **Data type override applies to** parameter. The following table details how these two parameters affect the data types in your model.

Fixed-Point Tool Settings		Block Local Settings	
Data type override	Data type override applies to	Floating-point types	Fixed-point types
Use local settings/Off	N/A	Unchanged	Unchanged
Double	All numeric types	Double	Double
	Floating-point	Double	Unchanged
	Fixed-point	Unchanged	Double
Single	All numeric types	Single	Single
	Floating-point	Single	Unchanged
	Fixed-point	Unchanged	Single
Scaled double	All numeric types	Double	Scaled double equivalent of fixed-point type
	Floating-point	Double	Unchanged
	Fixed-point	Unchanged	Scaled double equivalent of fixed-point type

Dependencies

- The following Simulink blocks allow you to set data types in their block masks, but ignore the **Data type override** setting:
 - Probe
 - Trigger
 - Width

Command-Line Alternative

Parameter: 'DataTypeOverride'

Type: string

Value: 'UseLocalSettings' | 'ScaledDouble' | 'Double' | 'Single' | 'Off'

Default: 'UseLocalSettings'

Data type override applies to

Specifies which data types the Fixed-Point Tool overrides

Settings

Default: All numeric types

All numeric types

Data type override applies to all numeric types, floating-point and fixed-point. It does not apply to `boolean` or enumerated data types.

Floating-point

Data type override applies only to floating-point data types, that is, `double` and `single`.

Fixed-point

Data type override applies only to fixed-point data types, for example, `uint8`, `fixdt`.

Tips

- Use this parameter with the **Data type override** parameter.
- Data type override never applies to `boolean` or enumerated data types or to buses.
- When you set the **Data type override** parameter of a parent system to `Double`, `Single`, `Scaled double` or `Off`, this setting also applies to all child subsystems and you cannot change the data type override setting for these child subsystems. When the **Data type override** parameter of a parent system is `Use local setting`, you can set the **Data type override** parameter for individual children.
- The following table details how these two parameters affect the data types in your model.

Fixed-Point Tool Settings		Block Local Settings	
Data type override	Data type override applies to	Floating-point types	Fixed-point types
Use local settings/Off	N/A	Unchanged	Unchanged
Double	All numeric types	Double	Double
	Floating-point	Double	Unchanged
	Fixed-point	Unchanged	Double
Single	All numeric types	Single	Single
	Floating-point	Single	Unchanged
	Fixed-point	Unchanged	Single

Fixed-Point Tool Settings		Block Local Settings	
Data type override	Data type override applies to	Floating-point types	Fixed-point types
Scaled double	All numeric types	Double	Scaled double equivalent of fixed-point type
	Floating-point	Double	Unchanged
	Fixed-point	Unchanged	Scaled double equivalent of fixed-point type

Dependencies

This parameter is enabled only when **Data type override** is set to Scaled double, Double or Single.

Command-Line Alternative

Parameter: 'DataTypeOverrideAppliesTo'

Type: string

Value: 'AllNumericTypes' | 'Floating-point' | 'Fixed-point'

Default: 'AllNumericTypes'

Name of shortcut

Enter a unique name for your shortcut. By default, the Fixed-Point Tool uses this name as the **Run name** for this shortcut.

If the shortcut name already exists, the new settings overwrite the existing settings.

See Also

- “Run Management” (Fixed-Point Designer)

Allow modification of fixed-point instrumentation settings

Select whether to change the model fixed-point instrumentation settings when you apply this shortcut to the model.

Settings

Default: On

On

When you apply this shortcut to the model, changes the fixed-point instrumentation settings of the model and its subsystems to the setting defined in this shortcut.

Off

Does not change the fixed-point instrumentation settings when you apply this shortcut to the model.

Tip

If you want to control data type override settings without altering the fixed-point instrumentation settings on your model, clear this option.

See Also

- “Run Management” (Fixed-Point Designer)

Allow modification of data type override settings

Select whether to change the model data type override settings when you apply this shortcut to the model

Settings

Default: On

On

When you apply this shortcut to the model, changes the data type override settings of the model and its subsystems to the settings defined in this shortcut .

Off

Does not change the fixed-point instrumentation settings when you apply this shortcut to the model.

Allow modification of run name

Select whether to change the run name on the model when you apply this shortcut to the model

Settings

Default: On

On

Changes the run name to the setting defined in this shortcut when you apply this shortcut to the model.

Off

Does not change the run name when you apply this shortcut to the model.

Run name

Specify the run name to use when you apply this shortcut.

By default, the run name uses the name of the shortcut. Run names are case sensitive.

Dependency

Allow modification of run name enables this parameter.

Capture system settings

Copy the model and subsystem fixed-point instrumentation mode and data type override settings into the Shortcut editor.

Fixed-point instrumentation mode

Control which objects in the shortcut editor log minimum, maximum and overflow data during simulation.

This information is stored in the shortcut. To use the current model setting, click **Capture system settings**.

Settings

Default: Same as model setting

Use local settings

Logs data according to the value of this parameter set for each subsystem. Otherwise, settings for parent systems always override those of child systems.

Minimums, maximums and overflows

Logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem during simulation.

Overflows only

Logs only overflow data for all blocks in the current system or subsystem.

Force off

Does not log data for any block in the current system or subsystem. Use this selection to work with models containing fixed-point enabled blocks if you do not have a Fixed-Point Designer license.

Dependency

Allow modification of fixed-point instrumentation settings enables this parameter.

Data type override

Control data type override of objects that allow you to specify data types in their dialog boxes.

This information is stored in the shortcut. To use the current model settings, click **Capture system settings**.

Settings

Default: Same as model

The value of this parameter for parent systems controls data type override for all child subsystems, unless Use local settings is selected.

Use local settings

Overrides data types according to the setting of this parameter for each subsystem.

Scaled double

Overrides the data type of all blocks in the current system and subsystem with doubles; however, the scaling and bias specified in the dialog box of each block is maintained.

Double

Overrides the output data type of all blocks in the current system or subsystem with doubles. The overridden values have no scaling or bias.

Single

Overrides the output data type of all blocks in the current system or subsystem with singles. The overridden values have no scaling or bias.

Off

No data type override is performed on any block in the current system or subsystem. The settings on the blocks are used.

Dependency

Allow modification of data type override settings enables this parameter.

Data type override applies to

Specifies which data types to override when you apply this shortcut.

This information is stored in the shortcut. To use the current model setting, click **Capture system settings**.

Settings

Default: All numeric types

All numeric types

Data type override applies to all numeric types, floating-point and fixed-point. It does not apply to boolean or enumerated data types.

Floating-point

Data type override applies only to floating-point data types, that is, double and single.

Fixed-point

Data type override applies only to fixed-point data types, for example, uint8, fixdt.

Dependency

Allow modification of data type override settings enables this parameter.

Model Advisor Checks

Simulink Checks

In this section...
“Simulink Check Overview” on page 8-4
“Migrating to Simplified Initialization Mode Overview” on page 8-5
“Identify unconnected lines, input ports, and output ports” on page 8-5
“Check root model Inport block specifications” on page 8-6
“Check optimization settings” on page 8-7
“Check diagnostic settings ignored during accelerated model reference simulation” on page 8-9
“Check for parameter tunability information ignored for referenced models” on page 8-10
“Check for implicit signal resolution” on page 8-11
“Check for optimal bus virtuality” on page 8-11
“Check for Discrete-Time Integrator blocks with initial condition uncertainty” on page 8-12
“Identify disabled library links” on page 8-13
“Check for large number of function arguments from virtual bus across model reference boundary” on page 8-14
“Identify parameterized library links” on page 8-15
“Identify unresolved library links” on page 8-16
“Identify model reference variants and variant subsystems that override variant choice” on page 8-17
“Identify configurable subsystem blocks for converting to variant subsystem blocks” on page 8-17
“Check usage of function-call connections” on page 8-18
“Check model for upgradable Simulink Scope blocks” on page 8-19
“Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues” on page 8-19
“Check if read/write diagnostics are enabled for data store blocks” on page 8-21
“Check data store block sample times for modeling errors” on page 8-22
“Check for potential ordering issues involving data store access” on page 8-23

In this section...

- “Check structure parameter usage with bus signals” on page 8-24
- “Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition” on page 8-25
- “Check for calls to `slDataTypeAndScale`” on page 8-27
- “Check bus signals treated as vectors” on page 8-29
- “Check for potentially delayed function-call subsystem return values” on page 8-30
- “Identify block output signals with continuous sample time and non-floating point data type” on page 8-31
- “Check usage of Merge blocks” on page 8-32
- “Check usage of Outport blocks” on page 8-35
- “Check usage of Discrete-Time Integrator blocks” on page 8-47
- “Check model settings for migration to simplified initialization mode” on page 8-48
- “Check S-functions in the model” on page 8-50
- “Check for non-continuous signals driving derivative ports” on page 8-51
- “Runtime diagnostics for S-functions” on page 8-52
- “Check model for foreign characters” on page 8-53
- “Identify unit mismatches in the model” on page 8-54
- “Identify automatic unit conversions in the model” on page 8-54
- “Identify disallowed unit systems in the model” on page 8-55
- “Identify undefined units in the model” on page 8-55
- “Check model for block upgrade issues” on page 8-56
- “Check model for block upgrade issues requiring compile time information” on page 8-57
- “Check that the model is saved in SLX format” on page 8-58
- “Check model for SB2SL blocks” on page 8-59
- “Check Model History properties” on page 8-59
- “Identify Model Info blocks that can interact with external source control tools” on page 8-60
- “Identify Model Info blocks that use the Configuration Manager” on page 8-61
- “Check model for legacy 3DoF or 6DoF blocks” on page 8-62

In this section...

- “Check model and local libraries for legacy Aerospace Blockset blocks” on page 8-62
- “Check and update masked blocks in library to use promoted parameters” on page 8-63
- “Check and update mask image display commands with unnecessary imread() function calls” on page 8-64
- “Check and update mask to affirm icon drawing commands dependency on mask workspace” on page 8-65
- “Identify masked blocks that specify tabs in mask dialog using MaskTabNames parameter” on page 8-66
- “Identify questionable operations for strict single-precision design” on page 8-66
- “Check get_param calls for block CompiledSampleTime” on page 8-68
- “Check model for parameter initialization and tuning issues” on page 8-70
- “Check for virtual bus across model reference boundaries” on page 8-71
- “Check model for custom library blocks that rely on frame status of the signal” on page 8-73
- “Check Rapid accelerator signal logging” on page 8-73
- “Check virtual bus inputs to blocks” on page 8-74
- “Check for root outports with constant sample time” on page 8-78
- “Analyze model hierarchy and continue upgrade sequence” on page 8-79
- “Check Access to Data Stores” on page 8-81

Simulink Check Overview

Use the Simulink Model Advisor checks to configure your model for simulation.

See Also

- “Run Model Checks”
- “Simulink Coder Checks” (Simulink Coder)
- “Simulink Check Checks” (Simulink Check)

Migrating to Simplified Initialization Mode Overview

Simplified initialization mode was introduced in R2008b to improve the consistency of simulation results. This mode is especially important for models that do not specify initial conditions for conditionally executed subsystem output ports. For more information, see “Simplified Initialization Mode” and “Classic Initialization Mode”.

Use the Model Advisor checks in **Migrating to Simplified Initialization Mode** to help migrate your model to simplified initialization mode.

See Also

- “Simplified Initialization Mode”
- “Classic Initialization Mode”
- “Underspecified initialization detection”
- “Check usage of Merge blocks” on page 8-32
- “Check usage of Outport blocks” on page 8-35
- “Check usage of Discrete-Time Integrator blocks” on page 8-47
- “Check model settings for migration to simplified initialization mode” on page 8-48

Identify unconnected lines, input ports, and output ports

Check ID: `mathworks.design.UnconnectedLinesPorts`

Check for unconnected lines or ports.

Description

This check lists unconnected lines or ports. These can have difficulty propagating signal attributes such as data type, sample time, and dimensions.

Note Ports connected to ground/terminator blocks will pass this test.

Results and Recommended Actions

Condition	Recommended Action
Lines, input ports, or output ports are unconnected.	Connect the signals. Double-click the list of unconnected items to locate failure.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Tips

Use the `PortConnectivity` command to obtain an array of structures describing block input or output ports.

See Also

“Common Block Properties” on page 6-109 for information on the `PortConnectivity` command.

“What Is a Model Advisor Exclusion?” (Simulink Check)

Check root model Inport block specifications

Check ID: `mathworks.design.RootInportSpec`

Check that root model Inport blocks fully define dimensions, sample time, and data type.

Description

Using root model Inport blocks that do not fully define dimensions, sample time, or data type can lead to undesired simulation results. Simulink software back-propagates dimensions, sample times and data types from downstream blocks unless you explicitly assign them values.

Results and Recommended Actions

Condition	Recommended Action
Root-level Inport blocks have undefined attributes.	Fully define the attributes of the root-level Inport blocks.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

Tips

The following configurations pass this check:

- **Configuration Parameters > Solver > Periodic sample time constraint** is set to `Ensure sample time independent`
- For export-function models, *inherited sample time* is not flagged.

See Also

- “About Data Types in Simulink”.
- “Determine Output Signal Dimensions”.
- “Specify Sample Time”.
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check optimization settings

Check ID: `mathworks.design.OptimizationSettings`

Check for optimizations that can lead to non-optimal code generation and simulation.

Description

This check reviews the status of optimizations that can improve code efficiency and simulation time.

Results and Recommended Actions

Condition	Recommended Action
The specified optimizations are off.	<p>Select the following optimization check boxes on the Optimization pane in the Configuration Parameters dialog box:</p> <ul style="list-style-type: none"> • “Remove root level I/O zero initialization” • “Remove internal data zero initialization” <p>Select the following optimization check boxes on the Optimization > Signals and Parameters pane in the Configuration Parameters dialog box:</p> <ul style="list-style-type: none"> • “Inline invariant signals” (only if you have a Simulink Coder license) <p>Select the following optimization check boxes in the Configuration Parameters dialog box:</p> <ul style="list-style-type: none"> • “Block reduction” • “Conditional input branch execution” • “Implement logic signals as Boolean data (vs. double)” • “Use memset to initialize floats and doubles to 0.0” • “Remove code from floating-point to integer conversions that wraps out-of-range values” (only if you have a Simulink Coder license) • “Signal storage reuse” • “Enable local block outputs” • “Reuse local block outputs” • “Eliminate superfluous local variables (Expression folding)” <p>Select the following optimization check boxes on the Optimization > Stateflow pane in the Configuration Parameters dialog box:</p> <hr/> <p>Note Model Advisor checks these parameters only if there is a Stateflow chart in the model.</p> <hr/> <ul style="list-style-type: none"> • “Use bitsets for storing state configuration”

Condition	Recommended Action
	<ul style="list-style-type: none"> • “Use bitsets for storing Boolean data”
“Application lifespan (days)” is set as infinite. This could lead to expensive 64-bit counter usage.	Choose a stop time if this is not intended.
The specified diagnostics, which can increase the time it takes to simulate your model, are set to warning or error.	Select none for: <ul style="list-style-type: none"> • Solver data inconsistency • Array bounds exceeded • Diagnostics > Data Validity > Simulation range checking
The specified Embedded Coder parameters are off.	If you have an Embedded Coder license and you are using an ERT-based system target file: <ul style="list-style-type: none"> • Select Single output/update function. For details, see “Single output/update function” (Simulink Coder). • Select Ignore test point signals. For details, see “Ignore test point signals” (Simulink Coder). • Set Optimization > Signals and Parameters > Pass reusable subsystem outputs as to <code>Individual arguments</code>. For details, see “Pass reusable subsystem outputs as”.

Tips

If the system contains Model blocks and the referenced model is in Accelerator mode, simulating the model requires generating and compiling code.

See Also

- “Optimization Pane: General”.

Check diagnostic settings ignored during accelerated model reference simulation

Check ID: `mathworks.design.ModelRefSIMConfigCompliance`

Checks for referenced models for which Simulink changes configuration parameter settings during accelerated simulation.

Description

For models referenced in accelerator mode, Simulink ignores the settings of the following configuration parameters that you set to a value other than `None`.

- **Array bounds exceeded**
- **Diagnostics > Data Validity > Inf or NaN block output**
- **Diagnostics > Data Validity > Division by singular matrix**
- **Diagnostics > Data Validity > Wrap on overflow**

Also, for models referenced in accelerator mode, Simulink ignores the following **Configuration Parameters > Diagnostics > Data Validity > Data Store Memory block** parameters if you set them to a value other than `Disable all`. For details, see “Data Store Diagnostics”.

- **Detect read before write**
- **Detect write after read**
- **Detect write after write**

Results and Recommended Actions

Condition	Recommended Action
You want to see the results of running the identified diagnostics with settings to produce warnings or errors.	Simulate the model in Normal mode and resolve diagnostic warnings or errors.

Check for parameter tunability information ignored for referenced models

Check ID: `mathworks.design.ParamTunabilityIgnored`

Checks if parameter tunability information is included in the Model Parameter Configuration dialog box.

Description

Simulink software ignores tunability information specified in the Model Parameter Configuration dialog box. This check identifies those models containing parameter

tunability information that Simulink software will ignore if the model is referenced by other models.

Results and Recommended Actions

Condition	Recommended Action
Model contains ignored parameter tunability information.	Click the links to convert to equivalent Simulink parameter objects in the MATLAB workspace.

See Also

“Block Parameter Representation in the Generated Code” (Simulink Coder).

Check for implicit signal resolution

Check ID: `mathworks.design.ImplicitSignalResolution`

Identify models that attempt to resolve named signals and states to `Simulink.Signal` objects.

Description

Requiring Simulink software to resolve all named signals and states is inefficient and slows incremental code generation and model reference. This check identifies those signals and states for which you may turn off implicit signal resolution and enforce resolution.

Results and Recommended Actions

Condition	Recommended Action
Not all signals and states are resolved.	Turn off implicit signal resolution and enforce resolution for each signal and state that does resolve.

See Also

“Resolve Signal Objects for Output Data”.

Check for optimal bus virtuality

Check ID: `mathworks.design.OptBusVirtuality`

Identify virtual buses that could be made nonvirtual. Making these buses nonvirtual improves generated code efficiency.

Description

This check identifies blocks incorporating virtual buses that cross a subsystem boundary. Changing these to nonvirtual improves generated code efficiency.

Results and Recommended Actions

Condition	Recommended Action
Blocks that specify a virtual bus crossing a subsystem boundary.	Change the highlighted bus to nonvirtual.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- “Signal Basics”.
- “Virtual and Nonvirtual Buses”.
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check for Discrete-Time Integrator blocks with initial condition uncertainty

Check ID: `mathworks.design.DiscreteTimeIntegratorInitCondition`

Identify Discrete-Time Integrator blocks with state ports and initial condition ports that are fed by neither an Initial Condition nor a Constant block.

Description

Discrete-Time Integrator blocks with state port and initial condition ports might not be suitably initialized unless they are fed from an Initial Condition or Constant block. This is more likely to happen when Discrete-Time Integrator blocks are used to model second-order or higher-order dynamic systems.

Results and Recommended Actions

Condition	Recommended Action
Discrete-Time Integrator blocks are not initialized during the model initialization phase.	Add a Constant or Initial Condition block to feed the external Initial Condition port.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- IC block
- Discrete-Time Integrator block
- Constant block
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Identify disabled library links

Check ID: `mathworks.design.DisabledLibLinks`

Search model for disabled library links.

Description

Disabled library links can cause unexpected simulation results. Resolve disabled links before saving a model.

Note This check may overlap with “Check model for block upgrade issues” on page 8-56.

Results and Recommended Actions

Condition	Recommended Action
Library links are disabled.	Click the Library Link > Resolve link option in the context menu.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Tips

- Use the Model Browser to find library links.
- To enable a broken link, right-click a block in your model to display the context menu. Select **Library Link > Resolve link**.

See Also

“Restore Disabled or Parameterized Links”

“What Is a Model Advisor Exclusion?” (Simulink Check)

Check for large number of function arguments from virtual bus across model reference boundary

Check ID: `mathworks.design.CheckVirtualBusAcrossModelReferenceArgs`

Checks virtual bus signals that cross model reference boundaries and flags cases where using virtual buses across a model reference boundary increases the number of function arguments significantly.

Description

To improve the speed of the code generation process, you can use this check to reduce the number of generated function arguments. If the check finds a model that where many arguments will be generated for a function, you can click **Update Model** to modify the model so that it generates fewer arguments.

Results and Recommended Action

Methods that generate many function arguments as the result of a virtual bus signal crossing model reference boundary slow down the code generation process.

Condition	Recommended Action
Methods are listed that generate a large number of arguments for the current the model configuration that this check can reduce by modifying the model.	Click Update Model .

Clicking **Update Model** resets Inport and Outport block parameters and inserts Signal Conversion blocks, as necessary, to reduce the number of generated function arguments for the model.

See Also

“Bus Data Crossing Model Reference Boundaries”

Identify parameterized library links

Check ID: `mathworks.design.ParameterizedLibLinks`

Search model for parameterized library links.

Description

Parameterized library links that are unintentional can result in unexpected parameter settings in your model. This can result in improper model operation.

Results and Recommended Actions

Condition	Recommended Action
Parameterized links are listed.	Verify that the links are intended to be parameterized.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Tips

- Right-click a block in your model to display the context menu. Choose `Link Options` and click `Go To Library Block` to see the original block from the library.
- To parameterize a library link, choose `Look Under Mask`, from the context menu and select the parameter.

See Also

“Restore Disabled or Parameterized Links”

“What Is a Model Advisor Exclusion?” (Simulink Check)

Identify unresolved library links

Check ID: `mathworks.design.UnresolvedLibLinks`

Search the model for unresolved library links, where the specified library block cannot be found.

Description

Check for unresolved library links. Models do not simulate while there are unresolved library links.

Results and Recommended Actions

Condition	Recommended Action
Library links are unresolved.	Locate missing library block or an alternative.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

“Fix Unresolved Library Links”

“What Is a Model Advisor Exclusion?” (Simulink Check)

Identify model reference variants and variant subsystems that override variant choice

Check ID: `mathworks.design.VariantOverride`

Identify model or subsystem for model reference variants and variant subsystems that specify variant choice using the override option instead of using the active variant object.

Results and Recommended Actions

Condition	Recommended Action
Model reference variants or variant subsystems that override variant choice are identified.	Specify variant choice using active variant object.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

“Working with Variant Choices”

“What Is a Model Advisor Exclusion?” (Simulink Check)

Identify configurable subsystem blocks for converting to variant subsystem blocks

Check ID: `mathworks.design.CSStoVSSConvert`

Search the model to identify configurable subsystem blocks at the model or subsystem level.

Results and Recommended Actions

Condition	Recommended Action
Configurable subsystem blocks are identified.	Convert these blocks to variant subsystem blocks to avoid compatibility issues. See Configurable Subsystem.

Capabilities and Limitations

You can run this check on your library models.

See Also

“Set up Model Variants Using a Model Block”

Check usage of function-call connections

Check ID: `mathworks.design.CheckForProperFcnCallUsage`

Check model diagnostic settings that apply to function-call connectivity and that might impact model execution.

Description

Check for connectivity diagnostic settings that might lead to non-deterministic model execution.

Results and Recommended Actions

Condition	Recommended Action
Diagnostics > Connectivity > Invalid function-call connection is set to warning. This might lead to non-deterministic model execution.	Set Diagnostics > Connectivity > Invalid function-call connection to error.
Diagnostic > Connectivity > Context-dependent inputs is set to Disable All or Use local settings. This might lead to non-deterministic model execution.	Set Diagnostics > Connectivity > Context-dependent inputs to Enable all as errors.

See Also

Function-Call Subsystem

Check model for upgradable Simulink Scope blocks

Check model for Simulink Scope blocks that you can upgrade to Simulink Time Scope blocks

Description

In a future release, Simulink Scope blocks will be removed and replaced with Simulink Time Scope blocks.

Results and Recommended Actions

Condition	Recommended Action
Model does not have Simulink Scope blocks.	No action required.
Model includes at least one Simulink Scope block.	Select the Check model for upgradable Simulink Scope blocks check, click the Run This Check button, review the list of scope blocks, and then click the Upgrade button.

Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues

Check ID: `mathworks.design.DataStoreMemoryBlkIssue`

Look for modeling issues related to Data Store Memory blocks.

Description

Checks for multitasking data integrity, strong typing, and shadowing of data stores of higher scope.

Results and Recommended Actions

Condition	Recommended Action
The Duplicate data store names check is set to none or warning.	Consider setting the “Duplicate data store names” check to error in the Configuration Parameters dialog box, on the Diagnostics > Data Validity pane.
<p>The data store variable names are not strongly typed in one of the following:</p> <ul style="list-style-type: none"> • Signal Attributes pane of the Block Parameters dialog for the Date Store Memory block • Global data store name 	<p>Specify a data type other than auto by taking one of the following actions:</p> <ul style="list-style-type: none"> • Choose a data type other than <code>Inherit: auto</code> on the Signal Attributes pane of the Block Parameters dialog for the Date Store Memory block. • If you are using a global data store name, then specify its data type in the <code>Simulink.Signal</code> object.
The Multitask data store check is set to none or warning.	Consider setting the “Multitask data store” check to error in the Configuration Parameters dialog box, on the Diagnostics > Data Validity pane.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- “Local and Global Data Stores”
- “Storage Classes for Data Store Memory Blocks” (Simulink Coder)
- Data Store Memory
- Data Store Read
- Data Store Write
- “Duplicate data store names”
- “Multitask data store”
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check if read/write diagnostics are enabled for data store blocks

Check ID: `mathworks.design.DiagnosticDataStoreBlk`

For data store blocks in the model, enable the read-and-write diagnostics order checking to detect run-time issues.

Description

Check for the read-and-write diagnostics order checking. By enabling the read-and-write diagnostics, you detect potential run-time issues.

Results and Recommended Actions

Condition	Recommended Action
The Detect read before write check is disabled.	Consider enabling “Detect read before write” in the Configuration Parameter dialog box Diagnostics> Data Validity pane.
The Detect write after read check is disabled.	Consider enabling “Detect write after read” in the Configuration Parameter dialog box Diagnostics> Data Validity pane.
The Detect write after write check is disabled.	Consider enabling “Detect write after write” in the Configuration Parameter dialog box Diagnostics> Data Validity pane.

Capabilities and Limitations

Exclude blocks and charts from this check if you have a Simulink Check license.

Tips

.

- The run-time diagnostics can slow simulations down considerably. Once you have verified that Simulink does not generate warnings or errors during simulation, set them to `Disable all`.

See Also

- “Local and Global Data Stores”
- Data Store Memory
- Data Store Read
- Data Store Write
- “Detect read before write”
- “Detect write after read”
- “Detect write after write”
- “Check for potential ordering issues involving data store access” on page 8-23
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check data store block sample times for modeling errors

Check ID: `mathworks.design.DataStoreBlkSampleTime`

Identify modeling errors due to the sample times of data store blocks.

Description

Check data store blocks for continuous or fixed-in-minor-step sample times.

Results and Recommended Actions

Condition	Recommended Action
Data store blocks in your model have continuous or fixed-in-minor-step sample times.	Consider making the listed blocks discrete or replacing them with either Memory or Goto and From blocks.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- “Local and Global Data Stores”
- Data Store Memory

- Data Store Read
- Data Store Write
- “Fixed-in-Minor-Step”
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check for potential ordering issues involving data store access

Check ID: `mathworks.design.OrderingDataStoreAccess`

Look for read/write issues which may cause inaccuracies in the results.

Description

During an **Update Diagram**, identify potential issues relating to read-before-write, write-after-read, and write-after-write conditions for data store blocks.

Results and Recommended Actions

Condition	Recommended Action
Reading and writing (read-before-write or write-after-read condition) occur out of order.	Consider restructuring your model so that the Data Store Read block executes before the Data Store Write block.
Multiple writes occur within a single time step.	Change the model to write data only once per time step or refer to the following Tips section.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

Tips

This check performs a static analysis which might not identify every instance of improper usage. Specifically, Function-Call Subsystems, Stateflow Charts, MATLAB for code generation, For Iterator Subsystems, and For Each Subsystems can cause both missed detections and false positives. For a more comprehensive check, consider enabling the following diagnostics on the **Diagnostics > Data Validity** pane in the Configuration Parameters dialog box: “Detect read before write”, “Detect write after read”, and “Detect write after write”.

See Also

- “Local and Global Data Stores”
- Data Store Memory
- Data Store Read
- Data Store Write
- “Detect read before write”
- “Detect write after read”
- “Detect write after write”
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check structure parameter usage with bus signals

Check ID: `mathworks.design.MismatchedBusParams`

Identify blocks and `Simulink.Signal` objects that initialize bus signals by using mismatched structures.

Description

In a model, you can use a MATLAB structure to initialize a bus signal. For example, if you pass a bus signal through a Unit Delay block, you can set the **Initial condition** parameter to a structure. For basic information about initializing buses by using structures, see “Specify Initial Conditions for Bus Signals”.

Run this check to generate efficient and readable code by matching the shape and numeric data types of initial condition structures with those of bus signals. Matching these characteristics avoids unnecessary explicit typecasts and replaces field-by-field structure assignments with, for example, calls to `memcpy`.

Partial Structures

This check lists blocks and `Simulink.Signal` objects that initialize bus signals by using partial structures. During the iterative process of creating a model, you can use partial structures to focus on a subset of signal elements in a bus. For a mature model, use full structures to:

- Generate readable and efficient code.

- Support a modeling style that explicitly initializes unspecified signals. When you use partial structures, Simulink implicitly initializes unspecified signals.

For more information about full and partial structures, see “Create Full Structures for Initialization” and “Create Partial Structures for Initialization”.

Data Type Mismatches

This check lists blocks and `Simulink.Signal` objects whose initial condition structures introduce data type mismatches. The fields of these structures have numeric data types that do not match the data types of the corresponding bus signal elements.

When you configure an initial condition structure to appear as a tunable global structure in the generated code, avoid unnecessary explicit typecasts by matching the data types. See “Generate Tunable Initial Condition Structure for Bus Signal” (Simulink Coder).

Results and Recommended Actions

Condition	Recommended Action
Block or signal object uses partial structure	Consider using the function <code>Simulink.Bus.createMATLABStructure</code> to create a full initial condition structure.
Data types of structure fields do not match data types of corresponding signal elements	Consider defining the structure as a <code>Simulink.Parameter</code> object, and creating a <code>Simulink.Bus</code> object to use as the data type of the bus signal and of the parameter object. To control numeric data types, use the <code>Simulink.BusElement</code> objects in the bus object.

See Also

- “Specify Initial Conditions for Bus Signals”
- “Generate Tunable Initial Condition Structure for Bus Signal” (Simulink Coder)
- “Data Stores with Signal Objects”
- `Simulink.Bus.createMATLABStruct`
- `Simulink.Signal`

Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition

Check ID: `mathworks.design.ReplaceZOHDelayByRTB`

Identify Delay, Unit Delay, or Zero-Order Hold blocks that are used for rate transition. Replace these blocks with actual Rate Transition blocks.

Description

If a model uses Delay, Unit Delay, or Zero-Order Hold blocks to provide rate transition between input and output signals, Simulink makes a hidden replacement of these blocks with built-in Rate Transition blocks. In the compiled block diagram, a yellow symbol and the letters “RT” appear in the upper-left corner of a replacement block. This replacement can affect the behavior of the model, as follows:

- These blocks lose their algorithmic design properties to delay a signal or implement zero-order hold. Instead, they acquire rate transition behavior.
- This modeling technique works only in specific transition configurations (slow-to-fast for Delay and Unit Delay blocks, and fast-to-slow for Zero-Order Hold block). Set the block sample time to be equal to the slower rate (source for the Delay and Unit Delay blocks and destination for the Zero-Order Hold block).
- When the block sample time of a downstream or upstream block changes, these Delay, Unit Delay and Zero-Order Hold blocks might not perform rate transition. For example, setting the source and destination sample times equal stops rate transition. The blocks then assume their original algorithmic design properties.
- The block sample time shows incomplete information about sample time rates. The block code runs at two different rates to handle data transfer. However, the block sample time and sample time color show it as a single-rate block. Tools and MATLAB scripts that use sample time information base their behavior on this information.

An alternative is to replace Delay, Unit Delay, or Zero-Order Hold blocks with actual Rate Transition blocks.

- The technique ensures unambiguous results in block behavior. Delay, Unit Delay, or Zero-Order Hold blocks act according to their algorithmic design to delay and hold signals respectively. Only Rate Transition blocks perform actual rate transition.
- Using an actual Rate Transition block for rate transition offers a configurable solution to handle data transfer if you want to specify deterministic behavior or the type of memory buffers to implement.

Use this check to identify instances in your model where Delay, Unit Delay or Zero-Order Hold blocks undergo hidden replacement to provide rate transition between signals. Click **Upgrade Model** to replace these blocks with actual Rate Transition blocks.

Results and Recommended Actions

Condition	Recommended Action
Model has no instances of Delay, Unit Delay, or Zero-Order Hold blocks used for rate transition.	No action required.
Model has instances of Delay, Unit Delay, or Zero-Order Hold blocks used for rate transition.	<p>The check identifies these instances and allows you to upgrade the model.</p> <ol style="list-style-type: none"> 1 Click Upgrade Model to replace with actual Rate Transition blocks. 2 Save changes to your model.

If you do not choose to replace the Delay, Unit Delay, and/or Zero-Order Hold blocks with actual Rate Transition blocks, Simulink continues to perform a hidden replacement of these blocks with built-in rate transition blocks.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- “Run Model Checks”
- “Model Upgrades”
- Rate Transition
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check for calls to `slDataTypeAndScale`

Check ID: `mathworks.design.CallsSlDataTypeAndScale`

Identify calls to the internal function `slDataTypeAndScale`.

Description

In some previous versions of Simulink, opening a model that had been saved in an earlier version triggers an automatic upgrade to code for data type handling. The automatic upgrade inserts calls to the internal function `slDataTypeAndScale`. Although Simulink continues to support some uses of the function, if you eliminate calls to it, you get cleaner and faster code.

Simulink does not support calls to `slDataTypeAndScale` when:

- The first argument is a `Simulink.AliasType` object.
- The first argument is a `Simulink.NumericType` object with property `IsAlias` set to `true`.

Running **Check for calls to `slDataTypeAndScale`** identifies calls to `slDataTypeAndScale` that are required or recommended for replacement. In most cases, running the check and following the recommended action removes the calls. You can ignore calls that remain. Run the check unless you are sure there are not calls to `slDataTypeAndScale`.

Results and Recommended Actions

Condition	Recommended Action
Required Replacement Cases	Manually or automatically replace calls to <code>slDataTypeAndScale</code> . Cases listed require you to replace calls to <code>slDataTypeAndScale</code> .
Recommended Replacement Cases	For the listed cases, it is recommended that you manually or automatically replace calls to <code>slDataTypeAndScale</code> .
Manual Inspection Cases	Inspect each listed case to determine whether it should be manually upgraded.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

Tips

- Do not manually insert a call to `slDataTypeAndScale` into a model. The function was for internal use only.

- Running **Check for calls to slDataTypeAndScale** calls the Simulink function `slRemoveDataTypeAndScale`. Calling this function directly provides a wider range of conversion options. However, you very rarely need more conversion options.

See Also

- For more information about upgrading data types and scales, in the MATLAB Command Window, execute the following:
 - `help slDataTypeAndScale`
 - `help slRemoveDataTypeAndScale`
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check bus signals treated as vectors

Check ID: `mathworks.design.BusTreatedAsVector`

Identify bus signals that Simulink treats as vectors.

Description

You cannot use bus signals that the Simulink software implicitly converts to vectors. Instead, either insert a Bus to Vector conversion block between the bus signal and the block input port that it feeds, or use the `Simulink.BlockDiagram.addBusToVector` command.

Results and Recommended Actions

Condition	Recommended Action
Bus signals are implicitly converted to vectors.	Use <code>Simulink.BlockDiagram.addBusToVector</code> or insert a Bus to Vector block.
Model is not configured to identify bus signals that Simulink treats as vectors.	In the Configuration Parameters dialog box, on the Diagnostics > Connectivity pane, set Bus signal treated as vector to <code>error</code> .

Action Results

Clicking **Modify** inserts a Bus to Vector block at the input ports of blocks that implicitly convert bus signals to vectors.

Tips

- Run this check before running **Check consistency of initialization parameters for Output and Merge blocks**.
- For more information, see “Correct Buses Used as Vectors”.

See Also

- “Correct Buses Used as Vectors”
- Bus to Vector block
- “Bus signal treated as vector”
- “Migrating to Simplified Initialization Mode Overview” on page 8-5
- `Simulink.BlockDiagram.addBusToVector`

Check for potentially delayed function-call subsystem return values

Check ID: `mathworks.design.DelayedFcnCallSubsys`

Identify function-call return values that might be delayed because Simulink software inserted an implicit Signal Conversion block.

Description

So that signals reside in contiguous memory, Simulink software can automatically insert an implicit Signal Conversion block in front of function-call initiator block input ports. This can result in a one-step delay in returning signal values from calling function-call subsystems. The delay can be avoided by ensuring the signal originates from a signal block within the function-call system. Or, if the delay is acceptable, insert a Unit Delay block in front of the affected input ports.

Results and Recommended Actions

Condition	Recommended Action
The listed block input ports could have an implicit Signal Conversion block.	Decide if a one-step delay in returning signal values is acceptable for the listed signals. <ul style="list-style-type: none"> • If the delay is not acceptable, rework your model so that the input signal originates from within the calling subsystem. • If the delay is acceptable, insert a Unit Delay block in front of each listed input port.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

Signal Conversion block

Unit Delay block

“What Is a Model Advisor Exclusion?” (Simulink Check)

Identify block output signals with continuous sample time and non-floating point data type

Check ID: `mathworks.design.OutputSignalSampleTime`

Find continuous sample time, non-floating-point output signals.

Description

Non-floating-point signals might not represent continuous variables without loss of information.

Results and Recommended Actions

Condition	Recommended Action
Signals with continuous sample times have a non-floating-point data type.	On the identified signals, either change the sample time to be discrete or fixed-in-minor-step ([0 1]).

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

“What Is Sample Time?”.

“What Is a Model Advisor Exclusion?” (Simulink Check)

Check usage of Merge blocks

Check ID: `mathworks.design.MergeBlkUsage`

Identify Merge blocks with parameter settings that can lead to unexpected behavior, and help migrate your model to simplified initialization mode.

Note Run this check along with the other checks in the “Migrating to Simplified Initialization Mode Overview” on page 8-5.

Description

Simplified initialization mode was introduced in R2008b to improve the consistency of simulation results. For more information, see “Simplified Initialization Mode” and “Classic Initialization Mode”.

This Model Advisor check identifies settings in the Merge blocks in your model that can cause problems if you use classic initialization mode. It also recommends settings for consistent behavior of Merge blocks. The results of the subchecks contain two types of statements: Failed and Warning. Failed statements identify issues that you must address manually before you can migrate the model to the simplified initialization mode.

Warning statements identify issues or changes in behavior that can occur after migration.

Results and Recommended Actions

Condition	Recommended Action
Check the run-time diagnostic setting of the Merge block.	<ol style="list-style-type: none"> <li data-bbox="790 421 1338 557">1 In the Configuration Parameters dialog box, set “Detect multiple driving blocks executing at the same time step” to <code>error</code>. <li data-bbox="790 565 1338 666">2 Verify that the model simulates without errors before running this check again.
Check for Model blocks that are using the PIL simulation mode.	The simplified initialization mode does not support the Processor-in-the-loop (PIL) simulation for model references.
Check for library blocks with instances that cannot be migrated.	Examine the failed subcheck results for each block to determine the corrective actions.
Check for single-input Merge blocks.	<p data-bbox="790 890 1338 986">Replace both the Mux block used to produce the input signal and the Merge block with one multi-input Merge block.</p> <p data-bbox="790 1012 1338 1112">Single-input Merge blocks are not supported in the simplified initialization mode.</p>
Check for root Merge blocks that have an unspecified Initial output value.	<p data-bbox="790 1121 1338 1286">If you do not specify an explicit value for the Initial output parameter of root Merge blocks, then Simulink uses the default initial value of the output data type.</p> <p data-bbox="790 1312 1338 1472">A root Merge block is a Merge block with an output port that does not connect to another Merge block. For information on the default initial value, see “Initializing Signal Values”.</p>

Condition	Recommended Action
Check for Merge blocks with nonzero input port offsets.	<p>Clear the Allow unequal port widths parameter of the Merge block.</p> <hr/> <p>Note Consider using Merge blocks only for signal elements that require true merging. You can combine other elements with merged elements using the Concatenate block.</p>
Check for Merge blocks that have unconnected inputs or that have inputs from non-conditionally executed subsystems.	<p>Set the Number of inputs parameter of the Merge block to the number of Merge block inputs. You must connect each input to a signal.</p> <p>Verify that each Merge block input is driven by a conditionally executed subsystem. Merge blocks cannot be driven directly by an Iterator Subsystem or a block that is not a conditionally executed subsystem.</p>
Check for Merge blocks with inputs that are combined or reordered outside of conditionally executed subsystems.	Verify that combinations or reordering of Merge block input signals takes place within a conditionally executed subsystem. Such designs may use Mux, Bus Creator, or Selector blocks.
Check for Merge blocks with inconsistent input sample times.	<p>Verify that input signals to each Merge block have the same Sample time.</p> <p>Failure to do so could result in unpredictable behavior. Consequently, the simplified initialization mode does not allow inconsistent sample times.</p>
Check for Merge blocks with multiple input ports that are driven by a single source.	Verify that the Merge block does not have multiple input signals that are driven by the same conditionally executed subsystem or conditionally executed Model block.

Condition	Recommended Action
Check for Merge blocks that use signal objects to specify the Initial output value.	<p>Verify that the following behavior is acceptable.</p> <p>In the simplified initialization mode, signal objects cannot specify the Initial output parameter of the Merge block. While you can still initialize the output signal for a Merge block using a signal object, the initialization result may be overwritten by that of the Merge block.</p> <hr/> <p>Note Simulink generates a warning that the initial value of the signal object has been ignored.</p>

See Also

- “Migrating to Simplified Initialization Mode Overview” on page 8-5
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check usage of Outport blocks

Check ID: `mathworks.design.InitParamOutportMergeBlk`

Identify Outport blocks and conditional subsystems with parameter settings that can lead to unexpected behavior, and help migrate your model to simplified initialization mode.

Note Run this check along with the other checks in the “Migrating to Simplified Initialization Mode Overview” on page 8-5.

Description

Simplified initialization mode was introduced in R2008b to improve the consistency of simulation results. This mode is especially important for models that do not specify initial conditions for conditionally executed subsystem output ports. For more information, see “Simplified Initialization Mode” and “Classic Initialization Mode”.

This Model Advisor check identifies Outport blocks and conditional subsystems in your model that can cause problems if you use the simplified initialization mode. It also recommends settings for consistent behavior of Outport blocks. The results of the subchecks contain two types of statements: Failed and Warning. Failed statements identify issues that you must address manually before you can migrate the model to the simplified initialization mode. Warning statements identify issues or changes in behavior can occur after migration.

Results and Recommended Actions

Condition	Recommended Action
Check for blocks inside of the Iterator Subsystem that require elapsed time.	Within an Iterator Subsystem hierarchy, do not use blocks that require a service that maintains the time that has elapsed between two consecutive executions. Since an Iterator Subsystem can execute multiple times at a given time step, the concept of elapsed time is not well-defined between two such executions. Using these blocks inside of an Iterator Subsystem can cause unexpected behavior.

Condition	Recommended Action
Check for Outport blocks that have conflicting signal buffer requirements.	<p>The Outport block has a function-call trigger or function-call data dependency signal passing through it, along with standard data signals. Some of the standard data signals require an explicit signal buffer for the initialization of the output signal of the corresponding subsystem. However, buffering function-call related signals lead to a function-call data dependency violation.</p> <p>Consider modifying the model to pass function-call related signals through a separate Outport block. For examples of function-call data dependency violations, see the example model <code>sl_subsys_semantics</code>.</p> <p>A standard data signal may require an additional signal copy for one of the following reasons:</p> <ul style="list-style-type: none">• The Outport block is driven by a block with output that cannot be overwritten. The Ground block and the Constant block are examples of such blocks.• The Outport block shares the same signal source with another Outport block in the same subsystem or in one nested within the current subsystem but having a different initial output value.• The Outport block connects to the input of a Merge block• One of the input signals of the Outport block is specifying a <code>Simulink.Signal</code> object with an explicit initial value.

Condition	Recommended Action
Check for Outport blocks that are driven by a bus signal and whose Initial output value is not scalar.	For Outport blocks driven by bus signals, classic initialization mode does not support Initial Condition (IC) structures, while simplified initialization mode does. Hence, when migrating a model from classic to simplified mode, specify a scalar for the Initial Output parameter. After migration completes, to specify different initial values for different elements of the bus signal, use IC structures. For more information, see “Create Initial Condition Structures”.

Condition	Recommended Action
<p>Check for Outport blocks that require an explicit signal copy.</p>	<p>An explicit copy of the bus signal driving the Outport block is required for the initialization of the output signal of the corresponding subsystem.</p> <p>Insert a Signal Conversion block before the Outport block, then set the Output parameter of the Signal Conversion block to <code>Bus copy</code>.</p> <p>A standard data signal may require an additional signal copy for one or more of the following reasons:</p> <ul style="list-style-type: none"> • A block with output that cannot be overwritten is driving the Outport block. The Ground block and the Constant block are examples of such blocks. • The Outport block shares the same signal source with another Outport block in the same subsystem or in one nested within the current subsystem but having a different initial output value. • The Outport block connects to the input of a Merge block • One of the input signals of the Outport block is specifying a <code>Simulink.Signal</code> object with an explicit initial value.
<p>Check for merged Outport blocks that inherit the Initial Output value from Outport blocks that have been configured to reset when the blocks become disabled.</p>	<p>When Outport blocks are driving a Merge block, do not set their Output when disabled parameters to <code>reset</code>.</p>

Condition	Recommended Action
Check for merged Output blocks that are driven by nested conditionally executed subsystems.	Determine if the new behavior of the Output blocks is acceptable. If it is not acceptable, modify the model to account for the new behavior before migrating to the simplified initialization mode.
Check for merged Output blocks that reset when the blocks are disabled.	Set the Output when disabled parameter of the Output block to <code>held</code> . This setting is required because the Output block connects to a Merge block. For more information, see Output .
Check for Output blocks that have an undefined Initial output value with invalid initial condition sources.	Verify that the following behavior is acceptable. When the Initial output parameter is unspecified (<code>[]</code>), it inherits the initial output from the source blocks. If at least one of the sources of the Output block is not a valid source to inherit the initial value, the block uses the default initial value for that data type. For simplified initialization mode, valid sources an Output blocks can inherit the Initial output value from are: Constant, Initial Condition, Merge (with initial output), Stateflow chart, function-call model reference, or conditionally executed subsystem blocks.

Condition	Recommended Action
Check Outport blocks that have automatic rate transitions.	<p>Simulink has inserted a Rate Transition block at the input of the Outport block. Specify the Initial output parameter for each Outport block.</p> <p>Otherwise, perform the following procedure:</p> <ol style="list-style-type: none"> 1 In the Configuration Parameters dialog box, on the Solver pane, clear the option “Automatically handle rate transition for data transfer”. 2 Run this Model Advisor check again.
Check Outport blocks that have a special signal storage requirement and have an undefined Initial output value.	<p>Verify that the following behavior is acceptable.</p> <p>Specify the Initial output parameter for the Outport block. Set this value to [] (empty matrix) to use the default initial value of the output data type.</p>
Check the Initial output setting of Outport blocks that reset when they are disabled.	<p>Specify the Initial output parameter of the Outport block.</p> <p>You must specify the Initial output value for blocks that are configured to reset when they become disabled.</p>
Check the Initial output setting for Outport blocks that pass through a function-call data dependency signal.	<p>You cannot specify an Initial output value for the Outport block because function-call data dependency signals are passing through it. To set the Initial output value:</p> <ol style="list-style-type: none"> 1 Set the Initial output parameter of the Outport block to []. 2 Provide the initial value at the source of the data dependency signal rather than at the Outport block.

Condition	Recommended Action
Check for Outport blocks that use signal objects to specify the Initial output value.	<p>Verify that the following behavior is acceptable.</p> <p>In the simplified initialization mode, signal objects cannot specify the Initial output parameter of an Outport block. You can still initialize the input or output signals for an Outport block using signal objects, but the initialization results may be overwritten by those of the Outport block.</p> <hr/> <p>Note If you are working with a conditionally executed subsystem Outport block, Simulink generates a warning that the initial value of the signal object has been ignored.</p>
Check for library blocks with instances that have warnings.	Examine the warning subcheck results for each block before migrating to the simplified initialization mode.
Check for merged Outport blocks that are either unconnected or connected to a Ground block.	<p>Verify that the following behavior is acceptable.</p> <p>The Outport block is driving a Merge block, but its inputs are either unconnected or connected to Ground blocks. In the classic initialization mode, unconnected or grounded outports do not update the merge signal even when their parent conditionally executed subsystems are executing. In the simplified initialization mode, however, these outports will update the merge signal with a value of zero when their parent conditionally executed subsystems are executing.</p>

Condition	Recommended Action
<p>Check for Outport blocks that obtain the Initial output value from an input signal when they are migrated.</p>	<p>Verify that the following behavior is acceptable.</p> <p>The Initial output parameter of the Outport block is not specified. As a result, the simplified initialization mode will assume that the Initial output value for the Outport block is derived from the input signal. This assumption may result in different initialization behavior.</p> <p>If this behavior is not acceptable, modify your model before you migrate to the simplified initialization mode.</p>
<p>Check for outer Outport blocks that have an explicit Initial output.</p>	<p>Verify that the following behavior is acceptable.</p> <p>In classic initialization mode, the Initial output and Output when disabled parameters of the Outport block must match those of their source Outport blocks.</p> <p>In simplified initialization mode, Simulink sets the Initial output parameter of outer Outport blocks to [] (empty matrix) and Output when disabled parameter to held.</p>
<p>Check for conditionally executed subsystems that propagate execution context across the output boundary.</p>	<p>Verify that the following behavior is acceptable.</p> <p>The Propagate execution context across subsystem boundary parameter is selected for the subsystem. Execution context will still be propagated across input boundaries; however, the propagation will be disabled on the output side for the initialization in the simplified initialization mode.</p>

Condition	Recommended Action
<p>Check for blocks that read input from conditionally executed subsystems during initialization.</p>	<p>Verify that the following behavior is acceptable.</p> <p>Some blocks, such as the Discrete-Time Integrator block, read their inputs from conditionally executed subsystems during initialization in the classic initialization mode. Simulink performs this step as an optimization technique.</p> <p>This optimization is not allowed in the simplified initialization mode because the output of a conditionally executed subsystem at the first time step after initialization may be different than the initial value declared in the corresponding Outport block. In particular, this discrepancy occurs if the subsystem is active at the first time step.</p>
<p>Check for a migration conflict for Outport blocks that use a <code>Dialog</code> as the Source of initial output value.</p>	<p>Other instances of Outport blocks with the same library link either cannot be migrated or are being migrated in a different manner. Review the results from the Check for library blocks with instances that cannot be migrated to learn about the different migration paths for other instances of each Outport block.</p> <p>The Outport block will maintain its current settings and use its specified Initial output value.</p>

Condition	Recommended Action
<p>Check for a migration conflict for Outport blocks that use <code>Input</code> signal as the Source of initial output value.</p>	<p>Other instances of Outport blocks with the same library link either cannot be migrated or are being migrated in a different manner. Review the results from the Check for library blocks with instances that cannot be migrated to learn about the different migration paths for other instances of each Outport block.</p> <p>The Outport block currently specifies an Initial output of <code>[]</code> (empty matrix), and the Output when disabled as <code>held</code>. This means that each outport does not perform initialization, but implicitly relies on source blocks to initialize its input signal.</p> <p>After migration, the parameter Source of initial output value will be set to <code>Input</code> signal to reflect this behavior.</p>
<p>Check for a migration conflict for Outport blocks that have <code>SimEvents@</code> semantics.</p>	<p>Other instances of Outport blocks with the same library link either cannot be migrated or are being migrated in a different manner. Review the results from the Check for library blocks with instances that cannot be migrated to learn about the different migration paths for other instances of each Outport block.</p> <p>The Outport blocks will continue to use an Initial output value of <code>[]</code> (empty matrix) and an Output when disabled setting of <code>held</code>. Simulink will maintain these settings because their parent conditionally executed subsystems are connected to <code>SimEvents</code> blocks.</p>

Condition	Recommended Action
<p>Check for a migration conflict for innermost Outport blocks with variable-size input and unspecified Initial output.</p>	<p>For these Outport blocks, the signal size varies only when the parent subsystem of the block is re-enabled. Therefore, Simulink implicitly assumes that the Initial output parameter is equal to 0, even though the parameter is unspecified, []. Consequently, unless you specify the parameter, the Model Advisor will explicitly set the parameter to 0 when the model is migrated to the simplified initialization mode.</p> <p>Other instances of Outport blocks with the same library link either cannot be migrated or are being migrated in a different manner. Review the results from the Check for library blocks with instances that cannot be migrated to learn about the different migration paths for other instances of each Outport block.</p>
<p>Check for a migration conflict for Outport blocks that use a default ground value as the Initial output.</p>	<p>The parameter Initial output is set to [] (empty matrix) and the source of the Outport is an invalid initial condition source. Thus, the block uses the default initial value as the initial output in the simplified initialization mode. Other instances of Outport blocks with the same library link either have errors or are being migrated differently.</p>

Condition	Recommended Action
Check for a migration conflict for merged Outport blocks without explicit specification of Initial output .	Review the results from the subcheck Check for library blocks with instances that cannot be migrated to learn about different migration paths for other instances of each Outport block. For the remaining Outport blocks, Initial output is set to [] (empty matrix) and Output when disabled is set to held respectively, in simplified initialization mode.

See Also

- “Migrating to Simplified Initialization Mode Overview” on page 8-5
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check usage of Discrete-Time Integrator blocks

Check ID: `mathworks.design.DiscreteBlock`

Identify Discrete-Time Integrator blocks with parameter settings that can lead to unexpected behavior, and help migrate your model to simplified initialization mode.

Note Run this check along with the other checks in the “Migrating to Simplified Initialization Mode Overview” on page 8-5.

Description

Simplified initialization mode was introduced in R2008b to improve the consistency of simulation results. For more information, see “Simplified Initialization Mode” and “Classic Initialization Mode”.

This Model Advisor check identifies settings in Discrete-Time Integrator blocks in your model that can cause problems if you use the simplified initialization mode. It also recommends settings for consistent behavior of Discrete-Time Integrator blocks. The results of the subchecks contain two types of statements: Failed and Warning. Failed statements identify issues that you must address manually before you can migrate the

model to the simplified initialization mode. Warning statements identify issues or changes in behavior that can occur after migration.

Results and Recommended Actions

Condition	Recommended Action
Check for Discrete-Time Integrator blocks whose parameter Initial condition setting is set to Output.	Determine if the new behavior of the Discrete-Time Integrator blocks is acceptable. If it is not acceptable, modify the model to account for the new behavior before migrating to the simplified initialization mode.
Check for Discrete-Time Integrator blocks whose Initial condition setting parameter is set to State (most efficient) and are in a subsystem that uses triggered sample time.	Use periodic sample time for the block, or set Initial Condition setting to Output.
Check for blocks inside of the Iterator Subsystem that require elapsed time.	<p>Within an Iterator Subsystem hierarchy, do not use blocks that require a service that maintains the time that has elapsed between two consecutive executions.</p> <p>Since an Iterator Subsystem can execute multiple times at a given time step, the concept of elapsed time is not well-defined between two such executions. Using these blocks inside of an Iterator Subsystem can cause unexpected behavior.</p>

See Also

- “Migrating to Simplified Initialization Mode Overview” on page 8-5
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check model settings for migration to simplified initialization mode

Note Do not run this check in isolation. Run this check along with the other checks in the “Migrating to Simplified Initialization Mode Overview” on page 8-5.

Check ID: `mathworks.design.ModelLevelMessages`

Identify settings in Model blocks and model configuration parameters that can lead to unexpected behavior, and help migrate your model to simplified initialization mode.

Description

Simplified initialization mode was introduced in R2008b to improve consistency of simulation results. For more information, see “Simplified Initialization Mode” and “Classic Initialization Mode”.

This Model Advisor check identifies issues in the model configuration parameters and Model blocks in your model that can cause problems when you migrate to simplified initialization mode. The results of the subchecks contain two types of statements: Failed and Warning. Failed statements identify issues that you must address manually before you can migrate the model to simplified initialization mode. Warning statements identify issues or changes in behavior that can occur after migration.

After running this Model Advisor consistency check, if you click **Explore Result** button, the messages pertain only to blocks that are not library-links.

Note Because it is difficult to undo these changes, select **File > Save Restore Point As** to back up your model before migrating to the simplified initialization mode.

For more information, see “Model Configuration Parameters: Connectivity Diagnostics”.

Results and Recommended Actions

Condition	Recommended Action
Verify that all Model blocks are using the simplified initialization mode.	Migrate the model referenced by the Model block to the simplified initialization mode, then migrate the top model.
Verify simplified initialization mode setting	Set Configuration Parameters > Underspecified initialization detection to Simplified.

Action Results

Clicking **Modify Settings** causes the following:

- The Model parameter is set to `simplified`
- If an Outport block has the **Initial output** parameter set to the empty character vector, `[]`, then the `SourceOfInitialOutputValue` parameter is set to **Input signal**.
- If an Outport has an empty **Initial output** and a variable-size signal, then the **Initial output** is set to zero.

See Also

- “Migrating to Simplified Initialization Mode Overview” on page 8-5
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check S-functions in the model

Check ID: `mathworks.design.SFuncAnalyzer`

Perform quality checks on S-functions in Simulink models or subsystems.

Description

The S-function analyzer performs quality checks on S-functions to identify improvements and potential problems in the specified model.

Results and Recommended Actions

Condition	Recommended Action
Continuous states are modified in <code>mdlOutputs</code> method.	Modify Continuous States at a major time step and use <code>ssSetSolverNeedsReset</code> function in S-function code.
Contionuous states are modified in the <code>mdlUpdate</code> method.	Modify Continous States only at a major time step and use <code>ssSetSolverNeedsReset</code> function in S-function code.
S-function discrete states are modified in the <code>mdlOutputs</code> at a minor step.	Modify the discrete states only at a major step guarded by <code>ssIsMajorTimeStep</code> function.

Condition	Recommended Action
S-function mode vector is modified in the <code>mdlOutputs</code> method at a minor step.	Modify the mode vector only at a major step guarded by <code>ssIsMajorTimeStep</code> function.
S-function is using static or global variables to declare internal states.	Declare the states explicitly using <code>ssSetNumDiscStates</code> function or “Model Global Data by Creating Data Stores”.
S-function has continuous states but sample time is not declared continuous.	Specify continuous sample time using <code>ssSetSampleTime</code> function.
S-function has discrete states but the <code>mdlOutputs</code> and <code>mdlUpdate</code> methods are combined.	Define the <code>mdlOutputs</code> and <code>mdlUpdate</code> methods separately and modify discrete states only in <code>mdlUpdate</code> method.
S-function sets the <code>SS_OPTION_CAN_BE_CALLED_CONDITIONALLY</code> option when having state-like data or multiple sample times.	Remove the options when the S-function has state-like data or multiple sample times.
MEX compilers do not exist on the machine.	Check for the presence or install MEX compilers on the machine.
S-function encounters errors while compiling the model.	Check the Diagnostics Viewer output and recompile the model.

Check for non-continuous signals driving derivative ports

Check ID: `mathworks.design.NonContSigDerivPort`

Identify noncontinuous signals that drive derivative ports.

Description

Noncontinuous signals that drive derivative ports cause the solver to reset every time the signal changes value, which slows down simulation.

Results and Recommended Actions

Condition	Recommended Action
There are noncontinuous signals in the model driving derivative ports.	<ul style="list-style-type: none"> • Make the specified signals continuous. • Replace the continuous blocks receiving these signals with discrete state versions of the blocks.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- “Modeling Dynamic Systems”
- “Simulation Phases in Dynamic Systems”
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Runtime diagnostics for S-functions

Check ID: `mathworks.design.DiagnosticSFcn`

Check array bounds and solver consistency if S-Function blocks are in the model.

Description

Validates whether S-Function blocks adhere to the ODE solver consistency rules that Simulink applies to its built-in blocks.

Results and Recommended Actions

Condition	Recommended Action
Solver data inconsistency is set to none.	In the Configuration Parameters dialog box, set Solver data inconsistency to warning or error.
Array bounds exceeded is set to none.	In the Configuration Parameters dialog box, set Array bounds exceeded to warning or error

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- “What Is an S-Function?”
- “How S-Functions Work”
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check model for foreign characters

Check ID: `mathworks.design.characterEncoding`

Check for characters that are incompatible with the current encoding

Description

Check for characters in the model file that cannot be represented in the current encoding. These can cause errors during simulation, and may be corrupted when saving the model.

Results and Recommended Actions

Condition	Recommended Action
Incompatible characters found	Change the current encoding to the encoding specified in the model file, using <code>slCharacterEncoding</code> . To change the current encoding you need to close the models, and this closes the Model Advisor.

Tips

The Upgrade Advisor report shows the encoding you need, or you can retrieve the encoding from the model using the command:

```
get_param(modelname, 'SavedCharacterEncoding')
```

Use `slCharacterEncoding` to change the encoding. This setting applies to the current MATLAB session, so if you restart MATLAB and want to open the same model, you will need to make the same change to the current encoding again.

For more information see:

- `slCharacterEncoding`
- “Open a Model with Different Character Encoding”
- “Save Models with Different Character Encodings”

See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Identify unit mismatches in the model

Check ID: `mathworks.design.UnitMismatches`

Identify instances of unit mismatches between ports in the model.

Description

Check for instances of unit mismatches between ports in the model.

Results and Recommended Actions

Condition	Recommended Action
Unit mismatches found	Change one of the mismatched unit settings to match the unit settings for the other port.

See Also

- “Unit Specification in Simulink Models”.

Identify automatic unit conversions in the model

Check ID: `mathworks.design.AutoUnitConversions`

Identify instances of automatic unit conversions in the model.

Description

Identify instances of automatic unit conversions in the model.

Results and Recommended Actions

Condition	Recommended Action
Automatic unit conversions found	Check that the converted units are expected for the model.

See Also

- “Unit Specification in Simulink Models”.

Identify disallowed unit systems in the model

Check ID: `mathworks.design.DisallowedUnitSystems`

Identify instances of disallowed unit systems in the model.

Description

Identify instances of disallowed unit systems in the model.

Results and Recommended Actions

Condition	Recommended Action
Disallowed unit systems found	Either choose a unit that conforms to the configured unit system, or select another unit system. For more information, see “Restricting Unit Systems”.

See Also

- “Unit Specification in Simulink Models”.

Identify undefined units in the model

Check ID: `mathworks.design.UndefinedUnits`

Identify instances of unit specifications, not defined in the unit database, in the model.

Description

Identify instances of unit specifications, not defined in the unit database, in the model.

Results and Recommended Actions

Condition	Recommended Action
Undefined units found	Change the unit to one that Simulink supports.

See Also

- “Unit Specification in Simulink Models”.
- Allowed Units

Check model for block upgrade issues

Check ID: `mathworks.design.Update`

Check for common block upgrade issues.

Description

Check blocks in the model for compatibility issues resulting from using a new version of Simulink software.

Results and Recommended Actions

Condition	Recommended Action
Blocks with compatibility issues found.	Click Modify to fix the detected block issues.
Check update status for the Level 2 API S-functions.	Consider replacing Level 1 S-functions with Level 2.

Action Results

Clicking **Modify** replaces blocks from a previous release of Simulink software with the latest versions.

See Also

- “Write Level-2 MATLAB S-Functions”.
- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Check model for block upgrade issues requiring compile time information

Check ID: `mathworks.design.UpdateRequireCompile`

Check for common block upgrade issues.

Description

Check blocks for compatibility issues resulting from upgrading to a new version of Simulink software. Some block upgrades require the collection of information or data when the model is in the compile mode. For this check, the model is set to compiled mode and then checked for upgrades.

Results and Recommended Actions

Condition	Recommended Action
Model contains Lookup Table or Lookup Table (2-D) blocks and some of the blocks specify Use Input Nearest or Use Input Above for a lookup method.	Replace Lookup Table blocks and Lookup Table (2-D) blocks with n-D Lookup Table blocks. Do not apply Use Input Nearest or Use Input Above for lookup methods; select another option.
Model contains Lookup Table or Lookup Table (2-D) blocks and some blocks perform multiplication first during interpolation.	Replace Lookup Table blocks and Lookup Table (2-D) blocks with n-D Lookup Table blocks. However, because the n-D Lookup Table block performs division first, this replacement might cause a numerical difference in the result.
Model contains Lookup Table or Lookup Table (2-D) blocks. Some of these blocks specify Interpolation-Extrapolation as the Lookup method but their input and output are not the same floating-point type.	Replace Lookup Table blocks and Lookup Table (2-D) blocks with n-D Lookup Table blocks. Then change the extrapolation method or the port data types for block replacement.
Model contains Unit Delay blocks with Sample time set to -1 that inherit a continuous sample time.	Replace Unit Delay blocks with Memory blocks.

Action Results

Clicking **Modify** replaces blocks from a previous release of Simulink software with the latest versions.

See Also

- n-D Lookup Table
- Unit Delay
- “Consult the Upgrade Advisor”
- “Model Upgrades”

Check that the model is saved in SLX format

Check ID: `mathworks.design.UseSLXFile`

Check that the model is saved in SLX format.

Description

Check whether the model is saved in SLX format.

Results and Recommended Actions

Condition	Recommended Action
Model not saved in SLX format	Consider upgrading to the SLX file format to use the latest features in Simulink.

Capabilities and Limitations

You can run this check on your library models.

Tips

Simulink Projects can help you upgrade models to SLX format and preserve file revision history in source control. See “Upgrade Model Files to SLX and Preserve Revision History”.

See Also

- “Save Models in the SLX File Format”
- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Check model for SB2SL blocks

Check ID: `mathworks.simulink.SB2SL.Check`

Check that the model does not have outdated SB2SL blocks.

Description

Check if the model contains outdated SB2SL blocks.

Results and Recommended Actions

Condition	Recommended Action
Model contains outdated SB2SL blocks	Consider upgrading the model to current SB2SL blocks.

Action Results

Clicking **Update SB2SL Blocks** replaces blocks with the latest versions.

See Also

- “Consult the Upgrade Advisor”.

Check Model History properties

Check ID: `mathworks.design.SLXModelProperties`

Check for edited model history properties

Description

Check models for edited Model History property values that could be used with source control tool keyword substitution. This keyword substitution is incompatible with SLX file format.

In the MDL file format you can configure some model properties to make use of source control tool keyword substitution. If you save your model in SLX format, source control tools cannot perform keyword substitution. Information in the model file from such keyword substitution is cached when you first save the MDL file as SLX, and is not updated again. The Model Properties History pane and Model Info blocks in your model show stale information from then on.

Results and Recommended Actions

Condition	Recommended Action
Edited model history properties	Manually or automatically reset the properties to the default values. Click the button to reset, or to inspect and change these properties manually, open the Model Properties dialog and look in the History pane.

Capabilities and Limitations

You can run this check on your library models.

See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Identify Model Info blocks that can interact with external source control tools

Check ID: `mathworks.design.ModelInfoKeywordSubstitution`

Use this check to find Model Info blocks that can be altered by external source control tools through keyword substitution.

Description

This check searches for character vectors in the Model Info block enclosed within dollar signs that can be overwritten by an external source control tool. Using third-party source control tool keyword expansion might corrupt your model files when you submit them. Keyword substitution is not available in SLX model file format.

For a more flexible interface to source control tools, use a Simulink project instead of the Model Info block. See “About Source Control with Projects”.

Results and Recommended Actions

Condition	Recommended Action
The Model Info block contains fields like this: \$keyword\$	Review the list of fields in the report, then remove the keyword character vectors from the Model Info block.

See Also

- “Consult the Upgrade Advisor”.
- “About Source Control with Projects”

Identify Model Info blocks that use the Configuration Manager

Check ID: `mathworks.design.ModelInfoConfigurationManager`

Use this check to find Model Info blocks that use the Configuration Manager.

Description

Model Info blocks using the Configuration Manager allow risky keyword substitution using external source control tools. Using third-party source control tool keyword expansion might corrupt your model files when you submit them. Keyword substitution is not available in SLX model file format. The Configuration Manager for the Model Info block will be removed in a future release.

For a more flexible interface to source control tools, use a Simulink project instead of the Model Info block. See “About Source Control with Projects”.

Results and Recommended Actions

Condition	Recommended Action
A Model Info block is using the Configuration Manager.	Click Remove the Configuration Manager .

See Also

- “Consult the Upgrade Advisor”.
- “About Source Control with Projects”

Check model for legacy 3DoF or 6DoF blocks

Check ID: `mathworks.design.Aeroblks.CheckDOF`

Lists 3DoF and 6DoF blocks are outdated.

Description

This check searches for 3DoF and 6DoF blocks from library versions prior to 3.13 (R2014a).

Results and Recommended Actions

Condition	Recommended Action
Blocks configured with old versions of 3DoF or 6DoF blocks found.	Click Replace 3DoF and 6DoF Blocks to replace the blocks with latest versions.

Action Results

Clicking **Replace 3DoF and 6DoF Blocks** replaces blocks with the latest versions.

See Also

- “Equations of Motion” (Aerospace Blockset)

Check model and local libraries for legacy Aerospace Blockset blocks

Check ID: `mathworks.design.Aeroblks.CheckFG`

Lists blocks configured to use FlightGear versions that are outdated or not supported.

Description

This check searches and lists blocks configured to use FlightGear versions that are outdated or not supported.

Results and Recommended Actions

Condition	Recommended Action
Blocks configured with old versions of FlightGear are found.	Click Update FlightGear blocks to change block settings to latest supported version of FlightGear. Then, download latest version of FlightGear that MATLAB supports.

Action Results

Clicking **Update FlightGear blocks** changes block settings to the latest supported version of FlightGear.

See Also

- “Flight Simulator Interfaces” (Aerospace Blockset)

Check and update masked blocks in library to use promoted parameters

Check ID: `mathworks.design.CheckAndUpdateOldMaskedBuiltinBlocks`

Check for libraries that should be updated to use promoted parameters.

Description

This check searches libraries created before R2011b for masked blocks that should be updated to use promoted parameters. Since R2011b, if a block parameter is not promoted, its value in the linked block is locked to its value in the library block. This check excludes blocks of type Subsystem, Model reference, S-Function and M-S-Function.

Results and Recommended Actions

Condition	Recommended Action
Libraries that need to be updated are found	Click Update . Once the libraries have been updated, run the check again

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check and update mask image display commands with unnecessary imread() function calls

Check ID: `mathworks.design.CheckMaskDisplayImageFormat`

Check identifies masks using image display commands with unnecessary calls to the `imread()` function.

Description

This check searches for the mask display commands that make unnecessary calls to the `imread()` function, and updates them with mask display commands that do not call the `imread()` function. Since 2013a, a performance and memory optimization is available for mask images specified using the image path instead of the RGB triple matrix.

Results and Recommended Actions

Condition	Recommended Action
Mask display commands that make unnecessary calls to the <code>imread()</code> function are found.	Click Update . Once the blocks have been updated, run the check again.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check and update mask to affirm icon drawing commands dependency on mask workspace

Check ID: `mathworks.design.CheckMaskRunInitFlag`

Check identifies if the mask icon drawing commands have dependency on the mask workspace.

Description

This check identifies if the mask icon drawing commands have dependency on the mask workspace and updates the `RunInitForIconRedraw` property accordingly. If there is no mask workspace dependency, the value of `RunInitForIconRedraw` is set to `off`, whereas, if there is mask workspace dependency the values is set to `on`.

Setting the values of `RunInitForIconRedraw` to `off` when there is no mask workspace dependency optimizes the performance by not executing the mask initialization code before drawing the block icon.

Results and Recommended Actions

Condition	Recommended Action
Mask drawing commands that are dependent or independent of the mask workspace are found.	Click Update . Once the blocks have been updated, run the check again.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Identify masked blocks that specify tabs in mask dialog using MaskTabNames parameter

Check ID: `mathworks.design.CheckAndUpdateOldMaskTabnames`

This check identifies masked blocks that specify tabs in mask dialog using the `MaskTabNames` parameter.

Description

This check identifies masked blocks that use the `MaskTabNames` parameter to programmatically create tabs in the mask dialog. Since R2013b, dialog controls are used to group parameters in a tab on the mask dialog.

Results and Recommended Actions

Condition	Recommended Action
Masked blocks that use the <code>MaskTabNames</code> parameter to create tabs programmatically in the mask dialog are found.	Click Upgrade available in the Action section. Once the blocks have been updated, run the check again.

Capabilities and Limitations

You can run this check on your library models.

See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Identify questionable operations for strict single-precision design

Check ID: `mathworks.design.StowawayDoubles`

For a strict single-precision design, this check identifies the blocks that introduce double-precision operations, and non-optimal model settings.

Description

For a strict single-precision design, this check identifies the blocks that introduce double-precision operations, and non-optimal model settings.

Results and Recommended Actions

Condition	Recommended Action
Double-precision floating-point operations found in model.	Verify that: <ul style="list-style-type: none"> • Block input and output data types are set correctly. • In the Configuration Parameters dialog box, on the Optimization pane, Default for underspecified data type is set to <code>single</code>.
Model uses a library standard that is not optimal for strict-single designs.	Verify that: <ul style="list-style-type: none"> • All target-specific math libraries used by the model support single-precision implementations. <p style="text-align: center;">Set Configuration Parameters > Standard math library to C99 (ISO).</p>
Logic signals are not implemented as Boolean data.	Verify that: <ul style="list-style-type: none"> • In the Configuration Parameters dialog box, Implement logic signals as Boolean data is selected.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- “Validate a Floating-Point Embedded Model”
- “Consult the Upgrade Advisor”.
- “Model Upgrades”
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check `get_param` calls for block `CompiledSampleTime`

Check ID: `mathworks.design.CallsGetParamCompiledSampleTime`

Use this check to identify MATLAB files in your working environment that contain `get_param` function calls to return the block `CompiledSampleTime` parameter.

Description

For multi-rate blocks (including subsystems), Simulink returns the block compiled sample time as a cell array of the sample rates in the block. The return value is a cell array of pairs of doubles. MATLAB code that accepts this return value only as pairs of doubles can return an error when called with a multi-rate block. Use this check to identify such code in your environment. Modify these instances of code to accept a cell array of pairs of doubles instead.

For example, consider a variable `blkTs`, which has been assigned the compiled sample time of a multi-rate block.

```
blkTs = get_param(block, 'CompiledSampleTime');
```

Here are some examples in which the original code works only if `blkTs` is a pair of doubles and the block is a single-rate block:

- Example 1

```
if isinf(blkTs(1))
    disp('found constant sample time')
end
```

Since `blkTs` is now a cell array, Simulink gives this error message:

```
Undefined function 'isinf' for input arguments of type 'cell'
```

Instead, use this code, for which `blkTs` can be a cell array or a pair of doubles.

```

if isequal(blkTs, [inf,0])
    disp('found constant sample time')
end

```

- **Example 2**

```

if all(blkTs == [-1,-1])
    disp('found triggered sample time')
end

```

For the above example, since blkTs is now a cell array, Simulink gives this error:

```
Undefined function 'eq' for input arguments of type 'cell'
```

Instead, use this code, for which blkTs can be a cell array or a pair of doubles.

```

if isequal(blkTs, [-1,-1])
    disp('found triggered sample time')
end

```

- **Example 3**

```

if (blkTs(1) == -1)
    disp('found a triggered context')
end

```

Again, since blkTs is now a cell array, Simulink gives this error:

```
Undefined function 'eq' for input arguments of type 'cell'
```

Instead, use this code.

```

if ~iscell(blkTs)
    blkTs = {blkTs};
end
for idx = 1:length(blkTs)
    thisTs = blkTs{idx};
    if (thisTs(1) == -1)
        disp('found a triggered context')
    end
end
end

```

The above code checks for a triggered type sample time (triggered or async). In cases in which a block has constant sample time ([inf,0]) in addition to triggered or async or when a block has multiple async rates, this alternative property detects the triggered type sample time.

This check scans MATLAB files in your environment. If the check finds instances of MATLAB code that contain `get_param` calls to output the block compiled sample time, Upgrade Advisor displays these results. It suggests that you modify code that accepts the block compiled sample time from multi-rate blocks.

Results and Recommended Actions

Condition	Recommended Action
No MATLAB files call <code>get_param(block,CompiledSampleTime)</code>	None
Some MATLAB files call <code>get_param(block,CompiledSampleTime)</code>	If files use the block <code>CompiledSampleTime</code> parameter from multi-rate blocks, modify these files to accept the parameter as a cell array of pairs of doubles.

See Also

- “Sample Times in Subsystems”
- “Block Compiled Sample Time”

Check model for parameter initialization and tuning issues

Check ID: `mathworks.design.ParameterTuning`

Use this check to identify issues in the model that occur when you initialize parameters or tune them.

Description

This check scans your model for parameter initialization and tuning issues like:

- Rate mismatch between blocks
- Divide by zero issue in conditionally executed subsystems
- Invalid control port value in Index Vector blocks

Results and Recommended Actions

Condition	Recommended Action
The model has rate transition issues.	Select Automatically handle rate transition for data transfer in the Solver pane of the model configuration parameters.
The model has a divide by zero issue in a conditionally executed subsystem with a control port.	At the command prompt, run <code>set_param(control_port, 'DisallowConstTsAndPrmTs')</code>
The model has an invalid control port value in a conditionally executed subsystem.	At the command prompt, run <code>set_param(control_port, 'DisallowConstTsAndPrmTs')</code>

Action Results

Select **Upgrade model** to resolve issues in the model related to parameter initialization and tuning.

See Also

- “Automatic Rate Transition”

Check for virtual bus across model reference boundaries

Check ID: `mathworks.design.CheckVirtualBusAcrossModelReference`

Check virtual bus signals that cross model reference boundaries.

Description

This check identifies root-level Inport and Outport blocks in referenced models and Model blocks with virtual bus outputs that require updates to change to nonvirtual bus signals.

If the check identifies issues, click **Update Model** to convert root-level Inport and Outport blocks configured for virtual buses to use nonvirtual buses in these situations:

- For root-level Inport blocks — Enable the **Output as nonvirtual bus** parameter and insert a Signal Conversion block after the Inport block. The Signal Conversion block is configured to output a virtual bus.

- For root-level Outputport blocks — Enable the **Output as nonvirtual bus in parent model** parameter.
- For Model blocks — For ports whose Outputport blocks were updated to address issues, insert a Signal Conversion block after the corresponding ports of the Model block. The Signal Conversion block is configured to output a virtual bus.

Recommended Action and Results

To resolve issues, click **Upgrade Model**.

Note Run the **Analyze model hierarchy and continue upgrade sequence** check on the top-level model and then down through the model reference hierarchy.

Clicking **Upgrade Model** converts affected root-level Inport and Outputport blocks configured for virtual buses to use nonvirtual buses in models where you:

- Use function prototype control
- Perform C++ code generation with the `I/O arguments step` method option.

Alternatively, you can change the C++ code generation function specification setting to `Default step method`:

- 1 In the **Configuration Parameters > Code Generation > Interface** pane, click **Configure C++ Class Interface**.
 - 2 In the dialog box, set the **Function specification** parameter to `Default step method`.
- Use buses that have variable-dimension signals
 - Use an associated non-auto storage class for Outputport block signals
 - The conversion for non-auto storage class occurs only if you have the target generation license that the model requires. For example, an ERT target requires an Embedded Coder license.
 - Use Export-function models where an Outputport block is driven by a nonvirtual bus
 - Have Model blocks that reference models containing Outputport blocks that have been fixed — Clicking **Upgrade Model** updates Model blocks referencing the models that had Outputport blocks fixed by the **Analyze model hierarchy and continue upgrade sequence** check.

See Also

- “Bus Data Crossing Model Reference Boundaries”

Check model for custom library blocks that rely on frame status of the signal

Check ID: `mathworks.design.DSPFrameUpgrade`

This check identifies custom library blocks in the model that depend on the frame status of the signal.

Description

This check searches for the custom library blocks in a model that depend on the frame status of the signal. The check analyzes the blocks, recommends fixes, and gives reasons for the fixes. You must make the fixes manually.

Results and Recommended Actions

Condition	Recommended Action
The check finds custom library blocks that depend on the frame status of the signal.	Follow the recommendation given by the Upgrade Advisor.

Capabilities and Limitations

You can run this check only on custom library blocks in your model.

You must make the fixes manually.

This check appears only if you have the DSP System Toolbox installed.

See Also

“Frame-based processing” (DSP System Toolbox)

Check Rapid accelerator signal logging

Check ID: `mathworks.design.CheckRapidAcceleratorSignalLogging`

When simulating your model in rapid accelerator mode, use this check to find signals logged in your model that are globally disabled. Rapid accelerator mode supports signal logging. Use this check to enable signal logging globally.

Description

This check scans your model to see if a simulation is in rapid accelerator mode and whether the model contains signals with signal logging. If the check finds an instance and signal logging is globally disabled, an option to turn on signal logging globally appears.

Results and Recommended Actions

Condition	Recommended Action
Simulation mode is not rapid accelerator.	None You can enable signal logging in rapid accelerator mode.
Simulation mode is rapid accelerator. Upgrade Advisor did not find signals with signal logging enabled.	NoneThe model does not use signal logging. Enable signal logging for signals and globally if you want to log signals.
Simulation mode is rapid accelerator. Upgrade Advisor found signals with signal logging enabled. However, global setting for signal logging was disabled.	Enable signal logging globally if you want to log signals with signal logging enabled.
Signal logging was already globally enabled.	None

Action Results

Selecting **Modify** enables signal logging globally in your model.

See Also

- “Signal Logging in Rapid Accelerator Mode”
- “Consult the Upgrade Advisor”.

Check virtual bus inputs to blocks

Check ID: `mathworks.design.VirtualBusUsage`

Check bus input signals for a set of blocks.

Description

Check bus input signals for a set of blocks.

Starting in R2015b, virtual bus signal inputs to blocks that require nonbus or nonvirtual bus input can cause an error. Examples of blocks that can specify a bus object as their output data type include a Bus Creator block and a root Inport block. The blocks that cause an error when they have a virtual bus input in this situation are:

- Assignment
- Delay

The Delay block causes an error only if you use the Block Parameters dialog box to:

- Set an initial condition that is a MATLAB structure or zero.
- Specify a value for **State name**.
- Permute Dimension
- Reshape
- Selector
- Unit Delay

The Unit Delay block causes an error only if you use the Block Parameters dialog box to:

- Set an initial condition that is a MATLAB structure or zero.
- Specify a value for **State name**.
- Vector Concatenate

Results and Recommended Actions

Condition	Recommended Action
Virtual bus signal input to these blocks: <ul style="list-style-type: none"> • Assignment • Delay (if you specify an initial condition from the dialog box that is a MATLAB structure or zero and the value for State name is not empty) • Permute Dimension • Reshape • Selector • Unit Delay (if you specify an initial condition that is a MATLAB structure or zero and the value for State name is not empty) • Vector Concatenate 	In the Upgrade Advisor, click Modify . The check inserts a Bus to Vector block to attempt to convert virtual bus input signals to vector signals. For issues that the Upgrade Advisor identifies but cannot fix, modify the model manually. For details, see “Correct Buses Used as Vectors”.

Action Results

Clicking **Modify** inserts a Bus to Vector block at the input ports of blocks.

For many models, running the Upgrade Advisor modifies your model so that bus signals are not treated as vectors. However, for some models you can encounter compatibility issues even after running the check. Modify your model manually to address those issues.

After you compile the model using Upgrade Advisor, the Simulink Editor sometimes indicates that you need to save the model (the model is dirty), even though you did not make changes. To prevent this issue from reoccurring for this model, save the model.

Modeling Pattern	Issue	Solution
Data Store Memory block with Data Type set to <code>Inherit: auto</code>	A Data Store Memory block whose associated Data Store Read or Data Store Write blocks read or write bus signal data must use a bus object.	In the Data Store Memory block, set the Data Type signal attribute to <code>Bus: <BusObject></code> .

Modeling Pattern	Issue	Solution
Signal Conversion block Output parameter matches input bus type	<p>A Signal Conversion block whose Output parameter is set to <code>Nonvirtual</code> bus requires a virtual bus input.</p> <p>A Signal Conversion block whose Output parameter is set to <code>Virtual</code> bus requires a nonvirtual bus input.</p>	To create a copy of the input signal, set Output to <code>Signal copy</code> .
Merge, Switch, or Multiport Switch block with multiple bus inputs	Merge, Switch, or Multiport Switch blocks with multiple bus inputs require those inputs to have the same names and hierarchy.	Reconfigure the model so that the bus inputs have the same names and hierarchy.
Root Inport block outputting a virtual bus and specifying a value for Port dimensions	A root Inport block that outputs to a virtual bus must inherit the dimensions.	Set the Inport block Port dimensions signal attribute to <code>1</code> or <code>-1</code> (<code>inherit</code>).
Mux block with nonvirtual bus inputs	A Mux block cannot accept nonvirtual bus signals.	<p>To treat the output as an array, replace the Mux block with a Vector Concatenate block.</p> <p>If you want a virtual bus output, use a Bus Creator block to combine the signals.</p>
Bus to Vector block without a virtual bus signal input	A nonbus signal does not need a Bus to Vector block.	Remove the Bus to Vector block.
Assignment block with virtual bus inputs	The Upgrade Advisor converts the Assignment block <code>Y0</code> port bus input to a vector.	Add a Bus to Vector block before the Assignment block.

Modeling Pattern	Issue	Solution
S-function using a nonvirtual bus	An S-function that is not a Level-2 C S-function does not support nonvirtual bus signals.	Change the S-function to be a Level-2 C S-function. Consider using an S-Function Builder block to create a Level-2 C S-function.
Stateflow chart with parameterized data type	In a Stateflow chart, you cannot parameterize the data type of an input or output in terms of another input or output if the data type is a bus object.	For the parameterized port, set Data Type to Bus: <object name>.
Subsystem with bus operations in a Stateflow chart	An Inport block inside a subsystem in a Stateflow chart requires a bus object data type if its signal is a bus.	In the Inport block, set Data type to Bus: <object name>.
Ground block used as a bus source	The output signal of a Ground block cannot be a source for a bus.	Use a Constant block with Constant value set to 0 and the Output data type signal attribute set to Bus: <object name>.
Root Outputport block with a single-element bus object data type	The input to the Outputport block must be a bus if it specifies a bus object as its data type.	In the Outputport block, set Data type to Inherit: auto.

See Also

- Bus to Vector block
- “Correct Buses Used as Vectors”
- “Migrating to Simplified Initialization Mode Overview” on page 8-5
- `Simulink.BlockDiagram.addBusToVector`

Check for root outputports with constant sample time

Check ID: `mathworks.design.CheckConstRootOutputportWithInterfaceUpgrade`

Use this check to identify root outputports with a constant sample time used with an AUTOSAR target, Function Prototype Control, or the model C++ class interface.

Description

Root outports with constant sample time are not supported when using an AUTOSAR target, Function Prototype Control, or the model C++ class interface. Use this check to identify root Output blocks with this condition and modify the blocks as recommended.

Results and Recommended Actions

Condition	Recommended Action
Root outport with constant sample time used with an AUTOSAR target, Function Prototype Control or the model C++ class interface.	Consider one of the following: <ul style="list-style-type: none"> • Set the sample time of the block to the fundamental sample time. • Identify the source of the constant sample time and set its sample time to the fundamental sample time. • Place a Rate Transition block with inherited sample time (-1) before the block.

See Also

- “Consult the Upgrade Advisor”.

Analyze model hierarchy and continue upgrade sequence

Check ID: `com.mathworks.Simulink.UpgradeAdvisor.UpgradeModelHierarchy`

Check for child models and guide you through upgrade checks.

Description

This check identifies child models of this model, and guides you through upgrade checks to run both non-compile and compile checks. The Advisor provides tools to help with these tasks:

- If the check finds child models, it offers to run the Upgrade Advisor upon each child model in turn and continue the upgrade sequence. If you have a model hierarchy you need to check and update each child model in turn.

- If there are no child models, you still need to continue the check sequence until you have run both non-compile and compile checks.

You must run upgrade checks in this order: first the checks that do not require compile time information and do not trigger an Update Diagram, then the compile checks.

Click **Continue Upgrade Sequence** to run the next checks. If there are child models, this will open the next model. Keep clicking **Continue Upgrade Sequence** until the check passes.

Results and Recommended Actions

Condition	Recommended Action
Child models found	Click Continue Upgrade Sequence to run the next checks. If there are child models, this will close the current Upgrade Advisor session, and open Upgrade Advisor for the next model in the hierarchy.
No child models, but more checks to run	If there are no child models, click Continue Upgrade Sequence to refresh the Upgrade Advisor with compilation checks selected. The compile checks trigger an Update Diagram (marked with ^). Run the next checks and take advised actions. When you return to this check, click Continue Upgrade Sequence until this check passes.

Tips

Best practice for upgrading a model hierarchy is to check and upgrade each model starting at the leaf end and working up to the root model.

When you click **Continue Upgrade Sequence**, the Upgrade Advisor opens the leaf model as far inside the hierarchy as it can find. Subsequent steps guide you through upgrading your hierarchy from leaf to root model.

When you open the Upgrade Advisor, the checks that are selected do not require compile time information and do not trigger an Update Diagram. Checks that trigger an Update

Diagram are not selected to run by default, and are marked with \wedge . When you use the Upgrade Advisor on a hierarchy, keep clicking **Continue Upgrade Sequence** to move through this sequence of analysis:

- 1 The Upgrade Advisor opens each model and library in turn, from leaf to root, and selects the non-compile checks. Run the checks, take any advised actions, then click **Continue Upgrade Sequence** to open the next model and continue.
- 2 When you reach the root end of the hierarchy, the Upgrade Advisor then opens each model again in the same order (but not libraries) and selects only the checks that require a model compile. Run the checks, take any advised actions, then click **Continue Upgrade Sequence** to open the next model. Continue until you reach the end of the hierarchy and this check passes.

See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”

Check Access to Data Stores

Check ID: `mathworks.design.ConflictsForDataStoreReadWriters`

Identify potential execution order sensitivity when reading and writing to data stores.

Description

The execution order of blocks that read and write to the same data store can change the simulation result. When blocks in the same hierarchy access the same data store, the execution order is not deterministic.

Results and Recommended Actions

Condition	Recommended Action
Data Store Memory block accessed by multiple blocks in the same hierarchy.	To enforce execution order for the blocks, consider the following: <ul style="list-style-type: none"><li data-bbox="635 423 1203 453">• Add a data dependency between the blocks.<li data-bbox="635 465 887 494">• Set block priority.<li data-bbox="635 506 1307 571">• Move blocks into separate Function-Call Subsystem blocks and schedule them.

See Also

- “Local and Global Data Stores”
- Data Store Memory
- Data Store Read
- Data Store Write

Model Reference Conversion Advisor

Model Reference Conversion Advisor

Check Conversion Input Parameters

Use input parameters to configure the actions the advisor performs and the output it produces.

You can use the default parameters to run the advisor without changing any parameters.

Input Parameter	Description
New model name	<p>The advisor provides a model name that is based on the Subsystem block name and is unique in the MATLAB path.</p> <p>The model name cannot exceed 59 characters.</p> <hr/> <p>Tip If the advisor generates an error indicating that the target referenced model already exists, then use the New model name parameter to specify a new file name.</p>
Conversion data file name	<p>The advisor creates a file for storing data created during the conversion. By default, the advisor uses the model name at the beginning of the file name and appending <code>_conversion_data.mat</code>. For example, if the subsystem is in a model named <code>myModel</code>, the conversion file name is <code>myModel_conversion_data.mat</code>.</p> <p>You can save the conversion data in a MAT-file (default) or a MATLAB file. If you use a <code>.m</code> file extension, the advisor serializes all variables to a MATLAB file.</p> <hr/> <p>Note If the top model uses a data dictionary, you cannot select this option.</p>
Fix errors automatically	<p>By default, if an advisor check finds any errors and the advisor can fix the error, the advisor provides a Fix button that you can click to have the advisor fix the issue.</p> <p>If you enable this parameter, the advisor fixes all conversion errors that it can, without displaying the Fix button.</p>

Input Parameter	Description
Replace content of a subsystem with a Model block	<p>By default, the advisor updates the original model by inserting a Model block in the model. The advisor action depends on whether you use the automatic fix options.</p> <ul style="list-style-type: none">• If you use the automatic fixes, then the advisor replaces the Subsystem block with a Model block unless the automatic fixes change the input or output ports. If the ports change, then the advisor includes the contents of the subsystem in a Model block that is inserted in the Subsystem block.• If you do not use the automatic fixes, then the advisor replaces the Subsystem block with a Model block. <p>Clear this parameter to have the advisor open a new Simulink Editor window that contains only a Model block that references the newly created referenced model. The advisor does not update the original model in the other Simulink Editor window.</p> <hr/> <p>Note If you are converting a variant subsystem, do not use this option. See “Convert Each Variant Subsystem”.</p>

Input Parameter	Description
Check simulation results after conversion	<p>Compare the results of simulating the top model for the referenced model to the results of simulating the baseline model that has the subsystem.</p> <p>To use this option, before performing the conversion, enable signal logging for the subsystem output signals of interest in the model. Set these advisor options:</p> <ul style="list-style-type: none"> • Model block simulation mode — Use the same simulation mode as in the original model. • Replace content of a subsystem with a Model block — Enable this option. • Stop time — Specify when you want the simulations to end. The default is the stop time of the top model. If the top model stop time is set to <code>inf</code>, the advisor stops after 10 seconds. • Absolute tolerance — Specify a value if you do not want to use the default of <code>'1e-06'</code>. • Relative tolerance — Specify a value if you do not want to use the default of <code>'1e-03'</code>. <p>To see the results after the conversion is complete, click View comparison results. The advisor displays the results of the comparison in the Simulation Data Inspector. For more information, see “Compare Simulation Results Before and After Conversion”.</p>
Stop time	<p>By default, the advisor uses the stop time of the top model, unless the stop time of the top model is <code>inf</code>. If the stop time of the top model is <code>inf</code>, the advisor uses a default stop time of 10. You can specify a different stop time. For details, see “Specify Simulation Start and Stop Time”.</p> <p>To use this option, select Check simulation results after conversion.</p>

Input Parameter	Description
Absolute tolerance	The absolute signal tolerance for the simulation run comparison. The default is $1e-06$. To use this option, select Check simulation results after conversion .
Relative tolerance	The relative signal tolerance for the simulation run comparison. The default is $1e-03$. To use this option, select Check simulation results after conversion .
Model block simulation mode	Simulation mode for the new Model block that references the referenced model. <ul style="list-style-type: none">• Normal (default)• Accelerator

After you configure the advisor, to start the conversion checks, click **Run this task**.

Performance Advisor Checks

Simulink Performance Advisor Checks

In this section...

- “Simulink Performance Advisor Check Overview” on page 10-2
- “Baseline” on page 10-3
- “Checks that Require Update Diagram” on page 10-3
- “Checks that Require Simulation to Run” on page 10-3
- “Check Simulation Modes Settings” on page 10-3
- “Check Compiler Optimization Settings” on page 10-4
- “Create baseline” on page 10-4
- “Identify resource-intensive diagnostic settings” on page 10-4
- “Check optimization settings” on page 10-4
- “Identify inefficient lookup table blocks” on page 10-5
- “Check MATLAB System block simulation mode” on page 10-5
- “Identify Interpreted MATLAB Function blocks” on page 10-6
- “Identify simulation target settings” on page 10-6
- “Check model reference rebuild setting” on page 10-7
- “Identify Scope blocks” on page 10-7
- “Identify active instrumentation settings on the model” on page 10-7
- “Check model reference parallel build” on page 10-8
- “Check Delay block circular buffer setting” on page 10-10
- “Check continuous and discrete rate coupling” on page 10-10
- “Check discrete signals driving derivative port” on page 10-11
- “Check solver type selection” on page 10-11
- “Select simulation mode” on page 10-12
- “Select compiler optimizations on or off” on page 10-13
- “Final Validation” on page 10-14

Simulink Performance Advisor Check Overview

Use Performance Advisor checks to improve model simulation time.

See Also

“Improve Simulation Performance Using Performance Advisor”

Baseline

Establish a measurement to compare the performance of a simulation after Performance Advisor implements improvements.

See Also

“Create a Performance Advisor Baseline Measurement”

Checks that Require Update Diagram

These checks require that **Update Diagram** occurs in order to run.

See Also

“Improve Simulation Performance Using Performance Advisor”

Checks that Require Simulation to Run

These checks require simulation to run in order to collect sufficient performance data. Performance Advisor reports the results after simulation completes.

See Also

“Improve Simulation Performance Using Performance Advisor”

Check Simulation Modes Settings

These checks evaluate simulation modes (Normal, Accelerator, Rapid Accelerator, Rapid Accelerator with up-to-date check off) and identify the optimal mode to achieve fastest simulation.

See Also

“What Is Acceleration?”

Check Compiler Optimization Settings

Use these checks to select compiler optimization settings for improved performance.

See Also

“Compiler optimization level”

Create baseline

Select this check to create a baseline when Performance Advisor runs. You can also create a baseline manually. A baseline is the measurement of simulation performance before you run checks in Performance Advisor. The baseline includes the time to run the simulation and the simulation results (signals logged). Before you create a baseline for a model, in the **Data Import/Export** pane of the Configuration Parameters dialog box:

- Select the **States** check box.
- Set the **Format** parameter to `Structure with time`.

See Also

“Create a Performance Advisor Baseline Measurement”

Identify resource-intensive diagnostic settings

To improve simulation speed, disable diagnostics where possible. For example, some diagnostics, such as **Solver data inconsistency** or **Array bounds exceeded**, incur run-time overheads during simulations.

See Also

- “Diagnostics”
- “Improve Simulation Performance Using Performance Advisor”

Check optimization settings

To improve simulation speed, enable optimizations where possible. For example, if some optimizations, such as Block Reduction, are disabled, enable these optimizations to improve simulation speed.

You can also trade off compile-time speed for simulation speed by setting the compiler optimization level. Compiler optimizations for accelerations are disabled by default. Enabling them accelerates simulation runs but results in longer build times. The speed and efficiency of the C compiler used for Accelerator and Rapid Accelerator modes also affects the time required in the compile step.

See Also

- “Optimization Pane: General”
- “Improve Simulation Performance Using Performance Advisor”

Identify inefficient lookup table blocks

To improve simulation speed, use properly configured lookup table blocks.

See Also

- “Lookup Tables”
- “Optimize Generated Code for Lookup Table Blocks”
- “Optimize Breakpoint Spacing in Lookup Tables”
- “Improve Simulation Performance Using Performance Advisor”

Check MATLAB System block simulation mode

In general, to improve simulation speed, choose `Code generation` for the **Simulate using** parameter of the MATLAB System block. Because data exchange between MATLAB and Simulink passes through several software layers, `Interpreted execution` usually slows simulations, particularly if the model needs many data exchanges.

This check identifies which MATLAB System blocks can generate code and changes the **Simulate using** parameter value to `Code generation` where possible.

While `Code generation` does not support all MATLAB functions, the subset of the MATLAB language that it does support is extensive. By using this `Code generation`, you can improve performance.

See Also

- MATLAB System
- “Simulation Modes”
- “Improve Simulation Performance Using Performance Advisor”

Identify Interpreted MATLAB Function blocks

To improve simulation speed, replace Interpreted MATLAB Function blocks with MATLAB Function blocks where possible. Because data exchange between MATLAB and Simulink passes through several software layers, Interpreted MATLAB Function blocks usually slow simulations, particularly if the model needs many data exchanges.

Additionally, because you cannot compile an Interpreted MATLAB Function, an Interpreted MATLAB Function block impedes attempts to use an acceleration mode to speed up simulations.

While MATLAB Function blocks do not support all MATLAB functions, the subset of the MATLAB language that it does support is extensive. By replacing your interpreted MATLAB code with code that uses only this embeddable MATLAB subset, you can improve performance.

See Also

- MATLAB Function
- “Improve Simulation Performance Using Performance Advisor”

Identify simulation target settings

To improve simulation speed, disable simulation target settings where possible. For example, in the Configuration Parameters dialog box, clear the **Simulation Target > Echo expression without semicolons** check box to improve simulation speed.

See Also

- “Model Configuration Parameters: Simulation Target”
- “Improve Simulation Performance Using Performance Advisor”

Check model reference rebuild setting

To improve simulation speed, in the Configuration Parameters dialog box, verify that the **Model Referencing > Rebuild** parameter is set to `If any changes in known dependencies detected`.

See Also

- “Rebuild”
- “Improve Simulation Performance Using Performance Advisor”

Identify Scope blocks

Opened and uncommented Scope blocks can impact simulation performance. To improve simulation performance, close and comment out Scope blocks. Right-click a scope block, and then select **Comment Out**.

For opened Scopes, you can improve simulation speed by reducing updates. From the Scope **Simulation** menu, select **Reduce Updates to Improve Performance**.

See Also

- “Improve Simulation Performance Using Performance Advisor”

Identify active instrumentation settings on the model

Identify active instrumentation settings on the model. The fixed-point instrumentation mode controls which objects log minimum, maximum, and overflow data during simulation. Instrumentation is required to collect simulation ranges using the Fixed-Point Tool. These ranges are used to propose data types for the model (requires Fixed-Point Designer). When you are not actively converting your model to fixed point, disable the fixed-point instrumentation to restore the maximum simulation speed to your model.

From the model **Analysis** menu, select **Data Type Design > Fixed-Point Tool**. Under **System under design**, click **Continue**.

In the Model Hierarchy pane, the Fixed-Point Tool denotes systems that currently have instrumentation turned on with `(mm0)`, or `(o)`. Right-click the system in the model hierarchy and, under `Fixed-point instrumentation mode`, select `Use local settings` or `Force off`.

See Also

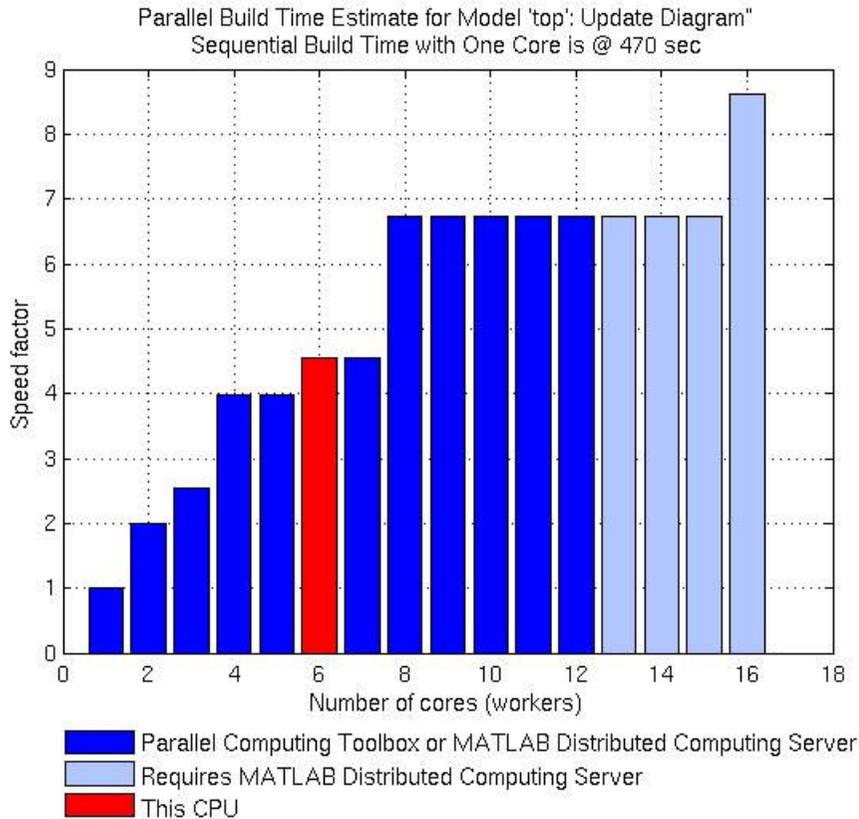
- Fixed-Point Instrumentation and Data Type Override (Fixed-Point Designer)

Check model reference parallel build

To improve simulation, verify the number of referenced models in the model. If there are two or more referenced models, build the model in parallel if possible.

Performance Advisor analyzes the model and estimates the build time on the current computer as if it were using several cores. It also estimates the parallel build time for the model in the same way an estimation would be performed if Parallel Computing Toolbox or MATLAB Distributed Computing Server™ software were installed on the computer. Performance Advisor performs this estimate as follows:

- 1** Search the model for referenced models that do not refer to other referenced models.
- 2** Calculate the average number of blocks in each of the referenced models that do not refer to other referenced models.
- 3** Of the list of referenced models that do not refer to others, select a referenced model whose number of blocks is closest to the calculated average.
- 4** Build this model to obtain the build time.
- 5** Based on the number of blocks and the build time for this referenced model, estimate the build time for all other referenced models.
- 6** Based on these build times, estimate the parallel build time for the top model.



To calculate the overhead time introduced by the parallel build mechanism, set the Parallel Build Overhead Time Estimation Factor. Performance Advisor calculates the estimated build time with overhead as:

$$(1 + \text{Parallel Build Overhead Time Estimation Factor}) * (\text{Build time on a single machine})$$

See Also

- “Enable parallel model reference builds”
- “Improve Simulation Performance Using Performance Advisor”

Check Delay block circular buffer setting

To improve simulation, check that each Delay block in the model uses the appropriate buffer type. By default, the block uses an array buffer (the **Use circular buffer for state** option is not selected). However, when the delay length is large, a circular buffer can improve execution speed by keeping the number of copy operations constant.

If the Delay block is currently using an array buffer, and all of the following conditions are true, Performance Advisor selects a circular buffer:

- The Delay block is in sample-based mode, i.e., either the **Input processing** parameter is set to `Elements as channels (sample based)`, or the input signal type is set to `Sample based`.
- The value or upper limit of the delay length is 10 or greater.
- The size of the state—equal to the delay length multiplied by the total of all output signal widths—is 1000 or greater.

See Also

- Delay
- “Improve Simulation Performance Using Performance Advisor”

Check continuous and discrete rate coupling

If your model contains both discrete and continuous rates, the coupling between these rates can slow down simulation. Performance Advisor checks for these conditions in your model.

- The model is using a variable step solver.
- The model contains both continuous and discrete rates.
- The fastest discrete rate is relatively smaller than **Max step size** determined by the solver.

Setting the `DecoupledContinuousIntegration` parameter to `on` might speed up simulation.

See Also

- “About Solvers”

- “Speed Up Simulation”
- “Improve Simulation Performance Using Performance Advisor”

Check discrete signals driving derivative port

Run this check if your simulation has many unnecessary resets. A discrete signal driving a block with continuous states triggers a reset at every sample time hit of the block. These resets are computationally expensive. Performance Advisor checks for these signals and blocks and provides a list of the same.

You can edit the model around the discovered discrete signals that drive these blocks to remove such cases. For example, inserting a Zero Order Hold block between the discrete signal and the corresponding block with continuous states might help resolve the issue.

See Also

- “Speed Up Simulation”
- “Modeling Techniques That Improve Performance”

Check solver type selection

To improve simulation, check that the model uses the appropriate solver type.

Explicit vs. Implicit Solvers

Selecting a solver depends on the approximation of the model stiffness at the beginning of the simulation. A stiff system has both slowly and quickly varying continuous dynamics. Implicit solvers are specifically designed for stiff problems, whereas explicit solvers are designed for non-stiff problems. Using non-stiff solvers to solve stiff systems is inefficient and can lead to incorrect results. If a non-stiff solver uses a very small step size to solve your model, check to see if you have a stiff system.

Model	Recommended Solver
Represents a stiff system	ode15s
Does not represent a stiff system	ode45

Performance Advisor uses the heuristic shown in the table to choose between explicit and implicit solvers.

Original Solver	Performance Advisor Action
Variable step solver	Calculates the system stiffness at 0 first. Then: <ul style="list-style-type: none"> • If the stiffness is greater than 1000, Performance Advisor chooses ode15s. • If the stiffness is less than 1000, Performance Advisor chooses ode45.
Fixed-step continuous solver	<ul style="list-style-type: none"> • If the stiffness is greater than 1000, Performance Advisor chooses ode14x. • If the stiffness is less than 1000, Performance Advisor chooses ode3.

This heuristic works best if the system stiffness does not vary during simulation. If the system stiffness varies with time, choose the most appropriate solver for that system rather than the one Performance Advisor suggests.

See Also

- “About Solvers”
- “Speed Up Simulation”
- “Improve Simulation Performance Using Performance Advisor”

Select simulation mode

To achieve fastest simulation time, use this check to evaluate the following modes and identify the optimal selection:

- Normal
- Accelerator
- Rapid Accelerator
- Rapid Accelerator with up-to-date check off

In Normal mode, Simulink interprets your model during each simulation run. If you change the model frequently, this is generally the preferred mode to use because it requires no separate compilation step. It also offers the most flexibility to make changes to your model.

In Accelerator mode, Simulink compiles a model into a binary shared library or DLL where possible, eliminating the block-to-block overhead of an interpreted simulation in

Normal mode. Accelerator mode supports the debugger and profiler, but not runtime diagnostics.

In Rapid Accelerator mode, simulation speeds are fastest but this mode only works with models where C-code is available for all blocks in the model. Also, this mode does not support the debugger or profiler.

When choosing Rapid Accelerator with up-to-date check off, Performance Advisor does not perform an up-to-date check during simulation. You can run the Rapid Accelerator executable repeatedly while tuning parameters without incurring the overhead of up-to-date checks. For instance, if you have a large model or a model that makes extensive use of model reference, this method of execution can increase efficiency.

For models with 3-D signals, Normal or Accelerator modes work best.

See Also

- “How Acceleration Modes Work”
- “Choosing a Simulation Mode”
- “Comparing Performance”
- “Run Simulations Programmatically”

Select compiler optimizations on or off

Use this check to determine whether performing compiler optimization can help improve simulation speed. The optimization can only be performed in Accelerator or Rapid Accelerator modes.

Note This check will be skipped if MATLAB is not configured to use an optimizing compiler.

See Also

- “How Acceleration Modes Work”
- “Choosing a Simulation Mode”
- “Comparing Performance”
- “Improve Simulation Performance Using Performance Advisor”

Final Validation

This check validates the overall performance improvement of simulation time and accuracy in a model. If the performance is worse than the original model, Performance Advisor discards all changes to the model and loads the original model.

Global settings for validation do not apply to this check. If you have not validated the performance improvement from changes resulting from other checks, use this check to perform a final validation of all changes to a model.

See Also

- “Comparing Performance”
- “Improve Simulation Performance Using Performance Advisor”

Simulink Limits

Maximum Size Limits of Simulink Models

The following table documents some limits on the size and complexity of Simulink models.

Model Feature	Limit
Maximum number of levels in a block diagram	1024
Maximum number of branches in a line	1024
Maximum length of a parameter name	63
Maximum length of a parameter character vector value	32768
Maximum value of a model window coordinate	32768
Maximum number of bytes of logged simulation data	$2^{31}-1$ bytes on 32-bit systems, $2^{48}-1$ bytes on 64-bit systems
Maximum number of bytes for the total block I/O buffer length in a model	$2^{31}-1$ bytes on 32-bit systems and on 64-bit systems
Maximum length of integer and fixed-point data types	128 bits

Block Reference Page Examples

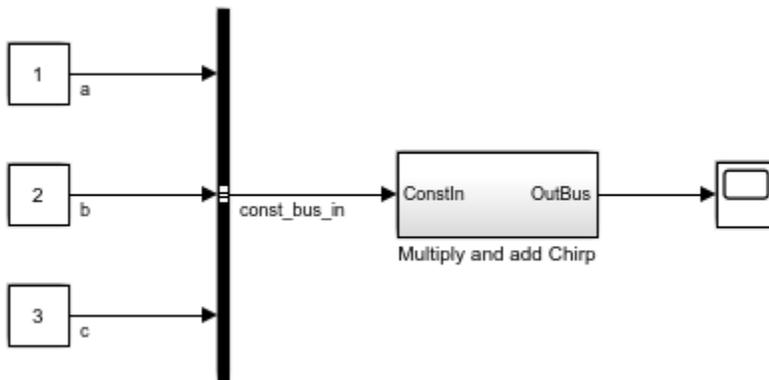
- “Create Bus Ports in a Subsystem” on page 12-2
- “Convert Bus Signal to a Vector” on page 12-5
- “Assign Signal Values to a Bus” on page 12-6
- “Initialize Your Model Using the Callback Button Block” on page 12-7
- “Control a Parameter Value with Callback Button Blocks” on page 12-9
- “Solve a Linear System of Algebraic Equations” on page 12-12
- “Model a Planar Pendulum” on page 12-13

Create Bus Ports in a Subsystem

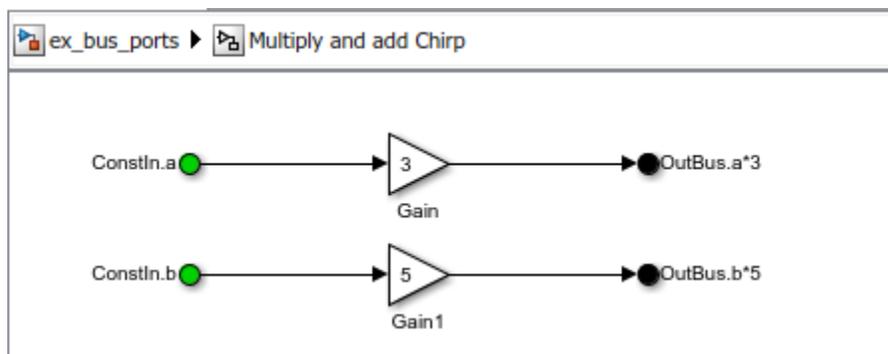
This example shows adding In Bus Element and Out Bus Element blocks to create bus element ports in a subsystem for selecting signals from an input bus and creating an output bus signal.

Model Structure

Open the model. The top model has three constant signals combined into a bus that feeds a subsystem input port outputs a bus signal to a Scope block.

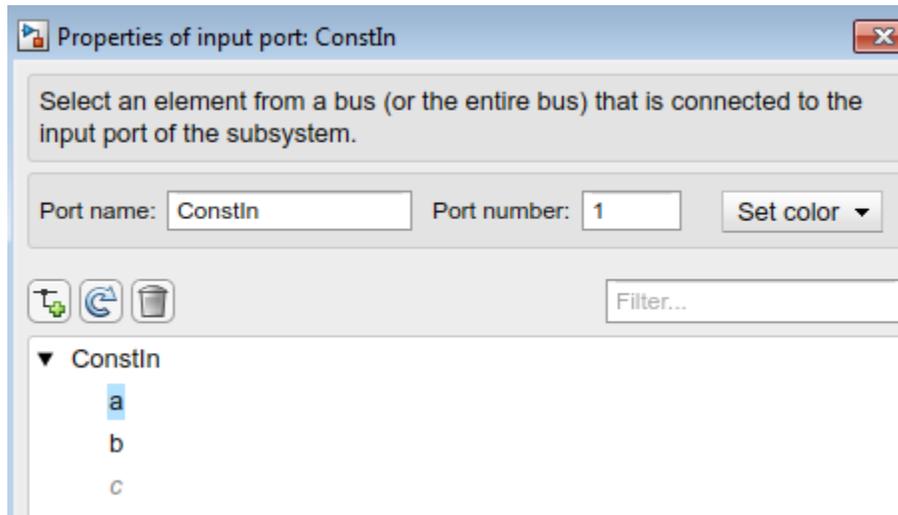


The subsystem includes two In Bus Element and Out Bus Element blocks.



Set the Port Name and Color for an In Bus Element Block

Open the Block Parameters dialog box for the ConstIn.a block. The Port name parameter is set to ConstIn, which changes the subsystem input input port name from the default InBus. This block selects the a signal. The block color is set to green instead of the default black.



The In Bus Element block feeds a Gain block, and the Gain block connects to an Out Bus Element block that includes an $a*3$ signal in the output bus signal.

Create an In Bus Element Block and an Out Bus Element Block

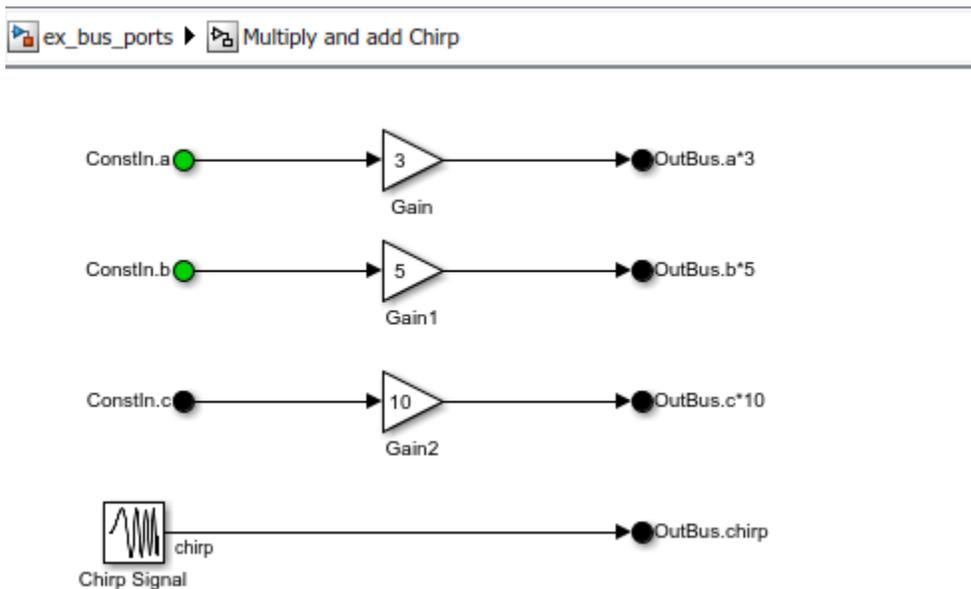
Create a third In Bus Element block for the c signal. In the Simulink Editor, right-click and drag the ConstIn.b block to make a copy of the block. Specify to use the existing port. Edit the block icon text to say ConstIn.c. Select the block and override the green block color by specifying black. Feed the output of the In Bus Element block to a third Gain block and set the gain to 10.

Copy the OutBus.b*5 block to create another Out Bus Element block, specifying to use the same port. Connect the Gain block to the Out Bus Element block that uses the same port. Connect the Gain block output signal to the Out Bus Element block and edit the icon text to say OutBus.c*10.

Add a Chirp Block and Include Its Output Signal in the Bus Output Signal

Add a Chirp Signal block and connect it to a new Out Bus Element block that uses the same port as the other Out Bus Element blocks (OutBus). Open the Out Bus Element dialog box, double-click the selected signal, and change the signal name to chirp.

The virtual bus signal that the subsystem outputs contains the output signals of the three Gain blocks and the Chirp block.

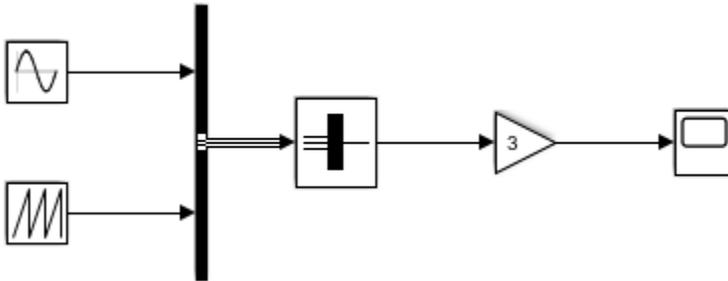


Convert Bus Signal to a Vector

This example shows how to use a Bus to Vector block to convert a bus signal to a vector, to provide a signal that the Gain block can accept.

Open a Simulink® model and simulate it.

```
model = fullfile(matlabroot, 'examples', 'simulink', 'ex_bus_to_mux_ok');  
open_system(model);  
sim(model)
```

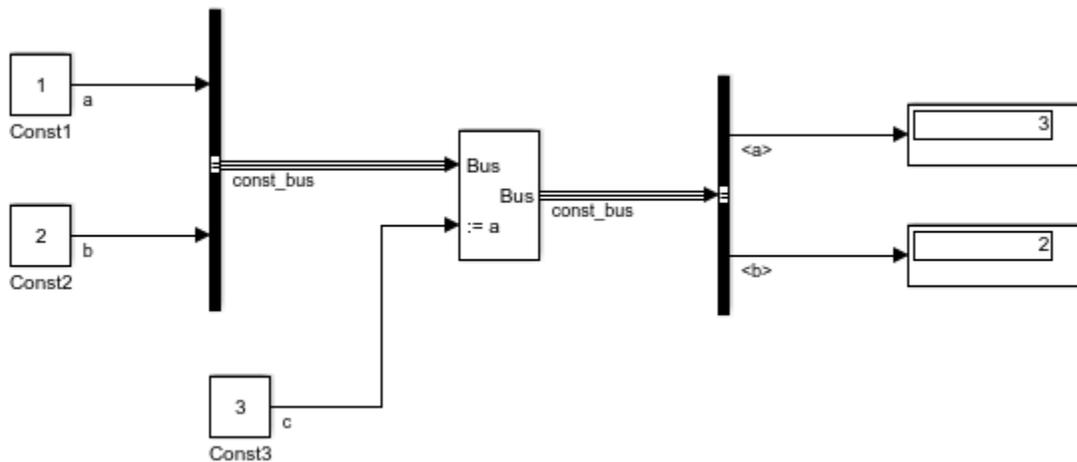


The Gain block cannot accept a bus signal. Inserting a Bus to Vector block provides the type of input signal (a vector) that the Gain block can accept. The Bus to Vector block has no user-accessible parameters.

Assign Signal Values to a Bus

This example shows how to use a Bus Assignment block to change a bus element value without adding Bus Selector and Bus Creator blocks to select bus elements and reassemble them into a bus.

Open the model and simulate it.



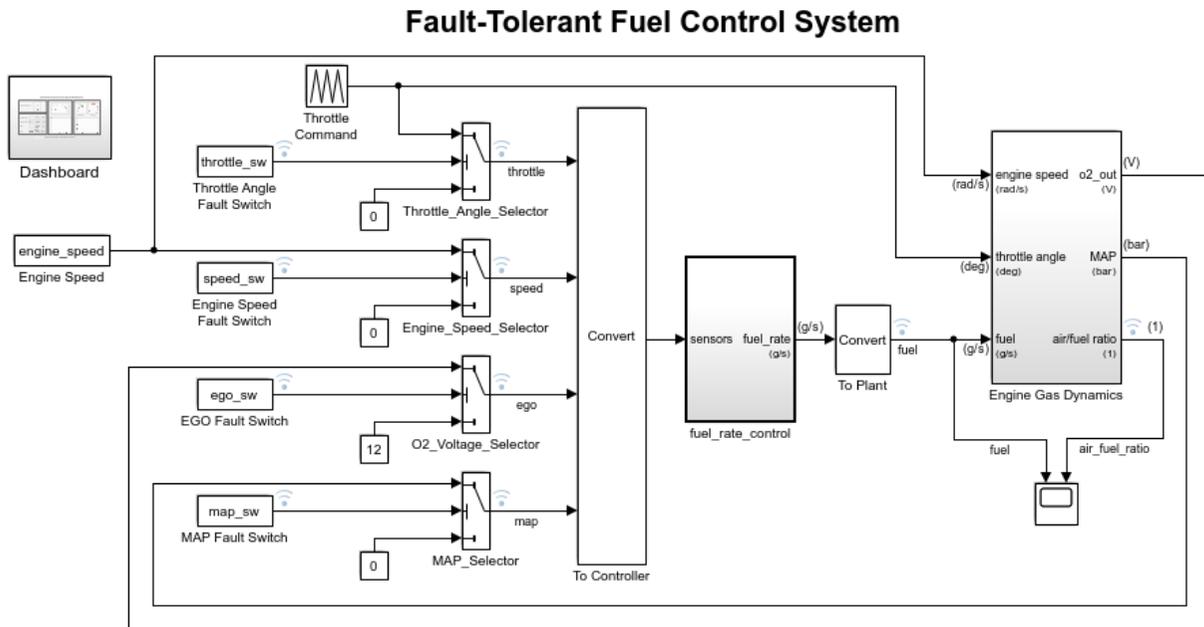
Initially, the value of signal a is 1. However, the Bus Assignment block replaces that initial value of signal a with the value of signal c, which is 3. The const_bus output signal has a value of 3 for signal a, as the Display block shows.

Initialize Your Model Using the Callback Button Block

This example shows how to use the Callback Button block to perform initialization routines on your model.

Explore the Model

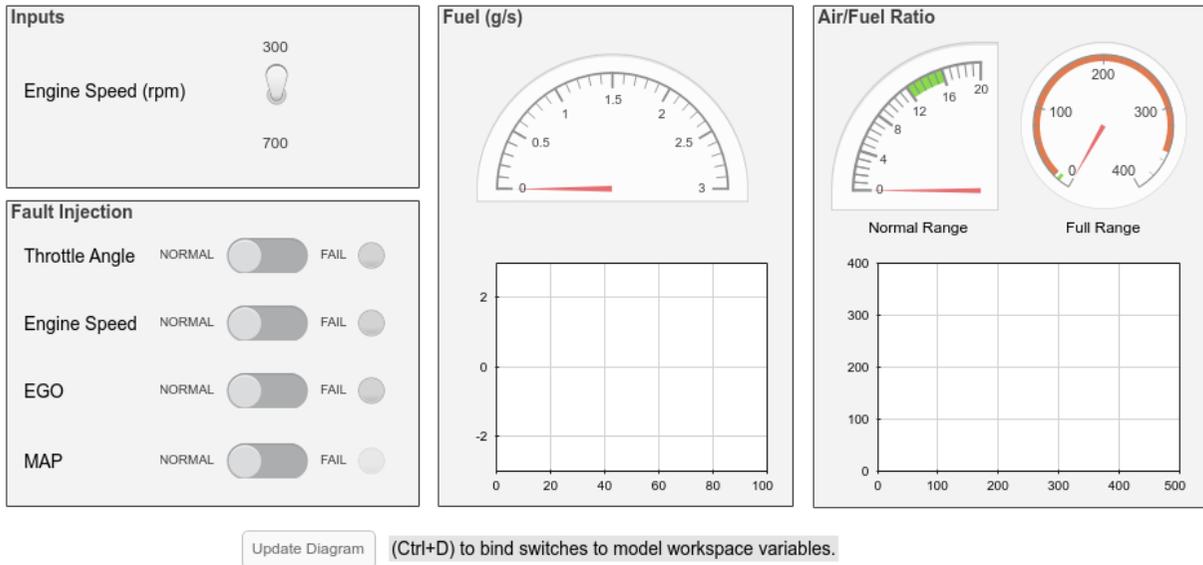
The example model builds on the `sldemo_fuelsys` featured model. When you open the model, to bind the workspace variables to their Dashboard blocks, you have to update the model diagram. Here, the Callback Button block at the bottom of the Dashboard subsystem in the model has been configured to update the diagram on the release of the mouse button when you click the block.



[Open the Dashboard](#) subsystem to simulate any combination of sensor failures.

Copyright 1990-2016 The MathWorks, Inc.

Fault-Tolerant Fuel Control System Dashboard



You do not need to start a simulation for the Callback Button to react to your input. Just select and then click the Callback Button to run the initialization code. Double-click the Callback Button block to view and edit its parameters, including the press and click scripts.

See Also

Callback Button

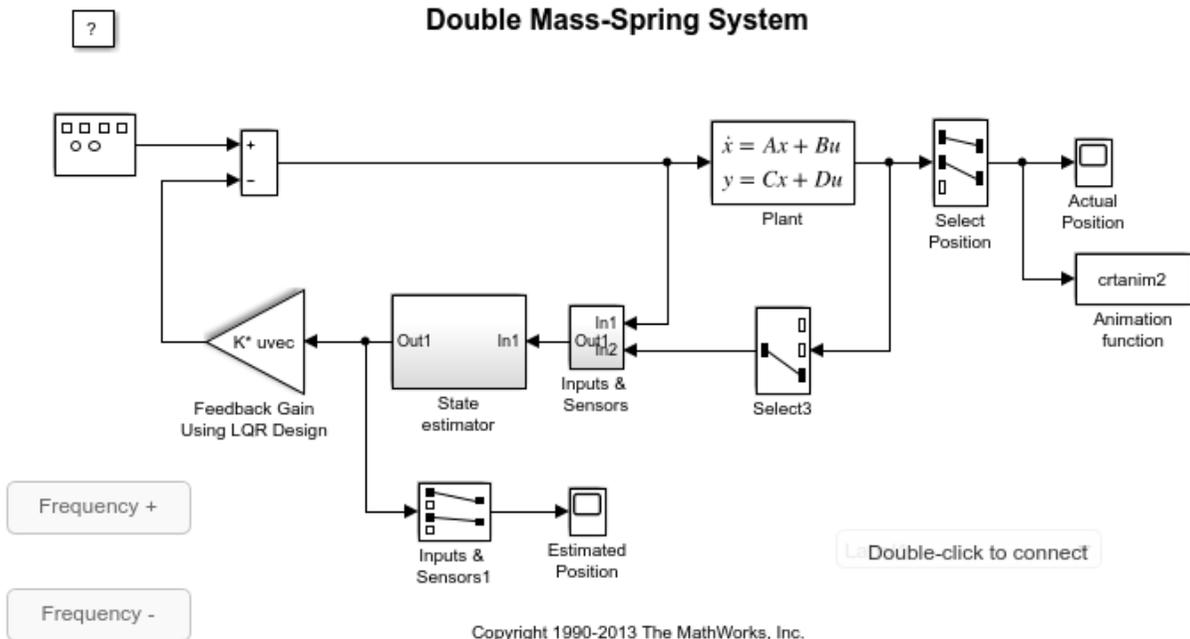
“Control a Parameter Value with Callback Button Blocks” on page 12-9

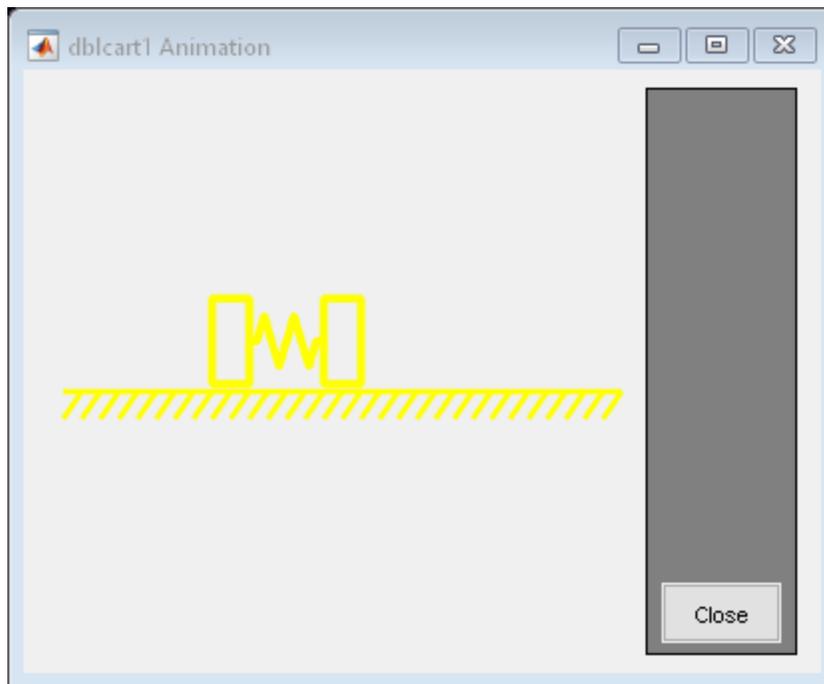
Control a Parameter Value with Callback Button Blocks

This example models control of a system that consists of two masses attached on either side of a spring. A control loop damps the oscillation of the spring that results when an external force acts on the system. The model uses Callback Button blocks to provide an interface for you to adjust the frequency of the external force before and during simulation.

Explore the Model

The model for this example adds two Callback Button blocks, labeled Frequency + and Frequency - to the Double Mass-Spring System model. When you simulate the model, an animation visualizes the system.





Click the button labeled `Frequency +` to increase the oscillation frequency. When you adjust the frequency of the external force, the `Callback Button` block displays a message in the command window indicating the new frequency value. You can adjust the parameter during a simulation and while the model is idle.

Both `Callback Button` blocks in this model are configured with a `ClickFcn` that responds to your clicks and a `PressFcn` that executes when you press the `Callback Button` block. Double-click the `Frequency + Callback Button` block to view its parameters.

When you click the `Frequency + Callback Button` block, the `ClickFcn` increases the frequency of the external force by `0.1`. If you press the `Callback Button` block for more than the `500 ms Press Delay`, the `PressFcn` increases the frequency of the external force by `0.1` every second.

See Also

[Callback Button](#)

“Initialize Your Model Using the Callback Button Block” on page 12-7

Solve a Linear System of Algebraic Equations

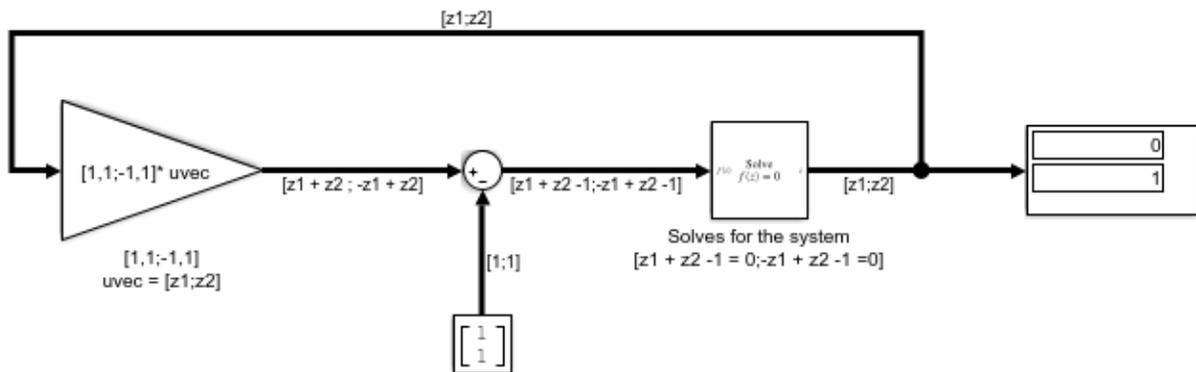
Use the Algebraic Constraint block to solve the system

$$z_1 + z_2 = 1$$

$$z_2 - z_1 = 1$$

The model represents the problem in a vectorized form as

$$\begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$



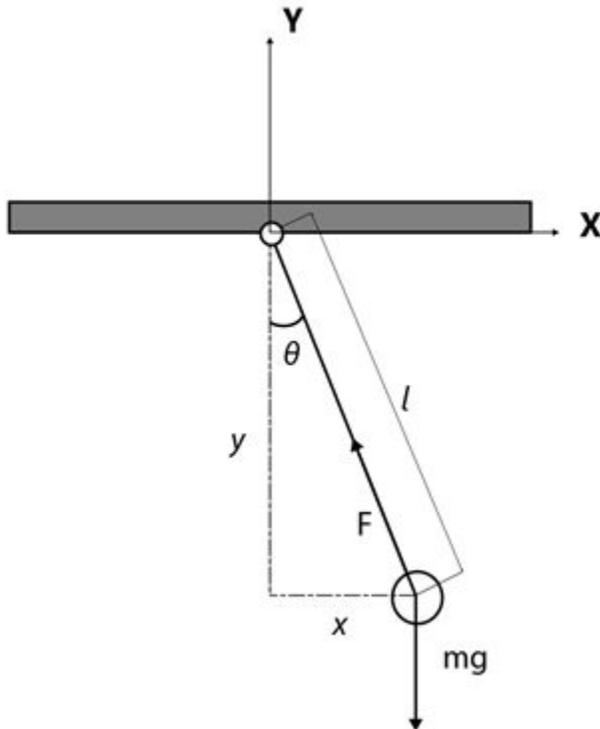
The signal fed to the Algebraic Constraint block $f(z)$ is a 2×1 vector of the form

$$\begin{bmatrix} z_1 + z_2 - 1 \\ -z_1 - 1 + z_2 - 1 \end{bmatrix}$$

The block is configured to constrain $f(z)$ to 0. Thus solving for $f(z) = 0$ yields the solution $z_1 = 0, z_2 = 1$

Model a Planar Pendulum

Consider a point mass m suspended by a massless rod of length l under the influence of gravity. The position of the mass can be expressed in Cartesian coordinates by (x,y) .



Modeling the System

A force balance of the mass gives the equations of motion in the x and y directions.

$$m\ddot{x} = F \sin \theta \quad (1)$$

$$m\ddot{y} + F \cos \theta = -mg \quad (2)$$

Let (u, v) be the velocities in (x, y) respectively. The system can be rewritten as a system of first order ODEs

$$\dot{x} = u \quad (3)$$

$$\dot{u} = -F \frac{x}{ml} \quad (4)$$

$$\dot{y} = v \quad (5)$$

$$\dot{v} = -F \frac{y}{ml} - g \quad (6)$$

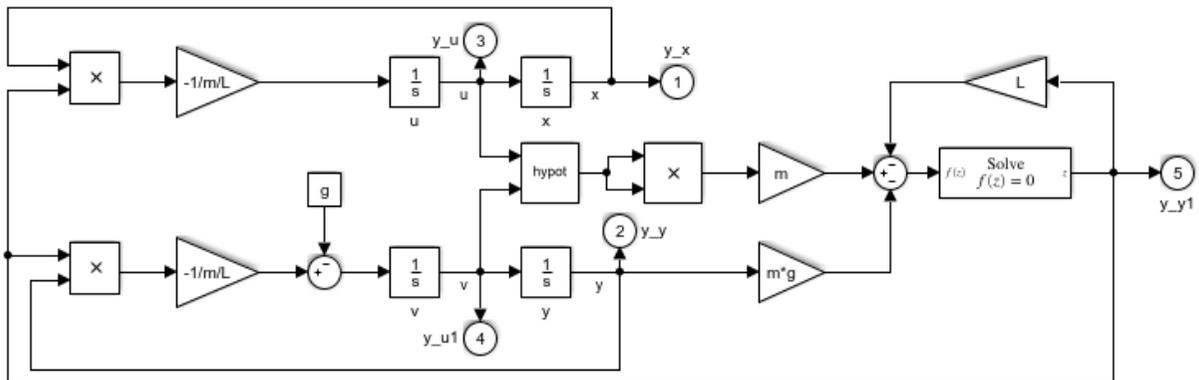
where F is the tension in the rod. The system also possesses the geometric constraint $x^2 + y^2 = l^2$ (7)

Differentiate (7) twice with respect to time t to arrive at $m(u^2 + v^2) - Fl - mgy = 0$ (8)

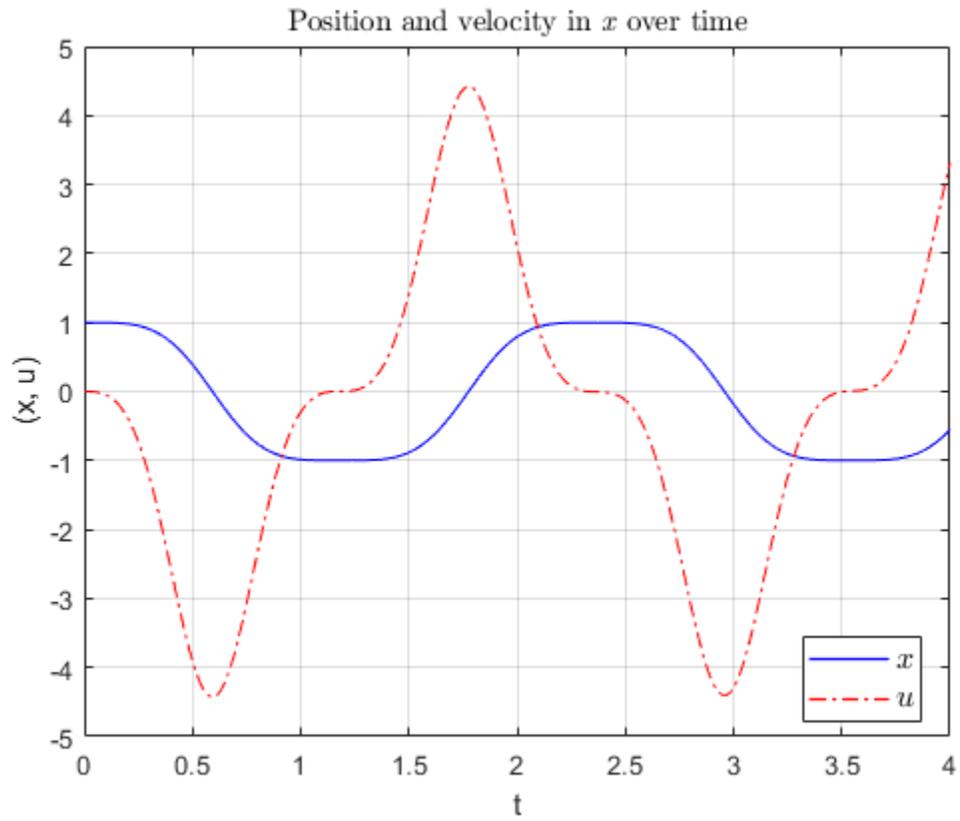
This relationship is useful since it allows F to be determined at every step for use in modeling the kinematics of the system.

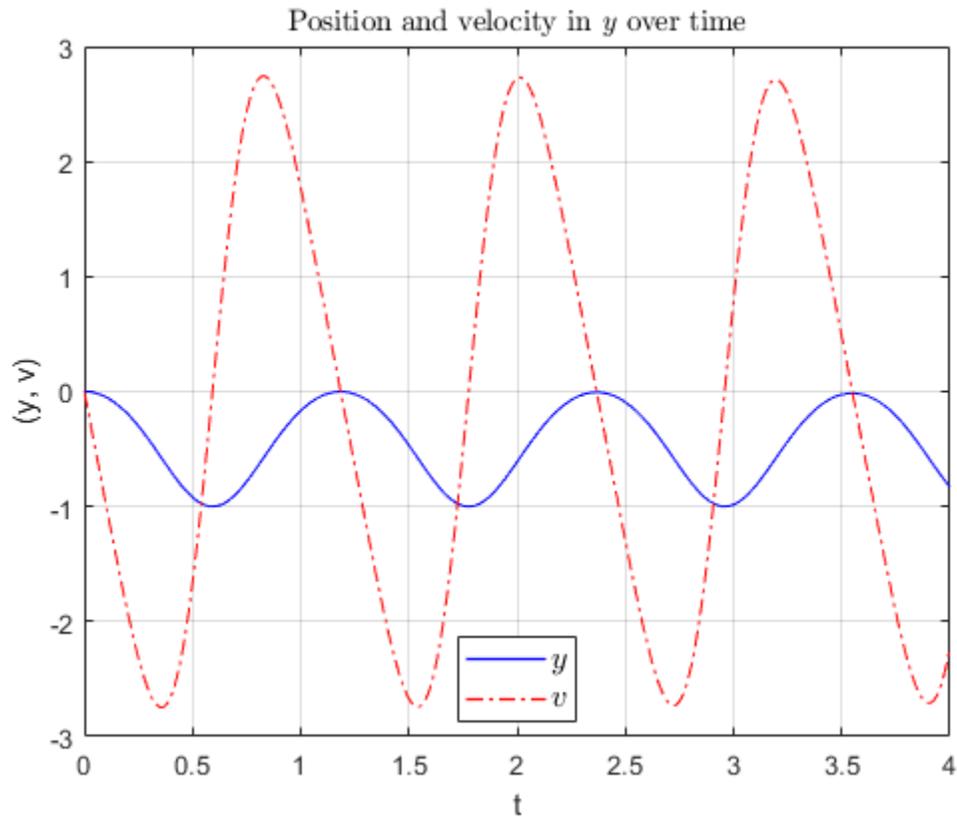
Simulating the System

The system is simulated as shown in the figure below



Equation (8) contains one unknown F and is of the form $f(z) = 0$ where $f(z) = m(u^2 + v^2) - Fl - mgy$. The Algebraic Constraint block constrains $f(z)$ to 0 and solves for F in accordance with (8).





References

Hairer, Ernst, Christian Lubich, and Michel Roche. "The Numerical Solution Of Differential-Algebraic Systems By Runge-Kutta Methods." *Lecture Notes in Mathematics*. Vol. 1409, Berlin: Springer-Verlag, 1989: pp. 8-9.